On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML

Matthias Boehm[1] [*], Berthold Reinwald[1], Dylan Hutchison[2] [†], Prithviraj Sen[1], Alexandre V. Evfimievski[1], Niketan Pansare[1]

[1] IBM Research – Almaden; San Jose, CA, USA
[2] University of Washington; Seattle, WA, USA

ABSTRACT

Many machine learning (ML) systems allow the specification of ML algorithms by means of linear algebra programs, and automatically generate efficient execution plans. The opportunities for fused operators—in terms of fused chains of basic operators—are ubiquitous, and include fewer materialized intermediates, fewer scans of inputs, and sparsity exploitation across operators. However, existing fusion heuristics struggle to find good plans for complex operator DAGs or hybrid plans of local and distributed operations. In this paper, we introduce an exact yet practical cost-based optimization framework for fusion plans and describe its end-to-end integration into Apache SystemML. We present techniques for candidate exploration and selection of fusion plans, as well as code generation of local and distributed operations over dense, sparse, and compressed data. Our experiments in SystemML show end-to-end performance improvements of up to 22x, with negligible compilation overhead.

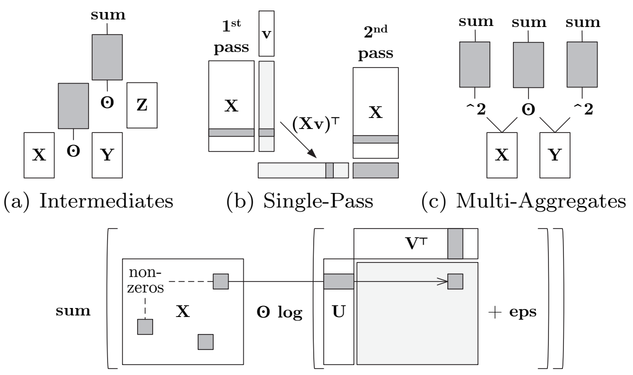
PVLDB Reference Format:
M. Boehm, B. Reinwald, D. Hutchison, P. Sen, A. V. Evfimievski, and N. Pansare. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *PVLDB*, 11(12): 1755-1768, 2018.
DOI: https://doi.org/10.14778/3229863.3229865

1. INTRODUCTION

Large-scale machine learning (ML) aims at statistical analysis and predictive modeling over large data collections [23], commonly using data-parallel frameworks like Spark [104]. State-of-the-art ML systems allow data scientists to express their ML algorithms—ranging from classification, regression, and clustering to matrix factorization and deep learning—in linear algebra and statistical functions [1, 14, 39, 60, 83, 87, 96, 102, 103], and automatically compile efficient execution plans. This high-level specification simplifies the development of custom ML algorithms, and allows the adaptation of execution plans to different data, hardware, and deployment characteristics.

[*]Email: mboehm7@gmail.com, Code: github.com/apache/systemml
[†]Work done during an internship at IBM Research – Almaden.

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3229865



(a) Intermediates (b) Single-Pass (c) Multi-Aggregates

(d) Sparsity Exploitation of $\mathbf{X}$ in $\text{sum}(\mathbf{X} \odot \log(\mathbf{U}\mathbf{V}^\top + \text{eps}))$

Figure 1: Examples of Fusion Opportunities.

Fusion Opportunities: There are many opportunities, where fused operators—in terms of fused chains of basic operators—can significantly improve performance. First, fusion allows eliminating the unnecessary materialization of intermediates, whose allocation and write is often a bottleneck. Examples are the two intermediates for $\text{sum}((\mathbf{X} \odot \mathbf{Y}) \odot \mathbf{Z})$ in Figure 1(a), where $\odot$ denotes an inexpensive element-wise multiplication. Second, fusion can eliminate unnecessary scans of inputs by exploiting temporal cell or row locality. For example, $\mathbf{X}^\top(\mathbf{X}\mathbf{v}) \rightarrow ((\mathbf{X}\mathbf{v})^\top \mathbf{X})^\top$ in Figure 1(b) can be realized in a single row-wise pass over the input because the intermediate is produced and consumed in a row-aligned manner. Third, multiple aggregates with shared inputs (e.g., $\text{sum}(\mathbf{X}^2)$, $\text{sum}(\mathbf{X} \odot \mathbf{Y})$, and $\text{sum}(\mathbf{Y}^2)$ in Figure 1(c)) leverage similar opportunities for DAGs (directed acyclic graphs) of multiple aggregates over common subexpressions (CSEs). Fourth, "sparse drivers"—i.e., sparse matrices with "sparse-safe" binary operations such as multiply $\mathbf{X}\odot$—allow sparsity exploitation across entire chains of operations. For example, $\text{sum}(\mathbf{X}\odot\log(\mathbf{U}\mathbf{V}^\top + \text{eps}))$ in Figure 1(d) can be computed for non-zeros in $\mathbf{X}$ only, which changes the asymptotic behavior by avoiding the computation of huge dense intermediates for $\mathbf{U}\mathbf{V}^\top$, $\mathbf{U}\mathbf{V}^\top + \text{eps}$, and $\log(\mathbf{U}\mathbf{V}^\top + \text{eps})$.

Existing Work on Operator Fusion: Given the ubiquitous opportunities and high performance impact, operator fusion has received a lot of attention in the literature. SystemML uses hand-coded fused operators to eliminate intermediates [40] or unnecessary scans [7], and to exploit sparsity across operations [14]. In contrast, Cumulon [39] and MatFast [103] use more generic masked and folded operators. However, these approaches require dedicated operators that are limited to fixed patterns of few operators and im-

$\mathbf{C} = \mathbf{A} + s * \mathbf{B}$
$\mathbf{D} = (\mathbf{C}/2)\hat{}(\mathbf{C}\text{-}1)$
$\mathbf{E} = \exp(\mathbf{C}\text{-}1)$

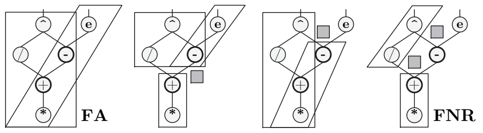


Figure 2: Alternative Fusion Plans in DAGs.

pose large development effort for combinations of dense and sparse inputs [29, 36]. Automatic operator fusion addresses this issue by access-pattern-aware fusion and subsequent code generation. Example systems include BTO [9], OptiML [97], Tupleware [25], Kasen [105], SystemML-SPOOF [29], Weld [76], TC [100], Julia [11, 43], MATLAB [61], TensorFlow XLA [1, 36], Nervana Graph [54], and TensorRT [75]. In contrast to traditional query optimization and compilation, these systems work with (1) operator DAGs instead of trees, and (2) dense or sparse linear algebra operations. However, existing work has limited support for sparse inputs and sparsity exploitation, and it relies on fusion plans that are derived with heuristics or even manual declaration.

A Case for Optimizing Fusion Plans: The lack of a principled approach for optimizing fusion plans becomes increasingly problematic as code generators cover more operation types. The key challenges are complex DAGs of operations, sparsity exploitation, and different access patterns, which create a search space that requires optimization:

- Materialization points (e.g., for multiple consumers),
- Sparsity exploitation and ordering of sparse inputs,
- Decisions on fusion patterns (e.g., template types), and
- Constraints (e.g., memory budgets and block sizes), and costs for local and/or distributed operations.

For a seamless compiler and runtime integration, operators are restricted to a single output. In DAGs, each intermediate might be consumed, however, by multiple operators, which requires materialization decisions to balance redundant compute and materialization costs. For example, Figure 2 shows a simple DAG with two materialization points (after + and -) and four valid, alternative fusion plans. Baseline solutions are heuristics like *fuse-all* (FA) or *fuse-no-redundancy* (FNR), but these struggle to find good plans for complex DAGs or hybrid plans of local and distributed operations.

Contributions: We introduce a practical framework for the *exact*, cost-based optimization of operator fusion plans over DAGs of linear algebra operations. This framework guarantees—under perfect cost estimates—finding the optimal plan regarding the considered decisions. In detail, we describe its end-to-end integration into open source Apache SystemML, major design decisions, and key components:

- *System Architecture:* In Section 2, we describe the compiler and runtime integration, including examples of code generation plans and generated operators.
- *Candidate Exploration:* In Section 3, we introduce a novel algorithm for the efficient exploration of valid partial fusion plans and our memoization table.
- *Candidate Selection:* Section 4 then presents our novel cost-based optimizer for plan selection with its search space, cost model, and enumeration algorithm.
- *Experiments:* In Section 5, we report on experiments in SystemML that cover micro benchmarks, local and distributed end-to-end experiments, as well as comparisons with Julia, TensorFlow, and fusion heuristics.

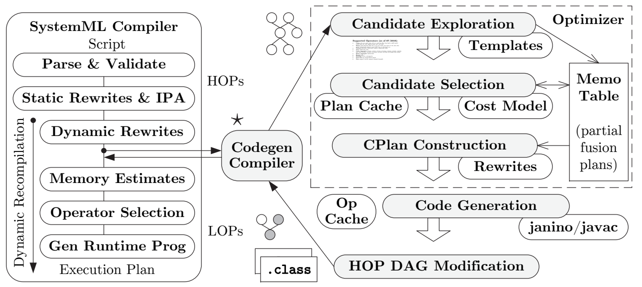


Figure 3: System Architecture Overview.

2. SYSTEM ARCHITECTURE

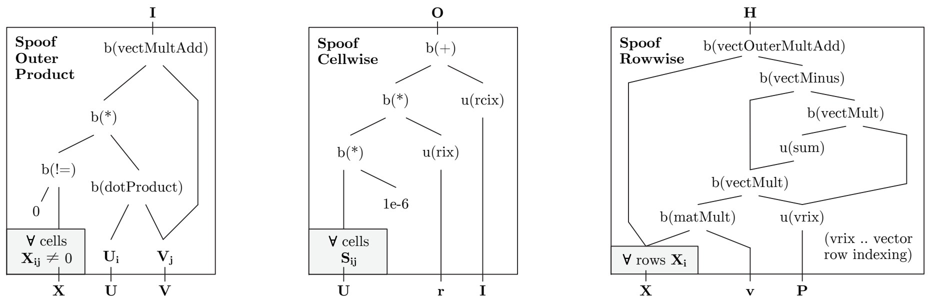
We describe the architecture of our code generator and its compiler integration into SystemML [13, 14]. Our optimization framework extends the SPOOF framework [29], which relied on ad-hoc candidate exploration and the FA heuristic. As background, we also sketch code generation plans, generated operators, and their runtime integration.

2.1 Compiler Integration

SystemML provides a high-level scripting language with R-like syntax, which includes linear algebra, element-wise and statistical operations. As shown in Figure 3 (left), a script is parsed into a hierarchy of statement blocks as delineated by control flow. Per block, we compile DAGs of high-level operators (HOPs). A HOP represents an operation output, and edges are data dependencies. These DAGs are modified via static—i.e., size-independent—rewrites, and interprocedural analysis (IPA) propagates matrix dimensions and sparsity from the inputs through the entire program. Based on this size information, we apply dynamic—i.e., size-dependent—rewrites and compute memory estimates per operation. These estimates are in turn used to select local or distributed execution types and physical operators. Similar to adaptive query processing [28], SystemML recompiles HOP DAGs during runtime (from dynamic rewrites) to adapt plans for initially unknown or changing sizes [13].

