

# Galley: Modern Query Optimization for Sparse Tensor Programs

Kyle Deeds  
kdeeds@cs.washington.edu  
University of Washington  
United States

Magda Balazinska  
magda@cs.washington.edu  
University of Washington  
United States

Willow Ahrens  
wahrens@mit.edu  
Massachusetts Institute of Technology  
United States

Dan Suciu  
suciu@cs.washington.edu  
University of Washington  
United States

## ABSTRACT

The tensor programming abstraction has become a foundational paradigm for modern computing. This framework allows users to write high performance programs for bulk computation via a high-level imperative interface. Recent work has extended this paradigm to sparse tensors (i.e. tensors where most entries are not explicitly represented) with the use of *sparse tensor compilers*. These systems excel at producing efficient code for computation over sparse tensors, which may be stored in a wide variety of formats. However, they require the user to manually choose the order of operations and the data formats at every step. Unfortunately, these decisions are both highly impactful and complicated, requiring significant effort to manually optimize. In this work, we present Galley, a system for declarative sparse tensor programming. Galley performs cost-based optimization to lower these programs to a logical plan then to a physical plan. It then leverages sparse tensor compilers to execute the physical plan efficiently. We show that Galley achieves high performance on a wide variety of problems including machine learning algorithms, subgraph counting, and iterative graph algorithms.

## CCS CONCEPTS

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

## KEYWORDS

Query Optimization, Sparse Tensors, Array Programming, Program Optimization

## 1 INTRODUCTION

In recent years, the tensor programming model has become ubiquitous across different fields of computation. Popularized by its use in deep learning, it has been applied to problems such as relational query processing [6, 18, 24], data cleaning [34], graph algorithms [35], and scientific computing [25, 32, 37] among others. This model promises a high-level imperative abstraction which is highly efficient as long as the problem can be posed in terms of tensor (i.e. array) operations. By doing so, it insulates the user from the many low level concerns that are crucial to achieving good performance. This has allowed experts in fields like data science and

machine learning to take advantage of a wide range of hardware infrastructure without having to become experts in high performance computing as well.

Traditional tensor processing frameworks are built on collections of hand-optimized functions, called kernels, which each compute an operation over *dense* tensors [1, 5, 17, 27]. The operation can be simple like a matrix-vector multiplication, or it can be a fusion of multiple semantic operations, like  $A(B + C)$ . However, most data is fundamentally *sparse* (i.e. most entries are a fill value like 0) such as graph data, one-hot encodings, relational data, meshes in physical simulations, sparse neural networks, and others. Even materializing these datasets as dense arrays can be prohibitively costly, so it’s crucial to perform the computation over its compressed, sparse format [23].

To support sparse data, each framework needs to offer many distinct implementations of each operation, one for each combination of input formats. Each input can either be stored densely or in one of many sparse formats, e.g. CSR, CSC, COO. This has led to an explosion of required kernels as the number of operations and formats both continue to grow. Understandably, these frameworks have been unable to keep up with this implementation effort, resulting in spotty coverage for operations over sparse data [21].

**EXAMPLE 1.** Consider logistic regression over  $n$  data points and  $d$  features where  $X \in \mathbb{R}^{n \times d}$  is the feature matrix and  $\theta \in \mathbb{R}^d$  is the parameter vector. Inference is defined as

$$P_i = \sigma\left(\sum_j X_{ij}\theta_j\right) \quad (1)$$

where  $\sigma$  is the sigmoid function. To compute  $Y_i = \sum_j X_{ij}\theta_j$  in the dense case, one could use the ‘matmul’ function from the Numpy library. On these inputs, this function specializes to an efficient, hand-coded implementation of dense matrix-vector multiplication. However, if any combination of  $X$ ,  $\theta$ , and  $Y$  can be sparse, Numpy needs to provide eight distinct implementations of matrix-vector multiplication.

To address this issue, the scientific computing community has adopted *sparse tensor compilers* (STCs) [2, 8, 20, 23, 31]. These compilers take in a high level *imperative* tensor program and, separately, a description of the input tensors’ formats<sup>1</sup>. They automatically produce an efficient kernel implementation in low level code (LLVM, MLIR, etc.). Thus, STCs offer a form of data independence, by allowing the user to separately specify the algorithm and the data layout.

<sup>1</sup>Some systems separate declarative and imperative concerns with a scheduling language. However, the user still controls both aspects.

```

0. # 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

```

**Figure 1: Logistic Regression Impl. in Simplified Sparse Tensor Compiler Language.**

For example, Fig. 1 shows a kernel definition for Ex. 1 written in Finch, an STC language [2]. The Dense/Sparse input formats are specified in line 1 and are separate from the imperative code in lines 5-15.

However, STCs require users to make a series of complex decisions to produce an *efficient* implementation. First, the user needs to break their program into a sequence of aggregations. Suppose that the feature matrix in Ex. 1 is the result of a matrix multiplication,  $X = A \cdot B$ , and Eq. (1) becomes  $Y_i = \sigma(\sum_{jk} A_{ij} B_{jk} \theta_j)$ . The user must choose to sum over  $j$  or  $l$  first, resulting in vastly different runtimes [4, 41]. Then, for each aggregate, the user needs to choose the output format, loop order, and iteration algorithm. Consider Fig 1. The user must choose the output format for the intermediate R (line 3). In this case, she chose a Dense format rather than a Sparse format which would be  $\approx 10\times$  slower. Then, the user chooses the loop order (lines 5-6). 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. Lastly, she 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  $\theta$ . If she chose to iterate through  $\theta$ , each inner loop would scan the entire vector. While STCs allow users to separate the algorithm from the data formats, the user is still responsible for optimizing her program to achieve good performance.

In this paper, we propose Galley, a system for declarative sparse tensor programming. Users write tensor programs in a declarative language based on the Einsum notation, similar to Equation (1), and Galley automatically produces an optimized STC implementation. First, it restructures the program into a sequence of aggregation steps, minimizing the total computation and materialization costs (Sec. 5). Then, Galley optimizes each step by selecting the optimal formats for all intermediate tensors, the loop order, and the access method for each index access (Sec. 6). This is all guided by a system for estimating sparsity via statistics on the input tensors (Sec. 7). 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[22].

In the first phase, Galley rewrites the input program into a sequence of aggregation steps by adapting the variable elimination framework. This reduces a complex rewriting problem to the problem of finding an optimal elimination order. However, this approach has previously only been defined for sum-product queries. To handle *arbitrary sparse tensor algebra* programs, we extend it to non-distributive functions (e.g.  $\sum_i \max(A_{ij}, B_j)$ ), disjunctive functions (e.g.  $\sum_i A_i + B_i = \sum_i A_i + \sum_i B_i$ ), and internal aggregates (e.g.  $\sum_j A_j \sqrt{\sum_k B_{jk}}$ ). In Sec. 9, we show that these optimizations produce up to a  $100\times$  speedup for machine learning algorithms over composite feature matrices. After this phase, our program has been converted to a sequence of aggregates over pointwise expressions, i.e. the Logical Plan dialect in Fig. 4.

