TreeToaster: Towards an IVM-Optimized Compiler

Darshana Balakrishnan, Carl Nuessle, Oliver Kennedy, Lukasz Ziarek
University at Buffalo
[dbalakri,carlnues,okennedy,lziarek]@buffalo.edu

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 ACM Conference (Conference'17).* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Typical database query optimizers, like Apache Spark's Catalyst [4] and Greenplum's Orca [37], 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 [1] the 22 queries of the TPC-H benchmark [40]. 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[2].

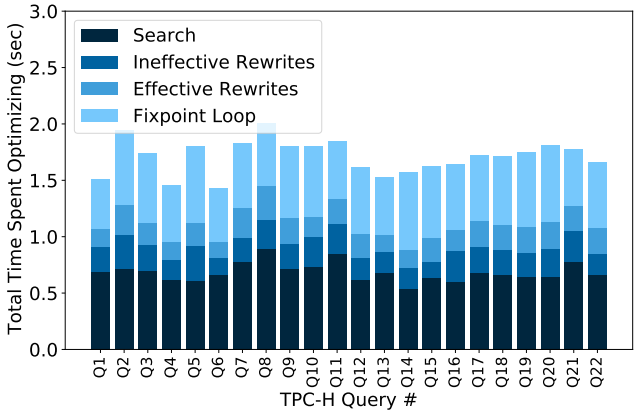


Figure 1. Time breakdown of the Catalyst optimizer on the 22 queries of the TPC-H Benchmark, including (i) searching the AST for candidate rewrites (**Search**), (ii) constructing new AST subtrees before aborting (**Ineffective**), (iii) constructing new AST subtrees (**Effective**), and (iv) In the optimizer's outer loop looking for a fixpoint (**Fixpoint**).

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 [24, 35]. 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 query are both trees, TreeToaster achieves similar

[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.

Conference'17, July 2017, Washington, DC, USA
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

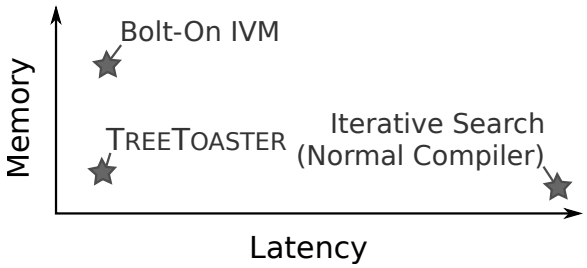


Figure 2. TreeToaster achieves AST pattern-matching performance competitive with "bolting-on" an embedded IVM system, but with negligible memory overhead.

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++ [24]) 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 [6, 21] 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 [19] or log-structured merges [31]. 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 [6, 21]. 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.

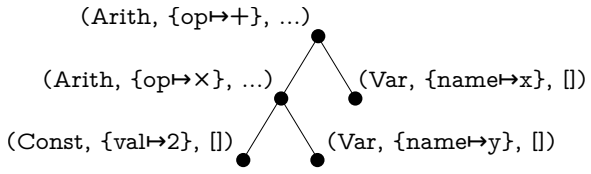


Figure 3. An AST for the expression `2 * y + x`

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., $\{\texttt{op} \mapsto +\}$ or $\{\texttt{val} \mapsto 2\}$).

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* $N = (\ell, A, \overline{N})$ *is a 3-tuple, consisting of (i) a label* ℓ *drawn from an alphabet* $\mathcal{L}$*; (ii) annotations* $A : \Sigma_M \rightarrow \mathbb{D}$*, a partial map from an alphabet of attribute names* Σ_M *to a domain* $\mathbb{D}$ *of attribute values; and (iii) an ordered list of children* $\overline{N}$*.*

We define a leaf node (denoted **isleaf**(N)) as a node that has no child nodes (i.e., $\overline{N} = \emptyset$). We assume that nodes follow a schema $\mathcal{S} : \mathcal{L} \rightarrow 2^{\Sigma_M} \times \mathbb{N}$; For each label ($\ell \in \mathcal{S}$), we fix a set of attributes that are present in all nodes with the label ($\overline{x} \in 2^{\Sigma_M}$), as well as an upper bound on the number of children ($c \in \mathbb{N}$).

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 [37] (written in C++) builds small ASTs to describe pattern matching structures, while Catalyst [4] (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

$$(\texttt{Arith}, \{\texttt{op} \mapsto +\}, [\ (\texttt{Const}, \{\texttt{val} \mapsto 0\}, []),\ (\texttt{Var}, \{\texttt{name} \mapsto b\}, [])\ ])$$

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

$$\Theta : \texttt{atom} = \texttt{atom} \mid \texttt{atom} < \texttt{atom} \mid \Theta \wedge \Theta \mid \Theta \vee \Theta \mid \neg\Theta \mid \mathbb{T} \mid \mathbb{F}$$
$$\texttt{atom} : \texttt{const} \mid \Sigma_I.\Sigma_M \mid \texttt{atom}\ [+, -, \times, \div]\ \texttt{atom}$$

Figure 4. Constraint Grammar

$$[\![q(\ell, A, [N_1 \ldots N_n])]\!] = \begin{cases} \mathbb{T}, \emptyset & \textbf{if } q = \texttt{AnyNode} \\ \mathbb{T}, \Gamma & \textbf{if } q = \texttt{Match}(\ell_q, i, [q_1 \ldots q_n], \theta) \\ & \quad \ell_q = \ell, \quad \theta(\Gamma), \\ & \quad [\![q_1(N_1)]\!] = \mathbb{T}, \Gamma_1 \\ & \quad \ldots [\![q_n(N_n)]\!] = \mathbb{T}, \Gamma_n, \\ & \quad \Gamma = \{\ i \to A\ \} \cup \bigcup_{k \in [n]} \Gamma_k \\ \mathbb{F}, \emptyset & \textbf{otherwise} \end{cases}$$

Figure 5. Semantics for pattern queries ($q \in \mathcal{Q}$)

2.1 Pattern Matching Queries

We formalize pattern matching in the following grammar:

Definition 2 (Pattern). *A pattern query $q \in \mathcal{Q}$ is one of*

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

The symbol $\texttt{Match}(\ell_q, i, \overline{Q}, \theta)$ 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 constraint θ over the attributes of the node and its children is satisfied. The node variable $i \in \Sigma_I$ is used to identify the node in subsequent use, for example to reference the node's attributes in the constraint (θ). The symbol AnyNode matches any node. Figure 5 formalizes the semantics of the $\mathcal{Q}$ grammar.

The grammar for constraints is given in Figure 4, 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 $\Gamma : \Sigma_I \to \Sigma_M \to \mathbb{D}$. 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 Arith nodes over Const and Var nodes as children are eligible for the simplification rule. The corresponding pattern query is:

$$\texttt{Match}(\texttt{Arith}, A, [\ \texttt{Match}(\texttt{Const}, B, [\,], \{B.\texttt{val} = 0\}),$$
$$\texttt{Match}(\texttt{Var}, C, [\,], \mathbb{T})\ ], \{A.\texttt{op} = +\})$$

Note the constraint on the Const match pattern; This sub-pattern only matches a node with a val(ue) attribute of 0.

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

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

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

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

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

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

Figure 6. Converting a pattern q to an equivalent SQL query.

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

$$q(N) \triangleq \{\ N' \mid N' \in \texttt{Desc}(N) \wedge \exists \Gamma : q(N') = \mathbb{T}, \Gamma\ \}$$

Pattern Matching is Expensive. Optimization 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 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 $\ell \to \langle\ \{\ x_1, \ldots, x_k\ \}, c\ \rangle \in \mathcal{S}$, we define a relation $R_\ell(\texttt{id}, x_1, \ldots, x_k, \texttt{child}_1, \ldots, \texttt{child}_c)$ with an id field, and one field per attribute or child. Each node $N = (\ell, A, [N_1, \ldots, N_c])$ 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 6. 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.

Example 3.1. Continuing Example 2.2, the AST nodes are encoded as relations: `Arith(id, op, child`$_1$`, child`$_2$`)`, `Const(id, val)`, and `Var(id, name)`. The corresponding pattern match query, following the process in Figure 6 is:

```
SELECT * FROM Arith a, Const b, Var c
WHERE a.child1 = b.id AND a.child2 = c.id
  AND a.op = '+' AND b.val = 0
```

3.1 Background: Incremental View Maintenance

Materialized views are used in production databases to accelerate query processing. If a view is accessed repeatedly, database systems materialize the view query Q 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, 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. A more efficient approach is to derive a so-called "delta query" (ΔQ) that computes a set of updates to the (already available) $Q(D)$. That is, denoting the view update operation by $\Leftarrow$:

$$Q(D + \Delta D) \quad \equiv \quad Q(D) \Leftarrow \Delta Q(D, \Delta D)$$

Example 3.2. Recall $Q(\texttt{Arith}, \texttt{Const}, \texttt{Var})$ from the prior example. After inserting a row c into `Const`, we want: $Q(\texttt{Arithmetic}, \texttt{Const} \uplus c, \texttt{Var})$

$$\begin{aligned} &= && \texttt{Arith} \bowtie (\texttt{Const} \uplus c) \bowtie \texttt{Var} \\ &= && (\texttt{Arith} \bowtie \texttt{Const} \bowtie \texttt{Var}) \uplus (\texttt{Arith} \bowtie c \bowtie \texttt{Var}) \\ &= && Q(\texttt{Arith}, \texttt{Const}, \texttt{Var}) \uplus (\texttt{Arith} \bowtie c \bowtie \texttt{Var}) \end{aligned}$$

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

The joint cost of $\Delta Q(D, \Delta D)$ and the $\Leftarrow$ operator is generally lower than re-running the query, significantly improving overall performance when database updates are small and infrequent. However, ΔQ can still be expensive. For larger or more frequent changes, we can further reduce the cost of computing ΔQ by caching intermediate results. Ross et. al. [35] proposed a form of cascading IVM that caches every intermediate result relation in the physical plan of the view query.