Codegen Compiler Integration: Conceptually, our code generator modifies the HOP DAGs—as shown in Figure 3 (middle $\star$)— after dynamic rewrites by replacing parts with fused operators. We do not consider rewrites and fusion jointly, which is an interesting direction for future work. Fused operators are represented via generic `SpoofOp`s that consist of meta data and the generated classes. These operators are still valid HOPs and thus, seamlessly leverage the remaining compilation steps such as memory estimates, operator selection (e.g., local/distributed), or runtime plan generation. We also invoke the code generator during dynamic recompilation, which is important for many algorithms in practice because our optimizer depends on known size information for costing and validity constraints. For example, $\text{sum}((\mathbf{X} \odot \mathbf{Y}) \odot \mathbf{Z})$ with unknown dimensions of $\mathbf{Y}$, only allows fusing $\text{sum}(\mathbf{TMP} \odot \mathbf{Z})$ to ensure correct handling of vector broadcasting. Such partial fusion is problematic because fused operators do not preserve their semantics (e.g., $\text{sum}(\odot)$) and thus, limit fusion potential once the size of $\mathbf{Y}$ becomes known. Therefore, our compiler integration retains the original HOP DAGs for dynamic recompilation. Applying codegen during recompilation is, however, challenging regarding its optimization and compilation overhead.

Codegen Architecture: Figure 3 (right) shows the codegen compiler architecture that comprises five well-



(a) Outer Template (b) Cell Template (c) Row Template

Figure 4: Example Code Generation Plans (CPlans).

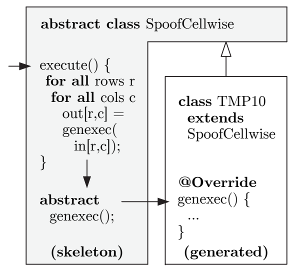


Figure 5: Runtime Integration of Fused Operators.

defined compilation steps. First, on candidate exploration (Section 3), we make a bottom-up pass over the HOP DAG to explore all valid partial fusion plans and store these plans in a memoization table. Second, candidate selection (Section 4) chooses the optimal subset of partial fusion plans using a fusion-aware cost model. Third, we construct code generation plans (CPlans, Section 2.2) for all selected fusion plans. These CPlans are further enhanced via low-level simplification rewrites for better code quality (Section 4.4). Fourth, we then recursively expand templates for all CPlans to generate Java source code per operator, as well as compile and load the classes with the `janino` [99] compiler. Generated operators are maintained in an operator cache—which identifies equivalent CPlans via hashing—to avoid redundant code generation and compilation. Finally, we replace the covered sub-DAGs with the fused operators. These separate compilation steps are very valuable for debugging without affecting fusion potential or compilation overhead.

2.2 Code Generation Plans

Code generation plans (CPlans) are a backend-independent representation of fused operators and allow for recursive code generation. We generate code via a depth-first template expansion to ensure a valid ordering of code fragments according to their data dependencies. Such plans consist of CNodes, which are either template or basic operation nodes. Template nodes represent generic fused operator skeletons that have a specific data binding and contain a DAG of basic operations that encodes the data flow.

Example Expressions: We illustrate CPlans for two typical expressions with high performance impact of fusion. The first expression is part of an inner-loop update rule of ALS-CG (alternating least squares via conjugate gradient) [14], which computes a low-rank factorization $\mathbf{X} \approx \mathbf{U}\mathbf{V}^\top$:

$$\mathbf{O} = ((\mathbf{X} \neq 0) \odot (\mathbf{U}\mathbf{V}^\top))\mathbf{V} + 10^{-6} \odot \mathbf{U} \odot \mathbf{r}, \tag{1}$$

where $\odot$ denotes an element-wise multiply. Typically, $\mathbf{X}$ is large but sparse, and the rank (i.e., ncol($\mathbf{U}$)) is in the tens to hundreds. This expression requires—similar to Figure 1(d)—a sparsity-exploiting operator to avoid computing and materializing the dense outer-product-like $\mathbf{U}\mathbf{V}^\top$. The second expression stems from the inner-loop of MLogreg (multinomial—i.e., multiclass—logistic regression):

$$\begin{aligned}\mathbf{Q} &= \mathbf{P}[\,,1:k] \odot (\mathbf{X}\mathbf{v})\\ \mathbf{H} &= \mathbf{X}^\top(\mathbf{Q} - \mathbf{P}[\,,1:k] \odot \text{rowSums}(\mathbf{Q})),\end{aligned} \tag{2}$$

where $\mathbf{X}$ is the feature matrix and k = #classes−1. This pattern requires—similar to Figure 1(b)—fusion to avoid multiple passes over $\mathbf{X}$ and intermediates of size nrow($\mathbf{X}$) × k.

Code Generation: Figure 4 shows the three CPlans of fused operators constructed for our example expressions. Figure 4(a) shows the CPlan of an Outer template operator for $\mathbf{I} = ((\mathbf{X} \neq 0) \odot (\mathbf{U}\mathbf{V}^\top))\mathbf{V}$, which is sparsity-exploiting and thus improves performance proportional to the sparsity of $\mathbf{X}$. From this CPlan, we generate the following code:

```
public final class TMP4 extends SpoofOuterProduct {
 public TMP4() {super(OuterType.RIGHT);}
 protected void genexec(double a,double[] a1,int a1i,
 double[] a2,int a2i,double[] c,int ci,...,int len) {
   double TMP0 = (a != 0) ? 1 : 0;  // see Sec. 4.4
   double TMP1 = dotProduct(a1, a2, a1i, a2i, len);
   double TMP2 = TMP0 * TMP1;
   vectMultAdd(a2, TMP2, c, a2i, ci, len);  }}
```

For each non-zero value $\mathbf{X}_{ij}$, we compute the scalar inner product w_{ij} of row vectors $\mathbf{U}_i$ and $\mathbf{V}_j$, scale $\mathbf{V}_j$ by w_{ij}, and add it to the output with $\mathbf{I}_i {+}{=} w_{ij} \odot \mathbf{V}_j$, where `dotProduct` and `vectMultAdd` refer to a library of vector primitives. Figure 4(b) shows the CPlan of an additional cell-wise operator for $\mathbf{I} + 10^{-6} \odot \mathbf{U} \odot \mathbf{r}$, which avoids two intermediates but cannot be fused into the previous Outer due to its aggregation and the sparse-unsafe addition. We use a `SideInput` abstraction to access additional dense or sparse inputs.

```
public final class TMP10 extends SpoofCellwise {
 public TMP10() {super(CellType.NO_AGG,null,false);}
 protected double genexec(double a, SideInput[] b,
 double[] scalars,..., int rix, int cix) {
   double TMP5 = getValue(b[0], n, rix, cix);
   double TMP6 = a * 1.0E-6;
   double TMP7 = getValue(b[1], rix);
   double TMP8 = TMP6 * TMP7;
   double TMP9 = TMP5 + TMP8;
   return TMP9;  }}
```

Figure 4(c) shows the row-wise CPlan of Expression (2). This single-pass operator exploits temporal row locality by accessing $\mathbf{X}_i$ twice (lines 6/11), and it avoids six large intermediates. The memory for row intermediates is managed via preallocated ring buffers per thread (here of size 5).

```
 1: public final class TMP25 extends SpoofRowwise {
 2:  public TMP25() {super(RowType.COL_AGG_B1_T,true,5);}
 3:  protected void genexecDense(double[] a, int ai,
 4:  SideInput[] b, double[] c,..., int len) {
 5:    double[] TMP11 = getVector(b[1].vals(rix),...);
 6:    double[] TMP12 = vectMatMult(a,b[0].vals(rix),...);
 7:    double[] TMP13 = vectMult(TMP11, TMP12, 0, 0,...);
 8:    double TMP14 = vectSum(TMP13, 0, TMP13.length);
 9:    double[] TMP15 = vectMult(TMP11, TMP14, 0,...);
10:    double[] TMP16 = vectMinus(TMP13, TMP15, 0, 0,...);
11:    vectOuterMultAdd(a, TMP16, c, ai, 0, 0,...); }
12:  protected void genexecSparse(double[] avals, int[]
13:  aix, int ai, SideInput[] b, ..., int len){...}}
```

Template Types: Generalizing the previous examples, we use four template types $T = (\text{Row}, \text{Cell}, \text{MAgg}, \text{Outer})$. The row-wise (Row) template binds to sparse/dense rows of a main input $\mathbf{X}_i$, a list of sparse/dense side inputs, and a vector of scalars. Similarly, the cell-wise (Cell) and multi aggregate (MAgg) templates bind to cells $\mathbf{X}_{ij}$ and side inputs. All templates can be marked sparse-safe in which case they only process non-zero rows or cells of the main input. Finally, the outer-product (Outer) template binds to non-zero cells in $\mathbf{X}$, rows in $\mathbf{U}$ and $\mathbf{V}$, as well as dense side inputs. Template variants include different aggregation types, such as none (e.g., `X+Y`), row (e.g., `rowSums(X+Y)`), column (e.g., `colSums(X+Y)`), or full (e.g., `sum(X+Y)`), which have different implementations and allow to infer the output dimensions.

2.3 Runtime Integration

SystemML uses—like many other ML systems [39, 83, 102, 103]—a blocked matrix format to store distributed matrices as a collection of block indexes and blocks, as well as local matrices as a single block to reuse the block runtime. Accordingly, the generated fused operators process blocks.

Block Operations: Templates refer to generic skeletons of fused operators, which are inspired by algorithmic skeletons [24]. Figure 5 shows the runtime integration of a Cell operator. Unlike other work [9, 25, 52, 100], we made the conscious design decision not to generate the data access into the fused operators. Instead, the hand-coded skeleton implements the data access of dense, sparse, or compressed [30] matrices and calls an abstract (virtual) `genexec` method for each value. Generated operators (e.g., `TMP10` in Figure 5) then inherit the skeleton and override `genexec`, which yields very lean yet efficient operators. The skeleton also handles multi-threading, cache blocking, memory management, and pseudo-sparse-safe aggregations[1]. Sharing skeletons and vector primitives can also reduce the instruction footprint and thus, instruction cache misses, which is a known bottleneck in OLTP [93] and scale-out workloads [32].

Distributed Operations: The generated operators are directly used for distributed Spark operations as well. We support data-parallel fused operators and fused operators in task-parallel `parfor` loops [12]. For both, we ship the generated class codes via task closures to the executors, where the classes are compiled, but reused across tasks. We use additional skeletons for (1) consolidating inputs via `join` and `broadcast`, and (2) aggregating results when necessary.

Since the codegen framework is part of recompilation during runtime, efficient candidate exploration and selection—as discussed in the next sections—is crucial for performance.

3. CANDIDATE EXPLORATION

The exploration of candidate fusion plans aims to identify all valid partial fusion plans to provide a common input for different plan selection policies and simplify optimization. However, the exponential search space prohibits the enumeration of all possible plans. Instead, we enumerate *partial fusion plans* per operator, which represent local fusion decisions but no combinations of these local plans. We describe the representation of partial fusion plans in our central memoization table, and an efficient algorithm for populating this memo table in a single pass over the HOP DAG.

[1]For example, `rowMins(X)` can be computed over non-zeros only, but it requires corrections for missing 0s if `nnz(X[i,])<ncol(X)`.

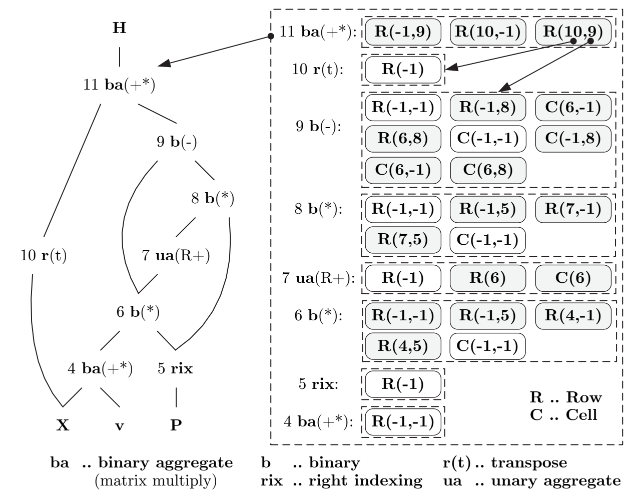

H
11 ba(+*)
9 b(-)
8 b(*)
10 r(t)
7 ua(R+)
6 b(*)
4 ba(+*)
5 rix
X
v
P
11 ba(+*): R(-1,9) R(10,-1) R(10,9)
10 r(t): R(-1)
9 b(-): R(-1,-1) R(-1,8) C(6,-1) R(6,8) C(-1,-1) C(-1,8) C(6,-1) C(6,8)
8 b(*): R(-1,-1) R(-1,5) R(7,-1) R(7,5) C(-1,-1)
7 ua(R+): R(-1) R(6) C(6)
6 b(*): R(-1,-1) R(-1,5) R(4,-1) R(4,5) C(-1,-1)
5 rix: R(-1)
4 ba(+*): R(-1,-1)
R .. Row
C .. Cell
ba .. binary aggregate (matrix multiply)
b .. binary
rix .. right indexing
r(t) .. transpose
ua .. unary aggregate

Figure 6: Example Memo Table (w/ basic pruning).