Next, Galley’s physical optimizer transforms each aggregate into an efficient STC kernel by choosing the loop order, output format, and merge algorithm. Because sparse outer loops avoid inner iterations, a good loop ordering can significantly improve performance. Galley finds the optimal loop order with a novel combination of branch-and-bound and dynamic programming techniques. Following the modern tensor format abstraction (i.e. the fibertree abstraction), Galley then chooses a format for each dimension of the output by examining both the sparsity and write pattern (random vs sequential)[8, 14, 36]. Lastly, for each loop index, we select one input to iterate over and perform lookups on the others.

All of Galley’s optimizations are guided by a framework for estimating the sparsity of intermediate results from statistics on the inputs. The core of this is a minimal interface for composing statistics about inputs to infer statistics about outputs. This makes incorporating a new kind of sparsity statistic as simple as implementing 5 functions to; 1) collect statistics on inputs 2-3) handle conjunctive and disjunctive point-wise operations 4) handle aggregates and 5) produce sparsity estimates. We demonstrate this by implementing a naive estimator based on uniformity assumptions and a worst-case estimator based on cardinality bounds.

**EXAMPLE 2.** *In Example 1, we assumed that  $X$  is given directly. However, feature matrices are generally assembled from more basic inputs. Suppose  $S_{ipc} \in \mathbb{B}^{n_i \times n_p \times n_c}$  is a boolean sparse tensor where each non-zero entry is a sale,  $i$ , of product,  $p$ , to customer,  $c$ .  $P_{pj} \in \mathbb{R}^{n_p \times d}$  and  $C_{cj} \in \mathbb{R}^{n_c \times d}$  are smaller matrices holding numeric features about products and customers. The feature dimension, indexed by  $j$ , represents both product and customer features, but each feature comes from either  $P_{pj}$  or  $C_{cj}$ . So, each column is non-zero in either  $P$  or  $C$ , and they are concatenated by addition.  $X$  is now defined as  $X_{ij} = \sum_{pc} S_{ipc} (P_{pj} + C_{cj})$ . Galley’s logical optimizer can take advantage of  $X$ ’s structure by pushing  $\theta$  down into its definition,*

$$Y_i = \sigma \left( \sum_{jpc} S_{ipc} P_{pj} \theta_j + \sum_{jpc} S_{ipc} C_{cj} \theta_j \right)$$

*We can then push down the summations,*

$$Y_i = \sigma \left( \sum_{pc} \left( S_{ipc} \sum_j (P_{pj} \theta_j) \right) + \sum_{pc} \left( S_{ipc} \sum_j (C_{cj} \theta_j) \right) \right) \quad (2)$$

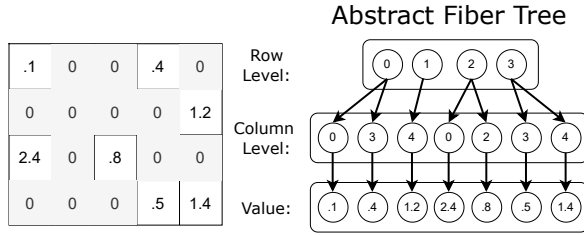


Figure 2: Fibertree Format Abstraction

By doing this, Galley only materializes vector intermediates rather than the full feature matrix. On similar examples, Sec. 9 shows that this can result in up to 100× faster execution.

After logical optimization, Galley’s physical optimizer produces a single STC kernel for each aggregate. For  $\sum_j P_{pj}\theta_j$ , it would produce,

```

I1 = Dense()
FOR p=_
  FOR j=_
    I1[p] += P[p::iter,j::iter]*θ[j::lookup]
  END
END
END

```

**Contributions** We claim the following contributions in this work,

- We present Galley, a system for declarative sparse tensor programming (Sec.4). Galley automatically translates high-level sparse tensor programs to efficient imperative programs targeting a sparse tensor compiler.
- Galley supports a highly expressive language for sparse tensor algebra with arbitrary algebraic operators, aggregates within expressions, and multiple outputs.
- Galley is the first system to perform cost-based logical optimization, redistributing aggregates and reorganizing the pointwise statements in the program. (Sec.5)
- Galley is the first system to perform cost-based physical optimization to determine loop orders, tensor formats, and merge algorithms for each dimension (Sec.6)
- We propose a minimal interface for sparsity estimation and demonstrate it by implementing two estimators (Sec.7)
- We evaluate Galley on several workloads and show that it is up to **100x** faster than hand-optimized kernels for mixed dense-sparse workloads and up to **100x** faster than a state of the art database for highly sparse workloads.

## 2 RELATED WORK

Our work differs from other proposals on cost-based optimization for tensor processing due to its targeting of STCs and its expressive input language. SystemDS, formerly SystemML, is a system for end-to-end machine learning over matrices and tabular data [7, 10, 11]. It primarily takes in 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[38]. Their core insight was that LA rewrites, which always match and

produce 0-2D expressions, are not sufficient. Optimal rewrites need to pass through higher order intermediate expressions. Other work translated sum-product expressions to SQL to leverage highly efficient database execution engines[9]. 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 which provides a strong foundation for automatically optimizing distributed dense tensor computations [40]. In the compilers community, there have been attempts to automatically optimize sparse tensor kernels based on asymptotic performance analyses[4, 16]. 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.

## 3 BACKGROUND

### 3.1 Tensor Index Notation

The input to our system, Galley, is an extended version of Einsum notation which we call *tensor index notation*. Traditional Einsum notation allows for a single summation wrapped around a multiplication operation. For instance, you can describe triangle counting in a graph where  $E_{ij}$  is the adjacency matrix with the following Einsum 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 of 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 they 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 otherwise.

$$L_i = \sigma\left(\sum_j X_{ij}\theta_j\right) > .5$$

$$V_i = L_i \cdot \max_{jk}(E_{ij}E_{jk}E_{ik})$$

Tensor compilers like Halide, TACO, and Finch each build off of similar core notations, adding additional structures like FOR-loops to allow users to specify algorithmic choices[2, 23, 29]. Crucially, the vast majority of operations in array programming frameworks like Numpy can be expressed as operations in tensor index notation. So, while we focus on this notation in this paper, traditional tensor workflows can be captured and optimized in this framework.