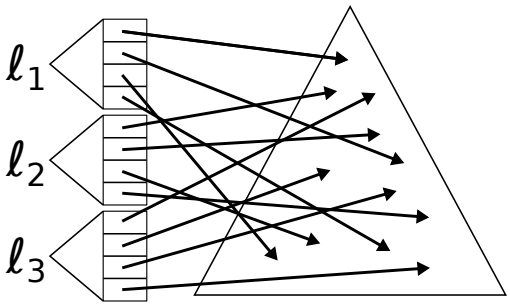


Figure 7. Indexing the AST by Label

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

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

In addition to materializing $Q(\cdot)$, Ross' scheme also materializes the results of $Q_1 = (\texttt{Arithmetic} \bowtie \texttt{Var})$. 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 [27]), but penalizes updates to tables early in the query plan. With DBToaster [24], Koch et. al. proposed instead materializing all possible query plans. Counterintuitively, 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 [24] 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:

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 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 insertions/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 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.

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[3]. 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.

[3]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.

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($N, q,$ Index$_N$)

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

```
if q = AnyNode then
    return N_match ← N
else if q = Match(ℓ, i, [q_1, ..., q_n], θ) then
    for N_idx ∈ Index_N[ℓ] do
        if q(N) = 𝕋, Γ then
            return N_match ← N_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 `Arith` 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., ΔD) 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 [9] that allows for elements with negative multiplicities. A generalized multiset $\mathbf{M} : \mathbf{dom}(\mathbf{M}) \to \mathbb{Z}$ 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 the integer 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 any node appears at most once in any AST.

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

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

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', the view should update accordingly. We initially assume that we have available a delta between the two ASTs (i.e., 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: $\texttt{View}'_q$

```
1 View'_q ← View_q
2 for N_i ∈ Δ  /* nodes with multiplicity ≠ 0 */
3 do
4     if q(N_i) = 𝕋, Γ then
5         View'_q ← View'_q ⊕ {| N_i → Δ(N_i) |}
```

Example 5.1. Consider the AST of Figure 3, which contains five nodes, and our ongoing example rule. Let us assume that the left subtree is replaced by `Const(0)` (e.g., if `Var(y)` is resolved to 0). The multiset of the corresponding delta is:

$$\{\!|\ (\texttt{Const}, \{\texttt{val} \mapsto 0\}, []) \mapsto 1,\quad (\texttt{Arith}, \{\texttt{op} \mapsto +\}, [\ldots]) \mapsto 1,$$
$$(\texttt{Const}, \{\texttt{val} \mapsto 2\}, []) \mapsto -1,\quad (\texttt{Var}, \{\texttt{name} \mapsto \texttt{y}\}, []) \mapsto -1,$$
$$(\texttt{Arith}, \{\texttt{op} \mapsto \times\}, [\ldots]) \mapsto -1\ |\!\}$$

Only one of the nodes with nonzero multiplicities matches, making the update: $\{\!|\ (\texttt{Arith}, \{\texttt{op} \mapsto +\}, [\ldots]) \mapsto 1\ |\!\}$

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

Lemma 5.2 (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. 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. $N[R\backslash R']$ is the node resulting from a replacement of R with R' in N:

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

We also lift this notation to collections:

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

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:

$$N[\ (\texttt{Arith}, \{\texttt{op} \mapsto \times\}, [\ldots])\ \backslash\ (\texttt{Const}, \{\texttt{val} \mapsto 0\}, [])\ ]$$

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.*

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

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 $\texttt{Ancestor}_i(N)$ the ith ancestor of N[4]. 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 $D(q)$.*

$$\begin{aligned} \lceil R, R' \rceil_q \triangleq\ & \texttt{Desc}(R) \oplus \{\!|\ \texttt{Ancestor}_i(R) \to 1 \mid i \in [n]\ |\!\} \\ & \ominus \texttt{Desc}(R') \ominus \{\!|\ \texttt{Ancestor}_i(R') \to 1 \mid i \in [n]\ |\!\} \end{aligned}$$

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 $\texttt{View}_q$ is correct for N. and $\texttt{View}'_q = \texttt{IVM}(q, \texttt{View}_q, \lceil R, R' \rceil_q)$, then $\texttt{View}'_q[R\backslash R']$ is correct for $N[R\backslash R']$*

Proof. Differs from the proof of Lemma 5.2 in three additional cases. If $q, N'' \mapsto \langle\ \mathbb{F}, \Gamma\ \rangle$, then $q(N) = N[R\backslash R'] = \texttt{View}_q = 0$. The condition on line 2 is false, and the multiplicity is unchanged at 0. Otherwise, if $N'' \in \texttt{Desc}(N)$ then $N''[R\backslash R'] \in \texttt{Desc}(N[R\backslash R'])$ 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,

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

and the proof is identical to Lemma 5.2. 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 $q = \texttt{AnyNode}$. 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. □

Example 5.5. The pattern depth of our running example is 1. Continuing the prior example, only the node, its 1-ancestor (i.e., parent), and the 1-descendents (i.e., children) of the replacement node would need to be examined for view updates.

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:

$$\mathcal{G} : \texttt{Gen}(\mathcal{L},\ \overline{\texttt{atom}},\ \overline{\mathcal{G}}) \mid \texttt{Reuse}(\Sigma_I)$$

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 $\Gamma : \Sigma_I \to \Sigma_M \to \mathbb{D}$. A Reuse term indicates the re-use of a subtree from the previous AST, provided by a node scope $\mu : \Sigma_I \to \mathcal{N}$. Node generators are evaluated by the $[\![\cdot]\!]_{\Gamma,\mu} : \mathcal{G} \to \mathcal{N}$ operator, defined as follows:

$$[\![g]\!]_{\Gamma,\mu} = \begin{cases} \mu(i) & \textbf{if } g = \texttt{Reuse}(i) \\ (\ell, \{a_1(\Gamma), \ldots, a_k(\Gamma)\}, [\![g_1]\!]_{\Gamma,\mu}, \ldots, [\![g_n]\!]_{\Gamma,\mu}) & \\ & \textbf{if } g = \texttt{Gen}(\ell, [a_1, \ldots, a_k], [g_1, \ldots, g_n]) \end{cases}$$

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 inserted back into the tree as replacements. As a simplification for clarity of presentation, we require that Reuse nodes reference nodes matched by AnyNode patterns. Define the set of matched node pairs as the set

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

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 a 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 –

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

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