3.1 Memoization Table

Our memoization (memo) table consists of a set of groups, where each group represents the output of an operator in the HOP DAG, i.e., a logical subexpression. Each group is identified by the operator ID, has access to its operator meta data, and contains a set of valid partial fusion plans for this operator. A partial fusion plan is called a memo table entry, and can reference other groups to represent fusion decisions. This structure is similar to *groups* and *group expressions* in the Cascades Optimization Framework [17, 37, 91], but we use it merely as a compact representation of fusion plans, which only includes operators that are amenable to fusion.

Memo Table Entries: A memo table entry is a tuple $(\text{type}, \{i_1, .., i_k\}, \text{closed})$, consisting of a template type a list of inputs, and a closed type. The inputs correspond to HOP inputs (i.e., data dependencies) by position, and each input is either a group reference to indicate fusion or -1 for materialization. A reference from an entry to a group implies that the group contains at least one compatible fusion plan. Finally, the close status can be open valid, open invalid (i.e., an invalid entry point), closed valid, and closed invalid.

Example: We use Expression (2) from Section 2.2 to illustrate the structure of our memo table. Figure 6 shows the HOP DAG and the related memo table after candidate exploration. All eight operators are represented by groups in the memo table. Group 11 refers to the final matrix multiplication (i.e., binary aggregate `ba(+*)`), and contains three memo table entries of type Row. These entries encode fusion alternatives: (1) fuse right R(-1,9), (2) fuse left R(10,-1), and (3) fuse both R(10,9). Instead of encoding all alternative subplans along inputs, we only reference the input groups. This memo table then allows for simple costing and fusion by traversing the HOP DAG top down (i.e., starting from the outputs), probing for fusion plans, traversing group references, and determining the input HOPs from where this process repeats until we reach the leaf HOPs.

3.2 Open-Fuse-Merge-Close Exploration

Given an operator DAG $\mathcal{G}$, a set of template types T, and an empty memo table $\mathcal{W}$, we aim to efficiently discover all valid partial fusion plans P. We introduce a bottom-up algorithm that is template-oblivious and populates the memo table in a single pass over the DAG.

Algorithm 1 OFMC Explore (recursive)

```
Input: An operator $g_i$ of DAG $\mathcal{G}$ w/ $|g_i|$ inputs, memo table $\mathcal{W}$
Output: A populated memo table $\mathcal{W}$
 1: // Memoization of processed operators – – – – – – – – – – –
 2: if $g_i \in \mathcal{W}[\star]$ then
 3:    return $\mathcal{W}$
 4: // Recursive candidate exploration – – – – – – – – – – – – –
 5: for all $j$ in 1 to $|g_i|$ do             // for all operator inputs
 6:    OFMCEXPLORE($g_j$, $\mathcal{W}$)
 7: // Open initial operator plans – – – – – – – – – – – – – – –
 8: for all $t \in T$ do                     // for all template types
 9:    if $t$.OPEN($g_i$) then               // probe opening condition
10:       $\mathcal{W}[g_i] \leftarrow$ CREATEPLANS($g_i$, null, $t$)
11: // Fuse and merge operators plans – – – – – – – – – – – – –
12: for all $j$ in 1 to $|g_i|$ do             // for all operator inputs
13:    for all $t$ in $\mathcal{W}[g_j]$ do          // for all distinct templates
14:       if $t$.FUSE($g_i, g_j$) then
15:          $\mathcal{W}[g_i] \leftarrow \mathcal{W}[g_i] \cup$ CREATEPLANS($g_i, g_j, t$)
16: // Close operator plans if required – – – – – – – – – – – –
17: for all $me$ in $\mathcal{W}[g_i]$ do            // for all memo entries
18:    $me.closed \leftarrow t(me.type)$.CLOSE($g_i$)
19:    if $me.closed < 0$ then                      // closed invalid
20:       $\mathcal{W}[g_i] \leftarrow \mathcal{W}[g_i] \setminus me$
21: $\mathcal{W}[\star] \leftarrow \mathcal{W}[\star] \cup g_i$             // mark operator as processed
22: return $\mathcal{W}$
```

OFMC Template Abstraction: As the basis of our candidate exploration algorithm, we define the *open-fuse-merge-close* (OFMC) template abstraction:

- `open(Hop h)`: Indicates if a new fused operator of this template can be started at HOP h, covering its operation and reading materialized inputs. For example, the condition of an Outer template is an outer-product-like matrix multiplication like $\mathbf{UV}^\top$ with size constraints.
- `fuse(Hop h, Hop in)`: Indicates if an open fused operator (of this template) at the input HOP *in* can be expanded to its consumer HOP h. For example, a Cell template can fuse valid unary, binary, or ternary operations, valid aggregations, and inner products.
- `merge(Hop h, Hop in)`: Indicates if an open fused operator at the consumer HOP h can be expanded to its input HOP *in*, i.e., if it can merge with fused operators of valid types at the input. An example is the merge of Cell templates into Row templates.
- `close(Hop h)`: Indicates the close status of the template after the HOP h and its validity. For example, any aggregation closes a Cell template (as valid or invalid), whereas only column-wise or full aggregations close a Row template. Outer templates are also validated for the existence of sparsity exploiting operators.

This abstraction separates template-specific conditions from the DAG traversal and the population of the memo table.

OFMC Algorithm: Based on the memo table and OFMC abstraction, we introduce the OFMC exploration algorithm shown by Algorithm 1. This algorithm is called recursively in a depth-first manner to populate the memo table bottom-up. First, we check for already processed operators—indicated by a set of visited operators $\mathcal{W}[\star]$—(lines 1-3) to avoid redundant exploration if nodes are reachable over multiple paths. Second, we recursively explore all $|g_i|$ input operators (lines 4-6) because these input data dependencies constitute potential fusion references. Third, we explore all templates for valid *opening* conditions at the current operator (lines 7-10). In case of a valid opening condition, we add this memo entry and enumerate merge plans with CREATEPLANS as discussed below. This merging is important to cover scenarios such as $\mathbf{X}^\top(\mathbf{y} \odot \mathbf{z})$, where the matrix-vector multiplication with $\mathbf{X}$ opens a Row template, which can also merge Cell templates over $\mathbf{y} \odot \mathbf{z}$. Fourth, we extend open fusion plans from the inputs (lines 11-15) via *fuse* (extend from input to consumer) and *merge* (extend from consumer to inputs). This step entails iterating over all distinct templates types of all inputs and probing the pair-wise *fusion* conditions `fuse(h,in)`. In case of a valid condition, we again call CREATEPLANS, which constructs a memo table entry for the fused operator, and enumerates all *local* plan combinations for inputs that satisfy the pair-wise *merge* condition `merge(h,in)`. These plan sets—of total size $\leq |T|\cdot 2^{|g_i|}$—are then added to the group of the current operator. Fifth, we check all group entries for *closing* conditions (lines 16-20). Entries that satisfy the closing condition of their templates are either removed (invalid) or marked as closed (valid), while all other entries remain open.

Algorithm Analysis: Overall, our algorithm has linear time and space complexity in the number of operators. Memoization ensures that we visit each operator exactly once and the OFMC conditions apply only locally to an operator and its inputs. These conditions still have access to the HOPs and thus, the entire DAG, but this flexibility is only exploited in rare cases such as recognizing `t(cumsum(t(X)))` as a row operation. For each operator g_i (with $|g_i|$ inputs), we enumerate and store up to $O(2^{|g_i|} \cdot |T|)$ memo entries, but the supported $|T| = 4$ templates and unary, binary, and ternary[2] basic operators (i.e., $\max(|g_i|) = 3$), give us an upper bound of $32 \cdot |\mathcal{G}|$ plans, and works very well in practice.

4. CANDIDATE SELECTION

Given an operator DAG $\mathcal{G}$ and a memo table of partial fusion plans P, candidate selection aims to choose the optimal subset of plans $P^\star$ that applied to $\mathcal{G}$ minimizes costs C. The optimal plans are subject to a set of constraints Z such as memory budgets, and block size restrictions imposed by distributed matrix formats. We describe the space of alternatives, the cost model, the cost-based enumeration algorithm MPSKIPENUM, and the final construction of CPlans. The basic ideas are to (1) split the set of partial fusion plans into independent partitions, (2) restrict the search per partition to interesting points, (3) linearize the resulting exponential search space, (4) enumerate and cost plans with skipping of search space areas that can be safely pruned, and (5) use a plan cache for repeated optimization problems.

Selection Heuristics: Common baseline solutions to the fusion problem for DAGs are the following heuristics:

- *Fuse-All (FA)* aims at maximal fusion, which leads to redundant compute on CSEs. This heuristic is similar to lazy evaluation in Spark [104], delayed arrays in Repa [47], and code generation in SPOOF [29].
- *Fuse-No-Redundancy (FNR)* takes another extreme of fusion without redundant compute, which leads to materializing all intermediates with multiple consumers. This heuristic is similar to caching in Emma [2].

In the following, we use these heuristics as baselines but focus solely on finding the cost-optimal set of fusion plans.

[2]An exception are nary `cbind`, `min`, and `max` operations, but these are uncommon and rarely have more than five inputs ($2^5 \cdot 1 = 32$).

4.1 Plan Partitions and Interesting Points

In preparation of plan enumeration, we analyze the fusion plans P to characterize the search space by independent partitions, and interesting materialization points per partition.

Plan Partitions: We define the *plan partitions* $\mathcal{P}$ of P as its connected components in terms of fusion references. Therefore, partitions are unreachable via fusion and thus, are optimized separately. Figure 7 shows an example with three partitions. A partitioning can originate from unsupported operations or aggregations like `sum` or `colSums`, which close all templates. In addition, we define the following terminology. First, *root nodes* $\mathcal{R}_i$ of a partition $\mathcal{P}_i$ (with $\mathcal{R}_i \subseteq \mathcal{P}_i$) are never referenced from fusion plans $g \in \mathcal{P}_i$ and thus, always materialized. These roots are the entry points for partition analysis and costing. Second, *input nodes* $\mathcal{I}_i$ of a partition $\mathcal{P}_i$ (with $\mathcal{I}_i \cap \mathcal{P}_i = \emptyset$) are nodes whose output is read by a node $g \in \mathcal{P}_i$. Third, *materialization points* $\mathcal{M}_i$ of a partition $\mathcal{P}_i$ (with $\mathcal{M}_i \subseteq \mathcal{P}_i \wedge \mathcal{M}_i \cap \mathcal{R}_i = \emptyset$) are nodes with multiple consumers of which at least one belongs to $\mathcal{P}_i$.

Interesting Points: Materialization points are interesting because fusion can cause redundant compute. Generalizing this notion, we define the search space per partition by its *interesting points* $\mathcal{M}'_i$. These points are boolean fusion decisions and our optimizer considers all $2^{|\mathcal{M}'_i|}$ plans:

- *Materialization Point Consumers:* Each data dependency $g \leftarrow \mathcal{M}_{ij} \mid g \in \mathcal{P}_i$ on a materialization point is considered separately, which is important for avoiding unnecessary reads in overlapping fused operators.
- *Template Switches* are data dependencies $(g_i \leftarrow g_j)$, where $\mathcal{W}[g_j]$ contains templates that are not in $\mathcal{W}[g_i]$, which is important for finding sparsity-exploiting plans such as $\mathbf{X} \odot \mathbf{U}\mathbf{V}^\top$ in $\mathbf{Y} + \mathbf{X} \odot \mathbf{U}\mathbf{V}^\top$.