### 3.2 Sparse Tensor Compilers

In the last decade, the compilers community has developed a series of sparse tensor compilers and shown that they can produce highly efficient code for sparse tensor computations. We use this work as our execution engine, so we take a moment here to explain their important concepts.

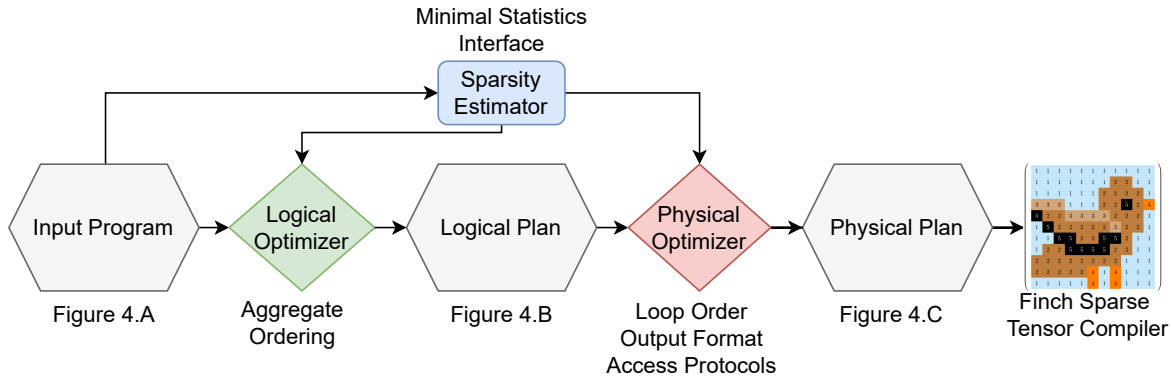


Figure 3: Galley Overview

*Tensor Formats.* There are many different ways to represent sparse tensors, and the best 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[23, 36]. In this formalism, a tensor format is a nested data structure like the one in Fig. 2. Each layer stores the non-fill (e.g. non-zero) indices in a particular dimension, conditioned on the earlier dimensions, and pointers to the next dimension’s non-fill indices. These layers can be represented in any format that allows for iteration and lookup. In this work, we consider sorted lists, hash tables, bytemaps, and dense vectors, and they each have different performance characteristics in terms of iteration, lookup, and memory footprint. For example, compressed sparse row (CSR) is a common format for sparse matrices. It stores the row dimension as 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 that 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) which 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  $i$  which map to non-zero entries in  $X$  and  $\theta$  need to be 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  $\theta$ . Considering the illustration in Fig. 2, we can see how this is simply co-iteration over the top levels of their formats. The inner loop then iterates over the  $j$  indices which are non-zero in  $X[i, \_]$ , i.e. the non-zero columns that occur in each row.

### 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 := Tns[PIdx...]   Alias := Name[PIdx...]
PIdx := Idx::Protocol
```

Figure 4: Query Plan Dialects

*Merge Algorithms.* Once the compiler has determined which tensors’ non-zero indices need to be merged to iterate over a particular index, there are several algorithms it can apply. All formats allow for both ordered iteration and lookup operations, so one algorithm is to iterate through the indices of all inputs, similar to a merge join. This is highly efficient per operation. However, it’s linear in the total size of all inputs, even if one is much smaller than the others. Another method is to iterate through one input’s level and lookup that index in the others. In this work, we generally take the latter approach, as described in Sec. 6.3. 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*[3].

## 4 GALLEY OVERVIEW

In this section, we provide a birds-eye view of Galley and show how an input program is transformed, first to a logical plan then to a physical plan then to an STC program, as illustrated in Fig. 3. The first transformation from input program to logical plan is handled by Galley’s logical optimizer, which extends the variable elimination framework to break the program down into a sequence of aggregations. The second transformation, from logical plan to physical plan, is performed by Galley’s physical optimizer, which determines the loop order, tensor formats, and merge algorithms for each logical query. 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 and show how our example program, logistic regression, would be transformed through these steps.

The input program dialect is equivalent to the tensor index notation defined in Sec. 3.1. Pointwise functions like  $A_{ij} * B_{jk}$  are represented with `Map`. Aggregates like  $\sum_i$  are denoted by `Agg`. Each assignment is a `Query`, and previous assignments are referenced with an `Alias`. Our logistic regression example from Eq. (1) is defined in this dialect as follows,

```
Query(P, Map( $\sigma$ , Agg(+, j, Map(*, X[i, j],  $\theta[j]$ ))))
```

Note that this notation is compatible with array APIs like Numpy which don’t 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. Also, note that aggregates can be over multiple indices, e.g. `Agg(+, i, j, k, ...)`.

### 4.1 Logical Plan

The first task in our optimization pipeline, handled by the logical optimizer, is to break down the input program into a sequence of simple aggregates. This is enforced by converting the input program to a logical plan. 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, it must minimize the total cost of all queries in the logical plan.

As an example, our logistic regression program above can be translated into the following logical plan,

```
Query(R, Agg(+, j, Map(*, X[i, j],  $\theta[j]$ )))
Query(P, Agg(no-op, Map( $\sigma$ , 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.

### 4.2 Physical Plan

Given the logical plan, Galley’s physical optimizer determines the implementation details needed to convert each logical query to a

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 verbose and 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],
                                      $\theta[j::lu]$ ))), i, j)
Query(P, Mat(dense, i, Map( $\sigma$ , 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  $\theta$ , and accumulating 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.

### 4.3 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, we return the tensors requested in the input program by looking them up in this dictionary. In Sec. 8, we describe additional optimizations that improve this workflow by doing just-in-time physical optimization and common sub-expression elimination.

## 5 LOGICAL OPTIMIZER

Given the plan dialects above, we now describe the logical optimizer, which takes in an input program and outputs a semantically equivalent *logical plan*. Specifically, the logical optimizer converts each query in the input program to a sequence of logical queries where the last one produces the same output as the input query. There are many valid plans, and the optimizer searches this space to identify the cheapest one. In this section, we briefly define “cheapest” in this context before outlining the complex space of valid logical plans that are considered. Lastly, we explain the algorithms that we use to perform this search.

### 5.1 Canonicalization & Pointwise Distributivity

The first step in our logical optimization is to canonicalize the input program with a few simple rules that we apply exhaustively; 1) merging nested `Map` operators 2) merging nested `Agg` operators 3) lifting `Agg` operators above `Map` operators when possible 4) renaming indices to ensure uniqueness. This compresses the input program and makes our reasoning simpler later by ensuring that operator boundaries are semantically meaningful.

Next, we consider whether to distribute point-wise expressions. This may or may not result in a better plan because it both makes operations more sparse and produces a larger expression.

