Asymptotically Better Query Optimization Using Indexed Algebra

Philipp Fent
Technische Universität München
fent@in.tum.de

Guido Moerkotte
Universität Mannheim
moerkotte@uni-mannheim.de

Thomas Neumann
Technische Universität München
neumann@in.tum.de

ABSTRACT

Query optimization is essential for the efficient execution of queries. The necessary analysis, if we can and should apply optimizations and transform the query plan, is already challenging. Traditional techniques focus on the availability of columns at individual operators, which does not scale for analysis of data flow through the query. Tracking available columns per operator takes quadratic space, which can result in multi-second optimization time for deep algebra trees. Instead, we need to re-think the naïve algebra representation to efficiently support data flow analysis.

In this paper, we introduce *Indexed Algebra*, a novel representation of relational algebra that makes common optimization tasks efficient. Indexed Algebra enables efficient reasoning with an auxiliary index structure based on link/cut trees that support dynamic updates and queries in $O(\log n)$. This approach not only improves the asymptotic complexity, but also allows elegant and concise formulations for the data flow questions needed for query optimization. While large queries see theoretically unbounded improvements, Indexed Algebra also improves optimization time of the relatively harmless queries of TPC-H and TPC-DS by more than 1.8×.

PVLDB Reference Format:
Philipp Fent, Guido Moerkotte, and Thomas Neumann. Asymptotically Better Query Optimization Using Indexed Algebra. PVLDB, 16(11): 3018 - 3030, 2023.
doi:10.14778/3611479.3611505

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tum-db/indexed-algebra.

1 INTRODUCTION

Optimizing the algebra plan of a query can take a significant portion of the overall runtime. The challenge for the optimizer here is that the data flow through the query, and its analysis can be astonishingly complex. Additionally, automatically generated queries with complex business logic amplify this problem [5, 18, 19, 37]. Query optimizers struggle to deal with such complex input, which is especially painful for small datasets where query optimization can be more expensive than query execution. Small data sizes are common during testing, but also in the real world, where, for example, Tableau reports that many workloads contain fewer than a million tuples [39]. As a result, query optimization usually operates on a budget, trading-off optimizations versus optimization time.

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611505

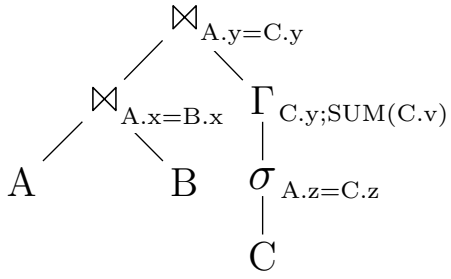


Figure 1: Relation algebra tree with subtle data flow. In this paper, we optimize queries by efficiently analyzing data flow.

Some typical questions that come up during query optimization are: From which part of the plan does a value come from? What are the join predicates? Can we push a predicate down into the inputs? Consider for example the SQL query below:

```
SELECT *
FROM   A, B, C LEFT OUTER JOIN D ON C.u = D.u
WHERE  A.v = 5 AND A.w = B.w AND B.x = C.x
  AND C.y = 7 AND D.z = 8
```

In this small example it is easy to see which attributes form join edges ({u, w, x}), which filters can be pushed down ({v, y}), and which not directly ({z}). In general, these questions are difficult because the FROM clause can contain arbitrary subqueries. The traditional solution to this problem is to keep track of the columns that are available in each step of the query in a set [9, 10] and to move predicates around step by step, checking the available columns in each transformation. But if we have a join tree of depth n, where each join produces at least one column, the construction time for these column sets grows with $\Omega(n^2)$, which is highly unattractive for large queries.

Even if we ignore the performance problems, this myopic look at individual operators is insufficient to express optimizations efficiently. In many cases we want to inspect the full data flow instead. Consider the small algebra tree shown in Figure 1. The top-most join predicate in this tree compares an attribute from its left input (A.y) with an attribute from its right input (C.y). While this data flow direction might be easy to *see* for such a small example, it is non-obvious when there are dozens of operators between the base tables and the predicate. And note that the example tree contains a non-trivial data flow that is not obvious at a first glance: The selection operator on the lower right uses an attribute that is produced in a different part of the operator tree, which effectively makes the top-most operator a *dependent join*. Evaluating such a join is highly inefficient, since it requires a nested loop join execution. The query optimizer has to detect these dependent joins and can then rewrite the query to remove the correlation between parts of the join tree [27]. While we can detect these dependent joins by

reasoning over the columns available after each operator, this again leads to highly unattractive quadratic (or even cubic) algorithms.

What we need instead is a framework to reason about complex data flows without inspecting individual operators. For example, we want to be able to correctly push down a predicate deep into a query tree in one holistic operation, skipping all intermediate operators, and we want to detect dependent joins without traversing the operator tree. This not only makes query optimization more efficient from an asymptotic perspective, it also makes the optimizer more pleasant to write, as we can ask data flow questions about the query itself instead of traversing the operator tree all the time.

In this paper we show how to answer these data flow questions that arise during query optimization with an *Indexed Algebra*. To avoid the frequent tree traversals, we maintain an auxiliary index of the algebraic query representation. This index answers data flow questions in $O(\log n)$ time and supports efficient query transformation. We implement this index structure as a link/cut tree [35] that builds a balanced search structure for the paths through our algebra. Using our Indexed Algebra to answer the data flow questions during query optimization is dramatically more efficient than traditional approaches, while additionally leading to concise and elegant formulations of optimization rules.

Indexed Algebra has the biggest advantage for complex queries, but also improves relatively harmless queries like the ones from TPC-H. Before developing Indexed Algebra, our research system Umbra [26] spent a significant time reasoning over columns. For example, in the intricate nested TPC-H Q2, which our system parses as 21 operators, Umbra spent more than 20% of the optimization time in naïve implementations answering data flow questions. Indexed Algebra helps to significantly improve this, while helping even more for complex queries.

The rest of this paper is structured as follows: We first discuss the algebraic representation of queries and common operations on the algebra in Section 2. We then introduce the idea of indexing paths through the algebra in Section 3, and how to efficiently support dynamic updates to that index. Then, we propose several optimizations using Indexed Algebra in Section 4. Furthermore, we discuss other techniques like property caching in Section 5. The benefits of these techniques are demonstrated in Section 6, and related work is discussed in Section 7.

2 QUERY REPRESENTATION

In this section, we introduce an algebraic representation that represents a query execution plan. We first discuss the general components of this algebra, before we show how we navigate and reason about this algebra.

2.1 Algebra: Operators, Expressions, and IUs

Operators. Like most relational systems, we represent a query in some form of extended relational algebra. *Operators* like joins $\bowtie$, selections σ, group by Γ, or table scans, make up the base structure of the algebra. The operators process a multiset of tuples from one or several inputs that have a clear parent-child relationship, where data conceptually flows from children inputs to the parent outputs. For now, we assume that an operator outputs its tuples to a single output operator, so that the query forms a tree of operators.

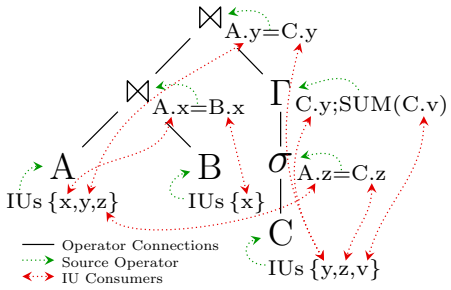


Figure 2: Our algebra representation interlinks operators, expressions, and IUs to allow efficient navigation.

We generalize this operator tree to a DAG in Section 5.3, and first concentrate on simpler tree structured queries.

Expressions. Attached to the operators are *expressions* that process scalar values instead of tuples, e.g., individual columns. Like operators, expressions are tree structured and can be arbitrary nested, however, the expression tree in anchored at exactly one operator. In contrast to relational operators, expressions are wide spread in general purpose programming languages, that also deal with reasoning and optimization of expressions. In Section 5.1, we detail relational algebra specific implementations that deal with expressions.