```
if q = AnyNode ∨ g = Reuse(μ) then
    f' ← (N ↦ { N })
else if q = Match(ℓ, i, [q_1, ..., q_n], θ) then
    g = Gen(ℓ', i', [g_1, ..., g_n]);
    if Align_0(q, g) then
        f'' ← (N ↦ { N })
    else
        f'' ← (N ↦ ∅)
    for i ∈ [n] do
        f_i ← Inline_gen(q, g_i)
    f' ← (N ↦ f''(N) ∪ ⋃_{i∈[n]} f_i(N))
A = { i | i ∈ [D(q)] ∧ Align_i(q, g) };
f ← (N ↦ f'(N) ∪ { Ancestor_i(N) | i ∈ A })
```

whether they have equivalent labels at equivalent positions.

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

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

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

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 [6, 7] compiler, a JIT compiler for data structures built around a complex AST[5]. 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.

[5]The full result set of our runs is available at https://github.com/UBOdin/jitd-synthesis/tree/master/treetoaster_scripts

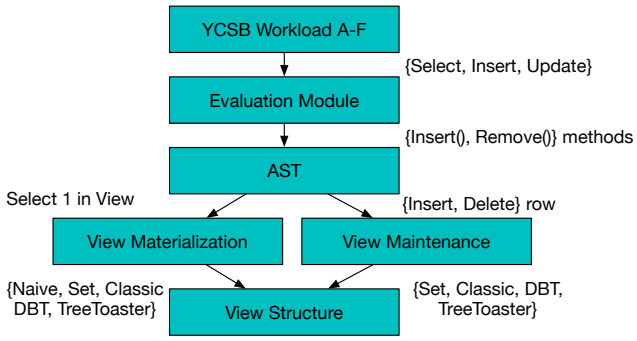


Figure 8. Benchmark Infrastructure

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,

7.1 Workload

To evaluate TreeToaster, we rely on a benchmark workload created by JustInTimeData [7, 22], 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 [6] to support more efficient reads. Data is organized through 5 node types that closely mimic the building blocks of typical index structures:

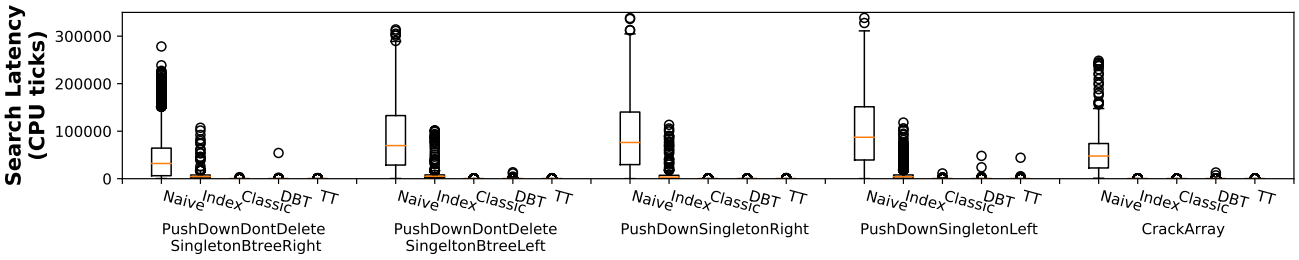
```
(Array, data:Seq[<key:Int,value:Int>], ∅)
(Singleton, data:<key:Int,value:Int>, ∅)
(DeleteSingleton, key:Int, N_1)
(Concat, ∅, N_1, N_2)
(BinTree, sep:Int, N_1, N_2)
```

JustInTimeData was configured to use five pattern-replacement rules that mimic Database Cracking [19] by incrementally building a tree, while pushing updates (`Singleton` and `DeleteSingleton` respectively) down into the tree.

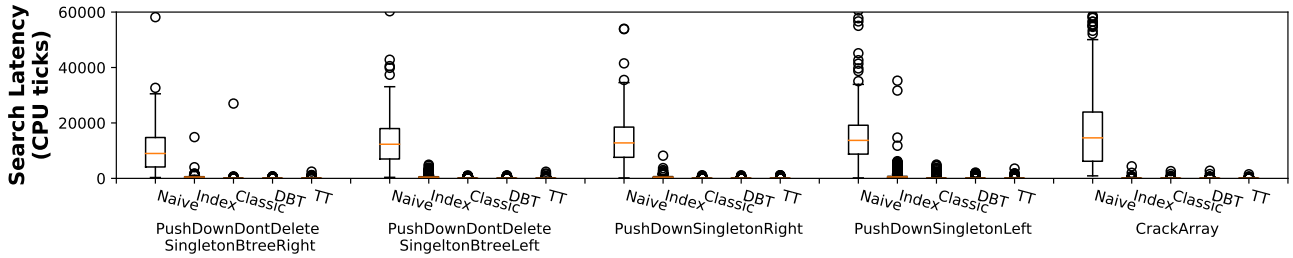
CrackArray: This rule matches `Array` nodes and partitions them on a randomly selected pivot `sep` ∈ `data`.

$$\texttt{Match}(\texttt{Array}, [\texttt{data}], \emptyset, \mathbb{T}) \rightarrow \texttt{Gen}(\texttt{BinTree}, [\texttt{sep}], [\texttt{Gen}(\texttt{Array}, [\{ x \mid x.\texttt{key} < \texttt{sep} \}], []), \texttt{Gen}(\texttt{Array}, [\{ x \mid x.\texttt{key} \geq \texttt{sep} \}], [])])$$

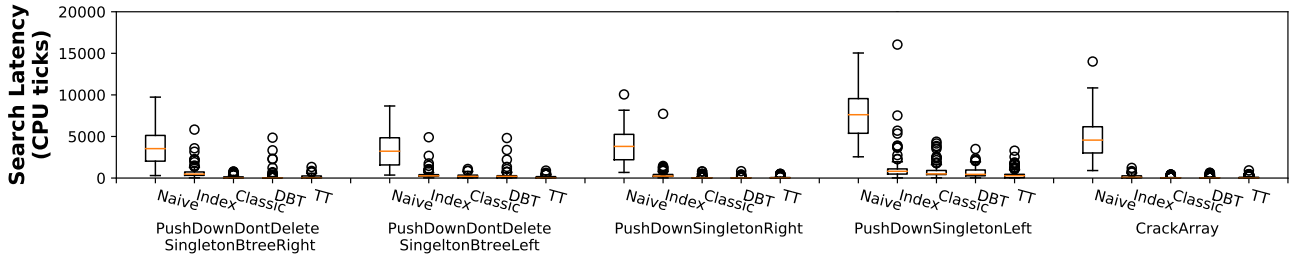
PushDownSingletonBtreeLeft/Right: These rules push `Singleton` nodes down into `BinTree` depending on the



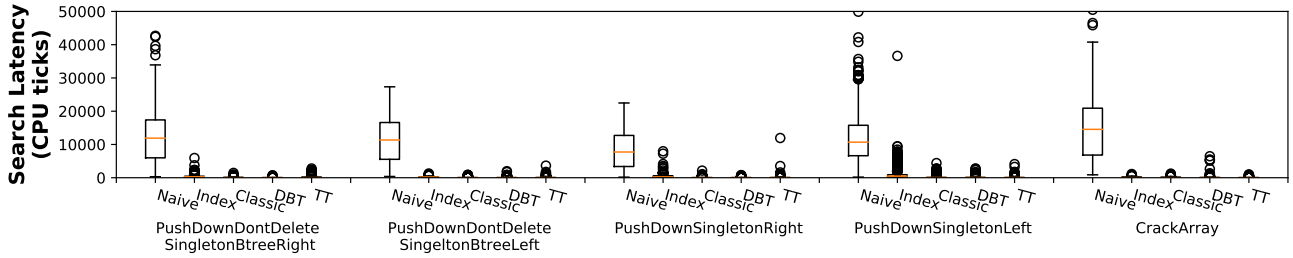
(a) Workload A



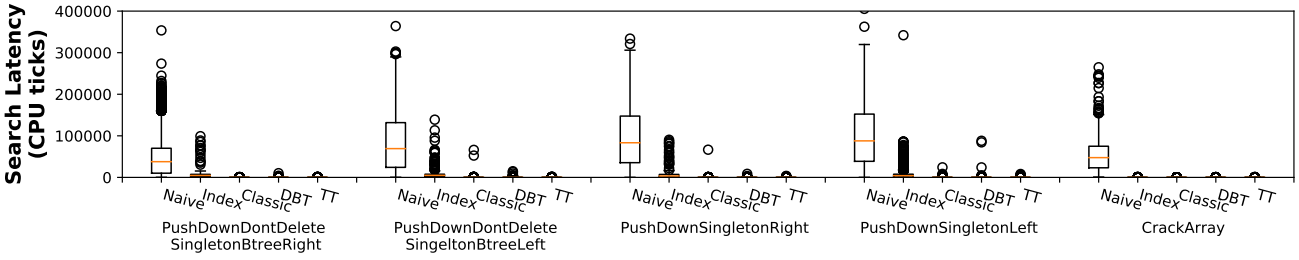
(b) Workload B



(c) Workload C

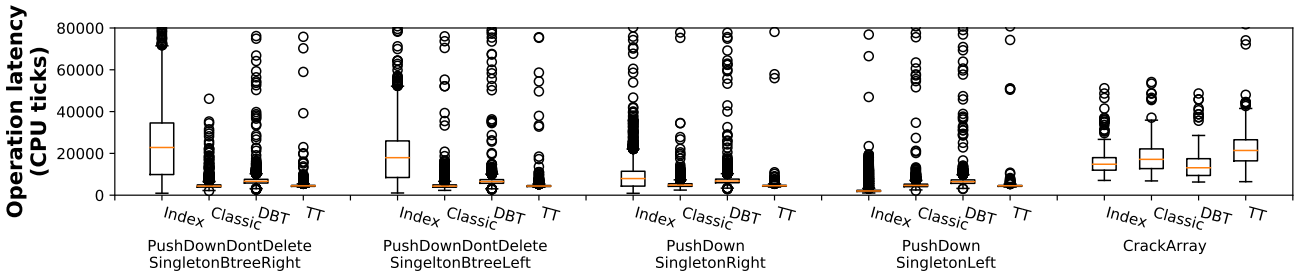


(d) Workload D

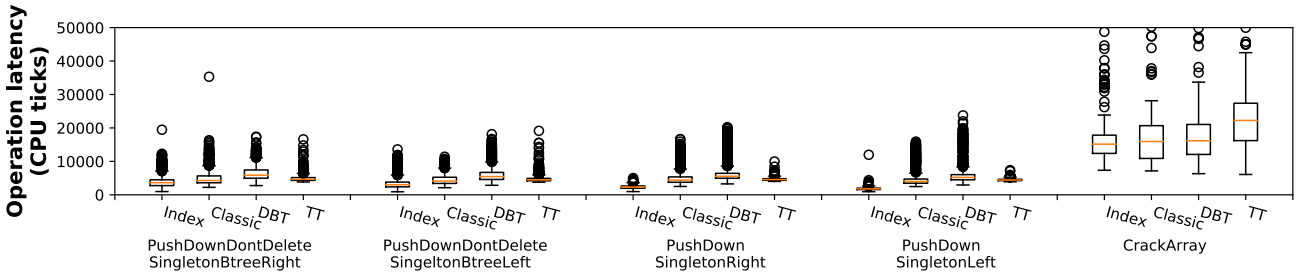


(e) Workload F

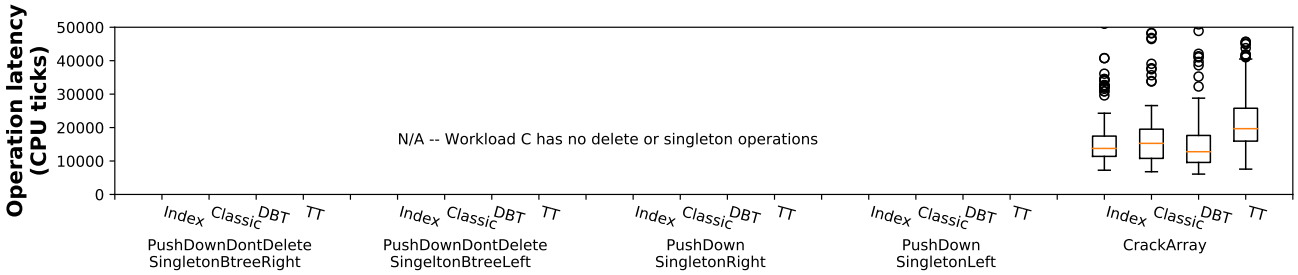
Figure 9. Relative Average Search Technique Performance by Rewrite Rule



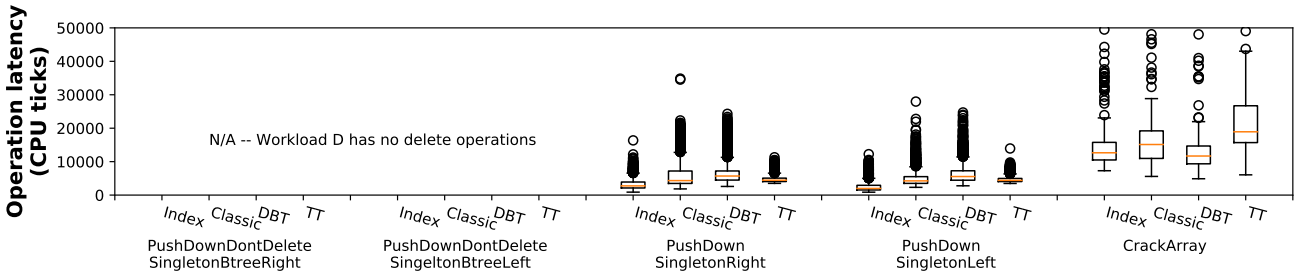
(a) Workload A



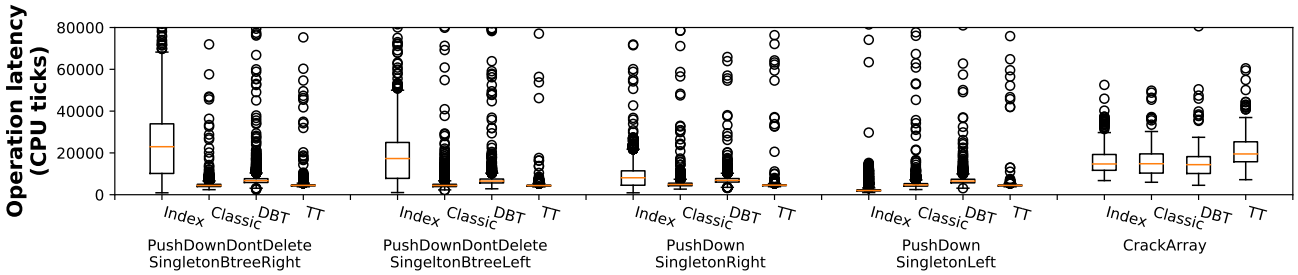
(b) Workload B



(c) Workload C



(d) Workload D



(e) Workload F

Figure 10. Relative Total Search + Maintenance Cost by Rewrite Rule

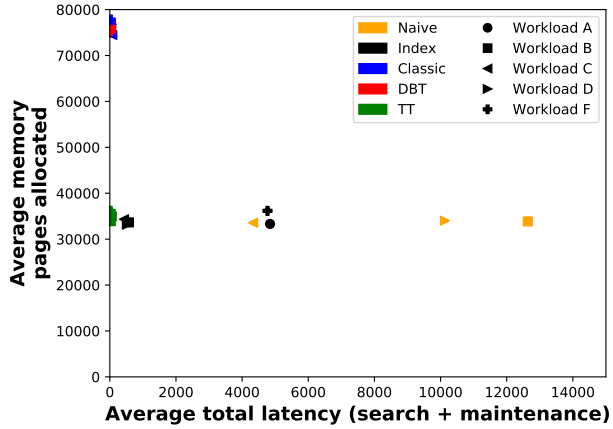


Figure 11. Total Latency (search cost + maintenance operations) and Memory Use, by method and node type

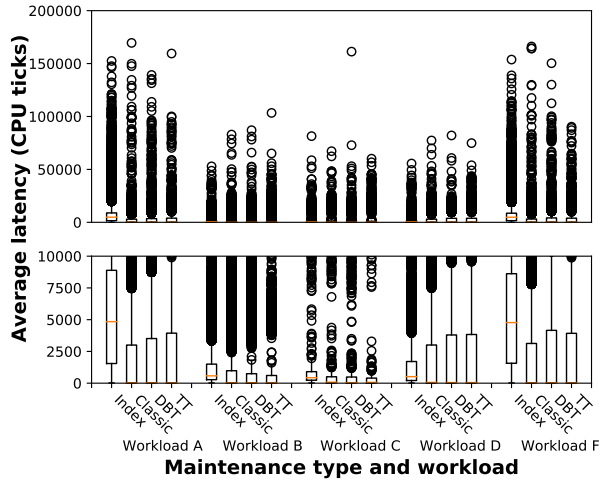


Figure 12. Average IVM Operational Latency. The bottom plot is a zoomed-in view of the top plot.

separator.

$$\texttt{Match}(\texttt{Concat}, C, [\texttt{Match}(\texttt{BinTree}, B, q_1, q_2, \emptyset), \texttt{Match}(\texttt{Singleton}, S, \emptyset, \emptyset)], S.\texttt{key} < \texttt{sep}) \rightarrow \texttt{Gen}(\texttt{BinTree}, [\texttt{sep}], [\texttt{Gen}(\texttt{Concat}, [], [\texttt{Reuse}(q_1), \texttt{Reuse}(S),]), \texttt{Reuse}(q_2),])$$

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 [4] and Orca [37]. A detailed discussion of these rules and how they relate to the example patterns is provided in an accompanying technical report [5].

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 YCSB [14] 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 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

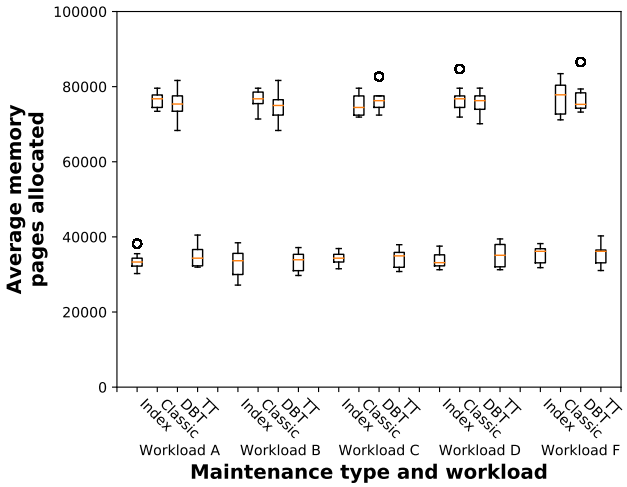


Figure 13. Average Process Memory Usage Summary.

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.1 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 IVM.

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 [13] for a survey. The area has been extensively studied, with techniques developed for incremental maintenance support for of a wide range of data models [8, 12, 35, 42] and query language features [20, 23, 26, 33].

Notable are techniques that improve performance through dynamic programming [24, 27, 35, 43]. A common approach is materializing intermediate results; For one plan as proposed by Ross et. al. [35], or all possible plans as proposed by Koch et. al [24]. A key feature of both approaches is computing the mininmal update – or slice – of the query result, an idea core to systems like Differential Dataflow [27]. 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 [43], and fixed point computations [27].

