Galley: Modern Query Optimization for Sparse Tensor Programs

KYLE DEEDS, University of Washington, United States
WILLOW AHRENS, Massachusetts Institute of Technology, United States
MAGDA BALAZINSKA, University of Washington, United States
DAN SUCIU, University of Washington, United States

The tensor programming abstraction is a foundational paradigm which allows users to write high performance programs via a high-level imperative interface. Recent work on *sparse tensor compilers* has extended this paradigm to sparse tensors (i.e., tensors where most entries are not explicitly represented). With these systems, users define the semantics of the program and the algorithmic decisions in a concise language that can be compiled to efficient low-level code. However, these systems still require users to make complex decisions about program structure and memory layouts to write efficient programs.

This work presents *Galley*, a system for declarative tensor programming that allows users to write efficient tensor programs without making complex algorithmic decisions. Galley is the first system to perform cost based lowering of sparse tensor algebra to the imperative language of sparse tensor compilers, and the first to optimize arbitrary operators beyond $\sum$ and $*$. First, it decomposes the input program into a sequence of aggregation steps through a novel extension of the FAQ framework. Second, Galley optimizes and converts each aggregation step to a concrete program, which is compiled and executed with a sparse tensor compiler. We show that Galley produces programs that are $1-300\times$ faster than competing methods for machine learning over joins and $5-20\times$ faster than a state-of-the-art relational database for subgraph counting workloads with a minimal optimization overhead.

CCS Concepts: • **Information systems** → **Query optimization**; • **Software and its engineering** → Domain specific languages; • **Mathematics of computing** → *Mathematical software*.

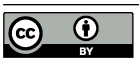
Additional Key Words and Phrases: Query Optimization, Sparse Tensors, Array Programming, Program Optimization

ACM Reference Format:
Kyle Deeds, Willow Ahrens, Magda Balazinska, and Dan Suciu. 2025. Galley: Modern Query Optimization for Sparse Tensor Programs. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 164 (June 2025), 24 pages. https://doi.org/10.1145/3725301

1 Introduction

In recent years, the tensor programming (eq. array programming) model has become ubiquitous for high performance computing tasks. It has been applied to problems such as deep learning [2, 8, 23, 36], data cleaning [46], graph algorithms [47], relational query processing [9, 24, 30], and scientific computing [32, 44, 49], among others. While this approach was originally limited to dense arrays, data from many domains is fundamentally *sparse* (i.e., most entries are a fill value like 0), including graph data, one-hot encodings, relational data, 3D physics meshes, sparse neural networks, and

Authors' Contact Information: Kyle Deeds, kdeeds@cs.washington.edu, University of Washington, United States; Willow Ahrens, wahrens@mit.edu, Massachusetts Institute of Technology, United States; Magda Balazinska, magda@cs.washington.edu, University of Washington, United States; Dan Suciu, suciu@cs.washington.edu, University of Washington, United States.



ACM 2836-6573/2025/6-ART164
https://doi.org/10.1145/3725301

```
 0. # Manually specified format for input tensors
 1. FUNC log_regression(X::Dense(Sparse()), θ::Dense())
 2.     # Manually defined intermediate format
 3.     R = Dense()
 4.     # Manually defined loop order
 5.     FOR i=_
 6.         FOR j=_
 7.             # Manually defined iteration algorithm
 8.             R[i] += X[i::iter,j::iter]*θ[j::lookup]
 9.         END
10.     END
11.     P = Dense()
12.     FOR i=_
13.         P[i] = σ(R[i::iter])
14.     END
15. END
```

Fig. 1. Logistic regression implemented in the language of a sparse tensor compiler.

others. However, modern tensor programming systems like NumPy, PyTorch, and sparse tensor compilers lack the advanced optimization capabilities of relational databases. Instead, users are forced to optimize their programs manually which is challenging and time consuming. In this work, we address this by introducing Galley, a system for declarative sparse tensor programming powered by advanced, cost-based program optimization.

Efficiently processing sparse tensors is challenging. Traditional tensor processing frameworks are collections of hand-optimized functions over *dense* tensors [2, 8, 23, 36]. To take advantage of sparsity, these frameworks need to provide implementations for every combination of input tensors' formats, resulting in spotty coverage for operations over sparse data [27]. Sparse tensor compilers (STCs) have been developed to automatically produce these implementations [4, 11, 26, 29, 43]. However, these compilers expose even more performance decisions than traditional frameworks, and they similarly lack automatic optimization capabilities.[1]

Example 1. *Consider Fig. 1 which implements logistic regression inference in the language of Finch, an STC[4]. Here, the user must choose the output format for the intermediate* `R` *(line 3). In this case, she chose a* `Dense` *rather than a* `Sparse` *format, which would be* $\approx 10\times$ *slower. Then, the user chooses the loop order (lines 5-6). In this case, she chose i-then-j, which is asymptotically faster than j-then-i because each out-of-order access to* X *requires a full scan of the tensor. Finally, the user picks a merge algorithm for each loop that describes how to iterate through the non-zero indices (line 8). Here,* X *is* `iterated` *through, and each non-zero* j *is* `looked up` *in* θ*. If she chose to* `iterate` *through* θ*, each inner loop would scan the entire vector. Even for a simple kernel, these decisions represent a minefield of potential slowdowns.*

In this paper, we propose *Galley*, a system for declarative sparse tensor programming. Galley makes algorithmic decisions on the users' behalf, freeing them to focus on the high-level semantics of their program without sacrificing computational efficiency. It accepts input programs written in a declarative language, equivalent to the core of the NumPy API, and automatically produces an optimized STC implementation using the Finch compiler [4]. To do so, it first restructures the program into a sequence of aggregation steps, minimizing total computation and materialization

[1]Some systems separate declarative and imperative concerns with a scheduling language. However, the user still controls both aspects. For a more detailed description of the prototypical STC, we direct the reader to [29].

costs (Sec. 4). It then optimizes each step by selecting the loop order, the optimal formats for all intermediate tensors, and the merge algorithm for each loop (Sec. 5). These decisions are all guided by a system for estimating sparsity via statistics on the input tensors (Sec. 6). *Galley builds on fundamental principles from cost-based query optimization while developing new techniques that are specific to producing optimized code for sparse tensor compilers.*

Designing Galley required overcoming three key challenges. First, the high-level optimization requires a complex rewriting of the original program which must respect the algebraic properties of the program. We addressed this by introducing a novel extension of the FAQ framework that can handle *arbitrary sparse tensor programs* [28]. Second, STCs provide a vast design space for kernel implementations which makes the per-aggregate optimization challenging. Galley's physical optimizer searches through this space efficiently by separating concerns (loop order, output format, and intersection algorithm) and applying branch-and-bound optimization. Lastly, the computational cost of a sparse tensor program depends on the data distribution of the input data which complicates the optimization process. Galley produces these data-dependent cost estimates by leveraging the similarity of sparsity estimation and relational cardinality estimation. By overcoming these challenges, we have attempted to design Galley for a broad set of use cases ranging from sparse ML to graph algorithms and scientific simulations. To this end, we have incorporated Galley into the PyData/Sparse library which implements the full NumPy API for sparse arrays [3, 23].

Example 2. *Let A, B, and C be sparse matrices, and suppose that you want to compute the matrix chain ABC. Because they do not consider the sparsity of the inputs, traditional systems will always perform this in the order $(AB)C$ where the intermediate, AB, is stored as a sparse matrix. When given this problem, Galley will optimize at runtime for the input's sparsities. This allows it to consider plans that are only efficient for specific inputs. For example, it may: 1) re-order the operations to perform BC before multiplying with A 2) store the intermediate as a dense matrix 3) transpose B to iterate over the shared dimension first. In Sec. 7, we show that this can provide a **10x** speedup over state-of-the-art tensor frameworks for this example.*

Contributions We claim the following contributions:

- We present Galley, a system for declarative sparse tensor programming (Sec.3). Galley is the first system to perform cost based lowering of sparse tensor algebra to the imperative language of sparse tensor compilers, and the first to optimize arbitrary operators beyond $\sum$ and $*$.
- Galley supports a *highly expressive language* for sparse tensor algebra with arbitrary algebraic operators, aggregates within expressions, and multiple outputs (Sec.3).
- Galley performs *cost-based logical optimization* with a novel extension of the variable elimination framework to handle arbitrary aggregations and pointwise operators (Sec.4). Galley performs *cost-based physical optimization* to determine loop orders, tensor formats, and merge algorithms (Sec.5).
- We propose a *minimal interface for sparsity estimation* to guide optimizations and implement two estimators (Sec.6).
- We evaluate Galley and show that it is **1-300x** faster than hand-optimized kernels for mixed dense-sparse workloads and **.25-100x** faster than a SOTA database for highly sparse workloads (Sec.7).
- We have implemented Galley as part of the PyData/Sparse sparse array project and the Finch tensor compiler[4, 37].

