**ABSTRACT**
A compiler’s optimizer operates over abstract syntax trees (ASTs), continuously applying rewrite rules to replace subtrees of the AST with more efficient ones. Especially on large source repositories, even simply finding opportunities for a rewrite can be expensive, as optimizer traverses the AST naively. In this paper, we leverage the need to repeatedly find rewrites, and explore options for making the search faster through indexing and incremental view maintenance (IVM). Concretely, we consider bolt-on approaches that make use of embedded IVM systems like DBToaster, as well as two new approaches: Label-indexing and TreeToaster, an AST-specialized form of IVM. We integrate these approaches into an existing just-in-time data structure compiler and show experimentally that TreeToaster can significantly improve performance with minimal memory overheads.

**CCS CONCEPTS**
- Information systems → Database views; Query optimization.
- Keywords
  - Abstract Syntax Tress, Compilers, Indexing, Incremental View Maintenance

**ACM Reference Format:**
Darshana Balakrishnan, Carl Nuessle, Oliver Kennedy, Lukasz Ziarek. 2021. TreeToaster: Towards an IVM-Optimized Compiler. In Proceedings of the 2021 International Conference on Management of Data (SIGMOD ’21), June 20–25, 2021, Virtual Event, China. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3459244

**1 INTRODUCTION**
Typical database query optimizers, like Apache Spark’s Catalyst [3] and Greenplum’s Orca [35], work with queries encoded as abstract syntax trees (ASTs). A tree-based encoding makes it possible to specify optimizations as simple, composable, easy-to-reason-about pattern/replacement rules. However, such pattern matching can be very slow. For example, Figure 1 shows a breakdown of how Catalyst spends its time optimizing the 22 queries of the TPC-H benchmark [38]. 33-45% of its time is spent searching for optimization opportunities, using Scala’s match operator to recursively pattern-match with every node of the tree. A further 27-43% of the optimizer’s time is spent in optimizer’s outer fixpoint loop (e.g., comparing ASTs to decide whether the optimizer has converged or if further optimization opportunities might exist). On larger queries, pattern matching can be tens or even hundreds of seconds.

In this paper, we propose TreeToaster, an approach to incremental view maintenance specialized for use in compilers. As we show, TreeToaster virtually eliminates the cost of finding nodes eligible for a rewrite. In lieu of repeated linear scans through the AST for eligible nodes, TreeToaster materializes a view for each rewrite rule, containing all nodes eligible for the rule, and incrementally maintains it as the tree evolves through the optimizer. Naively, we might implement this incremental maintenance scheme by simply reducing the compiler’s pattern matching logic to a standard relational query language, and “bolting on” a standard database view maintenance system [22, 33]. This simple approach typically reduces search costs to a (small) constant, while adding only a negligible overhead to tree updates. However, classical view maintenance systems come with a significant storage overhead. As we show in this paper, TreeToaster improves on the “bolt-on” approach by leveraging the fact that both ASTs and pattern queries are given as trees. As we show, when the data and benchmark [38].

![Figure 1: Time breakdown of the Catalyst optimizer on the 22 queries of the TPC-H Benchmark](image-url)

**CCS CONCEPTS**
- Information systems → Database views; Query optimization.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD ’21, June 20–25, 2021, Virtual Event, China
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8345-1/21/06...$15.00
https://doi.org/10.1145/3448016.3459244

---

1 The instrumented version of spark can be found at https://github.com/UBOdin/spark-instrumented-optimizer

2 This number is smaller, but still notable for Orca, accounting for 5-20% of the optimizer’s time on a similar test workload.
query are both trees, TreeToaster achieves similar maintenance costs without the memory overhead of caching intermediate results (Figure 2). 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 optimizations with standard techniques for view maintenance (e.g., inlining and compiling to C++ [22]) to produce an incremental-view maintenance engine that meets or beats state-of-the-art view maintenance on AST pattern-matching workloads, while using much less memory.

To illustrate the advantages of TreeToaster, we apply it to a recently proposed Just-in-Time Data Structure compiler [5, 19] that reframes tree-based index data structures as ASTs. Like other AST-based optimizers, pattern/replacement rules asynchronously identify opportunities for incremental reorganization like database cracking [17] or log-structured merges [29]. We implement TreeToaster within JustInTimeData and show that it virtually eliminates AST search costs with minimal memory overhead.