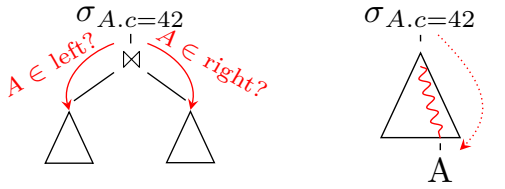
IUs. While our expressions consist of e.g., comparisons, arithmetic or constants, a relational algebra expression can also reference columns of data in relations. This way, an operator can execute, e.g., $x > 42$ as a filter, or $x = y$ as a join predicate. In our algebra, we generalize the notion of a column to an *information unit* (IU) [20]. IUs describe scalar values that can be referenced multiple times, e.g., as a cached expression for common subexpression elimination to avoid repeated computation.

IUs represent the data flow through a query. An IU has a single source operator, typically a table scan, that produces the values. Other operators then pass IUs through the algebra until they reach the consuming expressions. This makes IUs a key part of logical query planning, where we initially do not consider physical implementation details, like if an IU is pipelined and passed through a register or materialized in a hash table. Query optimization then brings the data flow of IUs in a form that is efficient to execute.

2.2 Efficiently Navigating Algebra

The relational algebra uses these components of operators, expressions, and IUs. While this is already enough to model and execute queries, optimizing this algebra is more complex. Simple analyses can be operator-local, e.g., to reorder predicates or to fold constant expressions. However, most optimizations require a structural analysis of the data flow through the relational algebra. For example, to drop unused IUs, we need to know which expressions, if any, use them. Likewise, to move around select σ or map χ operators, we need to consider the source of any consumed IU.

For these analyses, we set up explicit links between the components of our algebra that allow to navigate it efficiently. In Figure 2, we visualize these links as dotted arrows between the components in the example algebra tree of the introduction. Operators are connected via their parent-child relationships, which allow traversing the algebra tree. Expressions also have hierarchical links and are



(a) Operator-centric (b) Path-centric

Figure 3: Traditional predicate pushdown traverses the query tree operator by operator. Path-centric optimization can take a shortcut to the IU's source, but still needs to check if the path is sound, or if it contains e.g., an outer join.

additionally connected to their using operator, e.g., the top most comparison `A.y=C.y` is linked to its join ⋈, which transitively applies to the IU references in the expression. Similarly, IUs are linked to their source operator and are additionally connected to all references of that IU throughout the algebra.

With this setup of links, we can now traverse the tree to build reasoning grounds for optimizations. However, traversing this tree for each analysis, each optimization, each transformation, and each operator seems costly. Therefore, we want to build data structures that allow us to avoid duplicate traversal.

2.3 Reasoning about Column Sets

A traditional technique to reason about the query structure are column sets. For this method, we say that an operator produces a set of IUs, and an expression consumes a set of IUs for evaluation. This way, we can implement all kinds of optimizations, e.g., pushing predicates down a join, where we need to determine, which of the joins inputs produces the required IUs of the predicate.

Computing the required IUs of an operator also requires traversing the query tree. We can limit the algebra traversals by caching the produced IUs per operator, which allows us to calculate all sets in a single pass over the algebra. Unfortunately, we now use dynamic memory per operator to cache the column sets. For an algebra tree of n operators, we also need to cache n column sets. For most queries, the size of the set of produced IUs also grows, since regular joins produce the union of their input columns, which makes the size of these sets in $O(n)$. Consequently, our cache stores $n \cdot O(n) = O(n^2)$ produced IUs.

While reasoning about column sets is quite efficient, building the column sets is already expensive. This caching cost is additionally amplified by wide tables containing dozens of IUs. Also, column sets lead to complex code since transformations of the algebra might need to invalidate the cached column sets, potentially triggering a quadratic recalculation. As we will see in the evaluation in Section 6, the maintenance of these sets is too expensive to be worthwhile.

2.4 Reasoning by Path Traversal

We argue that instead of reasoning about column sets, we should reason about the paths that IUs take through the algebra. Another way to look at these optimizations is if we argue locally for each operator (i.e., with column sets), or if we argue about the whole algebra tree. Figure 3 contrasts these two approaches for predicate push down. In the operator-centric approach, we reason from the perspective of a join, where we can either push the predicate to its left or right inputs. On the other side, in path-centric reasoning, we use the link between IU reference and the producing operator to directly get to the IU source. Then, we traverse the algebra bottom-up to get the push down path and to detect operators like outer joins, through which we cannot directly push predicates.

For path-centric reasoning, we identify two common basic operations: *Finding the root* of an operator tree and intersecting data flows by finding the *lowest common ancestor*. We can find the root of an operator tree by traversing tree bottom up, which allows, e.g., to find the (sub)tree that contains the source of an IU. The lowest common ancestor of two operators is the place where the two paths from these operators to the root intersect, and is useful, e.g., as the earliest possible place to evaluate an expression consuming data from two IU sources. Based on these operations, one can implement many optimizations, which we discuss in Section 4.

The main advantage of this path-centric reasoning is that it requires no dynamic per operator state. Avoiding this state entirely avoids the quadratic space requirements, and additionally simplifies the transformation logic, since it no longer needs to invalidate the column set caches. Still, deep query trees are a problem. The tree traversal for each analysis still can lead to quadratic runtime. Nevertheless, we found that it is still more efficient than building column sets.

While better than column sets, walking the raw algebra tree is still quite naïve. In the following, we propose Indexed Algebra to make path-centric queries efficient. Our index structure maintains a balanced path structure to make traversal fast, while amortizing the balancing of the index during algebra transformations. In effect, we get the simplicity of path traversal with the analysis performance of set operations.

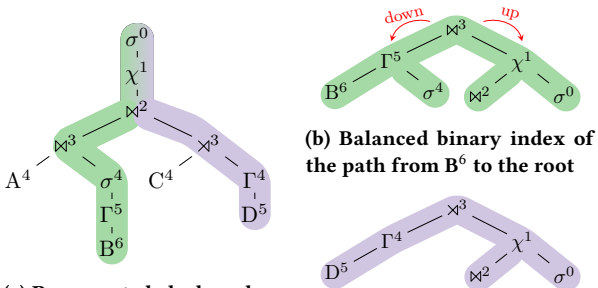
3 INDEXING THE ALGEBRA

To make query optimization analysis with path traversal efficient, Indexed Algebra uses an embedded index structure that makes direct traversal unnecessary. To build this index, we first start with the simpler problem of indexing static algebra trees in this section, before we generalize our Indexed Algebra to support dynamic updates with link/cut trees.

3.1 Simple Tree Indexes

For the path queries that we want to support efficiently, simple path traversals are only suitable for basic optimization. Consider, for example, again predicate push down. Placing predicates that use a single IU requires only traversing a single path through the algebra. However, if the predicate uses multiple IUs, i.e., it is actually a join condition, finding the appropriate place in the tree is more challenging. For this case, we need to determine the operator where all paths from IU sources converge, i.e., we need to find the *lowest common ancestor* (LCA) of the involved IUs.

To answer such queries efficiently, Indexed Algebra builds balanced binary search trees for paths through the relational algebra. These auxiliary search trees are keyed by the distance to the root of the algebra tree. The balance significantly shortens the $O(n)$ path from a base relation to the root. Figure 4 shows an example of such an algebra tree with the distance to the root annotated as



(a) Represented algebra plan

(b) Balanced binary index of the path from B^6 to the root

(c) Index from D^5 to the root

Figure 4: A binary search tree keyed on the distance to the root (annotated in superscript) allows efficient path queries on static algebra trees.

superscript. Consider the $O(n)$ path from σ^0 to B^6, marked on the left side of Figure 4a. Figure 4b shows the $O(\log n)$ balanced index that allows to take a shortcut to traverse the path.

As an example analysis, assume that the predicate σ^0 references a column of B and a column of D. To place this predicate, the query optimizer now needs to answer the path-query to find the LCA of B and D. We can find this join by traversing the search trees of both B and D upwards until we reach a common operator. In our example in Figure 4b and 4c, we traverse to parent nodes until we reach the root node of the auxiliary tree, then follow right up child pointers until we find the subtree of common ancestors rooted at χ^1. Since this subtree contains multiple nodes, but we want to find the lowest common ancestor, we still need to find the lowest operator in this subtree. By following the left down child pointers of χ^1, we find $\bowtie^2$ as the LCA of A and D.