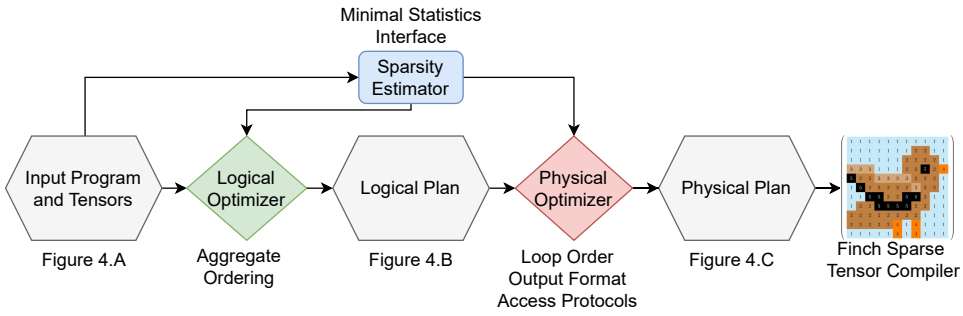


Fig. 2. Galley overview.

2 Background

2.1 Tensor Index Notation

Input to Galley is written in an extended version of Einstein Summation (Einsum) notation that we call *tensor index notation*[7]. Traditional Einsum notation permits a single summation wrapped around a multiplication. For instance, you can describe triangle counting in a graph with adjacency matrix E_{ij} using the following statement:

$$t = \sum_{ijk} E_{ij} E_{jk} E_{ik}$$

To capture the diverse workloads of tensor programming, we additionally allow the use of arbitrary functions for both aggregates and pointwise operations, nesting aggregates and pointwise operations, and defining multiple outputs. For example, a user could perform logistic regression to predict entities that might be laundering money. Then, they could filter this set based on whether the entities occur in a triangle in the transactions graph. This is represented by $\max_{jk}(E_{ij}E_{jk}E_{ik})$, which is 1 if i occurs in at least one triangle and 0. This can be written in tensor index notation as:

$$L_i = \sigma(\sum_j X_{ij}\theta_j) > .5$$
$$V_i = L_i \cdot \max_{jk}(E_{ij}E_{jk}E_{ik})$$

Tensor compilers like Halide, TACO, and Finch each build off of similar core notations, adding additional structures like FOR-loops to let users specify algorithmic choices [4, 29, 39]. Crucially, the vast majority of operations in array programming frameworks like NumPy can be expressed as operations in tensor index notation. Therefore, though we focus here on this notation, traditional tensor workflows can be captured and optimized in this framework.

2.2 Sparse Tensor Compilers

Over the last decade, compiler researchers have developed a series of sparse tensor compilers and shown that they produce highly efficient code for sparse tensor computations[4, 29]. We use this work as our execution engine, so we briefly explain its important concepts below.

Tensor Formats. There are many different ways to represent sparse tensors, and the optimal approach depends on the data distribution and the workload. Work in this space has converged on the *fibertree* abstraction for describing the space of formats [29, 48]. In this formalism, a tensor

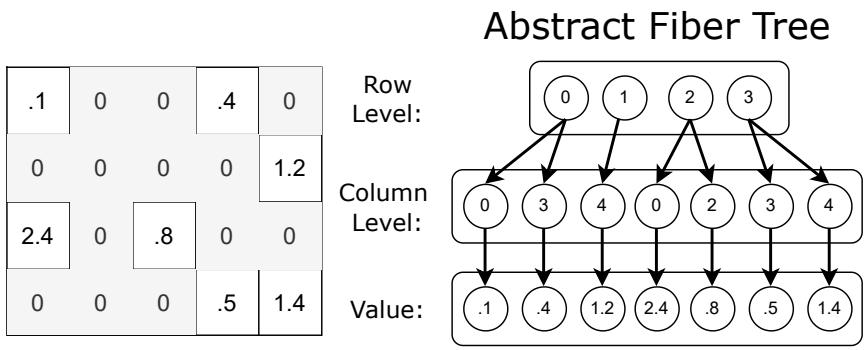


Fig. 3. Fibertree format abstraction.

```
A. Input Program
Plan := Query...    Query := (Name, Expr)
Agg := (Op, Idx..., Expr)   Map := (Op, Expr...)
Expr := Agg | Map | Input | Alias
Input := Tensor[Idx...]  Alias := Name[Idx...]
B. Logical Plan
Plan := Query...   Query := (Name, Agg)
Agg := (Op, Idx..., Expr)   Map := (Op, Expr...)
Expr := Map | Input | Alias
Input := Tensor[Idx...]  Alias := Name[Idx...]
C. Physical Plan
Plan := Query...   Query := (Name, Mat, Idx...)
Mat := (Format..., Idx..., Agg)
Agg := (Op, Idx..., Expr)   Map := (Op, Expr...)
Expr := Map | Input | Alias
Input := Tensor[PIdx...]  Alias := Name[PIdx...]
PIdx := Idx::Protocol
```

Fig. 4. Query plan dialects.

format is a nested data structure resembling the one in Fig. 3. Each layer stores the non-fill (e.g., non-zero) indices in a particular dimension, conditioned on earlier dimensions, and pointers to the next dimension's non-fill indices. These layers can be represented in any format that enables iteration and lookup.

In this work, we consider sorted lists, hash tables, bytemaps, and dense vectors, which each perform differently in terms of iteration, lookup, and memory footprint. For example, the compressed sparse row (CSR) is a common format for sparse matrices. It stores the row dimension as a dense vector, where each entry points to the set of non-zero columns for that particular row. This set of non-zero columns is then stored in a sorted list, i.e., in a compressed sparse format. Importantly, this abstraction requires tensors to be accessed in the order in which they are stored (e.g., row-then-column in the case of CSR), which restricts the set of valid loop orders, as we describe next.

Loop Execution Model. The input to a Sparse Tensor Compiler is a high-level domain specific language (DSL); it consists of for-loops, in-place aggregates (e.g., + =), and arithmetic over indexed tensors (e.g., $A[i, j] * B[j, k]$). Crucially, the for-loops in these expressions are not executed in a dense manner. Instead, these compilers analyze the input formats and the algebraic properties of the expression to determine which index combinations will produce non-fill entries. In Fig. 1, because 0 is the annihilator of multiplication (i.e., $x * 0 = 0$), only the values of i that map to non-zero entries in X *and* θ are processed. All other index values will return a zero. So, the outer loop is compiled to an iteration over the intersection of the non-zero i indices in X and θ; Fig. 3 shows how this is simply co-iteration over the top levels of their formats. The inner loop then iterates over the j indices that are non-zero in $X[i, \_]$, i.e., the non-zero columns that occur in each row.

Merge Algorithms. Once the compiler has determined which tensors' non-zero indices must be merged to iterate over a particular index, it can apply several algorithms. All formats enable both ordered iteration and lookup operations; therefore, one algorithm iterates through the indices of all inputs, similar to a merge join, which is highly efficient per operation. However, this algorithm is linear in the total size of all inputs even if one is much smaller than the others. Another method is to iterate through a single input's level and lookup that index in the others. In this work, we take the latter approach, as described in Sec. 5.3. We refer to the mode of an individual tensor (such as "iterate" or "lookup") as an *access protocol* and the overall strategy as a *merge algorithm* [5].

3 Galley Overview

We now provide a high-level view of Galley. We show how it transforms an input program to a logical plan then to a physical plan that is executed by an STC, as illustrated in Fig. 2. These steps are each represented by a dialect of our query plan language, whose grammar is defined in Fig. 4. In the following discussion, we use this grammar as a guide to show how our example program, i.e., logistic regression, would be transformed through these steps.

3.1 Input Program Space

The input program dialect is equivalent to the tensor index notation defined in Sec. 2.1. Pointwise functions such as $A_{ij} * B_{jk}$ are represented with `Map`. Aggregates such as $\sum_i$ are denoted by `Agg`. Each assignment is a `Query`, and previous assignments are referenced with an `Alias`. Crucially, the `Op` terminal used in both `Map` and `Agg` can be any user defined function (e.g. $f(x, y) = sin(1 + x * y)$) as long as it accepts the correct number of arguments (i.e. the number of expressions in the `Map` and two arguments in `Agg`). Galley takes advantage of properties of these functions during optimization, specifically distributivity, commutativity, associativity, identity, idempotency, and the existence of an annihilator. Further, users can declare these properties to Galley at runtime. This extensibility is a benefit of Galley's formal framework. Lastly, `Idxs` are named symbols (e.g. i, j), and `Tensors` are memory-resident input tensors. Our logistic regression example from Fig. 1 is defined in this dialect as

```
Query(P, Map(σ, Agg(+, j, Map(*, X[i,j], θ[j]))))
```

Note that this notation is compatible with array APIs like Numpy that do not have named indices. Operations like 'matmul' can be automatically mapped into this language by generating index names for inputs on the fly and renaming whenever operations imply equality between indices.

3.2 Logical Plan

The first task in our optimization pipeline, handled by the logical optimizer, breaks down the input program into a sequence of simple aggregates. This is enforced by converting the input program (4.A) to a logical plan (4.B). This dialect is a restriction of the input dialect, where each query

contains a single aggregate statement that wraps an arbitrary combination of `Map`, `Input`, and `Alias` statements. Intuitively, each logical query corresponds to a single STC kernel that produces a single intermediate tensor, but it does not specify details like loop orders and output formats. To perform this conversion soundly, each input query must correspond to a logical query, which produces a semantically equivalent output. To do this efficiently, Galley must minimize the total cost of all queries in the logical plan.