A true assignment of a point $(g_i \leftarrow g_j)$ dictates the materialized consumption of g_j. Hence, all fusion plans with a fusion reference from g_i to g_j are considered invalid for costing.

4.2 Cost Model

Given a plan assignment $\mathbf{q}$ of interesting points, we compute the costs $C(\mathcal{P}_i|\mathbf{q})$ of a plan partition $\mathcal{P}_i$ with an analytical cost model for DAG-structured fusion plans as follows:

$$C(\mathcal{P}_i|\mathbf{q}) = \sum_{p \in \mathcal{P}_i|\mathbf{q}} \left(\hat{T}^w_p + \max\left(\hat{T}^r_p, \hat{T}^c_p\right)\right). \quad (3)$$

Each p is a basic or fused operator defined by $\mathbf{q}$ and the DAG structure. $\hat{T}^w_p$, $\hat{T}^r_p$, and $\hat{T}^c_p$ are estimates of operator write, read, and computation times. The read and write estimates are derived from the input and output sizes, normalized by peak memory bandwidth. For example, reading a $100\text{M} \times 10$ (i.e., 1G) dense input matrix at 32 GB/s peak read bandwidth, gives us a time estimate of $\hat{T}^r_p = 1\text{G} \cdot 8\,\text{B}/32\,\text{GB/s} = 0.25\,\text{s}$. Similarly, the compute time is derived from the number of floating point operations and peak compute bandwidth. We take $\max(\hat{T}^r_p, \hat{T}^c_p)$ to account for overlapping read and compute costs, while adapting to I/O- and compute-bound operations. Sparsity-exploiting operators simply scale these estimates down by the sparsity of the main input. This simple cost model works well in practice because (1) it is only used for plan comparisons, (2) fusion-specific errors do not propagate (unlike in join ordering [41]), and (3) dynamic recompilation updates input sizes during runtime if needed.

Cost Computation via Cost Vectors: Efficient and correct costing requires (1) memoizing processed sub-DAGs

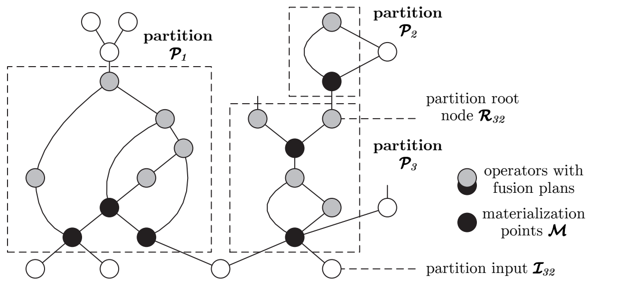


Figure 7: Example Plan Partitions and Terminology.

if they are reachable over multiple paths, (2) accounting for shared reads and CSEs within fused operators, and (3) accounting for redundant compute of overlapping fused operators. We tackle this challenge via DAG traversal and *cost vectors* $\mathbf{c}_p$ per fused operator. The partition costs $C(\mathcal{P}_i|\mathbf{q})$ are computed with getPlanCost starting from its roots $\mathcal{R}_i$ without cost vectors, indicating materialized outputs. At each operator, we then either open or extend a fused operator and query the memo table for the best valid fusion plan of the current template. For existing fusion references, we call getPlanCost with $\mathbf{c}_p$; otherwise, we add the input to $\mathbf{c}_p$ and cost it without $\mathbf{c}_p$. This vector $\mathbf{c}_p$ also captures the compute costs of included operators. After processing all inputs of an opened operator, we summarize $\mathbf{c}_p$ with formula (3) and add it to the total costs. Non-partition consumers of intermediates inside a fused operator trigger an additional call of getPlanCost without $\mathbf{c}_p$. Memoizing pairs of operators and cost vectors returns zero costs for processed operators, while correctly accounting for redundant compute.

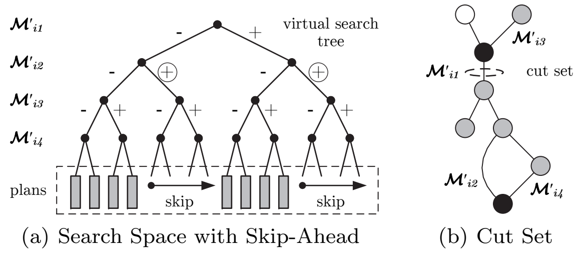
Constraints and Distributed Operations: We handle the constraints Z via a prefiltering of entries that are known to violate constraints. Remaining violations are then assigned infinite costs during enumeration and costing. Similarly, we also use different read bandwidths for inputs of distributed operations to reflect the cost of distributed joins and broadcasts, according to the input sizes of computed cost vectors and available memory budgets.

4.3 Enumeration Algorithm MPSkipEnum

Given a fusion partition $\mathcal{P}_i$ and its interesting points $\mathcal{M}'_i$, we aim to find the optimal plan $\mathbf{q}^\star$ that minimizes costs. We introduce the remarkably simple yet efficient MPSkipEnum algorithm that linearizes the exponential search space, enumerates and costs plans with the ability to skip entire subspaces using cost-based and structural pruning techniques.

Basic Enumeration: Algorithm 2 shows the basic enumeration approach. We iterate over the linearized search space of all $2^{|\mathcal{M}'_i|}$ plans (lines 3-21). In each iteration, we create the specific plan $\mathbf{q}$ (line 4), cost the plan with getPlanCost (line 18), and maintain the best plan $\mathbf{q}^\star$ and its costs $\overline{C}$ (lines 19-21). Figure 8(a) shows an example search space of $|\mathcal{M}'_i| = 4$ interesting points and its 16 plans. This basic enumeration approach is simple and has no space requirements. However, evaluating the exponential number of plans quickly becomes infeasible as $|\mathcal{M}'_i|$ increases. Therefore, we apply two high-impact, lossless pruning techniques.

Cost-Based Pruning (lines 12-16): We maintain the costs of the best plan $\overline{C}$, which is a monotonically decreasing upper bound of the optimal plan. By computing a lower bound $\underline{C}$ of unseen plans, we can safely prune these plans whenever $\underline{C} \geq \overline{C}$. We compute $\underline{C}$ from static and plan-



(a) Search Space with Skip-Ahead (b) Cut Set

Figure 8: Example Search Space and Cut Set.

dependent costs. The static costs $\underline{C}_{\mathcal{P}_i}$ comprise the read of inputs $\mathcal{I}_i$, the minimal compute (with no redundancy and corrections for sparsity-exploitation), and the write of roots $\mathcal{R}_i$. The plan-dependent costs are obtained by GETMPCOST without DAG traversal, based on the read and write costs of distinct materializations in $\mathcal{M}'_i|\mathbf{q}$. Our search space is then linearized from negative to positive assignments, which is crucial for effective pruning. We evaluate the left- and right-most plans—i.e., the FA and FNR heuristics—first (line 2), which yields a good $\overline{C}$ from the beginning. Our search space layout then allows skipping entire subspaces. Figure 8(a) shows an example, where $\mathcal{M}'_{i2}$ is set to true ($\oplus$). If $\underline{C}$ of the first plan in this subspace (with $\mathcal{M}'_{i3}$ and $\mathcal{M}'_{i4}$ set to false) exceeds $\overline{C}$, we can prune the subspace because other plans are known to increase the materialization costs and thus $\underline{C}$. The number of skipped plans (line 15) is $2^{|\mathcal{M}'_i|-x-1}$, where $x = \text{lastIndexOf}(\mathbf{q}, \mathbf{true})$. In addition, GETPLANCOST stops costing a plan whenever the partial costs exceed $\overline{C}$.

Structural Pruning (lines 7-11): Similar to state-of-the-art join enumeration [66, 67, 73], we exploit the graph structure of $\mathcal{P}_i$ and its interesting points $\mathcal{M}'_i$ for additional pruning. The key observation is that interesting points can—predicated on their assignment—create independent sub-problems because they act as fusion barriers. Figure 8(b) shows an example; if $\mathcal{M}'_{i1} = \mathbf{true}$, the two sub-problems of $\mathcal{M}'_{i3}$ and $(\mathcal{M}'_{i2}, \mathcal{M}'_{i4})$ are independent. Inspired by conditioning techniques for probabilistic databases [55], and the selection of optimization units for MapReduce workflows [59], we build a reachability graph RG over $\mathcal{M}'_i$ to determine *cut sets*. We use single points, composite points of equivalent inputs, and pairs of these as candidates. For each candidate cut set **cs**, we get the points S_1 reachable from the roots to **cs**, and the points S_2 reachable from the **cs**. Cut sets are then sorted in ascending order of their scores:

$$\left(\left(2^{|\mathbf{cs}|}-1\right)/2^{|\mathbf{cs}|}\cdot 2^{|\mathcal{M}'_i|}\right)+\left(1/2^{|\mathbf{cs}|}\cdot\left(2^{|S_1|}+2^{|S_2|}\right)\right) \quad \text{s.t. } S_1\cap S_2=\emptyset,\ S_1\neq\emptyset,\text{ and } S_2\neq\emptyset. \tag{4}$$

Intuitively, the two terms compute the total number of plans where **cs** is inactive and active; a **cs** is only active if $\forall c \in \mathbf{cs} : c = \mathbf{true}$. This order maximizes pruning and defines the top of the search tree. We store each **cs** along with its sub-problems. During enumeration, we then probe these cut sets (line 7), call MPSKIPENUM recursively for their sub-problems (lines 10-11), combine the results into a global plan for costing, and finally prune the subspace (line 21).

Graceful Approximation: Despite good pruning effectiveness, there are no guarantees on reducing the exponential search space for arbitrary DAGs. To ensure robustness in production, we further use—inspired by graceful degradation for join enumeration [69, 73]—approximate techniques for large problems of $|\mathcal{M}'_i| \geq 15$. First, we stop optimizing when $\overline{C}$ drops below $(1+\epsilon)\cdot\underline{C}_{\mathcal{P}_i}$, which helps for long tails

Algorithm 2 Materialization Point Skip Enumerate

```
Input: memo table $\mathcal{W}$, plan partition $\mathcal{P}_i$, reachability graph $RG$,
       interesting points $\mathcal{M}'_i$, offset off
Output: The best plan $\mathbf{q}^\star$
 1: // opening heuristic: evaluate FA and FNR heuristics
 2: $[\mathbf{q}^\star, \overline{C}] \leftarrow$ EVALFIRSTANDLASTPLAN($\mathcal{W}, \mathcal{P}_i, \mathcal{M}'_i$, off)
 3: for all $j$ in 2 to $2^{|\mathcal{M}'_i|-\text{off}} - 1$ do          // evaluate plans
 4:     $\mathbf{q} \leftarrow$ CREATEASSIGNMENT($|\mathcal{M}'_i|$ − off, off, $j$)
 5:     $pskip \leftarrow 0$
 6:     // pruning via skip-ahead − − − − − − − − − − − − − − −
 7:     if $RG \neq$ null $\wedge$ ISCUTSET($RG, \mathbf{q}$) then        // structural
 8:         $pskip \leftarrow$ GETNUMSKIPPLANS($RG, \mathbf{q}$)
 9:         $S \leftarrow$ GETSUBPROBLEMS($RG, \mathbf{q}$)
10:         for all $k$ in 1 to $|S|$ do
11:             $\mathbf{q}[S_k.\mathbf{ix}] \leftarrow$ MPSKIPENUM($\mathcal{W}, \mathcal{P}_i$, null, $S_k.\mathbf{m}, S_k$.off)
12:     else                                              // cost-based
13:         $\underline{C} \leftarrow \underline{C}_{\mathcal{P}_i}$ + GETMPCOST($\mathcal{W}, \mathcal{P}_i, \mathcal{M}'_i, \mathbf{q}$)
14:         if $\underline{C} \geq \overline{C}$ then
15:             $j \leftarrow j$ + GETNUMSKIPPLANS($\mathbf{q}$) − 1
16:             continue
17:     // plan costing and comparison − − − − − − − − − − − − −
18:     $C \leftarrow$ GETPLANCOST($\mathcal{W}, \mathcal{P}_i, \mathcal{M}'_i, \mathbf{q}, \overline{C}$)
19:     if $\mathbf{q}^\star$ = null $\vee$ $C < \overline{C}$ then
20:         $\mathbf{q}^\star \leftarrow \mathbf{q}$;   $\overline{C} \leftarrow C$
21:     $j \leftarrow j + pskip$
22: return $\mathbf{q}^\star$
```

with minor changes. Second, we reuse plans via a plan cache for repeated optimizations. This cache uses an inexpensive, approximate plan signature based on $\mathcal{P}_i$ and the initial costs $\overline{C}$. Collisions are highly unlikely for large problems and plan caching helps significantly for recompilation in mini-batch algorithms. Third, MPSKIPENUM is an *anytime* algorithm that allows returning the currently best plan at any time (e.g., once an optimization budget is reached).