With the help of these indexes, we can now reason efficiently without explicitly traversing the algebra plan, and also answer more complex path queries. Since we only traverse balanced auxiliary plans, the complexity of these improves from linear to logarithmic, and we can find the LCA in $O(\log n)$ [12]. Still, building these indexes is potentially expensive. Since we build an index from each leaf operator to the root, we again end up with quadratic complexity.

3.2 Path Labeling

Another well known technique to index hierarchical data are *labeling schemes*, e.g., pre/post encoding [11] or OrdPath [29]. In OrdPath, we label each node with its path from the root, with a special labeling algorithm that allows insertions without the need to relabel other nodes. Using these path labels, queries are efficient, i.e., we can answer most queries in $O(1)$, and LCA queries in $O(\log n)$.

However, the OrdPath scheme was originally developed for XML queries, which need to preserve document order. For relational algebra, we can relax this and label nodes not with ordinals, but with pointers, which makes accessing nodes referenced in the path more efficient. The fundamental downside of this approach is that relocating a whole subtree requires a relabeling of the whole subtree. As an example, when we push down a predicate from the root of the tree to a table scan, we need to update the OrdPath labels of all operators on this path to remove the predicate. Unfortunately, this means that the complexity for algebra transformations is $O(n)$.

Table 1: Complexity of operations on relational algebra.

Rel. Algebra	Transformation	Traversal
w/o index	$O(1)$	$O(n)$
static index	$O(n)$	$O(\log n)$
path labeling	$O(n)$	$O(1)$
Indexed Algebra	$O(\log n)$	$O(\log n)$

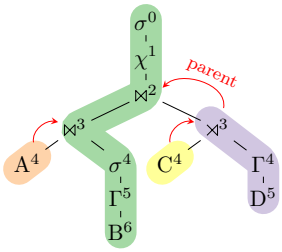


Figure 5: Partial path indexes of Figure 4. Link/cut trees build dynamically balanced splay trees over paths through the algebra and connect subtrees via path-parent pointers.

3.3 Link/Cut Trees

Static algebra indexes have the problem of efficient updates of the indexed algebra. While they allow efficient traversal over the algebra for queries, transforming and building the tree is now significantly more costly than in the raw algebra without an index. We now improve the transformation by using the amortization technique of link/cut trees proposed by Sleator and Tarjan [35, 36]. Table 1 summarizes the complexity of these different approaches to reason about relational algebra. While not indexing allows efficient transformations, traversing the algebra is expensive. Static indexes improve the traversal, but make updates costly. With Indexed Algebra, we achieve both logarithmic updates and traversal.

On a high level, link/cut trees do not maintain balanced indexes for all paths from leafs to the algebra root, but build them dynamically when needed. They do this by maintaining splay trees for (partial) paths, which are connected by path-parent pointers. We show an example in Figure 5, where only the path from B^6 has a complete splay index to the root. When we want to operate on another path, e.g., from D^5, we first need to transform the path-parent parent edge to a balanced edge.

Implementing a link/cut tree is relatively little effort, e.g., a public implementation of Sleator fits in about 100 lines of code [34]. In this implementation, efficient *link* operations connect two trees, and *cut* operations split a subtree from its parent. During all operations, the link/cut tree keeps the accessed path roughly balanced. The operation that enables the simple implementation of the other function is *expose*. Exposing a node brings its path to the root in a form that is suitable for queries, and keeps the nodes balanced enough for amortized logarithmic behavior by organizing them in a self-balancing splay tree [36].

3.4 Efficient Operations using the Link/Cut Tree

Operations on the link/cut tree work similar as with static index trees, except for the additional *expose* operations. In the following,

we describe the core path traversal operations that we use for algebra optimization.

Find Root: Finding the root of an algebra tree is useful to detect if two operators are in the same tree, or if they are separated, e.g., by a common table expression. Additionally, we also use it during join ordering to detect if subtrees are already connected via a transitive join edge. Finding the root requires a path index to the root, i.e., exposing that node. With an exposed operator, its path index is fully connected to the root and roughly balanced. Finding the root of the algebra tree now means following upwards index pointers. Since these are organized via a balanced index, this operation is efficient and reaching the root takes $O(\log n)$ steps.

Lowest Common Ancestor: Finding the lowest common ancestor *lca(A, B)* in the link cut tree differs from static indexes. We first expose A, which connects its path to the root. This path now necessarily also contains the LCA. Then, we also expose B, now connecting B's path to the root. The lowest intersection of these paths then mark the LCA.

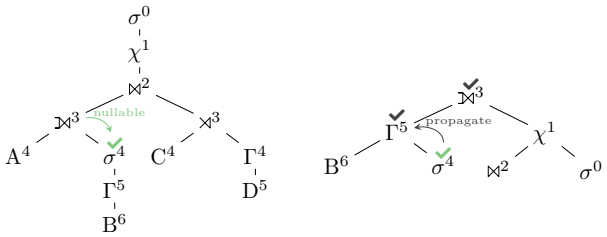
Path Aggregates: Finally, one advanced technique allows to efficiently answer queries about the paths between operators in a tree [15]. The idea here is to maintain the answers for calculations over a path in the index as *path aggregates*, where we calculate the aggregate from left and right index tree aggregates. This adds a constant maintenance overhead to the balancing during expose, but amortizes path queries over expose operations, which allows answering queries about a path of length n in $O(\log n)$ time.

One example of a useful path aggregate is determining if a path between two operators contains an outer join ⟕. Since outer joins make IUs nullable, it depends on if the data flow to a consumer crosses an outer join, if the column is actually nullable or not. Figure 6a shows the already familiar indexing example, where the left join $⟕^3$ now has left outer semantics. This outer join $⟕^3$ now marks its right, potentially nullable, child σ^4, indicating that IUs passing both, the child and the join, can be null due to a missing join partner. In the balanced path index in Figure 6b, we propagate this marker, marking a node when either of its children has a marker. Transitively, the marker at the root of the path index then indicates an outer join somewhere on the path from B^6 to σ^0. With a slight tweak, i.e., storing pointers instead of markers, we can also quickly determine the causing outer join. Similarly, we can also determine the lowest or highest outer join when we preferably propagate the outer join pointer from either the upwards or downwards direction.

With these three base operations, we can implement a plethora of useful optimizations. Since these operations take only logarithmic amortized time, these optimizations are also very effective. In the following, we describe how we apply these operations for query optimization, and how this makes the optimizations effective.

4 APPLICATIONS IN QUERY OPTIMIZATION

Indexed algebra not only enables efficient operations on relational algebra, but also allows elegant formulations of many query optimization techniques. Since our indexes ensure amortized $O(\log n)$ operations, we can formulate many optimizations that consider each operator individually without risking quadratic runtime. In the following, we discuss some query optimization problems, and how Indexed Algebra helps to efficiently apply them. This ties together the connections between operators and expressions of Section 2 with the indexing approach of Section 3. We start with simple, yet effective optimization techniques using path traversals and LCA queries, before we demonstrate the versatility of path aggregates.



(a) Outer joins mark their children as nullable.

(b) Path indexes propagate nullable markers.

Figure 6: Algebra indexes efficiently determine if a path contains an outer join by propagating a marker through the balanced auxiliary tree.

4.1 Determining Join Graph Edges

Join ordering algorithms to find the optimal execution order of joins usually operate on a join graph [28]. To construct this graph, we collect all subtrees (e.g., nested operators like group-by Γ, or base relations) that are connected via joins as graph nodes. In addition, we also collect all join conditions from join ⋈ nodes or selections σ. To determine the edges of this join graph, we need match to match the consumed IUs in the conditions to nodes in the graph. The difficulty here is that IUs in the join condition might have their source arbitrary deep in a nested operator of the graph's leaf nodes.

To avoid building explicit column sets for each node of the join graph, we use the efficient `findRoot` operation of our Indexed Algebra. Algorithm 1 formulates the base construction of this join graph. Note, that the following algorithms rely on the connections between expressions, IUs, and their source and consuming operators, as introduced in Section 2.2.