Our logistic regression program above is not a valid logical plan because the outer expression is a pointwise function not an aggregate. However, it can be translated into the following logical plan

```
Query(R, Agg(+, j, Map(*, X[i,j], θ[j])))
Query(P, Agg(no-op, Map(σ, R[i])))
```

In this plan, the first query isolates the sum over the j index, while the second query performs the remaining sigmoid operation on the result. Note that the latter query uses a no-op aggregate to represent an element-wise operation while conforming to the logical dialect.

3.3 Physical Plan

Given the logical plan, Galley's physical optimizer determines the implementation details needed to convert each logical query to an STC kernel. Specifically, it defines the loop order of each compiled kernel, the format of each output, and the merge algorithm for each index. As above, this is expressed by converting the logical plan to a physical plan described in the most constrained dialect. To avoid out-of-order accesses, we require that the index order of inputs and aliases are concordant with the loop order, so the physical optimizer may insert additional queries to transpose inputs. Therefore, each logical query corresponds to *one or more* physical queries.

Using this language, we can precisely express the program from Fig. 1 as follows, where `it` means iterate and `lu` means lookup.

```
Query(R,Mat(dense,i,Agg(+,j,Map(*, X[i::it,j::it],
                                        θ[j::lu]))), i, j)
Query(P,Mat(dense,i, Map(σ, P1[i::it])), i)
```

The first query computes the sum by iterating over the valid i indices for `X`, iterating over the j indices in the intersection of `X[i,_]` and θ, and materializing (hence `Mat`) their product in a dense vector over the i indices. The second query runs over this output and applies the sigmoid function, returning the result as a dense vector.

3.4 Execution

Once Galley has generated a physical plan, the execution is very simple. For each physical query, it first translates the expression into an STC kernel definition and calls the STC to compile it. Then, Galley injects the tensors referenced by inputs and aliases and executes the kernel, storing the resulting tensor in a dictionary by name. After all queries have been computed, it returns the tensors requested in the input program by looking them up in this dictionary.

4 Logical Optimizer

Given the plan dialects above, we now describe the logical optimizer, which receives an input program (Dialect 4.A) and outputs a semantically equivalent *logical plan* (Dialect 4.B). Specifically, the logical optimizer converts each query in the input program to a sequence of logical queries, where the last query produces the same output as the input query. There are many valid plans, and the optimizer searches this space to identify the cheapest one. We now briefly define "cheapest" in this context before outlining the complex space of logical plans that are considered. Finally, we explain the algorithms that we use to perform this search.

4.1 Normalization & Pointwise Distributivity

The first step in logical optimization is to normalize the input program with a few simple rules that we apply exhaustively: (1) merge nested `Map` operators, (2) merge nested `Agg` operators, (3) lift `Agg` operators above `Map` operators, when possible, and (4) rename indices to ensure uniqueness. Applying these rules compresses the input program and makes our reasoning simpler in later steps by ensuring that operator boundaries are semantically meaningful.

Next, we consider whether to distribute pointwise expressions. Doing so may or may not yield a better plan because it both makes operations more sparse and produces larger expressions.

Example 3. *Consider the following expression which computes the loss function for the alternating least squares (ALS) algorithm and its distributed form:*

$$\sum_{ij}(X_{ij} - U_i V_j)^2 = \sum_{ij} X_{ij}^2 - 2\sum_{ij} X_{ij} U_i V_j + \sum_i U_i^2 \sum_j V_j^2$$

If all inputs are dense, the non-distributed form is more efficient because it results in fewer terms and has the same computational cost per term. However, if X_{ij} is sparse and U_i, V_j are dense, then the distributed form is more efficient because all terms can be computed in time linear w.r.t. the sparsity of X_{ij}. Note that the squaring operation here is a pointwise function, not a matrix multiplication.

To take advantage of this potentially asymptotic performance improvement, Galley performs a greedy search for the optimally distributed expression. At each step, it considers all single applications of distributivity in the expression. It then runs variable elimination for each (described later in this section) and computes the cost an optimal logical plan. If applying distributivity improved on the cost of the original expression, it continues. If not, it returns the optimal logical plan discovered so far. Lastly, we additionally consider the expression derived from applying distributivity exhaustively.

4.2 Cost Model

Overall, Galley's logical optimizer attempts to minimize the time required to execute the logical program. Because logical queries do not correspond to concrete implementations, our logical cost model aims to approximate this time without reference to the particular implementation that the physical optimizer will eventually decide on. This approximation considers two factors: (1) the number of non-fill entries in the output tensor and (2) the amount of computation (i.e., the number of FLOPs) needed to produce the output. The former corresponds to the size of the tensor represented by `Agg`, $nnz(\texttt{Agg})$, and the latter corresponds to the tensor size represented by the `MapExpr` within, $nnz(\texttt{MapExpr})$. We assume that the inputs are in memory; hence, there is no cost for reading inputs from disk. We then perform a simple regression to associate each cost with a constant, and we add them to produce our overall cost, c, as follows:

$$cost \approx a * nnz(\texttt{Agg}) + b * nnz(\texttt{MapExpr})$$

To estimate $nnz(\texttt{Agg})$ and $nnz(\texttt{MapExpr})$, we use the sparsity estimation framework described in Sec. 6.

4.3 Variable Elimination

The core of our logical optimizer is an extension of the *variable elimination* (VE) (eq. FAQ) framework [19, 28]. In its original context, this algorithm described a means of marginalization for probabilistic models by removing one variable at a time. When applied to our setting, it allows us to define the logical plan for an input query via an order on the indices being aggregated over, i.e., an *elimination order*. If we are given this order, we can construct a valid logical plan by iterating through the

elimination order one index at a time in order to (1) identify the minimal sub-expression needed to aggregate over it, (2) create a new logical query representing the result of that sub-expression, and (3) replace it in the original query with an alias to the result. At the end of this process, the remaining query no longer requires any aggregation and therefore is itself a logical query.

Example 4. *Consider optimizing the following matrix chain multiplication:*

$$E_{im} = \sum_{jkl} A_{ij} B_{jk} C_{kl} D_{lm}$$

The elimination order jkl corresponds to a left-to-right multiplication strategy because eliminating j from the expression first requires performing the matrix multiplication between A and B. Eliminating k then requires multiplying that intermediate result with C, and so on. Concretely, this produces the following sequence of logical queries:

```
Query(I1, Agg(+, j, Map(*, A[i,j], B[j,k])))
Query(I2, Agg(+, k, Map(*, I1[i,k], C[k,l])))
Query(E, Agg(+, l, Map(*, I2[i,l], D[l,m])))
```

Similarly, the elimination order lkj corresponds to a right-to-left strategy, and the order klj to a middle-first strategy.

Unlike traditional VE for sum-product queries, we support complex trees of pointwise operators and aggregates. This makes identifying minimal sub-queries challenging since we must carefully examine the expression's algebraic properties. Given a strategy for this, the core problem of optimizing VE is to search the space of elimination orders for the most efficient one. In the worst case, this takes exponential time w.r.t. the number of indices being aggregated over. In the following sections, we describe how we identify minimal sub-queries and our search algorithm for finding the optimal elimination order.

4.4 Identifying Minimal Sub-Expressions

We now explain how to identify the minimal sub-expressions (MSEs) needed to eliminate an index. In sum-product expressions, the MSE is simply the product of the tensors that are indexed by it. For more complex input programs, we show that identifying MSEs corresponds to a careful traversal down the *annotated expression tree*, examining the algebraic properties of each operation to determine how to proceed.

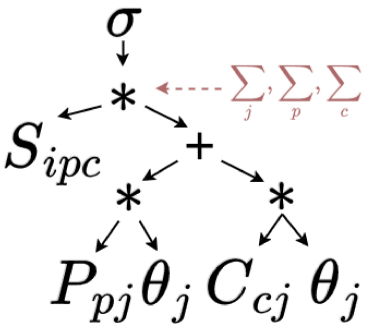


Fig. 5. Annotated expression tree for logistic regression over joins $\sigma(\sum_{jpc} S_{ipc}(P_{pj}\theta_j + C_{cj}\theta_j))$

Annotated Expression Tree. The annotated expression tree (AET) is constructed by examining the nested structure of `Agg`, `Map`, `Input`, and `Alias` nodes in the input query. To do this, Galley

first removes all `Agg` nodes and annotates their inner expressions with `(Idx, Op)`. It then replaces all `Map` nodes with their operator to get the final tree, where every internal node is a pointwise function and every leaf is either an `Input` or an `Alias`.

Example 5. *Fig. 5 shows the annotated expression tree for logistic regression where the input matrix is defined by a join-like expression $X_{ij} = S_{ipc}(P_{pj} + C_{cj})$. Further, Galley has pushed down θ_j into this expression. The sigmoid function is the outermost layer of the expression, so it appears at the top of the tree. The summations all occur just inside the sigmoid function, so they annotate the top multiplication operator.*

Given the AET, Galley identifies an index's MSEs by starting at the node where it is annotated and traversing downwards according to the algebraic properties of each internal node. We now describe the traversal rules for functions that are distributive, non-distributive, and commutative with respect to the aggregation operator.

Distributive Functions. When we reach a function that distributes over the aggregate (e.g., $*$ and $\sum$), we examine how many of the children, subtrees of the AET, contain the current index. If one child contains the index, we traverse down that child's branch, i.e., we factor the other children out of the aggregate. If multiple children contain the index, we wrap the sub-tree rooted at that node in the aggregate and return it as our MSE. If the function is commutative and associative, we only include the children that contain the index.