4.4 CPlan Construction and Rewrites

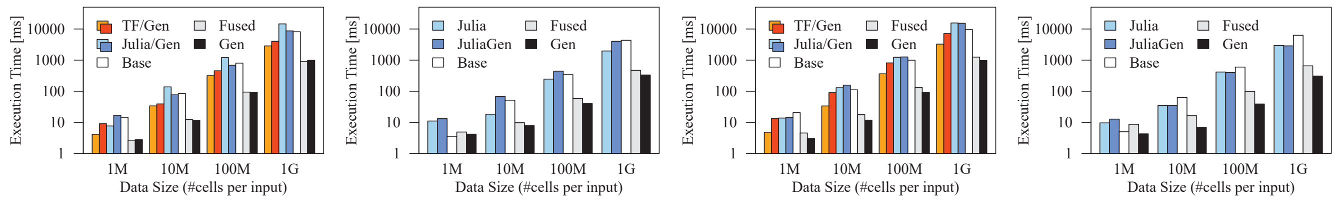
With the optimal plan $\mathbf{q}^\star$, we then prune the memo table P to obtain the optimal $P^\star$ and mechanically construct CPlans for all fused operators of the given DAG. Before code generation, these CPlans are enhanced by rewrites. First, we apply low-level simplification rewrites. For example, consider $\mathbf{I} = ((\mathbf{X} \neq 0) \odot (\mathbf{U}\mathbf{V}^\top))\mathbf{V}$ from Figure 4(a). Since the Outer template binds to non-zero input cells, we remove the unnecessary $(\mathbf{X} \neq 0)\odot$. Second, we apply common subexpression elimination to avoid redundancy within fused operators. This is important for operations such as `conv2d` that are broken-up into their elementary row-wise operations (e.g., `im2col`, `matrixMult`, and `reshape`), which avoids redundant `im2col` operations if a batch is fed into multiple `conv2d` operations, a pattern found in neural networks such as Sentence CNN [51] and Inception [98].

5. EXPERIMENTS

Our experiments study the performance characteristics of code generation for linear algebra programs and the optimization of fusion plans. To this end, we investigate (1) several interesting micro-benchmark patterns, (2) dense, sparse, ultra-sparse, and compressed data, as well as (3) single-node and large-scale, end-to-end experiments.

5.1 Experimental Setting

Setup: We ran the experiments on a 1+6 node cluster of one head node (2x4 Intel Xeon E5530 @ 2.40 GHz-2.66 GHz, hyper-threading, 64 GB RAM @800 MHz) and six worker



(a) sum($\mathbf{X} \odot \mathbf{Y} \odot \mathbf{Z}$), D (b) sum($\mathbf{X} \odot \mathbf{Y} \odot \mathbf{Z}$), S (c) sum($\mathbf{X} \odot \mathbf{Y}$), sum($\mathbf{X} \odot \mathbf{Z}$), D (d) sum($\mathbf{X} \odot \mathbf{Y}$), sum($\mathbf{X} \odot \mathbf{Z}$), S

Figure 9: Operations Performance of Example Cell and MAgg Patterns (D .. dense, S .. sparse).

Table 1: Real Datasets and their Characteristics.

Name	Data Size $m \times n$	Sparsity $\text{nnz}/(m \times n)$	Fmt
Airline78 [4]	14,462,943 × 29	0.73	D
Mnist8m [15]	8,100,000 × 784	0.25	S
Netflix [44]	480,189 × 17,770	0.012	S
Amazon [38, 62]	8,026,324 × 2,330,066	0.0000012	S

Table 2: ML Algorithms and Configurations.

Name	Type	Icpt	λ	ϵ	MaxIter
L2SVM	2 classes	0	10^{-3}	10^{-12}	20 (∞)
MLogreg	2 classes	0	10^{-3}	10^{-12}	20 (10)
GLM	bin.-probit	0	10^{-3}	10^{-12}	20 (10)
KMeans	1 run, k=5	N/A	N/A	10^{-12}	20
ALS-CG	rank=20, wL2	N/A	10^{-3}	10^{-12}	20 (rank)
AutoEncoder	|batch|=512 H_1=500, H_2=2	N/A	N/A	N/A	$\frac{\text{nrow}(\mathbf{X})}{\lvert\text{batch}\rvert}$

nodes (2x6 Intel Xeon E5-2440 @ 2.40 GHz-2.90 GHz, hyper-threading, 96 GB DDR3 RAM @1.33 GHz, registered ECC, 12x2 TB disks), 10Gb Ethernet, and CentOS Linux 7.4. The nominal peak memory bandwidth and compute per node are 2x32 GB/s from local memory, (47.9 GB/s measured with a modified STREAM [63]), 2x12.8 GB/s over QPI (Quick Path Interconnect), and 2x115.2 GFLOP/s. We used OpenJDK 1.8.0_161, Python 2.7.5, Apache Hadoop 2.7.3, and Apache Spark 2.2.0, in yarn-client mode, with 6 executors, 24 cores per executor, 35 GB driver memory, 65 GB executor memory, and default memory fractions (0.6/0.5), which results in an aggregate cluster memory of $6 \cdot 65\,\text{GB} \cdot 0.6 = 234\,\text{GB}$.

Datasets and ML Algorithms: To study different data characteristics, we use both synthetic and real datasets. The synthetic datasets were created with algorithm-specific data generation scripts, and the real datasets are summarized in Table 1. Airline78 refers to the years 2007/2008 of the Airline dataset [4], Mnist8m is a scaled version of the Mnist60k dataset of hand-written digits, created with the InfiMNIST data generator [15], Netflix is the Netflix Prize user-movie rating dataset [44], and Amazon is the books category of the Amazon product review dataset [38, 62]. With the goal of reflecting the diversity of ML algorithms, we conduct end-to-end experiments for six algorithms from classification, regression, clustering, matrix factorization, and neural networks. Table 2 shows these algorithms and their configurations. The parameters Icpt, λ, ϵ, and MaxIter refer to the intercept type, the regularization, the convergence tolerance, and the maximum number of outer (and inner) iterations.

Baselines: As baseline comparisons, we use the following systems with consistent double precision (i.e., `FP64`) inputs:

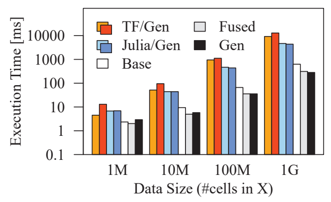
- *SystemML 1.0++ (Feb'18):* The baselines are **Base**, and **Fused** (hand-coded fused operators, SystemML's default). **Gen** is our exact, cost-based optimizer (without approximation), but we also compare the fuse-all (**FA**) and fuse-no-redundancy (**FNR**) heuristics.
- *Julia 0.6.2:* As a baseline with LLVM code generation, we use **Julia** [11] (without fusion), and **JuliaGen** (with fusion based on Julia's dot syntax) [43]. Similar to SystemML, Julia dispatches operations internally to sparse and dense kernels for all operations.
- *TensorFlow 1.5:* We also compare TensorFlow (**TF**) [1] (without fusion), and **TFGen**, i.e., TensorFlow XLA [36], but only for dense micro benchmarks due to very limited support for sparse tensors. We built TF from sources with `-march=native -O3` to enable XLA.

5.2 Operations Performance

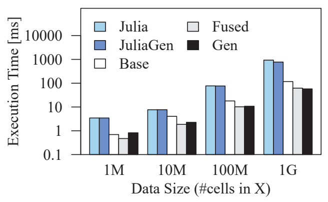
In a first set of experiments, we study the multi-threaded performance of our four templates on representative expressions, which have been introduced in Figure 1. These experiments were run on a single worker node, through SystemML's JMLC API (prepared scripts with in-memory inputs), and with the JVM flags `-Xmx80g -Xms80g -Xmn8g -server`. We used warmup runs for JIT compilation and report the mean runtime of 20 subsequent runs, including recompilation (and thus, CPlan construction) overhead.

Cell Operations: Figures 9(a) and 9(b) show the runtimes for sum($\mathbf{X} \odot \mathbf{Y} \odot \mathbf{Z}$) over dense and sparse data. Each input is of size $m \times 10^3$ (with sparsity 0.1 for sparse data), where we vary $m \in (10^3, 10^4, 10^5, 10^6)$. For the small $10^3 \times 10^3$ input (i.e., 8 MB), Fuse and Gen are only 4x faster because intermediates fit into the L3 cache (15 MB). As we increase the datasize, Fused and Gen yield a 10x improvement and reach peak single-socket/remote memory bandwidth of $\approx 25\,\text{GB/s}$. In contrast, JuliaGen shows only moderate improvements over Julia because the aggregation is not fused and both are single-threaded. TF's multi-threaded operations are competitive for small data due to the reuse of allocated intermediates. However, Gen is 2.4x faster for larger data because TF still writes intermediates. TFGen shows a consistent slowdown due to single-threaded operations. Operator fusion for sparse inputs is more challenging; in fact, JuliaGen causes a slowdown due to sparse lookups. Gen handles such cases more efficiently via stateful iterators under the covers of the stateless `getValue()` abstraction.

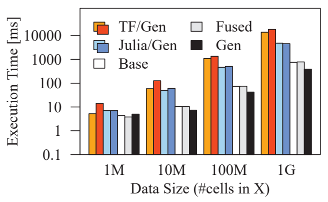
Multi-Aggregate Operations: Figure 9(c) and 9(d) show the runtimes for the two aggregates sum($\mathbf{X} \odot \mathbf{Y}$) and sum($\mathbf{X} \odot \mathbf{Z}$) over dense and sparse data as describe before. These aggregates qualify as multi-aggregate due to their shared input $\mathbf{X}$. The characteristics are similar to Cell operations with two notable differences. First, the performance of Julia and JuliaGen are identical—except for special cases with different garbage collection behavior—because Julia does neither fuse element-wise operations with aggregations nor consider multi-aggregates. Second, the hand-coded operators of Fused only apply to sum($\mathbf{X} \odot \mathbf{Y}$) and sum($\mathbf{X} \odot \mathbf{Z}$) individually, causing a redundant read of $\mathbf{X}$. In contrast, Gen



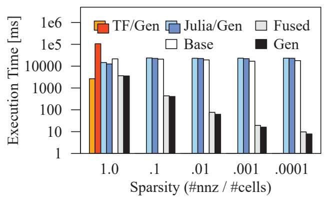
(a) $\mathbf{X}^\top(\mathbf{w} \odot (\mathbf{X}\mathbf{v}))$, D



(b) $\mathbf{X}^\top(\mathbf{w} \odot (\mathbf{X}\mathbf{v}))$, S



(c) $\mathbf{X}^\top(\mathbf{W} \odot (\mathbf{X}\mathbf{V}))$, D



(d) $\text{sum}(\mathbf{X} \odot \log(\mathbf{U}\mathbf{V}^\top + 10^{-15}))$

Figure 10: Operations Performance of Example Row and Outer Patterns (D .. dense, S .. sparse).