Algorithm 1: Determining join graph edges with Indexed Algebra.

```
nodes := The joined subtrees
for R ∈ nodes do
    // Split up the join edges between subtrees
    cut(R)
end
// Collect the join edges
edges = ∅
for (IUref(a), IUref(b)) ∈ conditions do
    edges += (findRoot(a.source), findRoot(b.source))
end
...
// After join ordering, rebuild tree with link()
```

In this algorithm, we first `cut` the joined subtrees from the overall algebra tree. This effectively makes each leaf node of the join graph a separate tree with a distinct root node. When we now consider

each condition, we identify each consumed IU of that expression. From these IUs, we can determine its source operator, which needs to be in one of the subtrees that make up the nodes of our join graph. The `findRoot()` operation now finds precisely these nodes. Subsequently, for each pair of referenced IUs within the conditions, we can add a join edge between the nodes we find this way.

During this join graph construction, the Indexed Algebra is implicitly maintained by `link`, `cut`, and `findRoot` operations. This allows us to efficiently constructs the join graph, even for complex input subtrees.

4.2 Detecting Dependent Joins

As already discussed in Section 1, detecting and eliminating dependent joins is one of the most important query optimization steps. Our algorithm to efficiently construct the join graph also relied on the absence of dependent join attributes. However, recognizing dependent joins is not trivial and needs some data flow analysis.

To recognize dependent IU references, we need to detect the situation where a consumed IU's source is not in the input relations of the operator containing that reference (e.g., the A.z in Figure 1). A possible implementation to detect these IUs would be to temporarily `cut` each operator consuming an IU and determine, if the root of the subtree containing the IU source is the consuming operator. This gives us a binary check if that reference is dependent, but we still do not know, which join to transform to eliminate the dependency.

Algorithm 2: Recognizing dependent joins.

```
for IUref(a) ∈ query do
    lca ← findLCA(IUref.operator, a.source)
    if lca ≠ IUref.operator then
        mark lca as dependent join
end
```

To simultaneously recognize the dependent join, we implement our dependency detection slightly differently. Algorithm 2 shows our implementation using Indexed Algebra and LCA operations. We again check all IU references, but now find the LCA of the IU's source operator and the reference's containing operator. For regular references, the reference is in an ancestor operator of the source, and thus equal to the LCA. Otherwise, the reference is dependent. Here, not only detects the dependent reference, but also directly determine the dependent join.

To efficiently process the query and make subsequent optimizations simpler, we eliminate all correlations as one of the first optimizations. To unnest this query, we use the transformation rules presented in earlier work [27], eliminating the need for expensive nested loop joins. In this optimization, Indexed Algebra again helps to skip large unrelated parts of the query that are in between IU source and its references.

4.3 Tracking IU Nullability

For query optimizers, *null* columns are especially unpleasant to deal with [23]. In our system, we therefore try to eliminate null values already in base table scans, where we can filter many tuples all at once [16]. When we can prove that an IU is not null, subsequent

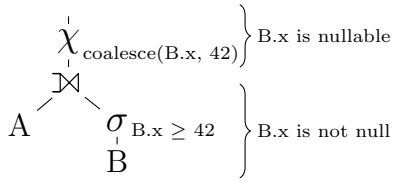


Figure 7: The nullability of IUs can depend on outer joins and their position in the algebra.

operations (e.g., a join condition) can efficiently compare values and ignore null bits. However, outer joins complicate this optimization significantly, since they can conditionally produce null values.

Figure 7 shows an example of a query with an outer join ⟕ between base relations A and B, where B can become null after the join. Below the join, we have a predicate σ that filters on a condition of B.x, which allows us to filter null values already at the scan of B. Subsequently, we can skip the null check for the predicate and any subsequent reference of that IU, and also optimize expressions based on the assumption that the IU is not null. Unfortunately, a naïve application would incorrectly transform the upper coalesce expression. Instead, we need to analyze the query further and prove the absence of outer joins.

Algorithm 3: Determine if the path between B and χ contains an outer join that makes B's columns nullable.

```
// Temporarly cut to limit the path B →* χ
cut(χ)
// Calculate the aggregate for B
expose(B)
result ← B.nullable marker
// Restore the full path index
link(χ, χ.parent)
```

To analyze this situation, we use Indexed Algebra's efficient path aggregates (cf. Figure 6), as shown in Algorithm 3. One catch here is, that aggregates by default propagate through the whole preferred-path. This means that aggregates for non-root algebra nodes do not directly contain the analysis result we are interested in, i.e., if the path from B to χ contains an outer join. Instead, we temporarily *cut* the χ subtree from the overall algebra tree, so that the operator of the reference becomes a temporary root node. Afterwards, we *expose* the source operator B to propagate our aggregate through that path and get the answer. Since all three operations, cut, expose, and linking the temporary cut again, are $O(\log n)$, these operations are surprisingly efficient.

4.4 Predicate Pushdown

Predicate pushdown is a ubiquitous optimization that needs data flow analysis. In a simple form, we push a predicate σ on the path towards the source of its referenced IUs. In a more complex optimization, we also want to capture transitive predicates that are conjunctive. So, when we encounter a join, we want to infer new predicates, e.g., $\sigma_{x=42}(\bowtie_{x<y}) \implies \sigma_{42<y}$.

A problem with the operator-by-operator detection of transitive edges is that views or common table expressions can also introduce

predicates in arbitrary deep subtrees. To infer predicates from all join edges, even above the initial predicate, the optimizer would need to first bubble predicates up, then push them down again. Transitively, this can transform join edges ($x < y$) to predicates that can be pushed further down ($42 < y$). To infer all transitive edges, we need to iteratively bubble up and push down all predicates until we find no new transitive predicates. Since transitive chains can be arbitrary long, traversing tree operator-by-operator for each step is inefficient. Indexed Algebra instead uses two distinct techniques to propagate transitive predicates:

- Predicate push down without considering transitivity
- Upward constant propagation generating new predicates

For the push down, we use the path-centric logic from Figure 3 that skips intermediary operators. With Indexed Algebra, we can avoid directly traversing the algebra by finding the LCA of the referenced IUs in the predicate. Certain operators, e.g., outer joins ⟕, require more logic, so we need a way to recognize these in the path. To detect these operators, we, again, use path aggregates that signal the presence of these operators on paths through the algebra. With these aggregates, we can effectively skip unrelated operators and push any predicate in $O(\log n)$ to the next interesting operator. With these pushed down predicates, we can now infer additional constants that we can propagate upwards.

4.5 Propagating Constants

Our predicate push down ensured that IUs are filtered as early as possible, ideally at base relations. After evaluating this predicate, we know that it holds in any downstream operator, which allows propagating it transitively. For some predicates like <, we can infer additional bounds for other comparisons, but for equality comparisons, we directly replace the IU references with constants.

Figure 8: SQL query with nested constants. We propagate and fold constants from inner queries to all uses to enable transitive pushdown.

```
with years_orders as
  (select * from orders where o_year = 2022)
select * from deliveries d, years_orders o
where d.order_id = o.id and d.d_year <= o.o_year + 1
```

For example, consider the SQL query shown in Figure 8. This query shows a factored subquery with the nested equality predicate $\sigma_{\text{year}=2022}$. Constant propagation would now replace all references of the year, i.e., the produced IU of both the orders subquery and the complete query, with the constant value. Then, we fold the resulting expression, which results in a simple predicate that we can push down to the deliveries table. Note that this is not limited to just constants, but this can be generalized to arbitrary predicates. For our example, the inferred bounds of `o_year` can be propagated again using the same technique, and we can deduce that any other comparisons must also maintain similar bounds.

A core assumption we have for constant propagation is that IUs have the same value during execution. As in the last sections, outer joins ⟕ are the exception to this rule. When there is an outer join on the path between IU and its reference, we cannot directly propagate a constant to that reference. Constant propagation uses the familiar path aggregates that track tuple nullability from Section 4.3 to detect this case any only propagate predicates when no outer join can introduce nulls.

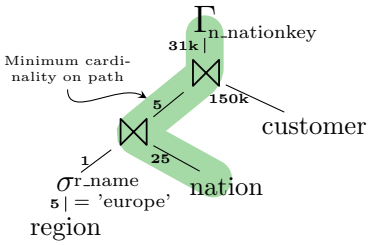


Figure 9: The cardinality of the top aggregation Γ depends on the distinct values reaching it from the base table. Indexed Algebra maintains a path aggregate to efficiently maintain this information.