Commutative, Identical Functions. When the node's function is the same as the aggregate function and is commutative, we can push the aggregate down to each child independently. For example, we can transform the expression $\sum_i A_i + B_i$ into $\sum_i A_i + \sum_i B_i$. For all children that contain the index, we add the result of traversing down its branch to the list of MSEs and replace it with an alias to the result. If a child does not contain the index, then we need to account for the repeated application of the aggregate function. For example, $\sum_i B = N_i * B$ where N_i is the size of the i dimension.

Blocking Functions. A function that does not distribute or commute with our aggregate function is called a *blocking function*. When we reach a blocking function in our traversal, we simply wrap it in our aggregate and return the sub-tree as an MSE. For example, the expression $\sum_j \sqrt{A_{ij} B_{jk}}$ cannot be rewritten as $\sqrt{\sum_j A_{ij} \sum_j B_{jk}}$ because $\sqrt{}$ is a blocking function.

Discussion. Galley builds upon and extends the theoretical FAQ framework for optimizing conjunctive queries with aggregation[28]. This framework explored the optimization of queries with the following form, where each $\bigoplus^{(i)}$ is either equal to or forms a semi-ring with $\otimes$:

$$\bigoplus_{v_1}^{(1)} \cdots \bigoplus_{v_k}^{(k)} F^1_{V_1} \otimes \cdots \otimes F^k_{V_k}$$

Similarly to Galley, the FAQ paper described the optimization problem as selecting an optimal elimination order over the aggregated variables. Though this framework captures many important problems, it lacks the flexibility needed to support a general tensor processing system. Consider a slightly modified version of the SDDMM kernel:

$$\sum_j A_{ik}(B_{ij} + C_{jk})$$

This expression is not an FAQ query because it mixes addition and multiplication in the pointwise expression. Galley extends this framework to accommodate arbitrary pointwise expressions and placement of aggregates within expressions.

4.5 Restricted Elimination Orders

Depending on the program structure, the order in which indices can be eliminated might be restricted. This could be due to *non-commutative aggregates* or *aggregate placement*. The former is when an aggregate wraps another aggregate that it does not commute with. For example, given $\max_i \sum_j A_{ij}$, we must perform the summation before handling the maximum because max and $\sum$ do not commute. The latter is when an aggregate wraps another aggregate but cannot reach it via the traversal described above, e.g., $\sum_i \sqrt{\sum_j A_{ij}}$; in this case, the inner aggregate must be performed first. Collectively, these restrictions form a partial ordering on the index variables that must be respected when we enumerate elimination orders.

4.6 Search Algorithms

With the VE approach, we have simplified the complicated issue of high-level optimization to the discrete problem of choosing an optimal order on the aggregated index variables. We start by revisiting our example from Fig. 5. The input query is the following,

```
Query(X, Map(σ,
            Agg(+,p,c,j,
                Map(*,S[i,p,c],
                    Map(+,
                        Map(*, P[p,j], θ[j]),
                        Map(*, C[c,j], θ[j])))))
```

The elimination order for this expression is an ordering of the indices $\{p, c, j\}$. Galley's logical optimizer searches through these possible orders to find the most efficient one. In this case, it would choose $[j, p, c]$, resulting in the following logical plan,

```
Query(A1, Agg(+, j, Map(*, P[p,j], θ[j])))
Query(A2, Agg(+, j, Map(*, C[c,j], θ[j])))
Query(A3, Agg(+, p, c, Map(*, S[i,p,c],
                           Map(+, A1[p], A2[c]))))
Query(X, Map(σ, A3[i]))
```

We now present two algorithms to search for that optimal order using the tools described above.

Greedy. The greedy approach chooses the cheapest index to aggregate at each step by finding the minimal sub-query for each index and computing its cost. The cheapest index's minimal sub-query is removed from the expression, appended to the logical plan, and replaced with an alias to the result. This continues until no aggregates remain in the expression.

Branch-and-Bound. The branch-and-bound approach computes the optimal variable order and occurs in two steps. The first step uses the greedy algorithm to produce an upper bound on the cost of the overall plan; the second performs a dynamic programming algorithm. In the dynamic programming step, the keys of the memo table are unordered sets of indices, and the values are tuples containing a partial elimination order, residual query, and cost. The algorithm initializes the table with the empty set and a cost of zero. At each step, it iterates through table entries and attempts to aggregate out another index. It then uses the cost bound from the first step to prune entries from the memo table whose cost exceeds the bound; doing so is valid because costs monotonically increase as more indices are added to the set. At the end of this step, the algorithm returns the index order associated with the full set of indices.

5 Physical Optimizer

Each query in the logical dialect roughly corresponds to a single loop nest and materialized intermediate. However, several decisions remain about *how* the kernel is computed, including: (1)

the loop order over the indices, (2) the format of the result, and (3) the merge algorithm for each index. The physical optimizer makes these decisions.

5.1 Loop Order

The loop order determines that inputs are accessed. An good loop order prunes the iteration space due to early intersection of sparse inputs. Intuitively, this is similar to selecting a variable order for a worst-case optimal join algorithm. Galley's physical optimizer searches the space of loop orders to find one with the minimum cost, defined below.

Cost Model. The cost of a loop order is composed of each loop's number of iterations and the cost of transposing inputs to make them concordant with the loop order.

Example 6. *Consider matrix chain multiplication over three sparse matrices, A, B, and C, where*

$$D[il] = \sum_{jk} A[ij] * B[jk] * C[kl] \tag{1}$$

Suppose that A has only a single non-zero entry and that B and C have 5 non-zero entries per column and per row. In this case, the loop order ijkl is significantly more efficient than lkji. In the former, the first two loops, over i and j, incur only a single iteration because they are bounded by the size of A. The third and fourth incur 5 and 5^2 iterations, respectively, because there are only 5 non-zero k's per j in B and 5 non-zero l's per k in C. In the latter, the first two loops iterate over the full matrix C despite most of those iterations not leading to useful computation.

Formally, let Q be the pointwise expression in our kernel, and let $Q_{(i_1,\ldots,i_k)}$ be the restriction of that expression to just the index variables $i_1, \ldots, i_k$. Let $\mathbf{A}^{(i_1,\ldots,i_k)}$ be the input tensors that are not concordant with $i_1, \ldots, i_k$. Then, we can define the cost of a loop order as follows,

$$cost(Q, (i_1, \ldots, i_k)) \approx \sum_{j=1}^{k} nnz(Q^{(i_1,\ldots,i_j)}) + \sum_{A \in \mathbf{A}^{(i_1,\ldots,i_k)}} |A|$$

In practice, we further refine this model to take into account the number and kind of tensor accesses at each loop.

Optimization Algorithm. To optimize the loop order, we combine this cost model with a branch-and-bound, dynamic programming algorithm. In the first pass, the optimization algorithm selects the cheapest loop index at each step until reaching a full loop order. This produces an upper bound on the optimal execution cost, which the algorithm uses to prune loop orders in the second step. This step applies a dynamic programming algorithm. Taking inspiration from Selinger's algorithm for join ordering, each key in the DP table is a set of index variables and a set of inputs. The former represents the loops that have been iterated so far, and the latter represents a set of inputs that must be transposed.

5.2 Intermediate Formats

Once the loop order has been determined, the physical optimizer selects the optimal format for each query's output. First, Galley sets the order of the indices to be concordant with either the loop order of the kernel where it will be consumed or the order requested by the user. Then, it selects a format for each index (e.g., dense vector, hash table, etc.). Two factors affect this decision: (1) the kind of writes being performed (sequential vs random) and (2) the sparsity of the tensor at this index. The former is important because many formats (e.g., sorted list formats) only allow sequential construction. These formats can only be used if the output indices form a prefix of the loop order.

When considering sparsity, Galley balances the fact that dense formats have better baseline efficiency, while sparse formats are asymptotically more efficient for highly sparse outputs. To describe this trade-off, we hand selected sparsity cutoffs between fully sparse, bytemap, and fully dense formats. To determine a particular output index's format, the physical optimizer first determines the sparsity at this index level and uses our cutoffs to determine which category of formats to consider. Then, it checks whether sequential or random writes are being performed and selects the most efficient format that supports the write pattern.

5.3 Merge Algorithms

The final decision the physical optimizer makes concerns the algorithm it will use to perform each loop's intersection. While there are more complex strategies, we adopt instead a minimal approach and select a single input to iterate over for each loop. The physical optimizer then probes into the remaining inputs. It makes this selection by estimating the number of non-zero indices that each input has, conditioned on the indices in the outer loops. This resembles the approach taken in [51] for optimizing WCOJ.

5.4 Common Sub-Expression Elimination

Galley takes a straightforward approach to avoiding redundant computation. Once a physical plan has been generated, the right hand side of each physical query is canonicalized and hashed. When two physical queries result in the same hash, the latter query is removed from the plan and all references to it are replaced with a reference to the result of the former. This is helpful for caching small computations like transpositions, but it is also useful for reducing the overhead of applying distributivity which often results in duplicate sub-expressions.