EXAMPLE 3. Consider the following expression 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 could be 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 point-wise function not a matrix multiplication.

Because apply pointwise distributivity can produce asymptotic performance improvements, it cannot be ignored. To address this, we perform logical optimization for both the distributed expression and the original and use the cheaper one.

## 5.2 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 on 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  $\text{Agg}$ ,  $\text{nnz}(\text{Agg})$ , and the latter corresponds to the size of the tensor represented by the  $\text{MapExpr}$  within,  $\text{nnz}(\text{MapExpr})$ . We assume that the inputs are in memory, hence there is no cost for reading inputs from disk. We then performed a simple regression to associate each of these costs with a constant, and we add them to produce our overall cost,  $c$ ,

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

To estimate  $\text{nnz}(\text{Agg})$  and  $\text{nnz}(\text{MapExpr})$ , we introduce a sparsity estimation framework in Sec. 7.

## 5.3 Variable Elimination

The core of our logical optimizer is the variable elimination (VE) framework. With this view, a logical plan for an input query is defined by an order on the indices being aggregated over, i.e. an *elimination order*. Given this order, we can construct a valid logical plan in the following way. Iterating through the elimination order one index at a time, we (1) identify the minimal sub-query needed to aggregate it out of the expression, (2) create a new logical query representing the result of that sub-query, 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,

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

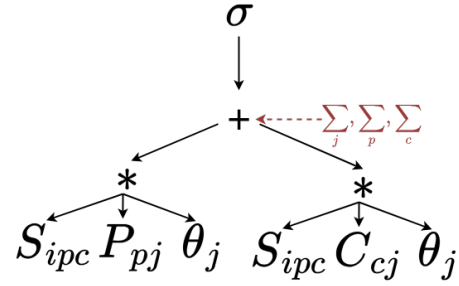


Figure 5: Annotated expression tree for logistic regression,  $\sigma(\sum_{jpc}(S_{ipc}P_{pj}\theta_j + S_{ipc}C_{cj}\theta_j))$

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 over sum-product queries, we need to handle complex trees of pointwise operators and aggregates. This makes the identification of minimal sub-queries challenging as we need to 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 1) how we identify minimal sub-queries and 2) our search algorithm for finding the optimal elimination order.

## 5.4 Identifying Minimal Sub-Queries

In this section, we show how to identify the minimal sub-queries (MSQ) needed to aggregate over an index. In sum-product expressions, the MSQ is just the sum over that index wrapping the product of the tensors that are indexed by it. However, it’s not immediately clear how to do this for more complex input programs which include an arbitrary mix of operators and aggregates. Fortunately, we show that identifying the MSQ corresponds to a careful traversal down the *annotated expression tree*, examining the algebraic properties of the operation at each node to determine how to proceed.

**Annotated Expression Tree** The annotated expression tree (AET) is constructed by examining the nested structure of  $\text{Agg}$ ,  $\text{Map}$ ,  $\text{Input}$ , and  $\text{Alias}$  nodes in the input query. We start by removing all  $\text{Agg}$  nodes and annotating their inner expressions with  $(\text{Idx}, \text{Op})$  for each index being aggregated over. We then replace all  $\text{Map}$  nodes with their operator to get the final tree where every internal node is a function and every leaf is either an  $\text{Input}$  or an  $\text{Alias}$ .

EXAMPLE 5. Fig. 5 shows the annotated expression tree for our logistic regression example after distributing the multiplication as in Eq. (2). 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 addition operator, as denoted by the red arrow. Then, each multiplication has three children which are each indexed input tensors.

Given the AET, we can identify the minimal sub-queries for a particular index by starting at the node where it’s annotated and traversing downwards according to the algebraic properties of each internal node. We now describe the traversal rules for functions which are distributive, non-distributive, and commutative with respect to the aggregation operator.

**Distributive Functions** When a function which distributes over the aggregate is reached (e.g. multiplication which distributes over summation), we examine how many of the children contain the current index. Each child is a subtree of the AET, and it contains the index iff it contains a tensor that is indexed by it. If one child contains the index, we simply traverse down that child’s branch, i.e. we factor the other children out of the aggregate. When multiple children contain the index, we can’t go any further down the tree, so we wrap the sub-tree rooted at that node in an aggregate and return it as our MSQ. If the function is commutative and associative, we are slightly more precise and only include the children which 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, if we have the expression  $\sum_i A_i + B_i$ , we can transform it into  $\sum_i A_i + \sum_i B_i$ . For all children which contain the index, we add the result of traversing down its branch to the MSQ and replace it with an alias to the result. If a child doesn’t contain the index, then we need to account for the repeated application of the aggregate function. To do this, we identify the function  $g(x, n) = f(x, \dots, x)$  which represents the repeated application of our aggregate function  $f$ . When  $f$  is addition,  $g$  is multiplication. When  $f$  is idempotent,  $g(x, n) = x$ . We then wrap each non-index child in a Map with function  $g$  and the size of the index’s dimension as the second child.

**Blocking Functions** A function which doesn’t 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 our MSQ. 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  $\sqrt{\cdot}$  is a blocking function.

**Discussion** Galley builds on and extends the FAQ framework for optimizing conjunctive queries with aggregation[22]. The FAQ framework explored the optimization of queries with the following form where each  $\bigoplus^{(i)}$  is either equal to or forms a semi-ring with  $\otimes$ ,

$$\bigoplus_{v_1}^{(1)} \dots \bigoplus_{v_k}^{(k)} F_{V_1}^1 \otimes \dots \otimes F_{V_k}^k$$

While this captures many important problems, it lacks the flexibility needed to support a general tensor processing system. For example,

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 within the pointwise expression. Similarly, our logistic regression example,  $\sigma(\sum_j X_{ij} \theta_j)$  cannot be expressed as an FAQ because the aggregate occurs within the  $\sigma$ . Galley’s logical optimizer extends this framework by accommodating arbitrary pointwise function composition and arbitrary placement of aggregates within expressions.

## 5.5 Restricted Elimination Orders

Lastly, before we can search for the optimal elimination order, we need to define the space of *valid* elimination orders. Depending on the structure of the input, the order in which indices can be eliminated might be restricted. This is due to two issues: (1) non-commutative aggregates and (2) aggregate placement. The former is when an aggregate wraps another aggregate which it doesn’t commute with. For example, if the expression is  $\max_i \sum_j A_{ij}$ , we have to perform the summation before handling the maximum because  $\max$  and  $\sum$  do not commute. The latter issue arises 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 which needs to be respected when we enumerate elimination orders.

## 5.6 Search Algorithm

We have now simplified the complicated issue of high level optimization to the discrete problem of choosing an optimal order on the aggregated index variables. So, in this section, we present two algorithms for searching for that optimal order using the tools described above.