In combination with predicate pushdown, the constant propagation transitively introduces predicates through the whole query. Again, Indexed Algebra ensures that the path traversals are cheap, and we can efficiently reach all relevant IUs.

4.6 Bounding Distinct Values Estimates

So far, we mainly discussed query optimization rules that are almost always beneficial. For cost based optimizations, e.g., join ordering, it is essential to have good cardinality estimations [6, 17]. By making cardinality estimations path sensitive with Indexed Algebra, we can improve the estimation bounds.

Cardinality estimation often uses distinct value counts, e.g., to estimate the size of aggregations that eliminate duplicates from keys. Usually, base table estimates derive these counts by calculating statistics, e.g., in the form of HyperLogLog sketches [7]. However, when we leave base tables and process predicates or joins, changes in cardinality do not translate easily to changes in distinct values. Due to these limitations, many systems just use base table estimates, which gives suboptimal estimates compared to a more precise tracking of distinct values through the algebra.

Figure 9 shows a simplified query on the TPC-H schema, where we can get better estimates by considering the path from source to root. Consider the marked path from nation to the group by Γ where we first have a filtering join with region, then a growing join with customers. If we just consider the distinct values at the base table (25), we loose the information about the filtering intermediate join. Inspecting the whole path between the operators to find the minimum path cardinality (5) captures a more precise upper bound for the distinct values.

To calculate this minimum path cardinality efficiently, we again use path aggregates. The path aggregates maintain the minimum cardinality of the path to the root by selecting the minimum aggregate from `lcUp` and `lcDown`. To estimate the minimum path cardinality to an arbitrary operator, we *cut* its subtree and *expose* the source operator. In our example, we cut the parent of the aggregate Γ to make it the root, and expose the nation table scan to ensure its path to the root is preferred. Then, we get the min

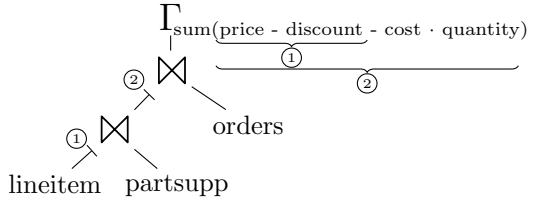


Figure 10: Materialization points allow evaluating expressions that reduce the size of the materialized data. We use Indexed Algebra's LCA and path operations to efficiently find suitable materializations points.

cardinality of 5, which gives us a precise estimate without the need to traverse the operator tree.

4.7 Placing Expression Evaluation

We also optimize the amount of data that we store in intermediary operators that need to materialize. These *pipeline breakers* are costly, since they break the data centric execution and allocate memory to store tuples that are read later. The most common pipeline breaker is the build side of a hash join that stores tuples in a hash table. Reducing the size of the tuples stored in this materialized state is advantageous, since it also increases cache locality.

For this optimization, we evaluate expressions at materialization points when this leads to a smaller materialized state. In this state, we need to store any produced IUs of the input that might be consumed by subsequent expressions. By evaluating expressions before this materialization, we can reduce the state, which reduces the memory consumption, and the overall cost of the query execution. While evaluating expression might incur some overhead, a smaller materialized state is usually more beneficial.

As an example, consider the algebra tree shown in Figure 10 that approximately follows the TPC-H schema. We explicitly annotate the pipeline breaking build side of the joins with $\top$, which gives us two locations to evaluate expressions. We can evaluate the first subexpression `price - discount` at the lower build side ①, which allows us to materialize the single result value instead of the two partial values. At the second hash table build ②, we have all required IUs to evaluate the whole expression. Thus, we can avoid materializing its four referenced IUs, and instead only store the single result value.

To reorder the evaluation of an expression, we need to consider its constraints in the algebra tree. We can only evaluate it when all its referenced IUs are available, which we can find by computing the LCA of the producing operators. Then, we can place it anywhere between the LCA position and its actual usage, at a suitable pipeline breaker. To also efficiently find pipeline breakers, we implement an aggregated property of Indexed Algebra, similar to the markers of Section 4.4. This way, we can efficiently find a suitable evaluation place for each subexpression of a complex expression that reduces the overall materialized state.

In summary, we presented several optimizations that benefit from Indexed Algebra. By using path-centric optimization techniques, we obtain concise and efficient algorithms.

5 BEYOND INDEXED ALGEBRA

Indexed Algebra allows to efficiently answer data flow questions that arise during query optimization. In the following, we discuss further implementation techniques that are less related to data flow.

5.1 Complex Expressions

Expressions in the relational algebra are often a significant challenge, and many systems optimize them, e.g., by compiling them to efficient machine code [30]. In practice, expressions can be very large, which makes optimizing and compiling them non-trivial. For example, we observed queries with a predicate of several thousand disjunctions. A naïve approach to such queries can result in large optimization states that exceed machine limits.

One problem with such large expressions is that recursive algorithms analyzing it reach stack-size limitations [25]. As with relational operators, we organize expressions in a tree structure that unfortunately is not balanced. This means that its depth can scale linearly with the size of the input expression. However, the stack space for recursive calls is limited and usually not very large.

We implement a technique to avoid the stack limits by recognizing large stack depths before an actual overflow. To recognize this situation, we can inspect the stack pointer, i.e., `rbp` on x86, and abort the query. This way, we avoid crashing the DBMS, but we degrade its usefulness, since now users need to work around the system's limitations. When we exceed the stack size in Umbra, we instead switch to a different stack, which allows processing such queries, albeit with some overhead during optimization. Note that since Umbra compiles queries, this is a one-time instead of a per-tuple overhead.

We also address the underlying issue of the large, deeply nested structure of expressions. One problem we identify is the representation of such predicates as *binary* boolean expressions. Instead, we increase the fan-out of our boolean expressions by representing they as *Nary* expressions, which, e.g., results in a single disjunctive expression with arbitrary many boolean inputs. In practice, this means that we get shallow expression trees, where optimizing scales nicely even for large input predicates. Note that this does not avoid any deeply nested expression, but we found that this significantly improves the common case.

Having such large expressions in mind, we also eagerly fold constants. Folding constants early not only reduces the size of the expression trees, but also reduces the load of subsequent code generation. In certain cases, compilers like LLVM use super-linear algorithms to compile code [24], which becomes painful when dealing with large generated expressions.

5.2 Lazy Property Evaluation

For individual operators, some properties are especially expensive to evaluate. For these properties, we only want to calculate them once we need them, and when they are unlikely to change again. In our implementation, we identified the most expensive properties:

- The estimated cardinality
- The functional dependencies

For cardinality estimation, loading data structures for statistics and evaluating predicates on samples of a base table is relatively expensive. Ideally, we only want to estimate base table cardinalities

once, when we pushed down all predicates that we can evaluate on that table. If we push down additional predicates after this cardinality estimation, we would need to invalidate previous estimations, which would cause duplicate work. Thus, we should lazily estimate cardinalities on demand after predicate push down.

However, we potentially access the cardinality multiple times during join ordering, where we treat filtered base tables as leaf nodes of the join graph. To estimate the cardinality only once, we additionally cache it for each operator. This unfortunately has the downside that we need cache invalidation logic when we alter operators in a way that affects the cardinality.

Similarly, calculating and maintaining functional dependencies between IUs can enable query optimizations, such as minimizing group-by keys. However, computing functional dependencies and equivalent IUs is relatively expensive, when only a handful of optimizations actually require functional dependencies. This lazy evaluation is mainly useful with complex expressions. For example in TPC-H, we see an overall improvement of 6%. However, the impact is larger for queries with complex predicates, e.g., TPC-H Q19, where we get a 23% improvement by avoiding unnecessary work.

5.3 DAG Structured Algebra

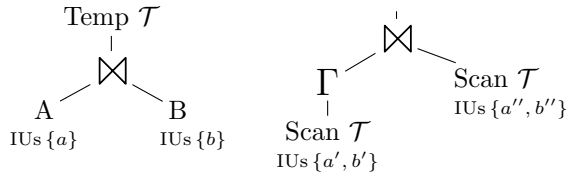
So far, we only looked at tree structured algebra plans, where each operator has a single parent. However, relational algebra can also have multiple outputs in the form of *common table expressions* (CTEs, i.e., `WITH` clauses or views in SQL). Additionally, we also want to be able to share intermediate results for push-based execution. Therefore, we introduce non-tree edges, which makes our algebra DAG structured in the general case.