6 Sparsity Estimation

We now describe how Galley performs the sparsity estimation that guides our logical and physical optimizers. First, we explore the subtle correspondence between sparsity and cardinality estimation. We then present a minimal interface for sparsity estimation inspired by this correspondence, after which we examine two implementations of this framework, i.e., the uniform estimator and the chain bound.

6.1 Sparsity and Cardinality Estimation

Sparsity estimation is highly related to cardinality estimation in databases. However, translating methods for the latter to the former requires analyzing the algebraic properties of our tensor programs. For example, let A_{ij} and B_{jk} be sparse matrices with a fill value of 0, and let $R_A(I, J)$ and $R_B(J, K)$ be relations that store the indices of their non-zero entries. Assume we are performing the following,

$$C_{ijk} = A_{ij}B_{jk}$$

In this case, the number of non-zero values in C is precisely equal to the size of the conjunctive query

$$nnz(C) = |R_A(I, J) \bowtie R_B(J, K)|$$

The correspondence results from the fact that 0 is the annihilator of multiplication (i.e., $x * 0 = 0 \forall x$), so any non-zero entry ijk in the output must correspond to a non-zero ij in A *and* a non-zero jk in B. Consider the following instead:

$$C_{ijk} = A_{ij} + B_{jk}$$

In this case, a nonzero ijk in the output can result from a non-zero ij in A *or* a non-zero jk in B. In traditional relational algebra, where relations are over infinite domains, this kind of disjunction would result in an infinite relation. However, tensors have finite dimensions, so we can introduce relations that represent the finite domains of each index, e.g., $D_i = \{1, ..., n_i\}$. This lets us represent the index relation of the output as

$$nnz(C) = |(R_A(I, J) \bowtie D_k(K)) \cup (D_i(I) \bowtie R_B(J, K))|$$

Finally, we can translate aggregations to the tensor setting as projection operations. Given the statement

$$C_{ik} = \sum_j A_{ijk}$$

we can express the non-zeros entries of C as

$$nnz(C) = |\pi_{I,K}(R_A(I, J, K))|$$

6.2 The Sparsity Statistics Interface

We use our statistics interface to annotate every node of the AST with statistics. Surprisingly, to support sparsity estimation over arbitrary tensor algebra expressions, we only need a few core functions: (1) a constructor, which produces statistics from a materialized tensor for `Input` and `Alias` nodes, (2-3) a function for (non) annihilating `Map` nodes (i.e., those whose children's fill values are the annihilator of its pointwise function), which merges the children's statistics, (4) a function for `Agg`, which adjusts the input's statistics to reflect an aggregation over some set of indices, and (5) an estimation procedure, which estimates the sparsity of a tensor based on its statistics.

6.3 Supported Sparsity Estimators

6.3.1 Uniform Estimator. The simplest statistic that can be kept about a tensor is the number of non-fill (e.g., non-zero) entries. The uniform estimator uses only this statistic and assumes these entries are uniformly distributed across the dimension space. This corresponds to System-R's cardinality estimator with the added assumption that the active domain is the whole dimension for each index [42].

Constructor. This function simply counts the non-fill values in the tensor, $nnz(A)$, and notes the dimension sizes $n_{i_1}, \ldots, n_{i_k}$.

Map (Annihilating). To handle an annihilating pointwise operation, this function calculates the probability that a point in the output was non-fill in all inputs, then multiplies this by the dimension space of the output. For a set of inputs $A^{(1)}_{I_1} \ldots A^{(l)}_{I_l}$ and output C_{I_C}, where each I_j is a set of indices, this probability is

$$nnz(C) \approx \left(\prod_{i \in I_C} n_i\right) \cdot \left(\prod_j \frac{nnz(A_j)}{\prod_{i \in I_j} n_i}\right)$$

Map (Non-Annihilating). To handle an non-annihilating pointwise operation, this function calculates the probability that an entry in the output was *fill* in all inputs. Then, it takes the compliment to get the probability that it was non-fill in all inputs and multiplies this by the output dimension space. Using the preceding notation:

$$nnz(C) \approx \left(\prod_{i \in I_C} n_i\right) \cdot \left(1 - \prod_j \left(1 - \frac{nnz(A_j)}{\prod_{i \in I_j} n_i}\right)\right)$$

Aggregate. Given an input tensor A_I to aggregate over the indices I', this function computes the probability that an output entry is non-fill by calculating the probability that at least one entry in the subspace of the input tensor was not fill:

$$nnz(C) \approx \left(\prod_{i \in I \setminus I'} n_i\right) \cdot \left(1 - \left(1 - \frac{nnz(A_I)}{\prod_{i \in I} n_i}\right)^{\prod_{i \in I'} n_i}\right).$$

Estimate. The estimation function simply returns the current tensor's stored cardinality statistic.

Example 7. *Suppose A_{ij} and B_{jk} are $100x100$ sparse matrices with $nnz(A) = 1000$, $nnz(B) = 200$, and we want to estimate $nnz(\sum_j A_{ij}B_{jk})$. We first compute $nnz(A_{ij}B_{jk})$ as $100^3 * \frac{1000}{100^2} * \frac{200}{100^2} = 2000$. The fractions are the probability that i, j or j, k was zero in A or B, respectively. Next, we factor in the aggregation over j to get $nnz(\sum_j A_{ij}B_{jk}) = 100^2 * (1 - (1 - \frac{nnz(A_{ij}B_{jk})}{100^3})^{100}) \approx 1800$. Here, the expression $(1 - \frac{nnz(A_{ij}B_{jk})}{100^3})^{100}$ is the probability that all entries were zero for a particular i, k pair.*

6.3.2 Degree Statistics and the Chain Bound. Galley stores degree statistics by default uses them to compute upper bounds on tensors' sparsities. A degree statistic, denoted as $D_A(X|Y)$, stores the maximum number of non-fill entries in the X dimensions conditioned on the Y dimensions for a tensor A. For example, given a matrix A_{ij}, $D_A(i|j)$ is the maximum number of non-fill entries per column, and $D_A(ij|\emptyset)$ is the total number of non-fill entries in the matrix. This approach follows work in cardinality bounding that has been shown to produce efficient query plans in the relational setting [16, 20, 25].

Constructor. This function first computes the boolean tensor representing the input's sparsity pattern. Then, to calculate each degree statistic, it sums over the X dimensions and takes the maximum over the Y dimensions. The set of degree statistics for a tensor A_I is denoted $\mathcal{D}_{A_I}$.

Map (Annihilating). Annihilating map operations can only reduce the degree for any X, Y pair. Therefore, every input's degree statistics are also valid for the output. If the inputs are $A^(1)_{I_1}, ..., A^(k)_{I_k}$, then the output's statistics are,

$$\mathcal{D}_C = \bigcup_j \mathcal{D}_{A^{(j)}_{I_j}}$$

Map (Non-Annihilating). In this case, Galley extends the degree constraints from each input to cover the full set of indices. Then, it computes degree statistics about the output, C, from the inputs $A^{(1)}_{I_1}, \ldots, A^{(k)}_{I_k}$ by addition:

$$D_C(X|Y) = \sum_j D_{A^{(j)}}(X|Y)$$

Estimator. This function calculates an upper bound (eq. performs sparsity estimation) using the breadth-first search approach described in [17]. Intuitively, each set of indices forms a node in the graph, and each degree constraint is a weighted edge from Y to X. Its search begins with the empty set; it then uses a breadth-first search to find the shortest weighted path to the full set of indices I. The product of the weights along this path bounds the number of non-zeros in the result.

Example 8. *Suppose A_{ij} and B_{jk} are $100x100$ sparse matrices with $D_A(ij|\emptyset) = 1000, D_A(i|j) = 10, D_B(jk|\emptyset) = 200$, and we want to bound $nnz(\sum_j A_{ij}B_{jk})$. Because multiplication is an annihilating operation in this case, the degree constraints of $\sum_j A_{ij}B_{jk}$ are simply the union of the constraints for A and B. To get a bound, we start by conditioning on the empty set and try to reach the output's index set, i, k, via the constraints, e.g. $D_B(jk|\emptyset) * D_A(i|j) = 2000$.*

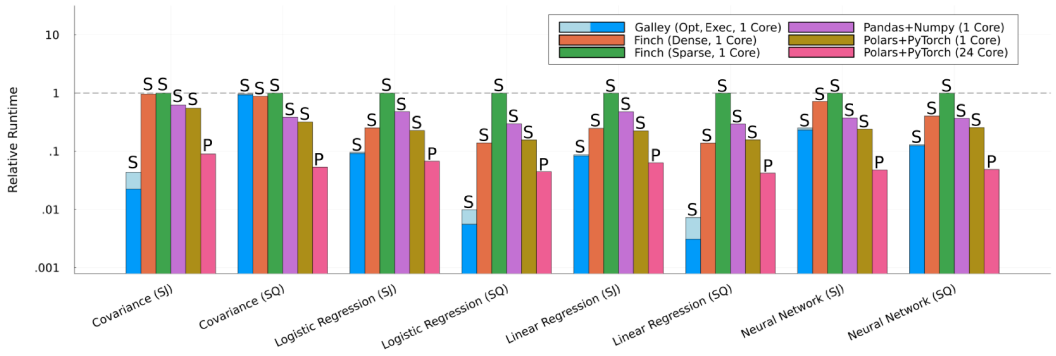
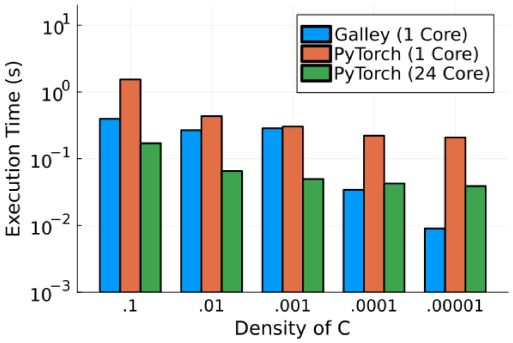
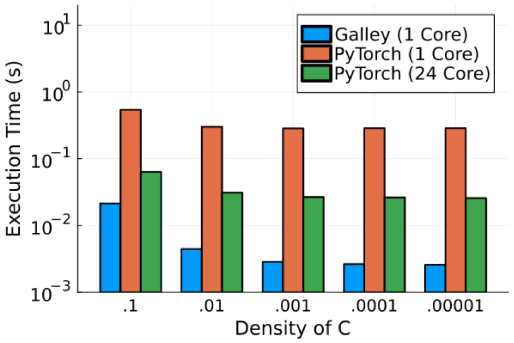