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 “bolt-on” DBToaster, an embeddable IVM system, onto a just-in-time data-structure compiler [5, 19]. This modification dramatically improves performance, but adds significant memory overheads; (iv) We present TreeToaster, a IVM system optimized for compiler construction. TreeToaster avoids the high memory overheads of bolt-on IVM; (v) We present experiments that show that TreeToaster significantly outperforms “bolted-on” state-of-the-art IVM systems and is beneficial to the JustInTimeData compiler.

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 of the source.

Example 2.1. Figure 3 shows the AST for the expression $2 * y + x$. AST nodes have labels (e.g., Arith, Var, or Const) and attributes (e.g., `{op↦+}` or `{val↦2}`).

![Figure 3: An AST for the expression $2 * y + x$](image)

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

Definition 1 (Node). An Abstract Syntax Tree node $N = (ℓ, A, D)$ is a 3-tuple, consisting of (i) a label $ℓ$ drawn from an alphabet $ℒ$; (ii) annotations $A : Σ_M → D$, a partial map from an alphabet of attribute names $Σ_M$ to a domain $D$ of attribute values; and (iii) an ordered list of children $N$.

We define a leaf node (denoted $\text{isleaf}(N)$) as a node that has no child nodes (i.e., $N = \emptyset$). We assume that nodes follow a schema $S : L → 2^{Σ_M} × D$; For each label ($ℓ ∈ S$), we fix a set of attributes that are present in all nodes with the label ($X ∈ 2^{Σ_M}$), as well as an upper bound on the number of children ($c ∈ N$).

Optimization. Next, the optimizer rewrites the AST, iteratively applying pattern-matching expressions and deriving replacement subtrees. We note that even compilers written in imperative languages frequently adopt a declarative style for expressing pattern-matching conditions. For example, ORCA [35] (written in C++) builds small ASTs to describe pattern matching structures, while Catalyst [3] (written in Scala) relies on Scala’s native pattern-matching syntax.

Example 2.2. A common rewrite rule for arithmetic eliminates no-ops like addition to zero. For example, the subtree

$$\text{Arith} (\text{op↦+}), \text{Const} (\text{val↦0}), \text{Var} (\text{name↦b}), \text{Var} (\text{name↦c})$$

can be replaced by $\text{Var} (\text{name↦b})$, $\text{Var} (\text{name↦c})$. If the optimizer encounters a subtree with an $\text{Arith}$ node at the root, $\text{Const}$ and $\text{Var}$ nodes as children, and a 0 as the $\text{val}$ attribute of the $\text{Const}$ node; it replaces the entire subtree by the $\text{Var}$ node.

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

Output. Finally, the compiler uses the optimized AST as appropriate (e.g., by generating bytecode or a physical plan).

2.1 Pattern Matching Queries

We formalize pattern matching in the following grammar:

Definition 2 (Pattern). A pattern query $q ∈ Q$ is one of

$$Q : \text{AnyNode} \mid \text{Match} (L, Θ)$$

The symbol $\text{Match}(ℓ_q, i, \bar{Q}, Θ)$ indicates a structural match that succeeds if (i) The matched node has label $ℓ_q$, (ii) the children of the matched node recursively satisfy $q_i ∈ \bar{Q}$, and (iii) the constraint $Θ$ over the attributes of the node and its children is satisfied. The node variable $i ∈ Σ_F$ is used to identify the node in subsequent use,
The symbol \( \Theta \) 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 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, and as shown in Figure 1 can take up to 45% of the optimizer’s time.

Example 2.4. Continuing the example, the optimizer would traverse the entire AST looking for Arith nodes with the appropriate child 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 simplification (e.g., by creating new Const nodes), and the tree traversal must be repeated.

3 BOLTING-ON IVM FOR PATTERN MATCHING

As a warm-up, we start with a simple, naive implementation of incremental view maintenance for compilers by mapping our pattern matching grammar onto relational queries, and “bolting on” an existing system for incremental view maintenance (IVM). Although this specific approach falls short, it illustrates how IVM relates to the pattern search problem. To map the AST to a relational encoding, for each label/schema pair \( \ell \rightarrow (\{ x_1, \ldots, x_k \}, c) \in S \), we define a relation \( R(\ell(id_{x_1}, \ldots, x_k, child_1, \ldots, child_d) \) with an id field, and one field per attribute or child. Each node \( N = (\ell, A, \{ N_1, \ldots, N_n \}) \) is assigned a unique identifier \( id_N \) and defines a row of relation \( R_\ell \).

\[
\begin{align*}
\text{select} \times \text{from } R_q \text{ where } \theta_q
\end{align*}
\]

Figure 6: Converting a pattern \( q \) to an equivalent SQL query.