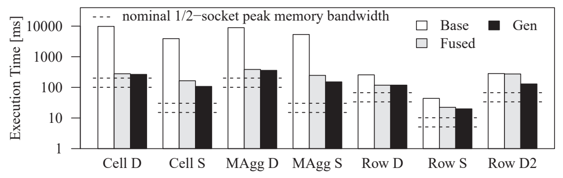


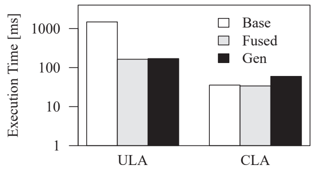
Figure 11: 1G Benchmarks on Xeon Gold 6138.

compiles a multi-aggregate (with 2×1 output vector), and in case of sparse data, correctly selects $\mathbf{X}$ as sparse driver which makes the entire multi-aggregate sparse-safe.

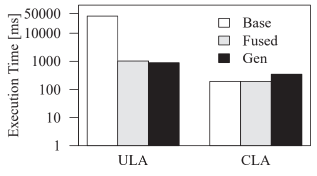
Row Operations: Row-wise operations are also very common. Figures 10(a) and 10(b) show the runtimes for a matrix-vector multiplication chain with weighting $\mathbf{X}^\top(\mathbf{w} \odot (\mathbf{X}\mathbf{v}))$ over sparse and dense data, where we vary the size of $\mathbf{X}$ as before and $\mathbf{v}$ is a $10^3 \times 1$ vector. Julia does not fuse these matrix-vector operations and suffers from single-threaded execution. TF's runtime is dominated by transposing $\mathbf{X}$ because $\mathbf{X}^\top(\mathbf{w} \odot (\mathbf{X}\mathbf{v}))$ is not rewritten into $((\mathbf{w} \odot (\mathbf{X}\mathbf{v}))^\top\mathbf{X})^\top$. With a modified DAG, the TF runtime improves from 9.2 s to 1.6 s. TFGen causes again a consistent slowdown. In contrast, Fuse, and Gen (283 ms) yield peak single-socket/remote memory bandwidth, and a 2x improvement over Base by exploiting temporal row locality, where each 8 KB row fits into the 32 KB L1 cache. Figure 10(c) further shows the results of a matrix-matrix multiplication chain $\mathbf{X}^\top(\mathbf{W} \odot (\mathbf{X}\mathbf{V}))$ over dense data, where $\mathbf{V}$ is a $10^3 \times 2$ matrix. Base and Fused are equivalent because the hand-coded `mmchain` operator only applies to matrix-vector chains. In contrast, Gen yields again a 2x improvement.

Outer-Product Operations: Figure 10(d) shows the runtime of an outer-product expression $\text{sum}(\mathbf{X} \odot \log(\mathbf{U}\mathbf{V}^\top + 10^{-15}))$, which can exploit sparsity over $\mathbf{X}\odot$. We set the size of $\mathbf{X}$ to $2 \cdot 10^4 \times 2 \cdot 10^4$, the rank of $\mathbf{U}$ and $\mathbf{V}$ to 100, and vary the sparsity of $\mathbf{X}$ with $\text{sp} \in (1, 10^{-1}, 10^{-2}, 10^{-3}, 10^{-4})$. Base, Julia, and JuliaGen show almost constant runtime, which means Julia does not exploit sparsity. Julia calls native BLAS matrix multiplications but this expression is largely dominated by the costs for log(), where $\mathbf{U}\mathbf{V}^\top$ attributes to less than 15%. For this reason, Base with native BLAS only slightly improved performance. For dense data, TF shows very good performance but does not support sparse operations. In contrast, Fused and Gen achieve, for $\text{sp} = 10^{-4}$, an improvement of three orders of magnitude and even if $\mathbf{X}$ is dense, an improvement of 5x compared to Base, due to multi-threaded execution without intermediates.

Modern Server HW: For validation, we repeated selected micro benchmarks on a modern server of 2x20 Intel Xeon Gold 6138 @ 2.00 GHz-3.70 GHz, hyper-threading, 768 GB DDR4 RAM @2.66 GHz, reg. ECC (peak bandwidth: memory 2x119.21 GB/s, compute 2x1.25 TFLOP/s). Figure 11 shows the results for the large 1G Cell, MAgg, and Row scenarios (9(a)-9(d), 10(a)-10(c)). Overall, we see similar characteristics but larger speedups due to the higher degree of parallelism (80 vcores), which increases the relative Base overheads for allocating and writing intermediates.



(a) Airline78 Dataset (dense)

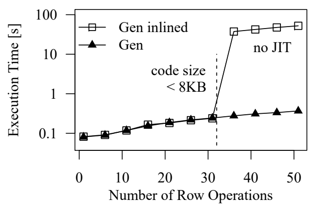


(b) Mnist8m Dataset (sparse)

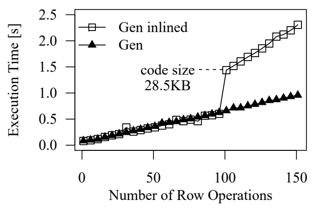
Figure 12: Compressed Operations: $\text{sum}(\mathbf{X}^2)$.

Compressed Linear Algebra (CLA): All templates support operations over compressed matrices with column-wise compression, heterogeneous encoding formats, and column co-coding [30]. Figure 12 shows the runtime for computing the sparse-safe expression $\text{sum}(\mathbf{X}^2)$ over Airline78 and Mnist8m. For these datasets, CLA achieves compression ratios of 7.44x and 7.32x. On uncompressed data (ULA), fused operators yield similar speedups as for synthetic data because they avoid the expensive materialization of $\mathbf{X}^2$. On compressed data (CLA), however, Base and Fused show equivalent performance for this special case, because $\mathbf{X}^2$ is only computed over the dictionary of distinct values with a shallow copy of the compressed data. CLA achieves substantial improvements due to computing the sum via counts per value and reduced memory bandwidth requirements. The Gen templates similarly call—under the conditions of a single input and sparse-safe operations—the generated operator only for distinct values, which achieves performance remarkably close to hand-coded CLA operations.

Instruction Footprint: Separating operator skeletons and vector primitives from the generated operators reduces the instruction footprint. To evaluate its impact, we use $\text{sum}(f(\mathbf{X}/\text{rowSums}(\mathbf{X})))$, where we generate f as a sequence of k row operations $\mathbf{X} \odot i$ and $\mathbf{X}$ as a dense $10^5 \times 10^3$ matrix (800 MB). Gen uses the vector primitive `vectMultWrite` (with 8-fold loop unrolling) and—independent of k—two vector intermediates per thread. Figure 13(a) shows the runtime of Gen and Gen inlined, where the latter inlines `vectMultWrite`. For more than 31 operations, Gen inlined is two orders of magnitude slower because the code size of its

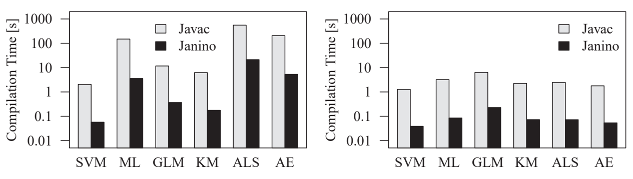


(a) Default Configuration



(b) `-XX:-DontCompileHugeMethods`

Figure 13: Impact of Instruction Footprint.



(a) Without Operator Cache (b) With Operator Cache

Figure 14: Java Class Compilation and Loading.

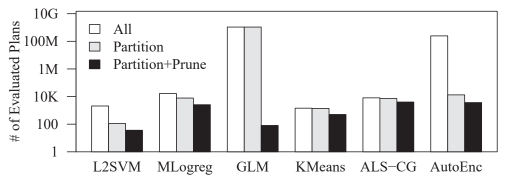


Figure 15: Plan Enumeration and Pruning.

`genexec` method exceeds 8 KB, which is the JVM threshold for JIT compilation. Figure 13(b) reports the runtime with disabled threshold, where both operators show the same runtime up to 96 operations. However, for 101 and more operations, Gen inlined does no longer fit into the L1 instruction cache (32 KB), which leads to a significant slowdown.

5.3 Compilation Overhead

In a second set of experiments, we investigate the compilation overhead of code generation and optimization. Since the relative overhead decreases with increasing input size, we use the very small Mnist60k dataset ($60\text{K} \times 784$, sparse). We use again the single worker node setup and report algorithm-level statistics as the mean of 5 runs, including read times.

Operator Compilation: Gen uses an operator cache for reusing compiled operators (across DAGs and during dynamic recompilation) as well as the fast `janino` compiler. Figure 14 shows the impact of these components on the compilation overhead. There are two major insights. First, `janino` consistently improves the performance of class compilation and loading by one and a half orders of magnitude compared to the standard `javac` compiler. Second, the operator cache significantly reduces the compilation overhead, especially for algorithms with dynamic recompilation (i.e., MLogreg, GLM, ALS-CG, and AutoEncoder). The observed operator cache hit rates of the six algorithms are 8/20, 1,492/1,520, 56/115, 46/67, 5,632/5,657, and 2,241/2,260. Besides the reduced compilation overhead, the plan cache also reduces the asynchronous JIT compilation overhead and thus, improves the end-to-end performance. For example, on ALS-CG, the JIT compilation time reduced from 262 s to 25 s, which improved the runtime from 171 s to 81 s.

Plan Enumeration: The second major overhead is the cost-based plan selection due to its exponential search space and the need for DAG traversal when costing a plan. Figure 15 shows the total number of evaluated—i.e., costed—plans, for the six algorithms and different configurations without partitioning (all), with partitioning (partition), and with partitioning and both pruning techniques (partition+prune). Overall, none of the algorithms requires more than a few thousand plans, for two reasons. First, focusing on interesting points and optimizing partitions—i.e., connected components of fusion plans—independently, is very impactful. For example, the largest DAG of AutoEncoder has 71 operators with partial fusion plans after candidate exploration, which would result in an infeasible number of $2^{71} > 10^{21}$ plans. Instead we only consider interesting points and optimize partitions independently. This partitioning reduces the number of plans by more than four orders of magnitude. Second, the individual pruning techniques, but especially cost-based pruning, are very effective as well. For example on GLM, pruning reduces the number of evaluated plans by almost seven orders of magnitude, which rendered an analysis of optimization time without pruning infeasible.

Table 3: End-to-End Compilation Overhead.

Name	Total [s]	# Compile	Compile [ms]
L2SVM	1.2	14 / 20 / 12	**55** (39)
MLogreg	3.2	426 / 1,580 / 28	**366** (85)
GLM	3.0	212 / 126 / 59	**350** (230)
KMeans	2.0	65 / 75 / 21	**112** (73)
ALS-CG	81.1	1,662 / 5,658 / 25	**1,067** (72)
AutoEncoder	26.1	132 / 2,260 / 19	**491** (63)

Codegen Statistics: Table 3 summarizes the resulting codegen statistics for our ML algorithms. These statistics include the execution time, number of compiled plans (optimized HOP DAGs, created CPlans, and compiled classes), as well as the compilation overhead (total codegen and class compilation time). Overall, the overhead is very small—below one second for most algorithms—despite a substantial number of optimized DAGs (up to 1,662), constructed CPlans (up to 5,658), and compiled operators (up to 59). To summarize, effective pruning techniques, a simple operator cache, and the fast `janino` compiler significantly reduce the compilation overhead, which makes codegen practical.