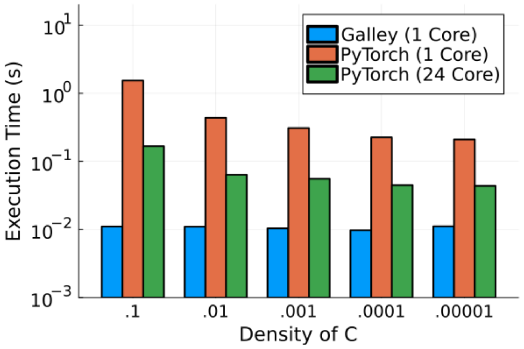
Fig. 6. ML Inference Over Joins



(a) Matrix Chain Multiplication (ABC)



(b) Elementwise Matrix Multiplication (A*B*C)



(c) Sum of Matrix Chain (SUM(ABC))

Fig. 7. Linear Algebra Kernels

7 Experimental Evaluation

In this section, we evaluate the effectiveness of our optimizer on a variety of workloads: (1) ML algorithms over joins, (2) unstructured sparse linear algebra (3) subgraph counting, and (4) breadth-first search. We choose those workloads because they exercise different aspects of our optimizer on real-world use-cases: ML algorithms over joins require careful logical optimizations over programs

Table 1. Experimental Dataset Sizes

Dataset	Size
TPCH (SF .25 - SF 5.0)	.3-6 GB
aids	11 MB
human	1.5 MB
yeast	1.2 MB
dblp	21 MB
youtube	63 MB
Epinions	5.1 MB
Kron	34 MB
LiveJournal	.5 GB
Orkut	1.7 GB
RoadNet	41 MB

with mixtures of dense and sparse inputs and non-linear operators; core linear algebra expressions demonstrate the broad utility of Galley; subgraph counting requires both logical and physical optimization of complex sum-product expressions over highly sparse inputs and demonstrates Galley's advantage over a relational engine even for very sparse workloads; breadth-first search requires careful selection of tensor formats over the course of the computation, showing the benefit of physical optimization for even simple computations. Compared to hand-optimized solutions and alternative approaches, Galley is highly computationally efficient while requiring only a concise, declarative input program from the user. Overall, we show that Galley:

- Performs logical optimizations resulting in **1-300**× faster execution for ML algorithms over joins compared to hand-optimized and Pandas implementations and **.5-20**× faster runtime when including optimization.
- Optimizes in a mean time of at most **0.1** seconds for all subgraph counting workloads, with **5-20**× faster median execution than DuckDB.
- Selects optimal formats for intermediates, outperforming both fully dense and sparse formats for 4/5 graphs in a BFS application.

Experiment Setup. These experiments are run on a server with an AMD EPYC 7443P Processor and 256 GB of memory. We implemented Galley in the programming language Julia, and the code is available at https://anonymous.4open.science/r/Galley-21BF/. We used the sparse tensor compiler Finch[2] for execution, and all experiments are executed using a single thread. Unless otherwise stated, Galley uses the chain bound described in Sec. 6.3.2 for sparsity estimation. Experiments for all methods are run five times, and the mean execution time is reported. We perform all experiments on a warm cache, and we separately report the compilation and optimization times.

7.1 Machine Learning Over Joins

To explore end-to-end program optimization, we experiment with simple ML algorithms over joins. This represents a typical machine learning use case where the feature matrix is constructed from a variety of tables stored within an enterprise database that are joined together before training. Prior work has shown that co-optimizing these mixed RA/LA problems can yield significant benefits [18, 22, 31, 34, 40]. For this, we consider two join queries over the TPC-H benchmark: a star join and a self join, at scale factor 5 and .25, respectively. The star join is expressed as follows, where

[2]https://github.com/FinchTensor/Finch.jl

Table 2. Total Subgraph Counting Execution Time (S)

Workload	Galley (Greedy)	Galley+DuckDB	DuckDB	Umbra 1 (24)
human	.17 (.43)	.156	.12	.04 (.02)
aids	32.1 (29.43)	43.32	78.16	7.47 (1.89)
yeast_lite	2.96 (3.85)	8.91	1633	367.28 (51.56)
dblp_lite	32.31 (30.44)	39.31	3294	75.51 (18.23)
youtube_lite	240.24 (219.47)	1591.27	17203	14208 (13866)

L, S, P, O, and C are tensors representing the line items, suppliers, parts, orders, and customers tables, respectively:

$$X_{ij} = \sum_{spoc} L_{ispoc}(S_{sj} + P_{pj} + O_{oj} + C_{cj})$$

The non-zero values in S, P, O and C are disjoint along the j axis, so the addition in this expression serves to concatenate features from each source, resulting in 139 features after one-hot encoding categorical features. The self-join query compares line items for the same part based on part and supplier features. In this case, the feature data is a 3D tensor because the data points are keyed by pairs of line items:

$$X_{i_1 i_2 j} = \sum_{s_1 s_2 p} L_{i_1 s_1 p} L_{i_2 s_2 p}(S_{s_1 j} + S_{s_2 j} + P_{pj})$$

We consider a range of ML algorithms: (1) linear regression inference, (2) logistic regression inference, (3) covariance matrix calculation, and (4) neural network inference. We implement two versions of each of these using the Finch compiler. The dense version uses a dense feature matrix, and the sparse version uses CSR matrix to compress the one-hot encoding. We also implemented two standard baselines; 1) using Pandas for the joins and Numpy for the linear algebra 2) using Polars for the joins and PyTorch for the linear algebra. The latter supports parallelism, so we've included parallel results for it as well (marking the parallel bars with P and serial bars with S).

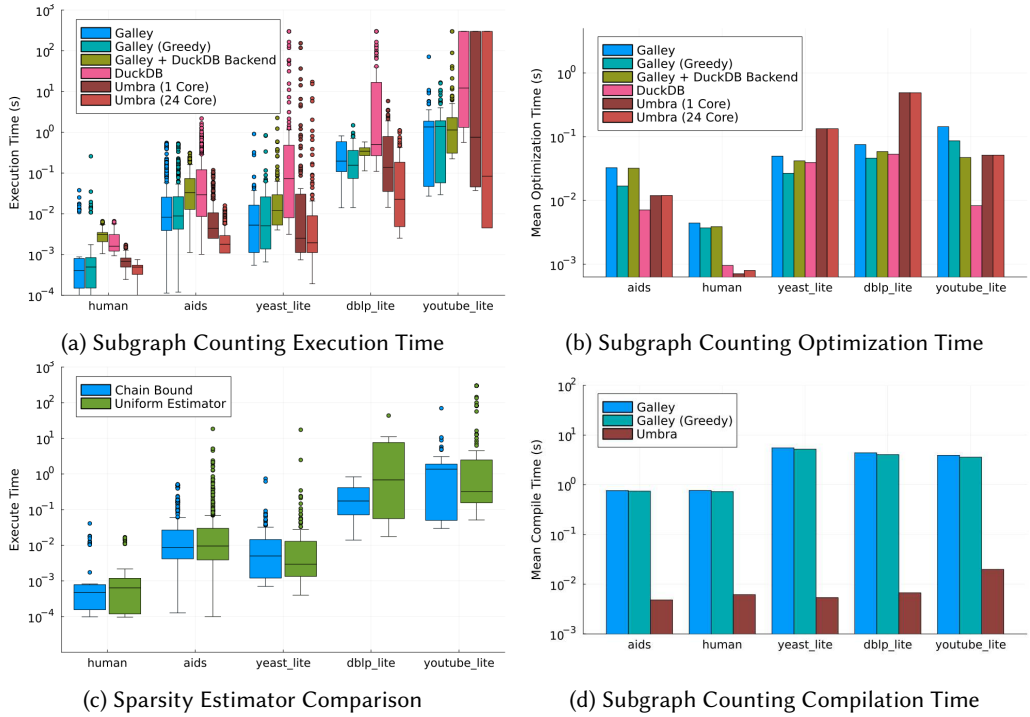
These algorithms stress the ability of Galley to handle complex operators and combinations of sparse and dense inputs. The definitions of the feature tensors combine pointwise multiplication and addition, and algorithms like logistic regression and neural networks wrap these definitions in non-linear operators (e.g. relu and sigmoid) and aggregates. Further, while the line item tensor is highly sparse, both the feature and parameter tensors are moderately to fully dense.