Materialized views are used in production databases to accelerate query processing. If a view is accessed repeatedly, database systems materialize the view query \( Q(D) \) by precomputing its results \( Q(D) \) on the database \( D \). When the database changes, the view must be
updated to match: Given a set of changes, $\Delta D$ (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 $\Delta D$ is small, most of this computation will be redundant. A more efficient approach is to derive a so-called “delta query” ($\Delta Q$) that computes a set of updates to the (already available) $Q(D)$. That is, denoting the view update operation by $\vDash$:

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

**Example 3.2.** Recall $Q(Arithmetic, Const, Var)$ from the prior example. After inserting a row $c$ into Const, we want:

$$Q(Arithmetic, Const \uplus c, Var)$$

$$= \text{Arith } (\text{Const }\uplus c) \bowtie \text{Var}$$

$$= (\text{Arith }\bowtie \text{Const }\bowtie \text{Var}) \uplus (\text{Arith }\bowtie c \bowtie \text{Var})$$

Instead of computing the full 3-way join, we can replace $c$ with a singleton and compute the cheaper query $(\text{Arith }\bowtie c \bowtie \text{Var})$, and union the result with our original materialized view to obtain an updated view.

The joint cost of $\Delta Q(D, \Delta D)$ and the $\vDash$ operator is generally lower than re-running the query, significantly improving overall performance when database updates are small and infrequent. However, $\Delta Q$ can still be expensive. For larger or more frequent changes, we can further reduce the cost of computing $\Delta Q$ by caching intermediate results. Ross et. al. [33] proposed a form of cascading IVM that caches every intermediate result relation in the physical plan of the view query.

**Example 3.3.** Consider the running example query with the following execution order:

$$(\text{Arithmetic }\bowtie \text{Var}) \bowtie \text{Const}$$

In addition to materializing $Q(\cdot)$, Ross’ scheme also materializes the results of $Q_1 = (\text{Arithmetic }\bowtie \text{Var})$. When $c$ is inserted into Const, the update only requires a simple 2-way join $c \bowtie Q_1$. However, updates are now (slightly) more expensive as multiple views may need to be updated.

Ross’ approach of caching intermediate state is analogous to typical approaches to fixpoint computation (e.g., Differential Dataflow [25]), but penalizes updates to tables early in the query plan. With DBToaster [22], Koch et al. proposed instead materializing all possible query plans. Counter-intuitively, this added materialization 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 Bolting DBToaster onto a Compiler

DBToaster [22] 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 changes required (i.e., “bolt-on”) to allow the compiler to use an incremental view maintenance data structure generated by DBToaster:

- The reduction above generates SQL queries for each pattern-match query used by the optimizer.
- DBToaster builds a view maintenance data structure.
- The compiler is instrumented to register changes in the AST with the view maintenance data structure.
- 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 AST scans. 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, our running example requires materializing 2 additional view queries, and this number grows combinatorially with the join width. Finally, DBToaster-generated view structures register updates at the granularity of individual node insertion/deletions, making it impossible for them to take advantage of the fact that most rewrites follow very structured patterns. For a relatively small number of pattern-matches, the memory use of the compiler with a DBToaster view structure bolted on, increases by a factor of 2.5×. Given that memory consumption is already a pain point for 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 project away, this still amounts to a shadow copy of the AST. Second, DBToaster 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.

Both benefits are of limited use in pattern-matching on ASTs. Pattern matches are 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 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 single join result. This also limits the value of materializing intermediate state for the sake of cache locality when updating downstream relations.

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 step to query optimization is indexing, for example with a secondary index on the node labels as illustrated in Figure 7. For each node label, the index maintains pointers to all nodes 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\((N, q, \text{Index}_N)\)

**Input:** \(N \in N, q \in Q, \text{Index}_N : \ell \to \{\text{Desc}(N)\}\)

**Output:** \(N_\text{match} \in \text{Desc}(N)\)

1. **if** \(q = \text{AnyNode}\) **then**
   2. **return** \(N_\text{match} \leftarrow N\)
2. **else** if \(q = \text{Match}(\ell, i, [q_1, \ldots, q_n], \theta)\) **then**
   3. **for** \(N_\text{idx} \in \text{Index}_N[\ell] \) **do**
   4. **if** \(q(N) = \top, \Gamma, \theta\) **then**
   5. **return** \(N_\text{match} \leftarrow N_\text{idx}\)

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 library unordered_set. Similarly, the maintenance overhead is low — one hash table insert and/or remove per AST node changed.