**Greedy** The greedy approach simply chooses the cheapest index to aggregate at each point. It does this by finding the minimal sub-query for each index and computing its cost. For the cheapest index, it’s minimal sub-query is removed from the expression and appended to the logical plan, then it is replaced in the remaining query 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 is broken into two steps. The first step uses the greedy algorithm to produce an upper bound on the cost of the overall plan. The second step performs a dynamic programming algorithm. In this algorithm, the keys of the DP table are unordered sets of indices, and the values are a tuple of ordered lists of indices, remaining queries, and costs. We initialize the table with the empty set and a cost of zero. At each step, we iterate through the entries of the DP table and attempt to aggregate out another index. We use the cost bound from the first step to prune entries from the DP table whose cost is greater than the bound which is valid because costs monotonically increase as more indices are added to the set. At the end, we return the index order associated with the full set of indices.

## 6 PHYSICAL OPTIMIZER

Each query in the logical dialect roughly corresponds to a single loop nest and materialized intermediate. However, there are still several decisions which need to be made about the way the kernel is computed; (1) the loop order over the indices, (2) the format (i.e. layout) of the result, and (3) the protocol (i.e. search algorithm) for accessing each index of each input. The physical optimizer makes these decisions.

### 6.1 Loop Order

The loop order of a tensor kernel determines which order the inputs are accessed in. A good order will result in early pruning of iterations due to early intersection of sparse inputs. Intuitively, this is similar to selecting a variable order for a worst-case optimal join algorithm.

*Cost Model.* The cost of a loop order is equal to how many iterations each level of the loop nest incurs.

EXAMPLE 6. Consider matrix chain multiplication over three sparse matrices,  $A$ ,  $B$ , and  $C$ .

$$D[il] = \sum_{jk} A[ij] * B[jk] * C[kl] \quad (3)$$

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. Then, the loop order  $ijkl$  is significantly more efficient than  $lkji$ . In the former, the first two loops, over  $i$  and  $j$ , only incur a single iteration because they are bounded by the size of  $A$ . The third and fourth occur 5 and  $5^2$  times 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.

The last piece of our cost model is the cost of transposition. If an input's index order is not concordant with the loop order, it must be transposed before the query can be executed. This imposes a cost that is linear in that input's size.

*Optimization Algorithm.* To optimize the loop order, we combine this cost model with a branch-and-bound, dynamic programming algorithm. In the first pass, it 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 we use 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 represent the loops that have been iterated so far, and the latter represents the inputs which will need to be transposed based on the optimal prefix.

### 6.2 Intermediate Formats

Once the loop order has been determined, we select the optimal format for each query's output. Because we operate within the fibertree abstraction, this means that we select a format for each index of the output (e.g. dense vector, hash table, etc.). There are two factors which affect this decision: (1) the kind of writes being

performed (sequential vs random) (2) the sparsity of the resulting tensor. The former is important because many formats (e.g. sorted list formats) only allow for sequential construction. This means that they can only be applied if the indices of the output form a prefix of the loop order. The latter factor balances the fact that denser formats tend to be more efficient due to their memory locality and simplicity while sparser formats are asymptotically better when dealing with significantly 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, we first determine the sparsity at this index level and use our cutoffs to determine which category of formats to consider. Then, we check whether we are performing sequential or random writes and select the most efficient format that supports our write pattern.

### 6.3 Merge Algorithms

The final decision for the physical optimizer to make is the algorithm it will use to perform each loop's intersection. While there are some more complex strategies, we adopt a minimal approach and select a single input to iterate over for each loop. The remaining inputs are then probed into. We make this selection by estimating the number of non-zero indices that each input has, conditioned on the indices in the outer loops. This is similar to the approach taken in [39] for optimizing WCOJ.

## 7 SPARSITY ESTIMATION

In this section, we describe how our system performs the sparsity estimation that guides our logical and physical optimizers. We start by discussing the subtle correspondence between sparsity estimation and cardinality estimation. Then, we describe a minimal interface for sparsity estimation inspired by this correspondence. Lastly, we discuss two implementation of this framework; the uniform estimator and the chain bound.

### 7.1 Sparsity & 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  $R_A(I, J)$  and  $R_B(J, K)$  be relations which store the indices of their non-zero entries. Suppose we're performing matrix multiplication,

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

Then, the number of non-zero values in  $C$  is precisely equal to the size of the following conjunctive query,

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

The correspondence is due to 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  $B$ . Now, 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.  $D_i = \{1, \dots, n_i\}$ . This allows us to represent the index relation of the output as,

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

Lastly, we can translate aggregations to the tensor setting as projection operations. If we have the following statement,

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

We can express the non-zeros entries of  $C$  as,

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

## 7.2 The Sparsity Statistics Interface

We use our statistics interface to annotate an expression with stats objects at every node of the AST in a bottom-up fashion. Each stats object then represents the sparsity patterns of the intermediate tensor output from that node. Surprisingly, to support sparsity estimation over the varied workloads and arbitrary operators of tensor algebra, we only need to implement a few core functions; 1) A constructor which produces statistics from a materialized tensor. This produces statistics for Input and Alias nodes. 2) A function for annihilating Map nodes (i.e. those whose children’s fill values are the annihilator of its pointwise function) which merges the children’s statistics. 3) A function for non-annihilating Map nodes 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. 5) An estimation procedure that estimates the number of non-fill entries based on statistics about a tensor.

## 7.3 Supported Sparsity Estimators

**7.3.1 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 relies on the assumption that these entries are uniformly distributed across the dimension space. This corresponds to System-R’s cardinality estimator with the added assumption that the size of the index attribute’s active domain is equal to the size of the dimension [30].

*Constructor.* Given a tensor  $A_{i_1, \dots, i_k} \in \mathbf{R}^{n_{i_1} \times \dots \times n_{i_k}}$ , we simply count the non-fill values in the tensor,  $nnz(A)$ , and note the dimension sizes  $n_{i_1}, \dots, n_{i_k}$ .

*Map (Annihilating).* To handle an annihilating pointwise operation, we calculate the probability that a point in the output was non-fill in all inputs, then multiply this with the dimension space of the output. For a set of inputs  $A_{I_1}^{(1)} \dots A_{I_l}^{(l)}$  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 a non-annihilating pointwise operation, we calculate the probability that an entry in the output was *fill* in all inputs. Then, we take the compliment to get

the probability that it was non-fill in all inputs and multiply this with the output dimension space. Using the notation from above,

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

*Aggregate.* Given an input tensor  $A_I$  which we are aggregating over the indices  $I'$ , we compute 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 wasn’t fill.