Execution Time. Fig. 6 shows that the execution time of Galley's optimized programs is $.5-300\times$ faster than the sparse Finch implementation. For the regression/neural network problems, this stems from pushing the multiplication with the parameter vector/matrix down to the feature matrices. For the covariance calculation over the self join, Galley fully distributes the multiplication over the addition then aggregates away the sparse i_1, i_2 dimensions. This produces small, dense intermediates which can be used to calculate the covariance efficiently.

Optimization Time. Fig. 6 also shows that Galley's optimizer has a reasonable overhead in this setting. Concretely, optimization takes $.5 - 5.0$ seconds across all workloads.

7.2 Linear Algebra Kernels

In Fig. 7, we show how Galley can provide significant benefits even for simple workloads without structured data. In these experiments, A, B, and C are 2000×2000 uniformly sparse matrices. A and B have a density of .1, and the density of C varies on the X axis. In the first experiment, Fig. 7a, Galley improves on PyTorch's execution in two ways: 1) when the matrices are less sparse, it



(a) Subgraph Counting Execution Time

(b) Subgraph Counting Optimization Time

(c) Sparsity Estimator Comparison

(d) Subgraph Counting Compilation Time

Fig. 8. Subgraph Counting Experiments

chooses fully dense formats to store then intermediates and outputs 2) when C is heavily sparse, it uses a right-to-left execution strategy for the chain, i.e. $(A(BC))$. In the second experiment, Fig. 7b, Galley is able to fuse the computation when PyTorch is unable to, removing an intermediate materialization. Further, when C is highly sparse, Galley iterates over the non-zeros of C and looks them up in A and B, rather than doing a symmetric intersection algorithm. In the third experiment, Fig. 7c, Galley's logical optimizer pushes down the outer summation to A and C first which avoids doing any expensive matrix multiplications. In almost all cases, Galley's optimizations even overcome the benefits of parallelism when PyTorch is provided with 24 cores.

7.3 Subgraph Counting

In this section, we stress test Galley's ability to optimize programs with a large number of highly sparse inputs by implementing several sub-graph counting benchmarks. These workloads represent the far end of the complexity and sparsity spectrum for sparse tensor compilers. Suppose you are counting the occurrences of $H(V, E)$ in a data graph G with adjacency matrix M; we can represent the count as,

$$c = \sum_{v_i \in V} \prod_{(v_i, v_j) \in E} M_{v_i v_j}$$

We add sparse binary vector factors for each labeled vertex. We use subgraph workloads from the G-Care benchmark and the paper "In-Memory Subgraph: an In-Depth Study" [35, 45]. We restrict them to query graphs with up to 8 vertices and 24 edges. Because this is a relational workload, we compare it with DuckDB and Umbra, two state-of-the-art modern OLAP databases [33, 38].

The latter is known to be one of the fastest databases for complex joins and aggregations, and we include both serial and parallel execution [1]. We did not include these systems in our other experiments due to the difficulty of framing the problems in SQL and because prior work has already demonstrated their challenges on pure LA workloads [41]. To separately discern the impact of logical vs physical optimization and our use of Finch, we provide a version of Galley that executes each logical query with a SQL query run on DuckDB. We also provide results for the greedy logical optimizer.

Logical Optimization. Fig. 8a shows the execution time for each method and benchmark. The comparison between ‘DuckDB‘ and ‘Galley + DuckDB Backend‘ demonstrates the benefits of Galley’s logical optimizer. Galley’s logical optimizer breaks down the program into a series of aggregations which minimizes the necessary computation and materialization. This has the largest impact on graphs with high skew like the social network graphs, ‘dblp_lite‘ and ‘youtube_lite‘. In these cases, pushing down aggregates avoids very large intermediate results. DuckDB hits the 300 second timeout on 56 out of 120 queries in the youtube_lite benchmark, as does Umbra on 46 queries. In contrast, Galley never times out across all workloads.

Physical Optimization. The impact of Galley’s physical optimizer can be seen by comparing ‘Galley‘ with ‘Galley + DuckDB‘. Galley’s median execution is up to 8x faster than DuckDB even with the same logical plan. This shows that Galley is selecting efficient loop orders and formats, effectively leveraging STCs.

Optimization Time. Fig. 8b shows the mean optimization time for each method on each workload. Galley has a mean optimization time of less than .15 seconds across all workloads, faster than Umbra’s optimizer for 2 workloads.

Compilation Time. Because it performs compilation at runtime, Galley incurs a compilation overhead when it invokes an STC kernel. These kernels are cached by Finch, reducing this cost when workloads repeatedly use similar kernels. We show the mean compilation time for each subgraph workload in Fig. 8d. On the simpler workloads, which often reuse kernels, this cost is lower. More complex workloads reuse kernels less, significantly increasing compilation time.

Comparing Figures 8 and 9, Galley’s optimization overhead is minimal (generally less than 1%) compared to the compilation overhead. Reducing this requires performance improvements to the underlying compiler which are out of scope for this work. Fortunately, the Finch project is working to improve this in two ways (1) by caching compilation to disk and (2) by migrating to the MLIR compiler infrastructure. As these improvements are made, Galley will immediately reap the benefits.

Sparsity Estimation. Finally, in Fig. 8c, we use the sub-graph matching workloads to compare sparsity estimators and their effect on performance. Across all workloads, we see that the chain bound has significantly better tail performance. This is because it encourages more conservative query plans which better handle correlated and skewed queries/datasets.

7.4 Breadth-First Search

To demonstrate the importance of format selection, we implement a breadth-first search algorithm using Galley and hand-coded Finch implementations. Both systems receive a single iteration at a time, and the total execution time is reported. The core computation is a masked sparse matrix times sparse vector to compute the new frontier vector. The main decision is the visited and frontier vectors’ formats. The former’s sparsity grows monotonically over iterations, while the latter peaks in the middle iterations. We provide two implementations of Finch, using either a sparse or a dense vector for both. Fig. 9 shows that Galley’s mixture of sparse and dense formats is significantly fastest for 4 of the 5 graphs and is competitive for all graphs. For 4/5 graphs, the total optimization time (not depicted in the figure) is less than .25 seconds. This experiment demonstrates the utility

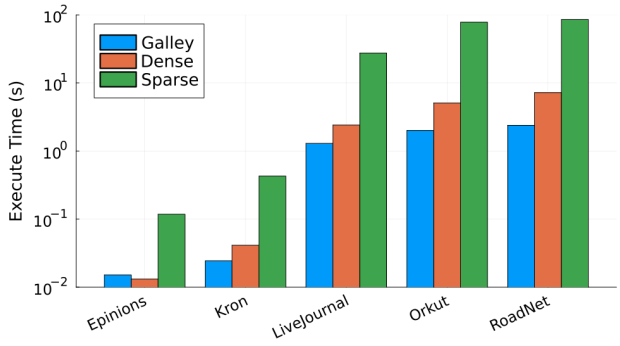


Fig. 9. BFS Execution Time

of sparsity-aware format selection, and future work should consider ways to amortize optimization time for iterative workloads.

8 Related Work

Galley differs from other work on cost-based optimization for tensor processing due to its targeting of STCs and its expressive input language. SystemDS, formerly SystemML, focuses on end-to-end ML over matrices and tabular data [10, 13, 14]; it takes as input linear algebra (LA) programs and targets a combination of LA libraries and distributed computing via Spark. Later work, SPORES, extended its logical optimizer to leverage relational algebra when optimizing sum-product expressions[50]; their core insight was that LA rewrites, which always match and produce 0-2D expressions, are not sufficient and that optimal rewrites must pass through higher order intermediate expressions. Other related work translated sum-product expressions to SQL to leverage highly efficient database execution engines [12]. These systems can perform well for highly sparse inputs but struggle on mixed dense-sparse workloads. Tensor relational algebra proposes a relational layer on top of dense tensor algebra that provides a strong foundation for automatically optimizing distributed dense tensor computations [15, 52]. The compiler community has made attempts to automatically optimize sparse tensor sum-product kernels based on asymptotic performance analyses[6, 21]. These systems each target a different execution context and focus on different aspects of optimization. Galley expands on this line of work by targeting a new execution engine, proposing novel optimization techniques, and handling a wider range of tensor programs.

9 Limitations

We are excited to enrich Galley with new optimizations in the future. Currently, Galley lacks support for complex loop structures (e.g., a single outer FOR loop that wraps multiple inner FOR loops), higher order functions (e.g. matrix inversion) or parallelism. However, we believe that these areas could benefit from cost-based optimization. Similarly, Galley does not consider hard memory constraints during optimization, but our use of cardinality bound methods provides an avenue for addressing this in future work.

References

[1] [n. d.]. https://benchmark.clickhouse.com/
[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon

Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zhang. 2016. TensorFlow: A system for large-scale machine learning. *CoRR* abs/1605.08695 (2016). arXiv:1605.08695 http://arxiv.org/abs/1605.08695

[3] Hameer Abbasi. 2018. Sparse: A more modern sparse array library.. In *SciPy*. 65–68.