**Example 4.1.** To find matches for the rule of Example 2.2, we retrieve a list of all Art1th nodes from the index and iteratively check each for a pattern match. Note that this approach only supports filtering on labels; Recursive matches and constraints both need to be re-checked with each iteration.

4.2 Incremental View Maintenance

While indexing works well for single-node patterns, recursive patterns require a heavier-weight approach. Concretely, when a node in the AST is updated, we need to figure out which new pattern matches the update creates, and which pattern matches it removes. As we saw in Section 3, this could be accomplished by “joining” the updated node with all of the other nodes that could participate in the pattern. However, to compute these joins efficiently, DBToaster and similar systems need to maintain a significant amount of supporting state: (i) The view itself, (ii) Intermediate state needed to evaluate subqueries efficiently (iii) A shadow copy of the AST. The insight behind TreeToaster is that the latter two sources of state are unnecessary when the AST is already available: (i) Subqueries (inter-node joins) reduce to pointer chasing when the AST is available, and (ii) A shadow copy of the AST is unnecessary if the IVM system can navigate the AST directly.

We begin to outline TreeToaster in Section 5 by defining IVM for immutable (functional) ASTs. This simplified form of the IVM problem has a useful property: When a node is updated, all of its ancestors are updated as well. Thus, we are guaranteed that the root of a pattern match will be part of the change set (i.e., \(\Delta D\)) and can restrict our search accordingly. 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, but note that this approach generalizes trivially to multiple patterns.

5 AST-OPTIMIZED IVM

We first review a generalization of multisets proposed by Blizard [8] that allows for elements with negative multiplicities. A generalized multiset \(M : \text{dom}(M) \to \mathbb{Z}\) maps a set of elements from a domain \(\text{dom}(M)\) 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:

\[
(M_1 \oplus M_2)(x) = M_1(x) + M_2(x)
\]

Difference, denoted \(\ominus\), is defined analogously:

\[
(M_1 \ominus M_2)(x) = M_1(x) - M_2(x)
\]

We write \(x \in M\) as a shorthand for \(M(x) \neq 0\). 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 the integer 1.

A view \(\text{View}_{aq}\) 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 any node appears at most once in any AST.

**Definition 4 (View Correctness).** A view \(\text{View}_{aq}\) is correct for an AST \(N\) if the view is the generalized multiset that maps the subset of \(\text{Desc}(N')\) that matches \(q\) to 1, and all other elements to 0:

\[
\text{View}_{aq} = \{ N' \to 1 | N' \in q(N) \}
\]

If we start with a view \(\text{View}_{aq}\) that is correct for the root of an AST \(N\) and rewire the AST’s root to \(N'\), the view should update...
IVM

Although correct, assum-
ing that the AST is immutable: When a node changes, each of its ancestors must be updated to reference the new node as well. Even when TreeToaster is built into a compiler with immutable ASTs, many of these pattern matches will be redundant. By lifting this restriction (if in spirit only), we can decrease the overhead of view maintenance by reducing the number of nodes that need to be checked with each AST update. To begin, we 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\R'] is the node resulting from a replacement of R with R' in N:

\[ N[R\R'] = \begin{cases} R' & \text{if } N = R \\ (\ell, A, [N_1[R'R'], \ldots, N_n[R'R']]) & \text{if } N = (\ell, A, [N_1, \ldots, N_n]) \end{cases} \]

We also lift this notation to collections:

\[ \text{View}[R[R']] = \| N[R[R']] \rightarrow c \mid (N \rightarrow c) \in \text{View} \]

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.

Example 5.3. The replacement of Example 5.1 is written:

\[ N[\{\text{Arith}, \{\text{op} \mapsto \times\}, \ldots\}] \setminus \{\text{Const}, \{\text{val} \mapsto 0\}, \{\} \} \]

In the mutable model, the root node itself does not change.