For regular queries, we expect DAG edges to be significantly fewer than the tree edges of our regular algebra. To minimize the DAG edges, we inline shared table scans to avoid unnecessary intermediate materialization, which makes them part of the algebra tree. CTEs can also be read multiple times, but a simple inlining transformation can similarly transform these DAG edges. Inlining is not always advisable, but helps to transform DAG to tree edges if a CTE is only read once.

To support DAG edges, it first seems that analyzing data flows crossing DAG edges complicates the reasoning over dependencies. When an IU crosses such a DAG edge, its data now takes two paths through the execution plan. However, to avoid ambiguity in the subsequent query, we also need to have distinct names for these IUs. Renaming is also a pragmatic solution for the data flow analysis. Within an algebra tree, we can use Indexed Algebra for efficient reasoning, but when the data flow crosses a DAG edge, we also switch the IU we reason about.

In our implementation, we use a special operator to cross DAG edges that share an input node. This parent operator of a shared node creates new IUs that are distinct from the input IUs. We call this operator a *PipelineBreakerScan* (PBS) that maintains the shared input in a referenced-counted state and explicitly maps from IUs in its input to IUs in its output. The PBS also generalizes over its input, which can be an arbitrary operator that can be scanned multiple times, which are usually pipeline breakers.

Figure 11 shows an example query plan that contains DAG edges. The scanned input in Figure 11a joins two base relations A and B, and materializes them in a Temp operator. Crossing the DAG edges, Figure 11b has two PBSs that rename all inputs to two sets of distinct IUs. The scanned output still references the IUs a and b, but we take special care to consider them part of the scanned Temp operator and not part of the disconnected output. This way, data flow questions reasoning about a are contained in one tree, while a' refers to the same data across the DAG edge.



(a) Shared input **(b) Scanning output**

Figure 11: A DAG structured query plan. Operators can be referenced multiple times, but need to rename their IUs to avoid ambiguity.

As a result, we generalize our algebra to DAG structured queries with the simple renaming abstraction. The resulting overall algebra now has DAG edges between algebra trees. Within these trees, we use Indexed Algebra that allows efficient reasoning. Thus, we can lift all techniques discussed before from trees to DAGs.

6 EVALUATION

We now evaluate the impact of our work on the performance of the query optimization engine in our research RDBMS Umbra. We compare our implementation of Indexed Algebra to our implementation using path traversal, and additionally evaluate an approach using column sets. We start with an evaluation that shows the asymptotic improvements for large queries, before we show the impact on popular benchmarks.

Our query optimizer produces state-of-the-art query plans that do not differ between any of the presented analysis approaches. To show the quality of our produced plans, we provide an online interactive query plan viewer for the evaluated benchmarks*.

Setup of Performance Measurements: We run all measurements on a system with an Intel Xeon W-2145 CPU with 8 cores, 2× hyper-threads, and 32 GB RAM. Since we measure the relatively small amount of optimization time in the benchmarks, we ensure consistent results by repeating every measurement 1000 times and reporting the average time. Compared to query execution, the small optimization time seem negligible, but as in most systems, query planning in Umbra is single threaded and every millisecond in query planning blocks potentially hundreds of cores for parallel query execution.

As benchmarks, we use TPC-H [4], TPC-DS [21], and the Join Order Benchmark (JOB) [17]. The complexity of the queries varies significantly between the benchmarks, with TPC-H having the least complex queries. In our implementation, the TPC-H queries involve on average 9 and a maximum of 20 operators, where JOB has an average of 18 and a maximum of 36, while TPC-DS is the most complex with an average of 20 and a maximum of 71 operators.

*https://umbra-db.com/interface/

Table 2: Impact of Indexed Algebra on TPC-DS optimization.

Optimization Pass	Avg. Time [µs]		Speedup
	Column Sets	Indexed Algebra	
Simplify Expressions	10.5	11.0	0.95
Unnesting	78.8	10.9	7.26
Predicate Pushdown	66.9	58.8	1.18
Cardinality Estimation	96.9	97.4	1.00
Join Ordering	63.5	65.7	0.97
Physical Planning	20.7	23.1	0.91
Total	339.9	269.5	1.28

Reports from industry, however, feature orders of magnitude more complex queries. SAP [5, 19] for example reports that their core data services contain over 100 views that reference more than 100 tables, with the largest view referencing over 4000 tables. Similarly, we hear reports from Tableau [39] and VMware [37] about auto generated queries that are dozens of pages of SQL.

To test the asymptotic optimization runtime for such large queries, we cannot compare queries with a fixed amount of operators. Instead, we use a synthetic workload, where we gradually increase the involved operators. In the following, we use a synthetic join workload that is inspired by a SQLite's sqllogictest[†]. In this workload, we gradually increase the number of involved base relations, which allows simulating large algebra trees.

In addition, query optimization has many parts that are unaffected by the size of the algebra. Table 2 shows a break-down of the average optimization times over Umbra's optimization passes in TPC-DS. For most passes, Indexed Algebra has no significant benefit, since these passes do not reason about the algebra itself, but use mostly operator local information, e.g., for constant folding during expression simplification. For these passes, Indexed Algebra adds some overhead to maintain the indexes, which only become relevant for the algebra centric optimizations. For these optimizations, unnesting and predicate pushdown, Indexed Algebra has the biggest impact. In the following, we concentrate on unnesting, since that is the optimization that sees the biggest improvement of using an index, and, since we need to check each IU reference if it is correlated, also scales with the size of the algebra tree.

6.1 Efficiency on Query Complexity

In a first experiment, we evaluate the impact of query complexity on optimization runtime. However, the implementation of the traditional operator-centric optimizations using column sets differs significantly from our proposed path-centric optimization (cf. Figure 3). For this evaluation, we implement all optimizations with the path-centric optimizations, and only calculate column sets once for the unnesting logic of Section 4.2. For path-centric optimization, we compare a naïve path traversal with Indexed Algebra.

For this experiment, we expect Indexed Algebra to have an advantage growing with the query complexity. Path traversal and column sets both have quadratic behavior, while Indexed Algebra runs in $O(n \log n)$. Between column sets and path traversal, we expect no dramatic difference.

[†]https://www.sqlite.org/sqllogictest/

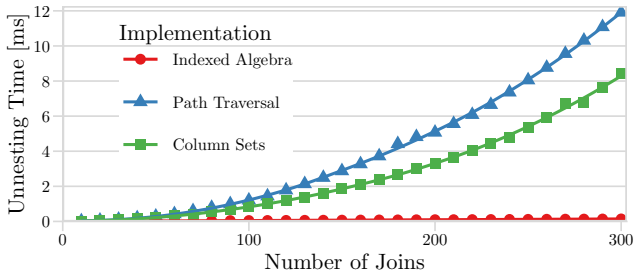


Figure 12: Query optimization time of synthetic join querys with many relations. Indexed Algebra has asymptotically better runtime for large queries.

Figure 12 shows the results measuring the time to optimize queries of increasing complexity. As a first micro benchmark, we only consider the time to execute the optimization to decorrelate any nested expressions. On the x-axis, the figure shows the increasing number of joins between 10 and 300 relations. As expected, the two traditional implementation approaches, path traversal and column sets, scale badly with them taking several milliseconds to perform the single optimization pass.

With increasing query complexity, the two traditional approaches show clear super linear execution time. For ten relations, path traversal takes 20 µs, while Indexed algebra is 4× faster and only takes 5 µs. With 300 relations, path traversal takes over 12 ms, where Indexed Algebra only takes 0.14 ms, over 85× faster. This means that using Indexing algebra, we can cope with very large queries. We tested even larger join sizes with 1k and 10k joins, where Umbra takes 0.16 and 13 seconds to optimize the query. For such large joins, join reordering becomes a bottleneck, where, e.g., join linearization shows quadratic runtime [28].

This shows the advantage of Indexed Algebra over the traditional approaches for complex queries. While this is not as pronounced for smaller queries, it is still a significant advantage there.