[4] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman P. Amarasinghe. 2024. Finch: Sparse and Structured Array Programming with Control Flow. *CoRR* abs/2404.16730 (2024). doi:10.48550/ARXIV.2404.16730 arXiv:2404.16730

[5] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. 2023. Looplets: A Language for Structured Coiteration. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2023)*. Association for Computing Machinery, New York, NY, USA, 41–54. doi:10.1145/3579990.3580020

[6] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. 2022. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 269–285. doi:10.1145/3519939.3523442

[7] Einstein Albert, W Perrett, and G Jeffery. 1916. The foundation of the general theory of relativity. *Annalen der Physik* 354, 7 (1916), 769.

[8] Edward C. Anderson, Zhaojun Bai, Jack J. Dongarra, Anne Greenbaum, A. McKenney, Jeremy Du Croz, Sven Hammarling, James Demmel, Christian H. Bischof, and Danny C. Sorensen. 1990. LAPACK: a portable linear algebra library for high-performance computers. In *Proceedings Supercomputing '90, New York, NY, USA, November 12-16, 1990*, Joanne L. Martin, Daniel V. Pryor, and Gary R. Montry (Eds.). IEEE Computer Society, 2–11. doi:10.1109/SUPERC.1990.129995

[9] Yuki Asada, Victor Fu, Apurva Gandhi, Advitya Gemawat, Lihao Zhang, Vivek Gupta, Ehi Nosakhare, Dalitso Banda, Rathijit Sen, and Matteo Interlandi. 2022. Share the Tensor Tea: How Databases can Leverage the Machine Learning Ecosystem. *Proc. VLDB Endow.* 15, 12 (2022), 3598–3601. doi:10.14778/3554821.3554853

[10] Sebastian Baunsgaard and Matthias Boehm. 2023. AWARE: Workload-aware, Redundancy-exploiting Linear Algebra. *Proc. ACM Manag. Data* 1, 1 (2023), 2:1–2:28. doi:10.1145/3588682

[11] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Transactions on Architecture and Code Optimization* 19, 4 (Sept. 2022), 50:1–50:25. doi:10.1145/3544559

[12] Mark Blacher, Julien Klaus, Christoph Staudt, Sören Laue, Viktor Leis, and Joachim Giesen. 2023. Efficient and portable einstein summation in SQL. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–19.

[13] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p22-boehm-cidr20.pdf

[14] Matthias Boehm, Michael Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick Reiss, Prithviraj Sen, Arvind Surve, and Shirish Tatikonda. 2016. SystemML: Declarative Machine Learning on Spark. *Proc. VLDB Endow.* 9, 13 (2016), 1425–1436. doi:10.14778/3007263.3007279

[15] Daniel Bourgeois, Zhimin Ding, Dimitrije Jankov, Jiehui Li, Mahmoud Sleem, Yuxin Tang, Jiawen Yao, Xinyu Yao, and Chris Jermaine. 2024. EinDecomp: Decomposition of Declaratively-Specified Machine Learning and Numerical Computations for Parallel Execution. *arXiv preprint arXiv:2410.02682* (2024).

[16] Walter Cai, Magdalena Balazinska, and Dan Suciu. 2019. Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 18–35. doi:10.1145/3299869.3319894

[17] Jeremy Chen, Yuqing Huang, Mushi Wang, Semih Salihoglu, and Kenneth Salem. 2023. Accurate Summary-based Cardinality Estimation Through the Lens of Cardinality Estimation Graphs. *SIGMOD Rec.* 52, 1 (2023), 94–102. doi:10.1145/3604437.3604458

[18] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2016. Towards linear algebra over normalized data. *arXiv preprint arXiv:1612.07448* (2016).

[19] Rina Dechter. 1999. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence* 113, 1-2 (1999), 41–85.

[20] Kyle B. Deeds, Dan Suciu, and Magdalena Balazinska. 2023. SafeBound: A Practical System for Generating Cardinality Bounds. *Proc. ACM Manag. Data* 1, 1 (2023), 53:1–53:26. doi:10.1145/3588907

[21] Adhitha Dias, Logan Anderson, Kirshanthan Sundararajah, Artem Pelenitsyn, and Milind Kulkarni. 2023. SparseAuto: An Auto-Scheduler for Sparse Tensor Computations Using Recursive Loop Nest Restructuring. *CoRR* abs/2311.09549 (2023). doi:10.48550/ARXIV.2311.09549 arXiv:2311.09549

[22] Yordan Grigorov, Haralampos Gavriilidis, Sergey Redyuk, Kaustubh Beedkar, and Volker Markl. 2023. P2D: A Transpiler Framework for Optimizing Data Science Pipelines. In *Proceedings of the Seventh Workshop on Data Management for End-to-End Machine Learning*. 1–4.

[23] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nat.* 585 (2020), 357–362. doi:10.1038/S41586-020-2649-2

[24] Dong He, Supun Chathuranga Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query Processing on Tensor Computation Runtimes. *Proc. VLDB Endow.* 15, 11 (2022), 2811–2825. doi:10.14778/3551793.3551833

[25] Axel Hertzschuch, Claudio Hartmann, Dirk Habich, and Wolfgang Lehner. 2021. Simplicity Done Right for Join Ordering. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper01.pdf

[26] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Transactions on Graphics* 38, 6 (Nov. 2019), 201:1–201:16. doi:10.1145/3355089.3356506

[27] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Saleh Ashkboos, and Torsten Hoefler. 2023. Sten: Productive and efficient sparsity in pytorch. *arXiv preprint arXiv:2304.07613* (2023).

[28] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 13–28. doi:10.1145/2902251.2902280

[29] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. 2017. The tensor algebra compiler. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 77:1–77:29. doi:10.1145/3133901

[30] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: an abstraction for general data processing. *Proceedings of the VLDB Endowment* 14, 10 (June 2021), 1797–1804. doi:10.14778/3467861.3467869

[31] Arun Kumar, Jeffrey Naughton, and Jignesh M Patel. 2015. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1969–1984.

[32] Chirag Modi, François Lanusse, and Uros Seljak. 2021. FlowPM: Distributed TensorFlow implementation of the FastPM cosmological N-body solver. *Astron. Comput.* 37 (2021), 100505. doi:10.1016/J.ASCOM.2021.100505

[33] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[34] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-end optimization of machine learning prediction queries. In *Proceedings of the 2022 International Conference on Management of Data*. 587–601.

[35] Yeonsu Park, Seongyun Ko, Sourav S. Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1099–1114. doi:10.1145/3318464.3389702

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[37] Pydata. [n. d.]. Pydata/sparse: Sparse multi-dimensional arrays for the PyData ecosystem. https://github.com/pydata/sparse

[38] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1981–1984. doi:10.1145/3299869.3320212

[39] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 519–530. doi:10.1145/2491956.2462176

[40] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. 2016. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*. 3–18.

[41] Maximilian E. Schüle, Thomas Neumann, and Alfons Kemper. 2023. The Duck's Brain: Training and Inference of Neural Networks in Modern Database Engines. *CoRR* abs/2312.17355 (2023). doi:10.48550/ARXIV.2312.17355 arXiv:2312.17355

[42] 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 *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, USA, May 30 - June 1*, Philip A. Bernstein (Ed.). ACM, 23–34. doi:10.1145/582095.582099

[43] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (April 2022), 89:1–89:33. doi:10.1145/3527333

[44] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. 2013. SciDB: A database management system for applications with complex analytics. *Computing in Science & Engineering* 15, 3 (2013), 54–62.

[45] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1083–1098. doi:10.1145/3318464.3380581

[46] Wenbo Sun, Asterios Katsifodimos, and Rihan Hai. 2023. Accelerating Machine Learning Queries with Linear Algebra Query Processing. In *Proceedings of the 35th International Conference on Scientific and Statistical Database Management, SSDBM 2023, Los Angeles, CA, USA, July 10-12, 2023*, Robert Schuler, Carl Kesselman, Kyle Chard, and Alejandro Bugacov (Eds.). ACM, 13:1–13:12. doi:10.1145/3603719.3603726

[47] Gábor Szárnyas, David A. Bader, Timothy A. Davis, James Kitchen, Timothy G. Mattson, Scott McMillan, and Erik Welch. 2021. LAGraph: Linear Algebra, Network Analysis Libraries, and the Study of Graph Algorithms. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, June 17-21, 2021*. IEEE, 243–252. doi:10.1109/IPDPSW52791.2021.00046

[48] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[49] Qing Wang, Matthias Ihme, Yi-Fan Chen, and John R. Anderson. 2022. A TensorFlow simulation framework for scientific computing of fluid flows on tensor processing units. *Comput. Phys. Commun.* 274 (2022), 108292. doi:10.1016/J.CPC.2022.108292

[50] Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow.* 13, 11 (2020), 1919–1932. http://www.vldb.org/pvldb/vol13/p1919-wang.pdf

[51] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2024. From Binary Join to Free Join. *SIGMOD Rec.* 53, 1 (2024), 25–31. doi:10.1145/3665252.3665259

[52] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. 2021. Tensor Relational Algebra for Distributed Machine Learning System Design. *Proc. VLDB Endow.* 14, 8 (2021), 1338–1350. doi:10.14778/3457390.3457399

Received October 2024; revised January 2025; accepted February 2025