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 \( q_i \):

\[ D(q) = \begin{cases} 0 & \text{if } q = \text{AnyNode} \\ 1 + \max_{i \in [n]} (D(q_i)) & \text{if } q = \text{Match}(i, [q_1, \ldots, q_n], \theta) \end{cases} \]

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 Ancestor_i(N) the \( i \)th ancestor of \( N \). 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 \( q \), \( [R[R'], q] \), is the difference between the generalized multiset of the respective nodes, their descendants, and their ancestors up to a height of \( D(q) \):

\[ [R[R'], q] = \text{Desc}(R) \oplus \| \text{Ancestor}_i(R) \rightarrow 1 \mid i \in [n] \| \oplus \text{Desc}(R') \oplus \| \text{Ancestor}_i(R') \rightarrow 1 \mid i \in [n] \| \]

Lemma 5.4. 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 View_q is correct for N, and View'_q = IVM(q, View_q, [R[R'], q]), then View'_q[R[R']] is correct for N[R[R']]

Proof Sketch. Shown by building on the proof of Lemma 5.2 with a recursive proof that changing a subtree cannot affect the matchability of a node more than \( D(q) \) ancestors above. The full proof appears in an associated tech report [4].
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:

\[
G : \text{Gen}(\mathcal{L}, \text{atom}, \mathcal{G}) | \text{Reuse}(\Sigma_f)
\]

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 \( \Gamma : \Sigma_f \rightarrow \Sigma_M \rightarrow \mathcal{D} \). A Reuse term indicates the re-use of a subtree from the previous AST, provided by a node scope \( \mu : \Sigma_f \rightarrow N \). Node generators are evaluated by the recursive node generator grammar follows:

\[
\llbracket G \rrbracket_{\Sigma_f, \mu} : \mathcal{G} \rightarrow N \text{ operator, defined as follows:}
\]

\[
\llbracket G \rrbracket_{\Sigma_f, \mu} = \begin{cases}
\{ (L, \ldots, L) \} & \text{if } G = \text{Gen}(L) \\
(t, \{ a_1(G), \ldots, a_k(G) \}, \llbracket G \rrbracket_{\Sigma_f, \mu}) & \text{if } G = \text{Gen}(t, \{ a_1, \ldots, a_k \}, \{ g_1, \ldots, g_n \}) \\
\text{if } G = \text{Reuse}(i) \\
\emptyset & \text{if } G = \text{Gen}(t) \\
\emptyset & \text{if } G = \text{Gen}(t, \{ a_1, \ldots, a_k \})
\end{cases}
\]

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

\[
pair(q, R) = \{ (q, R) \} \cup \ldots
\]

\[
\cup_{k \in [n]} \{ (\text{AnyNode}, N_k) \} \text{ if } q = \text{AnyNode}
\]

\[
\cup_{k \in [n]} \{ (q_k, N_k) \} \text{ if } q = \text{Match}(r, x, \{ q_1, \ldots, q_n \})
\]

\[
R = (t, A, \{ N_1, \ldots, N_n \})
\]

A set of generated node pairs \( \text{pair}(g, \Gamma, \mu) \) is defined analogously relative to the node \( \llbracket g \rrbracket_{\Gamma, \mu} \).

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

\[
(\text{AnyNode}, N \in \text{pair}(q, R)) \equiv (\text{Reuse}(N), N \in \text{pair}(g, \Gamma, \mu))
\]

Let \( g \in \mathcal{G} \) be a generator that is safe for \((m, R)\), where \( m \in Q \) is a pattern. The mutable update delta from \( N \) to \( N' \) is given by \( g \rrbracket_{\Gamma, \mu} \) is:

\[
\Delta = \llbracket N' \rightarrow 1 \mid (q', N') \in \text{pair}(g, \Gamma, \mu) \rrbracket \cup \llbracket N' \rightarrow 1 \mid (q', N') \in \text{pair}(m, R) \rrbracket
\]

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.

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

Input: \( q \in Q, g \in \mathcal{G} \)
Output: \( f : N \rightarrow \{ N \} \)
1 if \( q = \text{AnyNode} \lor g = \text{Reuse}(\mu) \) then
2 \( f' \leftarrow (N \rightarrow \{ N \}) \)
3 else if \( q = \text{Match}(t, i, \{ q_1, \ldots, q_n \}, \theta) \) then
4 \( g = \text{Gen}(t', i', \{ g_1, \ldots, g_n \}) \)
5 if \( \text{Align}(q, g) \) then
6 \( f'' \leftarrow (N \rightarrow \{ N \}) \)
7 else
8 \( f'' \leftarrow (N \rightarrow \emptyset) \)
9 for \( i \in [n] \) do
10 \( f_i \leftarrow \text{Inline}_{gen}(q_i, g_i) \)
11 \( f' \leftarrow (N \rightarrow f''(N) \cup \bigcup_{i \in [n]} f_i(N)) \)
12 \( \mathcal{A} = \{ i \mid i \in [D(q)] \land \text{Align}(q_i, g_i) \} \)
13 \( f \leftarrow (N \rightarrow f''(N) \cup \{ \text{Ancestor}(N) \mid i \in \mathcal{A} \}) \)