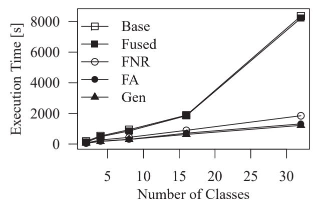
5.4 Single-Node End-to-End Experiments

Our third set of experiments studies the end-to-end performance impact on ML algorithms. Given the results of our micro benchmarks, we mostly restrict this comparison to Base, Fused, and Gen but also include the fusion heuristics FA, and FNR to evaluate the quality of fusion plans. We report the end-to-end algorithm runtime—invoked through `spark-submit` with 35 GB driver—as a mean of 3 runs.

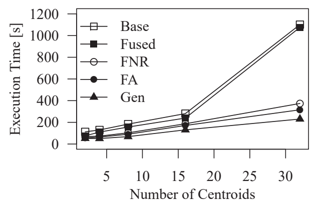
Data-Intensive Algorithms: Many traditional ML algorithms are data-intensive, i.e., memory-bandwidth bound. In addition to scans of the feature matrix $\mathbf{X}$, these algorithms often use many vector and matrix operations, which become a bottleneck for small or large numbers of features.

Table 4: Runtime of Data-Int. Algorithms [s].

Name	Data	Base	Fused	Gen	FA	FNR
L2SVM	$10^6 \times 10$	7	5	**3**	3	4
	$10^7 \times 10$	42	28	**6**	7	13
	$10^8 \times 10$	446	276	**37**	44	92
	Airline78	151	105	**24**	26	45
	Mnist8m	203	156	**113**	115	116
MLogreg	$10^6 \times 10$	10	9	**5**	5	6
	$10^7 \times 10$	65	55	**15**	17	27
	$10^8 \times 10$	733	538	**106**	132	307
	Airline78	190	142	**48**	52	74
	Mnist8m	478	308	**240**	307	291
GLM	$10^6 \times 10$	28	28	**11**	11	17
	$10^7 \times 10$	212	200	**22**	26	61
	$10^8 \times 10$	2,516	2,290	**140**	184	592
	Airline78	385	337	**50**	54	112
	Mnist8m	511	373	**207**	251	275
KMeans	$10^6 \times 10$	11	11	**4**	4	7
	$10^7 \times 10$	120	95	**15**	16	36
	$10^8 \times 10$	1,471	1,471	**136**	147	662
	Airline78	110	101	**32**	34	51
	Mnist8m	229	203	**160**	185	187



(a) MLogreg #Classes



(b) Kmeans #Centroids

Figure 16: Increasing Size of Intermediates.

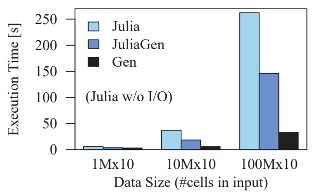
Table 5: Runtime of Compute-Int. Algorithms [s].

Name	Data	Base	Fused	Gen	FA	FNR
ALS-CG	$10^4 \times 10^4$	426	**20**	25	215	226
	$10^5 \times 10^4$	23,585	96	**80**	13,511	12,353
	$10^6 \times 10^4$	N/A	860	**722**	N/A	N/A
	Netflix	N/A	1,026	**789**	N/A	N/A
	Amazon	N/A	17,335	**7,420**	N/A	N/A
Auto-Encoder	$10^3 \times 10^4$	8	9	**7**	7	8
	$10^4 \times 10^4$	51	48	**31**	31	36
	$10^5 \times 10^4$	615	560	**286**	288	325
	Mnist1m	597	562	**379**	449	420

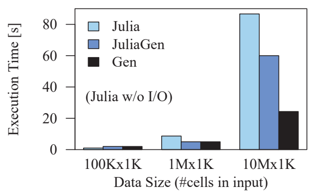
Accordingly, Table 4 shows the results for dense inputs with 10 features, but we also use real datasets. Fused shows only moderate improvements because its patterns are mostly limited to two or three operators. Compared to Fused, Gen shows significant end-to-end improvements due to fewer intermediates (which also reduces buffer pool evictions), fewer scans, and multi-threaded operations with fewer barriers and thus better utilization. On the $10^8 \times 10$ (8 GB) scenario, we see speedups of 7x, 5x, 16x, and 10x. Regarding heuristics, FA mostly outperforms FNR due to fewer intermediates. For robustness, Gen uses both FA and FNR as an opening heuristic. Gen—with its exact cost-based optimizer—then outperforms FA by up to 27% for algorithms such as L2SVM, MLogreg and GLM that exhibit complex DAG structures. In contrast to the FA and FNR heuristics, Gen also guarantees the optimality of the chosen fusion plans.

Hybrid Algorithms: MLogreg and KMeans are interesting hybrid algorithms, which change from memory-bandwidth- to compute-bound as we increase the number of classes/centroids k. Figure 16 shows the results for an input of size $10^7 \times 100$ (8 GB) and varying k. Apart from similar trends as before, there are three insights. First, the Gen runtime remains almost constant up until $k = 8$ because it is still memory-bandwidth-bound. Second, k also affects the size of intermediates ($10^7 \times k$, i.e., 2.5 GB for $k = 32$), which causes more evictions for Base and Fused. Third, for the case of $k = 2$, multiple rewrites and fused operators are applied, whereas, Gen shows very robust performance.

Compute-Intensive Algorithms: We also study the compute-intensive algorithms ALS-CG for matrix factorization and AutoEncoder for dimensionality reduction. Table 5 shows the results of ALS-CG on sparse data (sparsity 0.01), AutoEncoder on dense data, as well as real datasets. For ALS-CG, Fused and Gen show huge improvements due to sparsity exploitation in the update rules and loss computations. Gen outperforms Fused due to less evictions as the data size increases. The fusion heuristics fail to find good plans for the update rules because—without considering interesting points for sparsity exploitation—they fuse too many operations, which disallowed the use of sparse-safe Outer templates. Thus, Base, FA, and FNR are not applicable for larger datasets. Even for AutoEncoder, Gen and the fusion heuristics show a good 2x improvement, despite the mini-batch algorithm (with small intermediates), and many compute-intensive matrix-matrix multiplications.



(a) Dense



(b) Sparse (sparsity 0.01)

Figure 17: Julia Comparison for L2SVM.

Julia Comparison: In addition, we compare Julia for L2SVM as a selected ML algorithm, where manual fusion is feasible. For a fair comparison, we use the same SystemML setup as before (35 GB memory, read from HDFS), whereas Julia runs with unlimited memory and we exclude its read time. Julia and JuliaGen refer to two different scripts with (1) the materialization of all intermediates and (2) fused expressions using Julia's dot syntax with hand-tuned reuse of beneficial CSEs. Figure 17 shows the results for dense and sparse inputs. Julia performs similar to Base and Fused (see Table 4), while JuliaGen further improves performance by 2x. Gen is still almost 5x faster due to (1) fusion of element-wise operations with aggregation and matrix multiply, (2) cost-based optimization including the selection of multi-aggregates, and (3) multi-threaded execution.

5.5 Large-Scale End-to-End Experiments

Finally, we also study large-scale (i.e., distributed) algorithms. We use three datasets: D200m (200M × 100, dense, 160 GB), S200m ($200\text{M} \times 10^3$, sparsity 0.05, 121 GB), Mnist80m (81M × 784, sparsity 0.25, 204 GB), which all fit in aggregate memory (234 GB), and we report the end-to-end runtime, with 35 GB driver, as a mean of 3 runs in Table 6.

Distributed Algorithms: Gen shows again substantial improvements compared to Fused (up to 22x for KMeans). Unlike in the single-node experiments, however, the fusion heuristics show brittle performance characteristics. For example, FA even leads to slowdowns on L2SVM and MLogreg. This effect is caused by too eager fusion of vector operations—that could be executed at the driver—into distributed operations over large inputs. For distributed operations, these additional vector inputs (of up to 1.6 GB per vector) cause unnecessary broadcasting overhead to all 6 worker nodes and partial evictions of broadcasts—which are stored as `MEMORY_AND_DISK` in the executor's block managers—from aggregate memory. In contrast, Gen creates good plans by reasoning about template switches and broadcast costs.

Table 6: Runtime of Distributed Algorithms [s].

Name	Data	Base	Fused	Gen	FA	FNR
L2SVM	D200m	1,218	895	**347**	1,433	539
	S200m	1,481	1,066	**373**	2,205	575
	Mnist80m	1,593	1,114	**552**	1,312	896
MLogreg	D200m	5,435	3,872	**2,695**	3,591	5,943
	S200m	4,386	3,904	**2,705**	3,414	3,915
	Mnist80m	5,105	4,250	**3,377**	8,936	7,883
GLM	D200m	12,365	10,921	**1,913**	2,080	4,115
	S200m	11,852	10,681	**1,935**	2,269	3,770
	Mnist80m	5,303	3,876	**1,246**	1,607	2,399
KMeans	D200m	5,452	5,426	**321**	331	6,913
	S200m	5,255	5,205	241	**238**	7,325
	Mnist80m	2,182	2,164	**356**	510	3,881

6. RELATED WORK

We review work from query compilation, loop and operator fusion, and the optimization of DAGs and fusion plans.

Query Compilation: Already System R compiled SQL statements—for repetitive transactions—into machine code [19, 20], but compilation was later abandoned due to maintenance and debugging issues [82]. Motivated by the trend toward in-memory databases, query compilation was then reconsidered by JAMDB [82], HIQUE [56], DBToaster [50], and HyPer [70]. Kennedy et al. introduced the compilation of incremental view maintenance programs in DBToaster [50], while Neumann made a case for LLVM-based query compilation in HyPer to support ad-hoc queries with low compilation overhead [70]. LegoBase [53, 89] and DBLAB/L [88] focused on a modular compilation chain to exploit relational and compiler optimizations. Several systems also include restricted ML workloads into query compilation. Examples are the compilation of UDF-centric workflows in Tupleware [25], Lambda expressions in Hyper [78], the LLVM-based compilation of Java UDFs [86], and query compilation with UDFs in Flare [31]. Lang et al. explored the integration with scans over compressed blocks in Hyper [57], which—similar to our Row template over compressed matrices—extracts tuples to temporary storage. Menon et al. further introduced the notion of relaxed operator fusion to reason about temporary materialization in Peloton [65]. Additional directions are abstractions for different HW backends in Voodoo [80], operator fusion for GPUs in HorseQC [34], and compiled data access over heterogeneous formats in Proteus [45]. Meanwhile, query compilation is heavily used in many modern data systems such as Hyper [71], Impala [101], Hekaton [33], MemSQL [90], Tupleware [25], Peloton [79], and SparkSQL [5]. However, most of these systems do not handle DAGs, linear algebra, or the challenges of sparsity exploitation.

Loop and Operator Fusion: Loop fusion, tiling and distribution [3, 49] aim at merging multiple loops into combined loops and vice versa—without introducing redundancy or loop-carried dependencies—to improve locality, parallelism, or memory requirements. Existing work typically relies on the affine [58] or polyhedral [81, 100] models to build an inter-loop dependency graph [3]. Since loop fusion is known to be NP-complete [27, 49], typically greedy [48] or heuristic [64] methods are used. Also, loop fusion usually only considers dense data access. Recent research aims at specialized IRs for staged transformations—which does allow sparsity exploitation for restricted cases of unary operations—[84], normalization of comprehensions in Emma [2], distributed applications on heterogeneous hardware [16], and cross-library optimization in Weld [76, 77]. In ML systems, operator fusion aims at merging multiple operations over matrices or tensors into fused operations. In contrast to loop fusion, the dependencies are implicitly given by the data flow graph and operation semantics [9]. SystemML uses rewrites to replace patterns with hand-coded, local or distributed fused operators [7, 14, 40]. Other systems like Cumulon [39] and MatFast [103] use more generic masked and folded binary operators to exploit sparsity, which still require the materialization of masks and sparse intermediates. Automatic operator fusion addresses these limitations. BTO [9] introduced a refine-and-optimize approach for fusing BLAS Level 1/2 operations in local linear algebra kernels, whereas OptiML [97] provided operator fusion for both CPU and GPUs. Tupleware [25, 26] and Kasen [105] introduced operator fusion for distributed programs. SystemML-SPOOF [29] also supports operator fusion for local and distributed operations, as well as sparsity-exploitation. Additionally, Sparso [85] propagates context in sparse linear algebra programs. Recent work like TVM [21, 22] also considers FPGAs and ASICs. Meanwhile, operator fusion and code generation are being integrated into many systems in practice. Examples are SystemML, TensorFlow XLA [1, 36], Julia [11, 43], MATLAB [61], Intel Nervana Graph [54], NVIDIA TensorRT [75], and TC (Caffe2, PyTorch) [100]. However, all these systems rely on fusion heuristics or manual declaration of fusion plans.