6.2 Benchmarks

While we saw a definitive improvement for synthetic joins, as a workload, it is rather simplistic. The synthetic workload only has base relations and join operators, where the benchmark queries of TPC-H, TPC-DS and JOB better capture the real world applications that also contain business logic in aggregates and more complex expressions. For this experiment, we continue to measure unnesting time, which applies to all operators and expressions since the SQL standard allows correlated attributes at almost any point of a query.

In this experiment, we expect fewer gains than with the complex synthetic queries. Since the queries in this experiment contain significantly fewer operators on average, the quadratic behavior is not as dramatic. However, as we already saw in the last benchmark, Indexed algebra should still be several times faster.

Figure 13 shows a box plot of the results, which roughly follow our expectations. In TPC-H and JOB, Indexed Algebra is more than 4× faster than using column sets, while it is on average 7× faster in TPC-DS. In addition, Indexed Algebra significantly improves the situation for the outliers: Unnesting TPC-DS Q64 takes over 350 µs with column sets, while Indexed Algebra only takes about

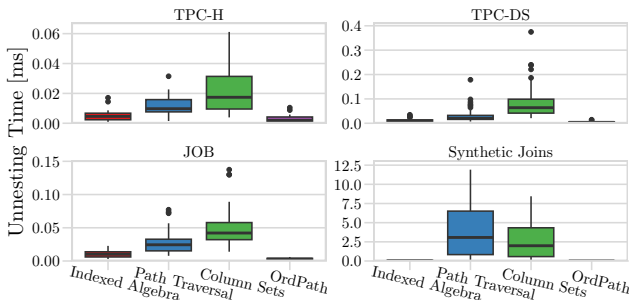


Figure 13: Query optimization time of various benchmarks.

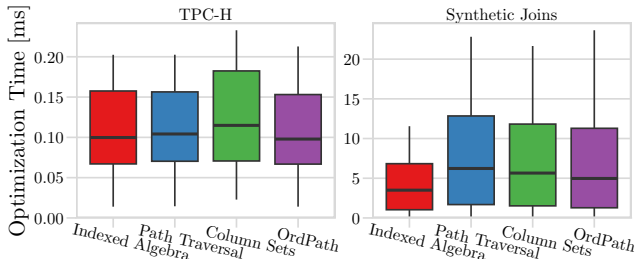


Figure 14: Comparison of total optimization times.

23 µs. Similarly, TPC-H Q8 takes over 80 µs with column sets, where Indexed Algebra takes 8.5 µs.

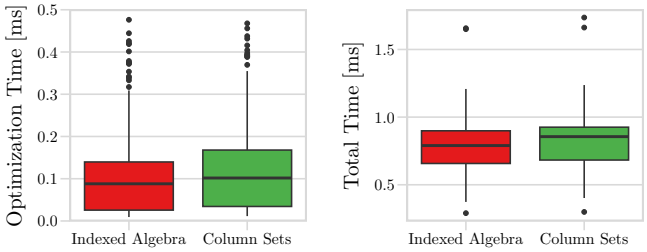
In comparison to the synthetic queries, the performance of path traversal and column sets is unexpected. On average, path traversal is faster for this workload of real queries. We suspect that this might be caused by memory allocation: The relations in real-world queries have many more columns than in the synthetic workload, which results in large dynamically allocated sets. Since these sets also scale quadratic with the query size, the ballooning memory results in poor cache locality, which path traversal avoids.

In contrast, OrdPath seems to be the best implementation, when just considering unnesting. However, unnesting mostly reads the query tree and favors optimization strategies that allow cheap path queries. When we also consider the total optimization time that includes transformations, this picture changes: Figure 14 shows the total optimization time, including query transformations. For the small queries in TPC-H, OrdPath is competitive with Indexed Algebra, but for larger synthetic join queries the asymptotically worse updates to OrdPath's path labels become costly.

To summarize, Index Algebra not only has sizable improvements for huge queries, but also significantly improves optimization of relatively small real world queries. In the following, we investigate how well the improvements of this specific task translate to the complete query optimization process.

6.3 Interactive Workloads

Benchmarks capture a narrow use-case with mostly static queries. Query optimization is more challenging for workbooks, which are popular tools for complex, interactive data analytics [40]. The data these workbooks run on is typically relatively small, but the queries



(a) Optimization time for over 1 000 queries from Tableau Public workbooks.

(b) Total end-to-end time for a comparably sized TPC-H workload.

Figure 15: Evaluation of interactive workloads.

can be quite complex. For the following evaluation, we use real-world queries from Tableau Public, which we convert to standard SQL and CSV files[‡].

For our evaluation of interactive workloads in Figure 15a, we consider over 1 000 queries from nine complex Tableau Public workbooks. The data queried in these workbooks is relatively small, and consequently, the median execution time for these queries is relatively short with 230 µs. In contrast, the median optimization time of these queries with Umbra's traditional column set implementation 105 µs. Using Indexed Algebra brings the median total optimization time down to 89 µs, which is an 18% speedup on real world queries.

As an additional data point on small data, we use TPC-H on a small scale factor 0.01. The small scale factor captures the optimization challenges of the interactive nature of such workbooks, and still captures the data size of the 75th percentile of Tableau Public workbooks [39]. In this configuration, the TPC-H queries have a median execution time of 369 µs. The median total processing time for column sets is 856 µs, using Indexed Algebra reduces this to 790 µs. Figure 15b shows this as a box plot. In total, Indexed Algebra can reduce the end-to-end latency of this workload by 8%.

6.4 Overall Results

Query optimization comprises many optimizations, where the unnesting is only one partial optimization. Many other analyses in other optimization passes can be similarly costly, but are not as dependent on the query structure as unnesting, which diminishes the improvement of Indexed Algebra. As discussed in Section 5.2, cardinality estimation using sample evaluation is expensive, and unaffected by Indexed Algebra. To quantify the overall improvement, we measure the speedup of using Indexed Algebra over the total query optimization time.

Over all optimizations, we expect less speedup than we saw for query unnesting in the last sections. Still, many optimizations besides unnesting also depend on the query structure, so we should still measurable a significant improvement with Indexed Algebra. Especially for complex queries, where the quadratic scaling of traditional methods has the most impact, we expect good improvements.

Figure 16 shows the aggregated speedup of query optimization time over the measured benchmarks. As expected, the speedup is less than for the specific optimization of the last measurements, but

[‡]https://github.com/tum-db/tableaupublic

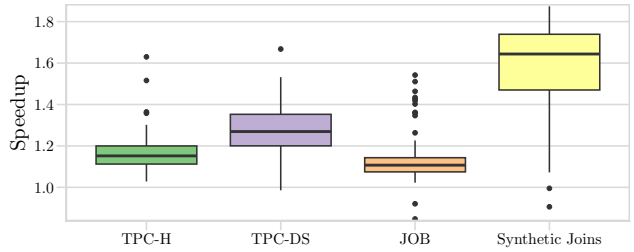


Figure 16: Improvements of total query optimization time. We compare the time to optimize queries using Indexed Algebra in contrast to column sets. The average improvements are: 12% for TPC-H, 29% for TPC-DS, and 10% for JOB.

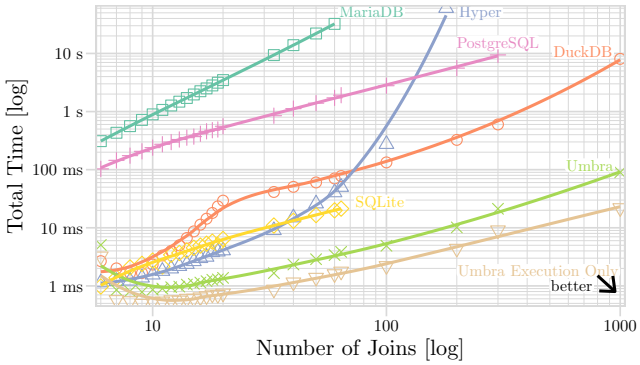


Figure 17: Total time spent processing synthetic join querys with many relations. Umbra uses Indexed Algebra to efficiently optimize large queries.

we still see some significant speedups for outliers, e.g., TPC-H Q8 and TPC-DS Q64, which improve by over 50%.