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 a function that, given the generated replacement node (i.e., \( \llbracket g \rrbracket_{\Gamma, \mu} \)) 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 re-used (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 \( \text{Align}_{gen} \) if the input pattern and generator align – whether they have equivalent labels at equivalent positions.

\[
\text{Align}_{gen}(q, g) = \begin{cases}
\top & \text{if } q = \text{AnyNode} \lor g = \text{Gen}(\mu) \\
\emptyset & \text{if } q = \text{Match}(t, A, \ldots, \theta) \land g = \text{Gen}(t', i, \ldots) \land t \neq t' \\
\forall k : \text{Align}_{gen}(q_k, g_k) & \text{if } q \neq \text{Match}(t, A, \ldots, \theta) \land g = \text{Gen}(t, i, \ldots)
\end{cases}
\]

The recursive case \( \text{Align}_{gen} \) checks for the existence an alignment among the \( d \)th level descendants of the input pattern.

\[
\text{Align}_{gen}(q, g) = \exists k : \text{Align}_{gen-1}(q_k, g_k)
\]

Example 6.1. Continuing the running example, only the Var node appears in both the pattern and replacement. Thus, when a replacement is applied we need only check the parent of a replaced node for new view updates.
7 EVALUATION

To evaluate TreeToaster, we built four IVM mechanisms into the JUSTINTIMEDATA [5, 6] compiler, a JIT compiler for data structures built around a complex AST. 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) Indexing labels, as proposed in Section 4.1, (iii) Classical incremental view maintenance implemented by bolting on a view maintenance data structure created by DBToaster with the −depth=1 flag, (iv) DBToaster’s full recursive view maintenance bolted onto the compiler, and (v) TreeToaster’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 consuming significantly less memory, (iii) On complex workloads, TreeToaster’s view maintenance latency is half of bolt-on approaches.

7.1 Workload

To evaluate TreeToaster, we rely on a benchmark workload created by JUSTINTIMEDATA [6, 20], an index designed like a just-in-time compiler. JUSTINTIMEDATA’s underlying data structure is modeled after an AST, allowing a JIT runtime to incrementally and asynchronously rewrite it in the background using pattern-replacement rules [5] to support more efficient reads. Data is organized through 5 node types that closely mimic the building blocks of typical index structures:

- \( Array, data : Seq[<key : Int, value : Int>], \emptyset \)
- \( Singleton, data : <key : Int, value : Int>, \emptyset \)
- \( DeleteSingleton, key : Int, N_1 \)
- \( Concat, \emptyset, N_1, N_2 \)
- \( BinTree, sep : Int, N_1, N_2 \)

JUSTINTIMEDATA was configured to use five pattern-replacement rules that mimic Database Cracking [17] by incrementally building a tree, while pushing updates (Singleton and DeleteSingleton respectively) down into the tree.

CrackArray: This rule matches Array nodes and partitions them on a randomly selected pivot \( sep \in data \).

- \( Match(Array, [data], \emptyset, T) \rightarrow Gen(BinTree, [sep], [\{} x | x.key < sep \}, [], [\{}]) \)
- \( Gen(Array, [\{} x | x.key \geq sep \}, [], [\{}]) \)

PushDownSingletonBtreeLeft/Right: These rules push Singleton nodes down into BinTree depending on the separator.

- \( Match(Concat, C, [Match(BinTree, B, q_1, q_2, \emptyset)], \emptyset) \)
- \( Match(Singleton, S, \emptyset, \emptyset) \rightarrow Gen(BinTree, [sep], [\{} Gen(Concat [\{}], [\{}], [\{}]
- \( Reuse(q_1), Reuse(S), \}) \rightarrow Gen(q_2) \)}

PushDownSingletonBtreeRight is defined analogously.

PushDownDeleteSingletonBtreeLeft/Right: These rules push DeleteSingleton nodes depending on the separator and are defined analogously to PushDownSingletonBtreeLeft.

Although these rewrite rules appear relatively simple, their pattern structures are representative of the vast majority of optimizer in both Apache Spark [3] and Orca [35]. A detailed discussion of these rules and how they relate to the example patterns is provided in an accompanying technical report [4].