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

*Estimate.* The estimation function simply returns the cardinality statistic stored about the current tensor.

**7.3.2 Degree Statistics & The Chain Bound.** We keep degree statistics as the default in Galley, and we use them to compute upper bounds on the number of non-fill entries in intermediate expressions. A degree statistic, denoted  $D_A(X|Y)$ , stores the maximum number of non-fill entries in the  $X$  dimensions conditioned on the  $Y$  dimensions for a tensor  $A$ . For example, if you have a matrix  $A_{ij}$ , then  $D_A(i|j)$  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 which has been shown to produce efficient query plans in the relational setting [12, 15, 19].

*Constructor.* We first compute the boolean tensor representing the input’s sparsity pattern. Then, to calculate each degree statistic, we sum over the  $X$  dimensions and take the maximum over the  $Y$  dimensions. The set of degree statistics for a tensor  $A_j$  is denoted  $\mathcal{D}_{A_j}$

*Map (Annihilating).* Annihilating map operations function as conjunctive queries with respect to the sparsity patterns of the inputs. Therefore, any degree statistics that are valid for an input are also valid about the output, so we compute the statistics about the output,  $C$ , from the inputs  $A_{I_1}^{(1)}, \dots, A_{I_k}^{(k)}$  by a union,

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

*Map (Non-Annihilating).* In this case, we need to be more careful to ensure that we maintain our upper bounds. First, we extend the degree constraints from each input to cover the full set of indices. For example, if we have  $DC_X(i|j)$  and want to extend it to the dimension  $k$ , then we compute  $D_X(ik|j) = DC(i|j) * n_k$ . Then, we compute degree statistics about the output,  $C$ , from the inputs  $A_{I_1}^{(1)}, \dots, A_{I_k}^{(k)}$  by addition,

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

*Estimator.* We calculate our upper bound (eq. perform sparsity estimation) using the breadth-first search approach described in [13]. Intuitively, each set of indices forms a node in the graph, and each degree constraint is a weighted edge from  $Y$  to  $X$ . Our search begins with the empty set, then we use breadth first search to find

the shortest weighted path to the full set of indices  $I$ . The product of the weights along this path bound the number of non-zeros in the result.

## 8 ADDITIONAL OPTIMIZATIONS

In this section, we outline a few additional techniques we apply to improve the optimization and execution of tensor programs.

### 8.1 Just-In-Time Physical Optimization

When optimizing a large program, estimates of sparsity become less reliable because they require more steps of inference on top of grounded inputs. This poses a challenge for the physical optimizer as it estimates the cost of different loop orders, formats, and access protocols. To ameliorate this, Galley performs a very simple form of adaptive optimization by waiting to perform physical optimization on each logical query until all of its aliases have already been executed. This allows it to calculate the size of prior intermediate results and update the relevant statistics before performing physical optimization. This makes that cost estimates in that optimization more accurate and produces a better physical plan.

### 8.2 Common Sub-Expression Elimination

We take a straightforward approach to avoiding redundant computation. Before executing each physical query, we first canonicalize the right hand side, hash it, and compute its hash value. We use this to check a cache to see if it has been executed already. If so, we immediately return the value from the cache rather than re-executing the query. Similarly, after computing a query, we store the result in this cache for later reuse.

## 9 EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness of our optimizer on a variety of workloads: (1) machine learning algorithms, (2) subgraph counting, and (3) breadth-first search. Compared to hand-optimized solutions and alternative approaches, Galley is highly computationally efficient while only requiring a concise, declarative input program from the user. Overall, we show that Galley:

- Performs logical optimizations resulting in up to **100×** faster ML algorithms over joins as compared to hand optimized implementations
- Has a mean optimization time of less than **0.15** seconds for all subgraph counting workloads with up to **100×** faster median execution than DuckDB
- Selects optimal tensor formats for intermediates, beating both the dense and sparse formats for 3/5 graphs

*Experimental 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://github.com/kylebd99/Galley>. We used the sparse tensor compiler Finch for execution, and all methods are executed using a single thread. Unless otherwise stated, Galley uses the chain bound described in Sec. 7.3.2 for sparsity estimation. Experiments for all methods are run three times and the minimum execution time is reported. This removes the compilation overhead which we address separately in Fig. 9.

## 9.1 Machine Learning Algorithms

To explore end-to-end program optimization, we experiment with simple machine learning algorithms over joins, represented entirely in tensor algebra. For this, we use the TPC-H benchmark and consider two join queries, a star query and a self-join query. First, we perform a star join over the line items table to bring together features about suppliers, parts, orders, and customers. This is expressed as follows where  $L, S, P, O$ , and  $C$  are tensors representing the lineitems, 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.

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

Given these definitions, we consider a range of ML algorithms; 1) linear regression inference 2) logistic regression inference 3) covariance matrix calculation and 4) neural network inference. As a comparison point, we also implement two versions of each of these using the Finch compiler. The dense version uses a fully dense matrix to represent the feature matrix whereas the sparse version uses a sparse level for the features to compress the one-hot encoded features. Fig. 6 shows that Galley is 2 – 100× faster for all but one experiment. This performance improvement is derived from Galley pushing the computation into the definition of  $X$ . For example, linear regression inference is fundamentally a matrix-vector multiplication, and Galley pushes the vector multiplication all the way down to the feature matrices  $S, P, O$ , and  $C$ . Note that we do not provide a dense implementation for the self-join query because it runs out of memory due to the highly sparse  $i_1, i_2$  dimensions.

## 9.2 Subgraph Counting

To test Galley’s performance on large, sparse problems, we implement a few common sub-graph counting benchmarks. The conversion from sub-graph counting to sparse tensor algebra is straightforward. Suppose you are counting the occurrences of  $H(V, E)$  in a data graph  $G$  with adjacency matrix  $M$ , then we can represent the count as,

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

To handle vertex labels, 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"[26, 33]. We restrict the latter benchmark to query graphs with up to 8 vertices, hence the "\_lite" suffix. Because this is a relational workload, we compare with DuckDB, a modern OLAP database [28]. To tease apart the impact of logical optimization from physical optimization and our use of Finch, we provide a version