Also relevant is the idea of embedding query processing logic into a compiled application [2, 18, 24, 28, 30, 32, 34, 36]. Systems like BerkeleyDB, SQLite, and DuckDB embed full query processing logic, while systems like DBToaster [2, 24, 30] and LinQ [28] 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 [28, 36, 41]. Major database engines have also recently been extended to compile queries to native code [11, 15, 29], albeit at query compile time.

To our knowledge, IVM over Abstract Syntax Trees has not been studied directly. The Cascades framework [17] considers streamlined approaches to scheduling rule application, a strategy that is used by the Orca [37] compiler. IVM approaches also exist for general tree and graph query languages and data models like XPath [16], Cypher [38, 39], and the Nested Relational Calculus [25]. 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, 44], 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 [44] 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, 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] Bahareh Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. 2018. GProM - A Swiss Army Knife for Your Provenance Needs. *IEEE Data Engineering Bulletin* 41, 1 (2018), 51–62.

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

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

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

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

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

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

[10] Michael Brachmann, William Spoth, Oliver Kennedy, Boris Glavic, Heiko Mueller, Sonia Castelo, Carlos Bautista, and Juliana Freire. 2020. Your notebook is not crumby enough, REPLace it. In *CIDR*.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

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

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

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

[20] 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.

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

[22] Oliver Kennedy and Lukasz Ziarek. 2015. Just-In-Time Data Structures.. In *CIDR*. Citeseer.

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

[24] 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.

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

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

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

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

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

[30] 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.

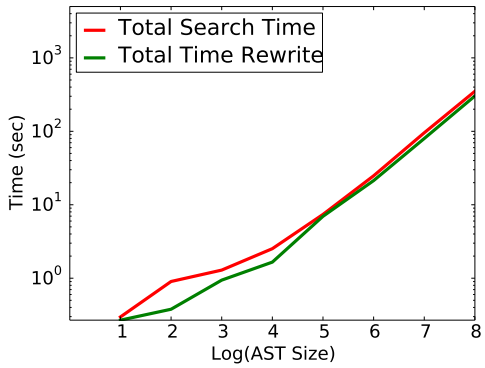
[31] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385.

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

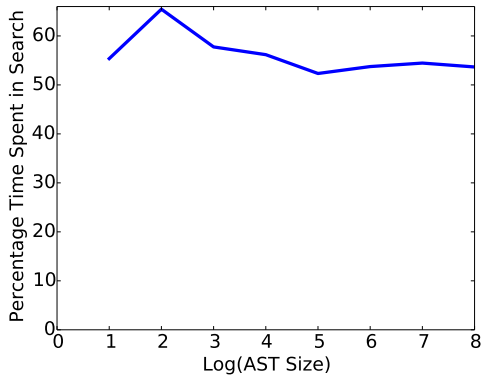
[33] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. 2002. Incremental Maintenance for Non-Distributive Aggregate Functions. In *VLDB*. Morgan Kaufmann, 802–813.

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

[35] 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.
[36] Amir Shaikhha. 2013. An Embedded Query Language in Scala. http://infoscience.epfl.ch/record/213124
[37] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: a modular query optimizer architecture for big data. In *SIGMOD Conference*. ACM, 337–348.
[38] Gábor Szárnyas. 2018. Incremental View Maintenance for Property Graph Queries. In *SIGMOD Conference*. ACM, 1843–1845.
[39] 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).
[40] The Transaction Processing Performance Council. [n.d.]. The TPC-H Benchmark. http://www.tpc.org/tpch/.
[41] Thomas Würthinger. 2014. Graal and truffle: modularity and separation of concerns as cornerstones for building a multipurpose runtime. In *MODULARITY*. ACM, 3–4.
[42] Jun Yang and Jennifer Widom. 2003. Incremental computation and maintenance of temporal aggregates. *VLDB J.* 12, 3 (2003), 262–283.
[43] Ying Yang and Oliver Kennedy. 2017. Convergent Interactive Inference with Leaky Joins. In *EDBT*. OpenProceedings.org, 366–377.
[44] Yue Zhuge and Hector Garcia-Molina. 1998. Graph Structured Views and Their Incremental Maintenance. In *ICDE*. IEEE Computer Society, 116–125.



(a) Spark: Total time spent in rewrite



(b) Spark: Percentage time in AST search.

Figure 14. Rewrite and Search Times for Spark's Optimizer

A Exploration of Spark and ORCA

In this section we present a thorough exploration of Spark's Catalyst optimizer and Greenplum's Orca. We have studied two open source SQL optimizers with an eye towards understanding how much of a time sink AST searches are. To accomplish this we used a simple, easily scalable query pattern:

```
CREATE VIEW TABLE_[N] AS
  SELECT * FROM (
SELECT * FROM TABLE_[N-1]
           UNION ALL SELECT * FROM TABLE_[N-1]
  ) a,
  SELECT * FROM (
SELECT * FROM TABLE_[N-1]
           UNION ALL SELECT * FROM TABLE_[N-1]
  ) b,
  WHERE a.attr = b.attr
```

This query structure is representative of an antipattern that arises naturally in query rewriting for provenance-tracking (e.g., [3, 10]), and that such compilers must explicitly guard against. Results appear below in Figures 14a, 14b, 15a and 15b. Spark results shown are the average of 5 runs. Orca timings were noisier, so we take the average of 10 runs. Spark's optimizer works through Scala's native pattern matching syntax (a recursive `match { case }` blocks, or more precisely calls to Spark's `transform { case }` utility function). We obtained these timings by logging the time spent in each state i.e, serach for a pattern and apply a pattern. Orca's compiler has a more intricate rule scheduling mechanism, but also works by recursive tree traversal during which a pairwise recursive traversal of the pattern AST and AST subtrees is used to check for matches. We measure the time taken in this match and contrast.

Take-away: Both Catalyst and Orca as seen in Figures 14b and 15b spend a non-negligible fraction of their optimization time searching for candidate AST nodes to rewrite (50-60% for Spark, 5-20% for Orca). In both cases, the relative fraction of time spent searching drops asymptotically (to about 50%, 5% respectively) as the AST size grows, but continues **scaling linearly with the AST size** Figures 14a and 15a. These results suggest that (i) At small scales, pattern-matching is a dominant cost for query optimization, and (ii) Any comprehensive strategy that will allow optimizers to scale to enormous ASTs will need to include a technique analogous to TreeToaster.

B JITD Compiler, Spark and ORCA

We did a thorough assessment of optimizer rules in Catalyst and ORCA, and found that most of the rules were comparable to, if not strict subsets of the rules used in the JustInTime-Data compiler we were evaluating — and all of these rules are local patterns. We first represent Spark and ORCA AST Nodes with our grammar in Appendix C and present a detailed overview of our assessment results in Appendices D and E.

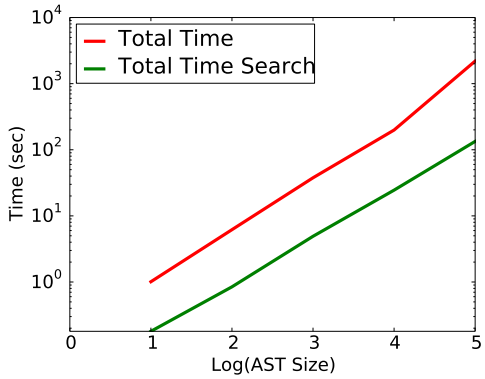
C Spark and ORCA AST Schemas

This subsection defines a simplified schema (in terms of the node's attributes and children) for nodes in Spark's `LogicalPlan` AST and ORCA's `CExpression` AST. Because all nodes in Spark share common attributes, we also allow pattern matching on a node with a variable label ℓ:

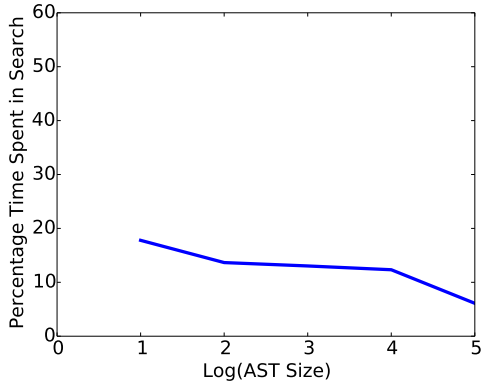
$$(\ell, \texttt{[o:Seq[Attribute],r:AttributeSet,...]}, \overline{N})$$

D Spark Transforms

Scala's optimizer makes extensive use of `LogicalPlan`'s `transform` method (among several variants), which does a pattern-matching search and replaces AST nodes based on a Scala pattern match (a `case` clause). In the following, we provide (i) the `case` clauses, (ii) The corresponding pattern matching expression in our grammar, and (iii) The most similar transform.



(a) Orca: Total time spent in rewrite



(b) Orca: Percentage time in AST search.

Figure 15. Rewrite and Search Times for Spark's Optimizer

D.1 Transform: RemoveNoopOperators

Scala:

```
case p : Project(_, child) if child.sameOutput(p)
case w: Window if w.windowExpressions.isEmpty
```

Patterns:

Match(Project, [o_1, r, p], Match(ℓ, [o_2, . . .], $\overline{Q}$, $\mathbb{T}$), $o_1 = o_2$)
Match(Window, [o, r, w, ps, so], q_1, w.empty)

Most Similar JITD Pattern: DeleteSingletonFromArray

D.2 Transform: CombineFilters

Scala:

```
case Filter(fc, nf : Filter(nc, grandChild))
        if fc.deterministic && nc.deterministic
```

Patterns:

Match(Filter, [o_1, r_1, c_1], Match(Filter, [o_2, r_2, c_2], q_1, c_2.deterministic), c_1.deterministic)

Most Similar JITD Pattern: MergeSortedConcat,PushDownDowntDeleteSingletonLeft/Right

D.3 Transform: PushPredicateThroughNonJoin

Scala:

```
case Filter(condition, project :
        Project(fields,grandChild))
            if fields.forall(_.deterministic) &&
                canPushThroughCondition(grandChild,
                condition)

case filter : Filter(condition,
        aggregate: Aggregate)
        if aggregate.aggregateExpressions.forall(
            _.deterministic) &&
            aggregate.groupingExpressions.nonEmpty

case filter : Filter(condition, w: Window)
        if w.partitionSpec.forall(
            _.isInstanceOf[AttributeReference])

case filter : Filter(condition, union: Union)

case filter : Filter(condition, watermark:
        EventTimeWatermark)

case filter : Filter(_, u: UnaryNode)
    if canPushThrough(u) && u.expressions.forall(
        _.deterministic)

def canPushThrough(p: UnaryNode): Boolean =
p match {
    case _: AppendColumns => true
    case _: Distinct => true
    case _: Generate => true
    case _: Pivot => true
    case _: RepartitionByExpression => true
    case _: Repartition => true
    case _: ScriptTransformation => true
    case _: Sort => true
    case _: BatchEvalPython => true
    case _: ArrowEvalPython => true
    case _: Expand => true
    case _ => false
  }
```

Patterns:

Match(Filter, [o_1, r_1, c], Match(Project, [o_2, r_2, p], Match(ℓ, [o_3, r_3, . . .], $\overline{Q}$, $\mathbb{T}$), p.deterministic), canPushThroughCondition(c, Match(ℓ, [o_3, r_3, . . .], $\overline{Q}$, $\mathbb{T}$)))
Match(Filter, [o_1, r_1, c], Match(Aggregate, [o_2, r_2, g, a], q_1, $\mathbb{T}$), [a.deterministic, g.empty])
Match(Filter, [o_1, r_1, c], Match(Window, [o_2, r_2, w, ps, so], q_1, ps.isInstanceOf[AttributeReference]), $\mathbb{T}$)
Match(Filter, [o_1, r_1, c], Match(Union, [o_2, r_2, p, a], [$q_1 \ldots q_n$], $\mathbb{T}$), $\mathbb{T}$)
Match(Filter, [o_1, r_1, c],

```
SubqueryExpr{[o:Seq[Attribute],r:AttributeSet,ce:Seq[Expression],e_id:ExprId]}{N_1}
UnaryNode{[o:Seq[Attribute],r:AttributeSet,e: Seq[Expression,..]}{N_1}
Project{[o:Seq[Attribute],r:AttributeSet,p:Seq[NamedExpr]]}{N_1}
Window{[o:Seq[Attribute],r:AttributeSet,w:Seq[NamedExpr],ps:Seq[Expr],...]}{N_1}
Filter{[o:Seq[Attribute],r:AttributeSet,c:Expression]}{N_1}
Aggregate{[o:Seq[Attribute],r:AttributeSet,g:Seq[Expr],a:Seq[NamedExpr]]}{N_1}
Union{[o:Seq[Attribute],r:AttributeSet,p:Bool,a:Bool]}{[N_1...N_n]}
WaterMark{[o:Seq[Attribute],r:AttributeSet,e:EventTime,d:Delay]]}{N_1}
Join{[o:Seq[Attribute],r:AttributeSet,j:JoinType,c:Expression,h:JoinHint]}{N_1,N_2}
Expand{[o:Seq[Attribute],r:AttributeSet,p:Seq[Seq[Expr]],op:Seq[Attribute]]}{N_1}
Deserialize{[o:Seq[Attribute],r:AttributeSet,e:Expr,a:Attr]}{N_1}
FlatMap{[o:Seq[Attribute],r:AttributeSet,e:Expr,a:Attr]}{N_1}
ScriptTransf{[o:Seq[Attribute],r:AttributeSet,i:Seq[Expr],s:String,...]}{N_1}
Distinct{[o:Seq[Attribute],r:AttributeSet]}{N_1}
Generate{[o:Seq[Attribute],r:AttributeSet,g: Generator, u: Seq[Int], ...]}{N_1}
SetOp{[o:Seq[Attribute],r:AttributeSet]}{N_1,N_2}
Subquery{[o:Seq[Attribute],r:AttributeSet,c:Bool]}{N_1}
Limit{[o:Seq[Attribute],r:AttributeSet,c:Expression]}{N_1}
LocalRel{[o:Seq[Attribute],r:AttributeSet,op:Seq[Attribute],...]}{}
Repartion{[o:Seq[Attribute],r:AttributeSet,n:Int,s:Bool]}{N_1}
GlobalLimit{[o:Seq[Attribute],r:AttributeSet,g:Expr]]}{N_1}
LocalLimit{[o:Seq[Attribute],r:AttributeSet,l:Expr]]}{N_1}
Sample{[o:Seq[Attribute],r:AttributeSet,l: Double, u: Double, w: Bool, ...]}{N_1}
```

Figure 16. A selection of Spark's LogicalPlan node types expressed as schemas for $\mathcal{G}$

```
CLogicalNAryJoin{[exprhdl:CExpressionHandle,p:childPredicateList]}{N̄}
CLogicalGet{[exprhdl:CExpressionHandle,t:CName,pt:CTableDescriptor,...]}{∅}
CLogicalSelect{[exprhdl:CExpressionHandle,m:CHashMapExpr2Expr,
        pt:CTableDescriptor,predicateExpr:CExpression]}{N_1}
CLogicalInnerJoin
        {[exprhdl:CExpressionHandle,predicateExpr:CExpression]}{N_1,N_2}
CLogicalUnionAll
        {[exprhdl:CExpressionHandle,i:Int,o:CColRefArray,k:CColRef2dArray]}{N̄}
```

Figure 17. A selection of Orca's AST node types expressed as schemas for $\mathcal{G}$

```
          Match(WaterMark,[o_2,r_2,e,d],
          q_1,𝕋),𝕋)
Match(Filter,[o_1,r_1,c],
          Match(UnaryNode,[o_2,r_2,e,...],q_1,
     [e.deterministic,canPushThrough(u)]),𝕋)
```

Most Similar JITD Pattern: MergeUnSortedConcatArray, PushDownDontDeleteSingletonLeft/Right

D.4 Transform: PushPredicateThroughJoin

Scala:

```
case f : Filter(filterCondition,
        Join(left, right, joinType,
            joinCondition, hint))
            if canPushThrough(joinType)

case j : Join(left, right, joinType,
        joinCondition, hint)
            if canPushThrough(joinType)

private def canPushThrough(joinType: JoinType):
Boolean = joinType match {
    case _: InnerLike | LeftSemi |
            RightOuter | LeftOuter | LeftAnti |
            ExistenceJoin(_) => true
    case _ => false
  }
```

Patterns:
Match(Filter, [o_1,r_1,c_1],
Match(Join, [o_2,r_2,j,c_2,h], q_1, q_2,
canPushThrough(j)), $\mathbb{T}$)
Match(Join, [o,r,j,c,h], q_1, q_2, canPushThrough(j))
Most Similar JITD Pattern: PushDownAndCrack, MergeSortedBTrees

D.5 Transform: ColumnPruning

Scala:

```
case p : Project(_, p2: Project)
       if !p2.outputSet.subsetOf(p.references)
case p : Project(_, a: Aggregate)
       if !a.outputSet.subsetOf(p.references)
case a : Project(_, e : Expand(_, _, grandChild))
       if !e.outputSet.subsetOf(a.references)
case d : DeserializeToObject(_, _, child)
       if !child.outputSet.subsetOf(d.references)
case a : Aggregate(_, _, child)
       if !child.outputSet.subsetOf(a.references)
case f : FlatMapGroupsInPandas(_, _, _, child)
       if !child.outputSet.subsetOf(f.references)
case e : Expand(_, _, child)
       if !child.outputSet.subsetOf(e.references)
case s : ScriptTransformation(_, _, _, child, _)
       if !child.outputSet.subsetOf(s.references)
case p : Project(_, g: Generate)
       if p.references ≠ g.outputSet
case j : Join(_, right, LeftExistence(_), _, _)
case p : Project(_, _: SetOperation)
case p : Project(_, _: Distinct)
case p : Project(_, u: Union)
case p : Project(_, w: Window)
       if
       !w.windowOutputSet.subsetOf(p.references)
case p : Project(_, _: LeafNode)
case p : Project(_, child)
       if !child.isInstanceOf[Project]
case GeneratorNestedColumnAliasing(p)
case NestedColumnAliasing(p)
```

Patterns:
Match(Project, [o_1,r_1,p_1],
Match(Project, [o_2,r_2,p_2], q_1, $\mathbb{T}$), $o_2 \subseteq r_1$)
Match(Project, [o_1,r_1,p_1],
Match(Aggregate, [o_2,r_2,g,a], q_1, $\mathbb{T}$),
$o_2 \subseteq r_1$)
Match(Project, [o_1,r_1,p_1],
Match(Expand, [o_2,r_2,p_2,op], q_1, $\mathbb{T}$),
$o_2 \subseteq r_1$)
Match(Deserialize, [o_1,r_1,e,a],
Match(ℓ, [o_2,r_2,. . .], $\overline{Q}$, $\mathbb{T}$), $o_2 \subseteq r_1$)
Match(Aggregate, [o_1,r_1,g,a],
Match(ℓ, [o_2,r_2,. . .], $\overline{Q}$, $\mathbb{T}$), $o_2 \subseteq r_1$)
Match(FlatMap, [o_1,r_1,e,a],
Match(ℓ, [o_2,r_2,. . .], $\overline{Q}$, $\mathbb{T}$), $o_2 \subseteq r_1$)
Match(Expand, [o_1,r_1,p,o],
Match(ℓ, [o_2,r_2,. . .], $\overline{Q}$, $\mathbb{T}$), $o_2 \subseteq r_1$)
Match(ScriptTransf, [o_1,r_1,i,s,op,io],
Match(ℓ, [o_2,r_2,. . .], $\overline{Q}$, $\mathbb{T}$), $o_2 \subseteq r_1$)
Match(Project, [o_1,r_2,p],
Match(Generate, [o_2,r_2,g,u,ob,q,go],
q_1, $\mathbb{T}$), $r_1 \neq o_2$)
Match(Join, [o,r,j,c,h:leftExistence], q_1, q_2, $\mathbb{T}$)
Match(Project, [o_1,r_1,p],
Match(SetOp, [o_2,r_2], q_1, q_2, $\mathbb{T}$), $\mathbb{T}$)
Match(Project, [o_1,r_1,p],
Match(Distinct, [o_2,r_1], q_1, $\mathbb{T}$), $\mathbb{T}$)
Match(Project, [o_1,r_1,p_1],
Match(Union, [o_2,r_2,p_2,a],
[$q_1 \ldots q_n$], $\mathbb{T}$), $\mathbb{T}$)
Match(Project, [o_1,r_1,p_1],
Match(Window, [o_2,r_2,w,p_2], q_1, $\mathbb{T}$),
$o_2 \subseteq r_2$)
Match(Project, [o_1,r_1,p_1],
Match(ℓ, [o_2,r_2,. . .], $\emptyset$, $\mathbb{T}$), $\mathbb{T}$)
Match(Project, [o_1,r_1,p_1],
Match(Project, [o_2,r_2,p_2], q_1, $\mathbb{T}$), $\mathbb{T}$)
Match(Project, [o_1,r_1,p_1], N_1, $\mathbb{T}$)
Match(Generate, [o_2,r_2,g,u,ob,q,go], q_1, $\mathbb{T}$)
Most Similar JITD Pattern: MergeConcatNodes

D.6 Transform: RewritePredicateSubquery

Scala:

```
case Filter(condition, child)
```

Patterns:
Match(Filter, [o,r,c], q_1, $\mathbb{T}$)
Most Similar JITD Pattern: DeleteSingletonFromArray

D.7 Transform: RemoveRedundantAliases

Scala:

```
case Subquery(child, correlated)

case Join(left, right, joinType, condition, hint)
```

Patterns:
Match(Subquery, [o,r,c], q_1, $\mathbb{T}$)
Match(Join, [o,r,j,c,h], q_1, q_2, $\mathbb{T}$)
Most Similar JITD Pattern: CrackArray,DeleteSingletonFromArray

D.8 Transform: InferFiltersFromConstraints

Scala:

```
case filter : Filter(condition, child)

case join : Join(left, right,
                 joinType, conditionOpt, _)
```

Patterns:
Match(Filter, [o,r,c], q_1, $\mathbb{T}$)
Match(Join, [o,r,j,c,h], q_1, q_2, $\mathbb{T}$)

Most Similar JITD Pattern: CrackArray,DeleteSingletonFromArray

D.9 Transform: ConvertToLocalrelation

Scala:

```
case Project(projectList,
        LocalRelation(output, data, isStreaming))
        if !projectList.exists(hasUnevaluableExpr)

case Limit(IntegerLiteral(limit),
        LocalRelation(output, data, isStreaming))

case Filter(condition,
        LocalRelation(output, data, isStreaming))
            if !hasUnevaluableExpr(condition)

private def hasUnevaluableExpr(expr: Expression):
Boolean = {
    expr.find(e => e.isInstanceOf[Unevaluable] &&
    !e.isInstanceOf[AttributeReference]).isDefined
 }
```

Patterns:

Match(Project, $[o_1, r_2, p]$,
Match(LocalRelation, $[o_2, r_2, op, d, i], \emptyset, \mathbb{T}$),
p.exists(hasUnevaluableExpr))
Match(Limit, [c],
Match(LocalRelation, [o,d,i], $\emptyset, \mathbb{T}$), $\mathbb{T}$)
Match(Filter, $[o_1, r_1, c]$,
Match(LocalRelation, $[o_2, r_2, op, d, i], \emptyset, \mathbb{T}$),
c.exists(hasUnevaluableExpr))

Most Similar JITD Pattern: DeleteSingletonFromArray

D.10 Transform: CollapseProject

Scala:

```
case p1 : Project(_, p2: Project)

case p : Project(_, agg: Aggregate)

case Project(l1, g : GlobalLimit(_, limit :
        LocalLimit(_, p2 : Project(l2, _))))
            if isRenaming(l1, l2)

case Project(l1, limit :
        LocalLimit(_, p2 : Project(l2, _)))
            if isRenaming(l1, l2)

case Project(l1, limit :
        LocalLimit(_, p2 : Project(l2, _)))
            if isRenaming(l1, l2)

case Project(l1, r :
        Repartition(_, _, p : Project(l2, _)))
            if isRenaming(l1, l2)

case Project(l1, s :
        Sample(_, _, _, _, p2 : Project(l2, _)))
            if isRenaming(l1, l2)

private def isRenaming(list1: Seq[NamedExpression],
list2: Seq[NamedExpression]): Boolean =
{
    list1.length == list2.length &&
    list1.zip(list2).forall {
      case (e1, e2) if e1.semanticEquals(e2) => true
      case (Alias(a: Attribute, _), b)
      if a.metadata == Metadata.empty && a.name ==
      b.name => true
      case _ => false
    }
  }
}
```

Patterns:

Match(Project, $[o_1, r_1, p_1]$,
Match(Project, $[o_2, r_2, p_2], q_1, \mathbb{T}), \mathbb{T}$)
Match(Project, $[o_1, r_1, p]$,
Match(Aggregate, $[o_2, r_2, g, a], q_1, \mathbb{T}), \mathbb{T}$)
Match(Project, $[o_1, r_1, p_1]$,
Match(Globallimit, $[o_2, r_2, g]$,
Match(LocalLimit, $[o_3, r_3, l]$,
Match(Project, $[o_4, r_4, p_2], q_1, \mathbb{T}), \mathbb{T}), \mathbb{T}$),
isRenaming(p_1, p_2))
Match(Project, $[o_1, r_1, p_1]$,
Match(LocalLimit, $[o_2, r_2, l]$,
Match(Project, $[o_3, r_3, p_1], q_1, \mathbb{T}), \mathbb{T}$),
isRenaming(p_1, p_2))
Match(Project, $[o_1, r_1, p_1]$,
Match(Repartition, $[o_2, r_2, n, s]$,
Match(Project, $[o_3, r_3, p_2], q_1, \mathbb{T}), \mathbb{T}$),
isRenaming(p_1, p_2))

Match(Project, $[o_1, r_1, p_1]$,
Match(Sample, $[o_2, r_2, l, u, w, s]$,
Match(Project, $[o_3, r_3, p_2], q_1, \mathbb{T}), \mathbb{T}$),
isRenaming(p_1, p_2))

Most Similar JITD Pattern: PivotLeft-/Right,PushDownAndCrack The third match pattern represents a 4-way join which is a exception. Most other look at a 3-level deep subtree representative of a 3-way join.

E ORCA Transforms

Orca defines specific patterns for rewrites and matches at runtime the AST nodes. If the node matches the pattern a rewrite is defined over the node is replaced. Orca's AST grammar is slightly more expressive than our won in that certain `CExpression` nodes in ORCA like `CLogicalNaryJoin` support multiple children; however (i) this is a limitation we impose largely for simplicity of presentation, and (ii) none of the patterns we encountered include recursive patterns among the children of such variable-child nodes.

In the following, we provide (i) the Pattern for rewrites along with the function that computes the rewrite's promise

which determines the priority of the rewrite, (ii) The corresponding pattern matching expression in our grammar, and (iii) The most similar transform. Determining the priority of a rewrite is encoded in the grammar as a constraint over the pattern match.

E.1 Transform: ExpandNaryJoin

Cpp:

```
CXformExpandNAryJoin::
CXformExpandNAryJoin(CMemoryPool *mp)
   : CXformExploration(
        // pattern
        GPOS_NEW(mp) CExpression(
           mp, GPOS_NEW(mp) CLogicalNAryJoin(mp),
           GPOS_NEW(mp)
           CExpression(mp, GPOS_NEW(mp)
           CPatternMultiLeaf(mp)),
           GPOS_NEW(mp)
           CExpression(mp, GPOS_NEW(mp)
           CPatternTree(mp))))
{
}
CXform::EXformPromise
CXformExpandNAryJoin::
Exfp(CExpressionHandle &exprhdl) const
{
   if (exprhdl.DeriveHasSubquery(exprhdl.Arity() - 1))
   {
      // subqueries must be unnested before
      // applying xform
      return CXform::ExfpNone;
   }
#ifdef GPOS_DEBUG
   CAutoMemoryPool amp;
   GPOS_ASSERT(!CXformUtils::
   FJoinPredOnSingleChild(amp.Pmp(), exprhdl) &&
            "join_predicates_are_not_pushed_down");
#endif   // GPOS_DEBUG

   return CXform::ExfpHigh;
}
```

Patterns:

Match(CLogicalNAryJoin, [exprhdl, p], $[q_1 \dots q_n]$, exprhdl.hasSubQuery)

Most Similar JITD Pattern: PushDownAndCrack

E.2 Transform: ExpandNaryJoinMinCard

Cpp:

```
CXformExpandNAryJoinMinCard::
CXformExpandNAryJoinMinCard(CMemoryPool *mp)
   : CXformExploration(
        // pattern
        GPOS_NEW(mp) CExpression(
           mp, GPOS_NEW(mp) CLogicalNAryJoin(mp),
           GPOS_NEW(mp)
           CExpression(mp, GPOS_NEW(mp)
           CPatternMultiTree(mp)),
           GPOS_NEW(mp)
           CExpression(mp, GPOS_NEW(mp)
           CPatternTree(mp))))
{
}
CXform::EXformPromise
CXformExpandNAryJoinMinCard::
Exfp(CExpressionHandle &exprhdl) const
{
   return CXformUtils::
   ExfpExpandJoinOrder(exprhdl, this);
}
```

Patterns:

Match(CLogicalNAryJoin, [exprhdl, p], $[q_1 \dots q_n]$, exprhdl.expandJoinOrd)

Most Similar JITD Pattern: PushDownAndCrack

E.3 Transform: ExpandNaryJoinGreedy

Cpp:

```
CXformExpandNAryJoinGreedy::
CXformExpandNAryJoinGreedy(CMemoryPool *pmp)
   : CXformExploration(
        // pattern
        GPOS_NEW(pmp) CExpression(
           pmp, GPOS_NEW(pmp)
           CLogicalNAryJoin(pmp),
           GPOS_NEW(pmp)
              CExpression(pmp, GPOS_NEW(pmp)
              CPatternMultiTree(pmp)),
           GPOS_NEW(pmp) CExpression(pmp,
           GPOS_NEW(pmp) CPatternTree(pmp))))
{
}
CXform::EXformPromise
CXformExpandNAryJoinGreedy::
Exfp(CExpressionHandle &exprhdl) const
{
   return CXformUtils::
   ExfpExpandJoinOrder(exprhdl, this);
}
```

Patterns:

Match(CLogicalNAryJoin, [exprhdl, p], $[q_1 \dots q_n]$, exprhdl.expandJoinOrd)

Most Similar JITD Pattern: PushDownAndCrack

E.4 Transform: ExpandNAryJoinDP

Cpp:

```
CXformExpandNAryJoinDP::
CXformExpandNAryJoinDP(CMemoryPool *mp)
   : CXformExploration(
        // pattern
        GPOS_NEW(mp) CExpression(
           mp, GPOS_NEW(mp) CLogicalNAryJoin(mp),
           GPOS_NEW(mp) CExpression(mp, GPOS_NEW(mp)
           CPatternMultiLeaf(mp)),
           GPOS_NEW(mp) CExpression(mp, GPOS_NEW(mp)
           CPatternTree(mp))))
{
}

CXform::EXformPromise
CXformExpandNAryJoinDP::
Exfp(CExpressionHandle &exprhdl) const
{
   COptimizerConfig *optimizer_config =
      COptCtxt::PoctxtFromTLS()->GetOptimizerConfig();
   const CHint *phint = optimizer_config->GetHint();

   const ULONG arity = exprhdl.Arity();

   // since the last child of the join operator is a
```

```
   // scalar child
   // defining the join predicate, ignore it.
   const ULONG ulRelChild = arity - 1;

   if (ulRelChild > phint->UlJoinOrderDPLimit())
   {
      return CXform::ExfpNone;
   }

   return CXformUtils::
   ExfpExpandJoinOrder(exprhdl, this);
}
```

Patterns:
Match(CLogicalNAryJoin, [exprhdl,p], $[q_1 \dots q_n]$, exprhdl.Arity-1 > x || exprhdl.expandJoinOrd)
Code:
Most Similar JITD Pattern: PushDownAndCrack

E.5 Transform: Get2TableScan

Cpp:

```
CXformGet2TableScan::CXformGet2TableScan(CMemoryPool *mp)
   : CXformImplementation(
        // pattern
        GPOS_NEW(mp) CExpression(mp, GPOS_NEW(mp)
        CLogicalGet(mp)))
{
}
CXform::EXformPromise
CXformGet2TableScan::
Exfp(CExpressionHandle &exprhdl) const
{
   CLogicalGet *popGet =
   CLogicalGet::PopConvert(exprhdl.Pop());

   CTableDescriptor *ptabdesc = popGet->Ptabdesc();
   if (ptabdesc->IsPartitioned())
   {
      return CXform::ExfpNone;
   }

   return CXform::ExfpHigh;
}
```

Patterns:
Match(CLogicalGet, [exprhdl,t,pt,o,k,c], $\emptyset$, pt.isPartitioned)

Most Similar JITD Pattern: CrackArray

E.6 Transform: Select2Filter

Cpp:

```
CXformSelect2Filter::CXformSelect2Filter(CMemoryPool *mp)
   :  // pattern
     CXformImplementation(GPOS_NEW(mp) CExpression(
        mp, GPOS_NEW(mp) CLogicalSelect(mp),
        GPOS_NEW(mp) CExpression(
           mp, GPOS_NEW(mp) CPatternLeaf(mp)),
// relational child
        GPOS_NEW(mp)
           CExpression(mp, GPOS_NEW(mp) CPatternLeaf(mp))
// predicate
        ))
{
}
CXform::EXformPromise
CXformSelect2Filter::Exfp(CExpressionHandle &exprhdl) const
{
   if (exprhdl.DeriveHasSubquery(1))
   {
      return CXform::ExfpNone;
   }

   return CXform::ExfpHigh;
}
```

Patterns:
Match(CLogicalSelect, [exprhdl,m,pt,PredicateExpr], q_1, exprhdl.hasSubQuery)

Most Similar JITD Pattern: CrackArray

E.7 Transform: InnerJoin2NLJoin

Cpp:

```
CXformInnerJoin2NLJoin::
CXformInnerJoin2NLJoin(CMemoryPool *mp)
   :  // pattern
     CXformImplementation(GPOS_NEW(mp) CExpression(
        mp, GPOS_NEW(mp) CLogicalInnerJoin(mp),
        GPOS_NEW(mp)
           CExpression(mp, GPOS_NEW(mp)
           CPatternLeaf(mp)),  // left child
        GPOS_NEW(mp)
           CExpression(mp, GPOS_NEW(mp)
           CPatternLeaf(mp)),  // right child
        GPOS_NEW(mp)
           CExpression(mp, GPOS_NEW(mp)
           CPatternLeaf(mp))  // predicate
        ))
{
}
CXform::EXformPromise
CXformInnerJoin2NLJoin::
Exfp(CExpressionHandle &exprhdl) const
{
   return CXformUtils::
   ExfpLogicalJoin2PhysicalJoin(exprhdl);
}
```

Patterns:
Match(CLogicalInnerJoin, [exprhdl,predicateExpr], q_1, q_2, exprhdl.ExfpLogicalJoin2PhysicalJoin)

Most Similar JITD Pattern: PushDownAndCrack

E.8 Transform: InnerJoin2HashJoin

Cpp:

```
CXformInnerJoin2NLJoin::
CXformInnerJoin2NLJoin(CMemoryPool *mp)
   :  // pattern
     CXformImplementation(GPOS_NEW(mp) CExpression(
        mp, GPOS_NEW(mp) CLogicalInnerJoin(mp),
        GPOS_NEW(mp)
           CExpression(mp, GPOS_NEW(mp)
           CPatternLeaf(mp)),  // left child
        GPOS_NEW(mp)
           CExpression(mp, GPOS_NEW(mp)
           CPatternLeaf(mp)),  // right child
        GPOS_NEW(mp)
```

```
            CExpression(mp, GPOS_NEW(mp)
            CPatternLeaf(mp))  // predicate
        ))
{
}
CXform::EXformPromise
CXformInnerJoin2NLJoin::
Exfp(CExpressionHandle &exprhdl) const
{
   return CXformUtils::
   ExfpLogicalJoin2PhysicalJoin(exprhdl);
}
```

Patterns:
Match(CLogicalInnerJoin,
[exprhdl,predicateExpr], q_1, q_2,
exprhdl.ExfpLogicalJoin2PhysicalJoin)

Most Similar JITD Pattern: PushDownAndCrack

E.9 Transform: JoinCommutativity

Cpp:

```
CXformJoinCommutativity::
CXformJoinCommutativity(CMemoryPool *mp)
   : CXformExploration(
        // pattern
        GPOS_NEW(mp) CExpression(
           mp, GPOS_NEW(mp) CLogicalInnerJoin(mp),
           GPOS_NEW(mp)
              CExpression(mp, GPOS_NEW(mp)
              CPatternLeaf(mp)),  // left child
           GPOS_NEW(mp) CExpression(
              mp, GPOS_NEW(mp)
              CPatternLeaf(mp)),  // right child
           GPOS_NEW(mp)
              CExpression(mp, GPOS_NEW(mp)
              CPatternLeaf(mp)))  // predicate
     )
{
}
```

```
BOOL
CXformJoinCommutativity::FCompatible(CXform::EXformId exfid)
{
   BOOL fCompatible = true;

   switch (exfid)
   {
      case CXform::ExfJoinCommutativity:
         fCompatible = false;
         break;
      default:
         fCompatible = true;
   }

   return fCompatible;
}
```

Patterns:
Match(CLogicalInnerJoin, [exprhdl,predicateExpr],
q_1, q_2, exprhdl.id)
Most Similar JITD Pattern: PivotLeft/Right

E.10 Transform: ImplementUnionAll

Cpp:

```
CXformImplementUnionAll::
CXformImplementUnionAll(CMemoryPool *mp)
   :  // pattern
     CXformImplementation(GPOS_NEW(mp)
     CExpression(
        mp, GPOS_NEW(mp) CLogicalUnionAll(mp),
        GPOS_NEW(mp) CExpression(mp, GPOS_NEW(mp)
        CPatternMultiLeaf(mp))))
{
}
```

Patterns:
Match(CLogicalUnionAll, [exprhdl,i,o,k], $[q_1 \dots q_n]$, $\mathbb{T}$)

Most Similar JITD Pattern: MergeUnsorted/SortedConcat