7.2 Data Gathering and Measurement

We instrumented JUSTINTIMEDATA to collect updates to AST nodes as allocation (\( insert() \)) and garbage collection (\( remove() \)) operations. To vary the distribution of optimization opportunities we used the six baseline YCB [12] benchmark workloads as input to JUSTINTIMEDATA. Each workload exercises a different set of node operations, resulting in ASTs composed of different node structures, patterns, and the applicability of different rewrite rules. We built a testing module in C++, allowing us to replace JUSTINTIMEDATA’s naive tree traversal with the view maintenance schemes described above for an apples-to-apples comparison. Figure 8 illustrates the benchmark generation process.

Views for TreeToaster and label indexing were generated by declarative specification as described in Section 6 and views for DBToaster were generated by hand, translating rules to equivalent SQL as described in Section 2.

We instrumented TreeToaster to collect in-structure information pertaining to view materialization and maintenance: the time to identify potential JUSTINTIMEDATA transform operations (rows in the materialized views), and the time to maintain the views in response to updates. The test harness also records database operation latency and process memory usage, as reported by the Linux /proc interface. To summarize, we measure performance along three axes: (i) Time spent finding a pattern match, (ii) Time spent maintaining support structures (if any), and (iii) Memory allocated.

7.3 Evaluation

JUSTINTIMEDATA is configured to use 5 representative rewrite rules listed above. Detailed results are grouped by the triggering rule. Each combination was run 10 times, with the search and operation results aggregated. Experiments were run on Ubuntu 16.04.06 LTS with 192GB RAM and 24 core Xeon 2.50GHz processors. All
Figure 9: Relative Average Search Technique Performance by Rewrite Rule
Figure 10: Relative Total Search + Maintenance Cost by Rewrite Rule
Figure 11: Total Latency (search cost + maintenance operations) and Memory Use, by method and node type

Figure 12: Average IVM Operational Latency. The bottom plot is a zoomed-in view of the top plot.

Figure 13: Average Process Memory Usage Summary.

the results are obtained from the instrumented `JustInTimeData` compiler run on YCSB workloads with 300M keys.

7.4 Results
We first evaluate how IVM performs relative to other methods of IVM in identifying target nodes in tree structure. We compare the latency in identifying potential nodes (i.e., materializing views) using the 5 IVM methods.

Figure 9 shows there were 5 sets of views one per transform (e.g., CrackArray) that were materialized, representing target nodes in the underlying tree structure. The 5 boxplot clusters compare the relative average latency of identifying 1 such node, using each of 5 identification methods. In each case, the naive search approach exhibits the worst performance. The label index approach also yields worse results than either of the IVM approaches. For identifying target nodes, we conclude that an IVM approach performs better.

We compare TreeToaster to IVM alternatives, including label indexing as well as a classic IVM system and a hash-based IVM implemented using DBToaster. Figure 11 shows for each system, the graph shows both memory and overall performance, in terms of the combined access latency and search costs per optimizer iteration. TreeToaster easily outperforms naive iteration and has an advantage over the label index, with only a slight memory penalty relative to the former. It slightly outperforms DBToaster in general, but the total system memory for TreeToaster is less than half of DBToaster.

Figure 10 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 of the 4 IVM target node identification methods. While the label-index approach performs well on workloads where the structure is allowed to converge (loads B and D) it scales poorly under increased pressure (update heavy loads A and F). Average total time was significantly worse than that of TreeToaster. In terms of total cost, TreeToaster outperforms both classic IVM and DBToaster.

Finally, Figure 13 shows the average memory use in pages. For all the methods, the memory footprint was a relatively stable constant within each run, with only small inter-run variance. Comparing across materialization and maintenance methods, both classic IVM and DBToaster 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 12 shows an aggregate summary of all workloads. Overall, TreeToaster offers both better memory and latency across all alternatives.

8 RELATED WORK
TreeToaster builds on decades of work in Incremental View Maintenance (IVM) — See [11] for a survey. The area has been extensively
studied, with techniques developed for incremental maintenance support for a wide range of data models [7, 10, 33, 40] and query language features [18, 21, 24, 31].

Notable are techniques that improve performance through dynamic programming [22, 25, 33, 41]. A common approach is materializing intermediate results; for one plan as proposed by Ross et. al. [33], or all possible plans as proposed by Koch et. al. [22]. A key feature of both approaches is computing the minimal update – or slice – of the query result, an idea core to systems like Differential Dataflow [25]. Both approaches show significant performance gains on general queries. However, the sources of these gains: selection-pushdown, aggregate-pushdown, and cache locality are less relevant in the context of abstract syntax trees. Similarly-spirited approaches can be found in other contexts, including graphical inference [41], and fixed point computations [25].