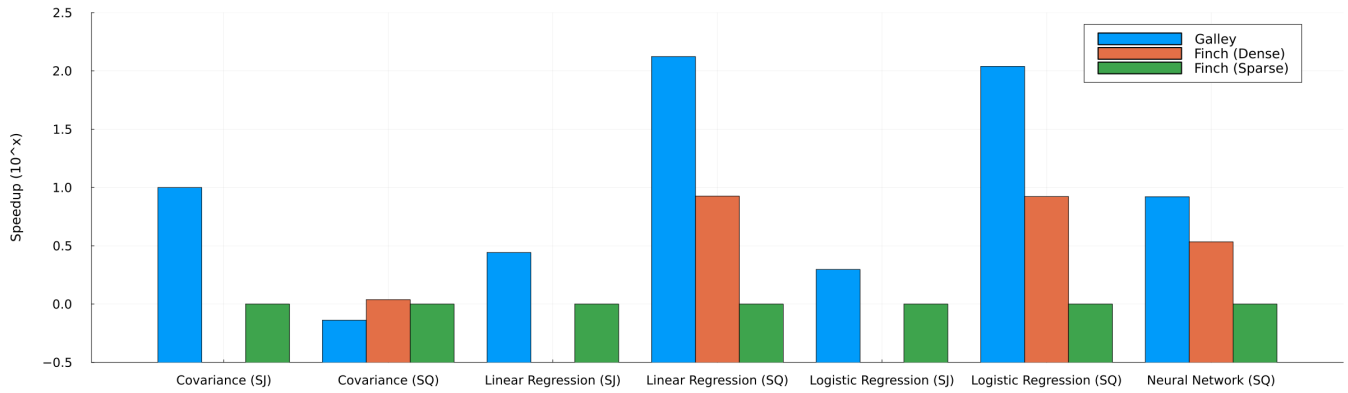


Figure 6: Machine Learning Algorithms Over Joins

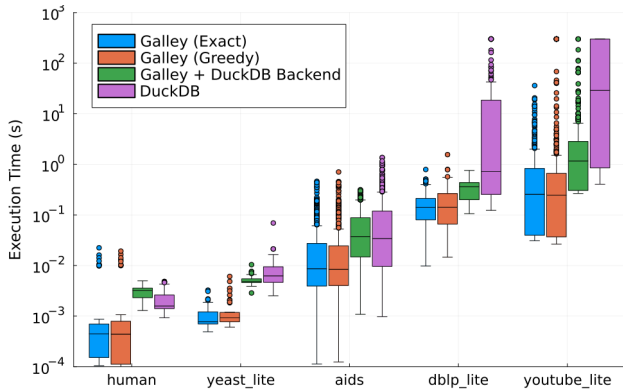


Figure 7: Subgraph Counting Execution Time

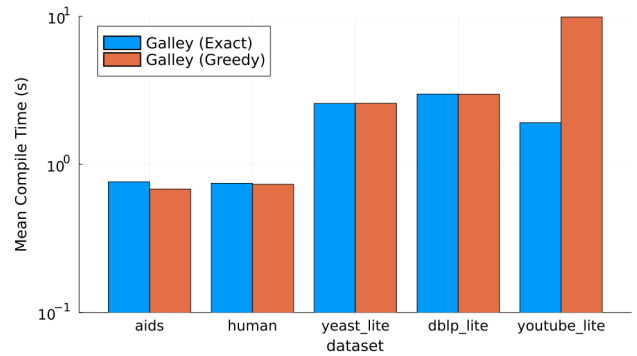


Figure 9: Subgraph Counting Compilation Time

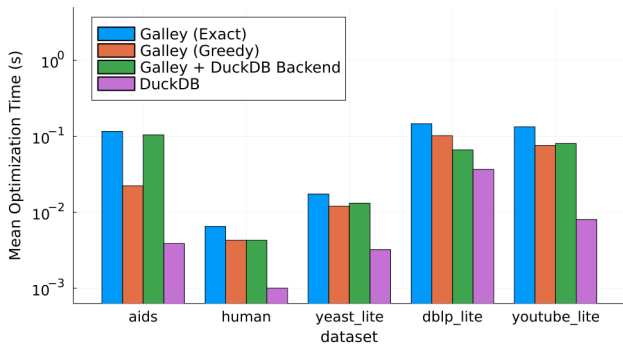


Figure 8: Subgraph Counting Optimization Time

of Galley that executes each logical query with a SQL query run on DuckDB. Lastly, we provide results for the greedy logical optimizer as well.

In Fig. 7, we see median execution times which are up to 100× lower when comparing Galley to DuckDB. When using Galley’s physical optimizer with Finch as opposed to using DuckDB as an execution engine, we still see up to 10× lower median execution

time. The former shows the benefit of Galley’s logical optimizer and the variable elimination framework. The latter demonstrates the benefits of using STCs which enable a wide range of formats and produce highly efficient code. Further, DuckDB hits the 300 second timeout on 196 out of 400 queries in the youtube\_lite benchmark. This is reduced to 5 and 18 timeouts when using Galley with a DuckDB and Finch execution engine, respectively. The greedy optimizer performs similarly to the exact optimizer on these workloads with the latter providing improvements on the slowest queries. Fig. 8 shows the mean optimization time for each of the methods on each of the workloads. Galley has a mean optimization time of less than .15 seconds across all workloads, approaching the time taken by the highly efficient DuckDB optimizer.

Lastly, because it performs compilation using Finch at runtime, Galley incurs a compilation latency when it first requests each tensor kernel implementation. Fortunately, these kernels are cached automatically by Finch, reducing this cost when workloads repeatedly use similar kernels. We show the mean compilation time for each subgraph workload in Fig. 9. On the simpler workloads which often reuse kernels, this cost is minimal. However, the more complex workloads both reuse kernels less and require compiling more complex kernels, significantly increasing compilation time. This

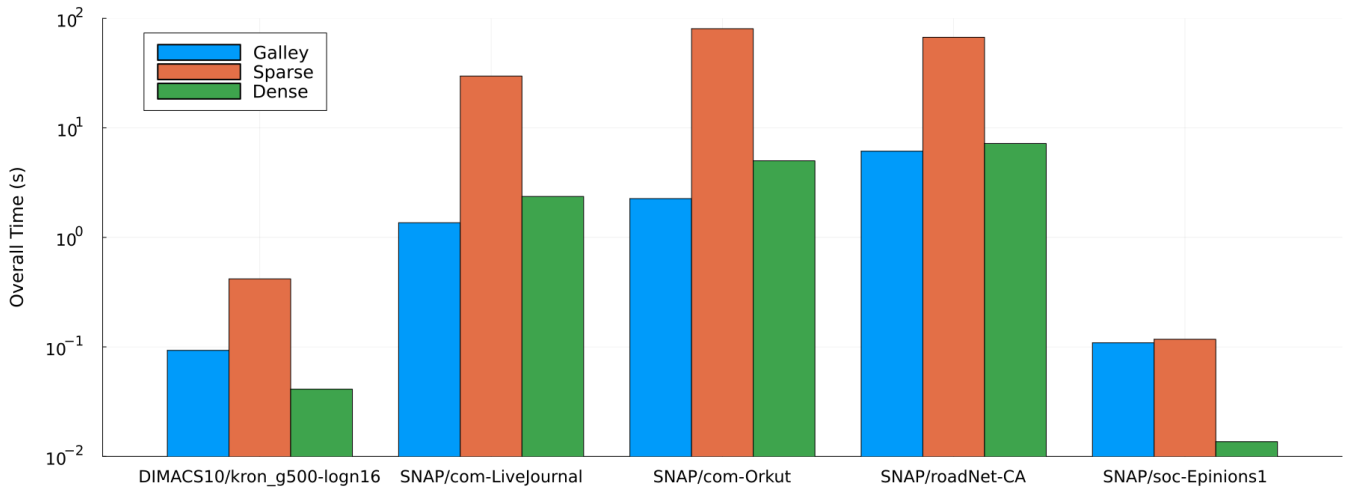


Figure 10: BFS Runtime

suggests a need for faster STC compilation or the use of an interpreted engine for complex queries.