We conclude with a systems comparison of the total processing time for synthetic join queries in Figure 17. This experiment shows MariaDB 10.9.4, DuckDB 0.6.1, PostgreSQL 14.6, Hyper 0.0.16377, and SQLite 3.40.1 over an average of three runs for join sizes of up to 1000 joins. As workload, we adapted the synthetic joins that we already used in the last experiments, but with base tables and a result of 1000 tuples so that the query execution is not trivial. We measure all systems with an increasing number of joins, until we hit the limits of the systems, e.g., a parse error for SQLite, or excessive processing time. While the excessive runtime could have multiple causes, inefficient planning or execution, Umbra still achieves subsecond processing times even for more than a thousand joins using Indexed Algebra.

7 RELATED WORK

After the classical approach in System R [31], Goetz Graefe pioneered the implementation of optimizations on relational algebra with the EXODUS [10], Volcano [8], and Cascades [9] systems. Modern optimizers like Calcite [3] or Orca [37] still use the same concepts. These systems all rely on operator centric optimizations that transform the plan with predefined rules. Indexed Algebra works on path-centric algorithms instead, which allows more efficient plan transformations.

A newer development is the development of advanced query compilers. Query compilers nowadays build upon data centric code generation [22, 32], which translates query plans into an intermediate language that a compiler like LLVM can optimize and transform to machine code. Subsequent work in this area advanced the used intermediate representations (IR) to fit the needs of query processing systems [13, 14, 33, 38]. Indexed Algebra optimizes the logical plan from a high level, where IRs focus on the lowering to machine code for the physical query plan. This also allows powerful inter-operator optimizations such as operator fusion. However, a series of operators forming a pipeline become a function or loop that is a boundary where imperative compilers cannot easily optimize. In contrast, Indexed Algebra specializes for data flow questions inherent to query languages, where we can introduce optimizations across pipelines. In summary, we see these approaches as complementary: IRs allow powerful low-level optimizations on individual expressions, while Indexed Algebra optimizes the high-level plan.

Another related work is TreeToaster [2], which builds efficient pattern matching based on the incremental view maintenance engine DBToaster [1]. TreeToaster recognizes that pattern matching is a bottleneck, and makes pattern matching on dynamic algebra trees efficient. In contrast, our work reengineers the pattern matching to operate on paths instead of individual operators.

8 CONCLUSION

In this paper we introduced Indexed Algebra as an efficient solution to optimize relational algebra. Traditional techniques to query optimization did not scale for complex queries, often showing quadratic runtime with increasing operators. While complex queries previously took a long time to optimize, our technique helps to reduce the average optimization time and makes processing the most complex queries viable.

Indexed Algebra tames the quadratic complexity of query optimization by building an index structure of the data flow paths though the query. In combination with our proposed path-centric query optimization, this reduces the time spent optimizing. With Indexed Algebra, the runtime of both queries and transformations of the algebra is logarithmic in the number of operators. In total, this query optimization can implement optimizations looking at all operators in $O(n \log n)$.

Furthermore, we have shown that path-centric query optimization not only allows efficient, but also expressive implementation of query optimization. For path-centric optimization, especially the least common ancestor operation is a convenient way to directly find interesting points where the data flow intersects. This approach also does not require additional data structures that we would need to maintain and update, which significantly reduces the implementation effort. In effect, this allows Indexed Algebra to offer the best of both worlds, efficiency and ease of use.

Even for moderately complex queries as we find in TPC-H, Indexed Algebra improves the total optimization time by up to 1.8×. Due to the asymptotically better runtime, complex queries show an even larger improvement. For such complex queries, Indexed Algebra allows to efficiently optimize queries that previously were considered impossible to optimize.

REFERENCES

[1] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. 2012. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *Proc. VLDB Endow.* 5, 10 (2012), 968–979.

[2] Darshana Balakrishnan, Carl Nuessle, Oliver Kennedy, and Lukasz Ziarek. 2021. TreeToaster: Towards an IVM-Optimized Compiler. In *SIGMOD Conference*. ACM, 155–167.

[3] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD*. ACM, 221–230.

[4] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC (Lecture Notes in Computer Science)*, Vol. 8391. Springer, 61–76.

[5] Nicolas Dieu, Adrian Dragusanu, Françoise Fabret, François Llirbat, and Eric Simon. 2009. 1,000 Tables Under the From. *Proc. VLDB Endow.* 2, 2 (2009), 1450–1461.

[6] Philipp Fent, Altan Birler, and Thomas Neumann. 2022. Practical planning and execution of groupjoin and nested aggregates. *VLDB J.* (2022).

[7] Michael J. Freitag and Thomas Neumann. 2019. Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates. In *CIDR*. www.cidrdb.org.

[8] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Trans. Knowl. Data Eng.* 6, 1 (1994), 120–135.

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

[10] Goetz Graefe and David J. DeWitt. 1987. The EXODUS Optimizer Generator. In *SIGMOD*. ACM Press, 160–172.

[11] Torsten Grust. 2002. Accelerating XPath location steps. In *SIGMOD Conference*. ACM, 109–120.

[12] Dov Harel and Robert Endre Tarjan. 1984. Fast Algorithms for Finding Nearest Common Ancestors. *SIAM J. Comput.* 13, 2 (1984), 338–355.

[13] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an Open Framework for Query Optimization and Compilation. *Proc. VLDB Endow.* 15, 11 (2022), 2389–2401.

[14] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905.

[15] Philip N. Klein and Shay Mozes. 2021. Optimization Algorithms for Planar Graphs. https://planarity.org.

[16] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. ACM, 311–326.

[17] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.

[18] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD*. ACM, 2530–2542.

[19] Norman May, Alexander Böhm, and Wolfgang Lehner. 2017. SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads. In *BTW (LNI)*, Vol. P-265. GI, 545–563.

[20] Guido Moerkotte. 2020. Building Query Compilers. http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf.

[21] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *VLDB*. ACM, 1049–1058.

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

[23] Thomas Neumann. 2018. Reasoning in the Presence of NULLs. In *ICDE*. IEEE Computer Society, 1682–1683.

[24] Thomas Neumann. 2020. Linear Time Liveness Analysis. https://databasearchitects.blogspot.com/2020/04/linear-time-liveness-analysis.html.

[25] Thomas Neumann. 2020. Taming Deep Recursion. https://databasearchitects.blogspot.com/2020/11/taming-deep-recursion.html.

[26] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[27] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *BTW (LNI)*, Vol. P-241. GI, 383–402.

[28] Thomas Neumann and Bernhard Radke. 2018. Adaptive Optimization of Very Large Join Queries. In *SIGMOD*. ACM, 677–692.

[29] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. 2004. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD Conference*. ACM, 903–908.

[30] Ravindra Pindikura. 2018. Gandiva Initiative: Improving SQL Performance by 70x. https://www.dremio.com/gandiva-performance-improvements-production-query/.

[31] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD Conference*. ACM, 23–34.

[32] Hesam Shahrokhi and Amir Shaikhha. 2023. Building a Compiled Query Engine in Python. In *CC*. ACM, 180–190.

[33] Amir Shaikhha, Yannis Klonatos, Lionel Parreaux, Lewis Brown, Mohammad Dashti, and Christoph Koch. 2016. How to Architect a Query Compiler. In *SIGMOD*. ACM, 1907–1922.

[34] Daniel Dominic Sleator. 2011. Submission #860934 - Codeforces. https://codeforces.com/contest/117/submission/860934.

[35] Daniel Dominic Sleator and Robert Endre Tarjan. 1983. A Data Structure for Dynamic Trees. *J. Comput. Syst. Sci.* 26, 3 (1983), 362–391.

[36] Daniel Dominic Sleator and Robert Endre Tarjan. 1985. Self-Adjusting Binary Search Trees. *J. ACM* 32, 3 (1985), 652–686.

[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*. ACM, 337–348.

[38] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *SIGMOD Conference*. ACM, 307–322.

[39] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD*. ACM, 1:1–1:6.

[40] Adrian Vogelsgesang, Tobias Mühlbauer, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Domain Query Optimization: Adapting the General-Purpose Database System Hyper for Tableau Workloads. In *BTW (LNI)*, Vol. P-289. Gesellschaft für Informatik, Bonn, 313–333.