Also relevant is the idea of embedding query processing logic into a compiled application [2, 16, 22, 26, 28, 30, 32, 34]. Systems like BerkeleyDB, SQLite, and DuckDB embed full query processing logic, while systems like DBToaster [2, 22, 28] and LinQ [26] 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 [26, 34, 39]. Major database engines have also recently been extended to compile queries to native code [9, 13, 27], albeit at query compile time.

To our knowledge, IVM over Abstract Syntax Trees has not been studied directly. The Cascades framework [15] considers streamlined approaches to scheduling rule application, a strategy that is used by the Orca [35] compiler. IVM approaches also exist for general tree and graph query languages and data models like XPath [14], Cypher [36, 37], and the Nested Relational Calculus [23]. These schemes address recursion, with which join widths are no longer bounded; and aggregates without an abelian group representation (e.g., min/max), where deletions are more challenging.

However, two approaches aimed at the object exchange model [1, 42], 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 [42] follows a similar model to our immutable IVM scheme, enforcing pattern matches on ancestors by recursively triggering shadow updates to all ancestors.

9 Conclusion and future work

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 shows that the TreeToaster approach works.

Many compilers choose to specify rewrites over the AST as pattern match rules, whether declaratively through language features (e.g., Spark), or a customized implementation of the same (e.g., Orca). We would like to extend our work, integrating it with language-based pattern matching to automate the matching process. For example, a DSL implemented with Scala macros could extract declarative pattern matches, plug them into our grammar, and provide a nearly drop-in replacement for its existing optimizer infrastructure. We also plan on 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.

Acknowledgments

This work is supported by NSF grants: SHF-1749539, IIS-1617586, and IIS-1750460. All opinions presented are those of the authors. The authors wish to thank the reviewers and shepherd for their substantial contributions.

References

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

[2] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Miloš Nikolic. 2012. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. arXiv preprint arXiv:2007.0137 (2012).

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Aftan, Michael J. Franklin, Ali Ghodsi, and Maiti Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In SIGMOD Conference. ACM, 1383–1394.

[4] Darshana Balakrishnan, Carl Naeveis, Oliver Kennedy, and Lukasz Ziaerek. 2021. TreeToaster: Towards an IVM-Optimized Compiler. UB CSE Technical Report (2021). http://www.cse.buffalo.edu/tech-reports/2021-01.pdf

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

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

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

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

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

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

[11] Latha S. Colby, Timothy Griffin, Leonid Libkin, Inderpal Singh Mumick, and Howard Trickey. 1996. Algorithms for Deferred View Maintenance. In SIGMOD Conference. ACM Press, 469–488.

[12] 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.

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

[14] Katja Dimitrova, Magd 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.

[15] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. IEEE Data Eng. Bull. 18, 3 (1995), 19–29.

[16] D. Richard Hipp. 2000. SQLite: Small. Fast. Reliable. Choose any three. SQLite.org.

[17] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2007. Database Cracking. In CIDR. www.cidrdb.org, 68–78.

[18] Akira Kawaguchi, Daniel F. Laurie, Indepal 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.

[19] Oliver Kennedy and Lukasz Ziaerek. 2015. Just-In-Time Data Structures. In CIDR. www.cidrdb.org.

[20] Oliver Kennedy and Lukasz Ziaerek. 2015. Just-In-Time Data Structures. In CIDR. Citeseer.

[21] Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In PODS. ACM, 87–98.
[22] 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.

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

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

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

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

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

[28] Milos Nikolic, Mohammad Daarbi, 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.

[29] Patrick E. O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O’Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). Acta Informatica 33, 4 (1996), 351–385.

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

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

[32] Mark Raasveld and Hannes Mühliesen. 2019. DuckDB: an Embeddable Analytical Database. In SIGMOD Conference. ACM, 1981–1984.

[33] 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.

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

[35] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongsian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvaredo, Foyzur Rahman, Michalis Petropoulos, Florian Waw, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orcas: a modular query optimizer architecture for big data. In SIGMOD Conference. ACM, 337–348.

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

[37] 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).

[38] The Transaction Processing Performance Council. [n.d.]. The TPC-H Benchmark. http://www.tpc.org/tpch/.

[39] Thomas Würthinger. 2014. Graal and truffle: modularity and separation of concerns as cornerstones for building a multipurpose runtime. In MODULARITY. ACM, 3–4.

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

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

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