### 9.3 Breadth First Search

To demonstrate the importance of selecting optimal formats, we implement a simple push-based breadth first search algorithm using Galley and different hand coded Finch implementations. Both systems are provided with a single iteration at a time and the total execution time across all iterations is reported. This means that the core optimization question is how to represent the vector of visited vertices and the vector of frontier vertices. The former grows monotonically over the course of the algorithm while the number of non-zeros in the latter forms a curve, peaking in the middle iterations. We provide two implementations of Finch 1) using a sparse vector for both intermediates and 2) using a dense vector for both intermediates. Fig. 10 shows that Galley is faster than both dense and sparse approaches for 3 of the 5 graphs and is competitive for all graphs. Note that this includes the optimization time for Galley.

## 10 CONCLUSION & LIMITATIONS

In this paper, we presented Galley, our system for declarative sparse tensor programming. We described and then demonstrated how it optimizes high-level program structure with its logical optimizer and how it lowers that program to an efficient implementation with its physical optimizer. These decisions are guided by sparsity estimates of intermediate expressions, and we show that these estimates can be computed for arbitrary tensor algebra programs by implementing a minimal 5-function interface. Lastly, we present an experimental evaluation of this work on ML algorithms over structured feature data, sub-graph counting, and breadth-first search.

There are a few optimizations which we are excited to bring to Galley in the future. Currently, it does not support complex loop structures (e.g. a single outer FOR loop which wraps multiple inner FOR loops) or parallelism. Both of these areas can benefit from cost-based optimization, and we're excited to explore them in future

work. Similarly, we do not consider memory constraints during the optimization process, but our focus on cardinality bounds provides an exciting avenue for approaching this in the future.

## REFERENCES

- [1] 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](https://arxiv.org/abs/1605.08695) <http://arxiv.org/abs/1605.08695>
- [2] 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). <https://doi.org/10.48550/ARXIV.2404.16730> [arXiv:2404.16730](https://arxiv.org/abs/2404.16730)
- [3] 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. <https://doi.org/10.1145/3579990.3580020>
- [4] 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. <https://doi.org/10.1145/3519939.3523442>
- [5] Edward C. Anderson, Zhaojun Bai, Jack J. Dongarra, Anne Greenbaum, A. McKeeney, 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. <https://doi.org/10.1109/SUPERC.1990.129995>
- [6] 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. <https://doi.org/10.14778/3554821.3554853>
- [7] Sebastian Baunsgaard and Matthias Boehm. 2023. AWARE: Workload-aware, Redundancy-exploiting Linear Algebra. *Proc. ACM Manag. Data* 1, 1 (2023), 2:1–2:28. <https://doi.org/10.1145/3588682>
- [8] 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. <https://doi.org/10.1145/3544559>

- [9] 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.
- [10] 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>
- [11] 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. <https://doi.org/10.14778/3007263.3007279>
- [12] 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. <https://doi.org/10.1145/3299869.3319894>
- [13] 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. <https://doi.org/10.1145/3604437.3604458>
- [14] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format abstraction for sparse tensor algebra compilers. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 123:1–123:30. <https://doi.org/10.1145/3276493>
- [15] 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. <https://doi.org/10.1145/3588907>
- [16] 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). <https://doi.org/10.48550/ARXIV.2311.09549> arXiv:2311.09549
- [17] 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. <https://doi.org/10.1038/S41586-020-2649-2>
- [18] 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. <https://doi.org/10.14778/3551793.3551833>
- [19] 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](http://cidrdb.org/cidr2021/papers/cidr2021_paper01.pdf)
- [20] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédéric 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. <https://doi.org/10.1145/3355089.3356506>
- [21] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Saleh Ashkboos, and Torsten Hoefer. 2023. Sten: Productive and efficient sparsity in pytorch. *arXiv preprint arXiv:2304.07613* (2023).
- [22] 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. <https://doi.org/10.1145/2902251.2902280>
- [23] 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. <https://doi.org/10.1145/3133901>
- [24] 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. <https://doi.org/10.14778/3467861.3467869>
- [25] Chirag Modi, François Lanusse, and Uros Seljak. 2021. FlowPM: Distributed TensorFlow implementation of the FastPM cosmological N-body solver. *Astron. Comput.* 37 (2021), 100505. <https://doi.org/10.1016/J.ASCOM.2021.100505>
- [26] 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. <https://doi.org/10.1145/3318464.3389702>
- [27] 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/bdca288fee7f92f2bfa9f7012727740-Abstract.html>
- [28] 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. <https://doi.org/10.1145/3299869.3320212>
- [29] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédéric 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. <https://doi.org/10.1145/2491956.2462176>
- [30] 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. <https://doi.org/10.1145/582095.582099>
- [31] 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. <https://doi.org/10.1145/3527333>
- [32] 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.
- [33] 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. <https://doi.org/10.1145/3318464.3380581>
- [34] 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, SSDM 2023, Los Angeles, CA, USA, July 10-12, 2023*, Robert Schuler, Carl Kesselman, Kyle Chard, and Alejandro Bugacov (Eds.). ACM, 13:1–13:12. <https://doi.org/10.1145/3603719.3603726>
- [35] 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. <https://doi.org/10.1109/IPDPSW52791.2021.00046>
- [36] 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.
- [37] 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. <https://doi.org/10.1016/J.CPC.2022.108292>
- [38] 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>
- [39] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2024. From Binary Join to Free Join. *SIGMOD Rec.* 53, 1 (2024), 25–31. <https://doi.org/10.1145/3665252.3665259>
- [40] 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. <https://doi.org/10.14778/3457390.3457399>
- [41] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. Gamma: leveraging Gustavson’s algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Virtual USA, 687–701.

<https://doi.org/10.1145/3445814.3446702>