Optimizing DAGs and Fusion Plans: Large operator DAGs are ubiquitous in ML workloads, which is challenging due to their scale and missing optimal substructure. Neumann pioneered the work on generating optimal DAG-structured query plans [68, 72], while others heuristically share CSEs via materialized views [68, 92, 106] or common operators [6, 18, 35]. Recent work further introduced a greedy algorithm with guaranteed approximation factor [46]. Sideways information passing such as semi-join reductions [10], magic sets [8], bypass plans for disjunctive queries [95], or adaptive information passing [42, 74] also deal with DAGs, but are not integrated with query compilation. Although most ML systems have compiler and runtime support for DAGs, their rewrite systems—such as SystemML's static and dynamic rewrites [13] or KeystoneML's cache management [94]—handle DAGs in a heuristic or greedy manner. Similarly, the literature on optimizing fusion plans is very sparse. Frameworks such as OptiML [97], Emma [2], Kasen [105], Voodoo [80], SystemML-SPOOF [29], Weld [76, 77], and TensorFlow XLA [1, 36] all use fusion heuristics, which miss opportunities. Tupleware [25] combines heuristics and cost-based decisions for micro-optimizations such as predication and loop tiling. Furthermore, BTO (Build to Order BLAS) [9] and TC (Tensor Comprehensions) [100] use k-greedy and evolutionary algorithms to iteratively refine linear algebra kernels. In contrast, we focus on entire operator DAGs, *exact* cost-based optimization (with optimality guarantee), and sparsity exploitation across operators.

7. CONCLUSIONS

We introduced an exact, cost-based optimization framework for operator fusion plans over DAGs of linear algebra operations, and described its compiler and runtime integration into SystemML. Our experiments show that optimized fusion plans match the performance of hand-coded fused operators, and yield—due to their generality and cost-awareness—significant end-to-end improvements compared to hand-coded operators and fusion heuristics. In conclusion, we believe that the optimization of fusion plans is a cornerstone of future declarative, large-scale ML systems. The major benefits are the high performance impact, the reduced development effort, and the broad applicability with regard to ML algorithms, dense, sparse, and compressed data, as well as local and distributed operations. Interesting future work includes—as outlined in the SPOOF vision [29]—the holistic optimization of fusion plans and rewrites, the inclusion of additional operation types, and code generation for heterogeneous hardware including GPUs.

Acknowledgement: We thank Tarek Elgamal for his contributions to the original SPOOF codegen framework, and our anonymous reviewers for their constructive feedback.

8. REFERENCES

[1] M. Abadi et al. TensorFlow: A System for Large-Scale Machine Learning. In *OSDI*, pages 265–283, 2016.

[2] A. Alexandrov et al. Implicit Parallelism through Deep Language Embedding. In *SIGMOD*, pages 47–61, 2015.

[3] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[4] American Statistical Association (ASA). Airline on-time performance dataset. `stat-computing.org/dataexpo/2009/the-data.html`.

[5] M. Armbrust et al. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*, pages 1383–1394, 2015.

[6] S. Arumugam et al. The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses. In *SIGMOD*, pages 519–530, 2010.

[7] A. Ashari et al. On Optimizing Machine Learning Workloads via Kernel Fusion. In *PPoPP*, pages 173–182, 2015.

[8] F. Bancilhon et al. Magic Sets and Other Strange Ways to Implement Logic Programs. In *PODS*, pages 1–15, 1986.

[9] G. Belter et al. Automating the Generation of Composed Linear Algebra Kernels. In *SC*, pages 59:1–59:12, 2009.

[10] P. A. Bernstein and D. W. Chiu. Using Semi-Joins to Solve Relational Queries. *J. ACM*, 28(1):25–40, 1981.

[11] J. Bezanson et al. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, 2017.

[12] M. Boehm et al. Hybrid Parallelization Strategies for Large-Scale Machine Learning in SystemML. *PVLDB*, 7(7):553–564, 2014.

[13] M. Boehm et al. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.*, 37(3):52–62, 2014.

[14] M. Boehm et al. SystemML: Declarative Machine Learning on Spark. *PVLDB*, 9(13):1425–1436, 2016.

[15] L. Bottou. The infinite MNIST dataset. `leon.bottou.org/projects/infimnist`.

[16] K. J. Brown et al. Have Abstraction and Eat Performance, Too: Optimized Heterogeneous Computing with Parallel Patterns. In *CGO*, pages 194–205, 2016.

[17] N. Bruno and R. V. Nehme. Configuration-Parametric Query Optimization for Physical Design Tuning. In *SIGMOD*, pages 941–952, 2008.

[18] G. Candea et al. A Scalable, Predictable Join Operator for Highly Concurrent Data Warehouses. *PVLDB*, 2(1):277–288, 2009.

[19] D. D. Chamberlin et al. A History and Evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.

[20] D. D. Chamberlin et al. Support for Repetitive Transactions and Ad Hoc Queries in System R. *ACM Trans. Database Syst.*, 6(1):70–94, 1981.

[21] T. Chen et al. TVM: End-to-End Compilation Stack for Deep Learning. In *SysML*, 2018.

[22] T. Chen et al. TVM: End-to-End Optimization Stack for Deep Learning. *CoRR*, 2018.

[23] J. Cohen et al. MAD Skills: New Analysis Practices for Big Data. *PVLDB*, 2(2):1481–1492, 2009.

[24] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1991.

[25] A. Crotty et al. An Architecture for Compiling UDF-centric Workflows. *PVLDB*, 8(12):1466–1477, 2015.

[26] A. Crotty et al. Tupleware: "Big" Data, Big Analytics, Small Clusters. In *CIDR*, 2015.

[27] A. Darte. On the complexity of loop fusion. *Parallel Computing*, 26(9):1175–1193, 2000.

[28] A. Deshpande et al. Adaptive Query Processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[29] T. Elgamal et al. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. In *CIDR*, 2017.

[30] A. Elgohary et al. Compressed Linear Algebra for Large-Scale Machine Learning. *PVLDB*, 9(12):960–971, 2016.

[31] G. M. Essertel et al. Flare: Native Compilation for Heterogeneous Workloads in Apache Spark. *CoRR*, 2017.

[32] M. Ferdman et al. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*, pages 37–48, 2012.

[33] C. Freedman et al. Compilation in the Microsoft SQL Server Hekaton Engine. *IEEE Data Eng. Bull.*, 37(1):22–30, 2014.

[34] H. Funke et al. Pipelined Query Processing in Coprocessor Environments. In *SIGMOD*, pages 1603–1618, 2018.

[35] G. Giannikis et al. Shared Workload Optimization. *PVLDB*, 7(6):429–440, 2014.

[36] Google. TensorFlow XLA (Accelerated Linear Algebra). `tensorflow.org/performance/xla`.

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

[38] R. He and J. McAuley. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *WWW*, pages 507–517, 2016.

[39] B. Huang et al. Cumulon: Optimizing Statistical Data Analysis in the Cloud. In *SIGMOD*, pages 1–12, 2013.

[40] B. Huang et al. Resource Elasticity for Large-Scale Machine Learning. In *SIGMOD*, pages 137–152, 2015.

[41] Y. E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results. In *SIGMOD*, pages 268–277, 1991.

[42] Z. G. Ives and N. E. Taylor. Sideways Information Passing for Push-Style Query Processing. In *ICDE*, pages 774–783, 2008.

[43] S. G. Johnson. More Dots: Syntactic Loop Fusion in Julia. `julialang.org/blog/2017/01/moredots`.

[44] Kaggle. Netflix Prize Data. `kaggle.com/netflix-inc/netflix-prize-data`.

[45] M. Karpathiotakis et al. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB*, 9(12):972–983, 2016.

[46] T. Kathuria and S. Sudarshan. Efficient and Provable Multi-Query Optimization. In *PODS*, pages 53–67, 2017.

[47] G. Keller et al. Regular, Shape-polymorphic, Parallel Arrays in Haskell. In *ICFP*, pages 261–272, 2010.

[48] K. Kennedy. Fast Greedy Weighted Fusion. *International Journal of Parallel Programming*, 29(5):463–491, 2001.

[49] K. Kennedy and K. S. McKinley. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *LPPC*, pages 301–320, 1993.

[50] O. Kennedy et al. DBToaster: Agile Views for a Dynamic Data Management System. In *CIDR*, pages 284–295, 2011.

[51] Y. Kim. Convolutional Neural Networks for Sentence Classification. In *EMNLP*, pages 1746–1751, 2014.

[52] F. Kjolstad et al. The Tensor Algebra Compiler. *PACMPL*, 1(OOPSLA):77:1–77:29, 2017.

[53] Y. Klonatos et al. Building Efficient Query Engines in a High-Level Language. *PVLDB*, 7(10):853–864, 2014.

[54] J. Knight. Intel Nervana Graph Beta. `intelnervana.com/intel-nervana-graph-and-neon-3-0-updates`.

[55] C. Koch and D. Olteanu. Conditioning Probabilistic Databases. *PVLDB*, 1(1):313–325, 2008.

[56] K. Krikellas et al. Generating Code for Holistic Query Evaluation. In *ICDE*, pages 613–624, 2010.

[57] H. Lang et al. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*, pages 311–326, 2016.

[58] A. W. Lim and M. S. Lam. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *POPL*, pages 201–214, 1997.

[59] H. Lim et al. Stubby: A Transformation-based Optimizer for MapReduce Workflows. *PVLDB*, 5(11):1196–1207, 2012.

[60] S. Luo et al. Scalable Linear Algebra on a Relational Database System. In *ICDE*, pages 523–534, 2017.

[61] MathWorks. GPU Coder: Generate CUDA code for NVIDIA GPUs. `mathworks.com/products/gpu-coder`.

[62] J. McAuley. Amazon Product Data - Books. `jmcauley.ucsd.edu/data/amazon`.

[63] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. `cs.virginia.edu/stream`.

[64] S. Mehta et al. Revisiting Loop Fusion in the Polyhedral Framework. In *PPoPP*, pages 233–246, 2014.

[65] P. Menon et al. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *PVLDB*, 11(1):1–13, 2017.

[66] G. Moerkotte and T. Neumann. Analysis of Two Existing and One New Dynamic Programming Algorithm for the Generation of Optimal Bushy Join Trees without Cross Products. In *VLDB*, pages 930–941, 2006.

[67] G. Moerkotte and T. Neumann. Dynamic Programming Strikes Back. In *SIGMOD*, pages 539–552, 2008.

[68] T. Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs*. PhD thesis, Universitaet Mannheim, 2005.

[69] T. Neumann. Query Simplification: Graceful Degradation for Join-Order Optimization. In *SIGMOD*, pages 403–414, 2009.

[70] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB*, 4(9):539–550, 2011.

[71] T. Neumann and V. Leis. Compiling Database Queries into Machine Code. *IEEE Data Eng. Bull.*, 37(1):3–11, 2014.

[72] T. Neumann and G. Moerkotte. Generating Optimal DAG-structured Query Evaluation Plans. *Computer Science - R&D*, 24(3):103–117, 2009.

[73] T. Neumann and B. Radke. Adaptive Optimization of Very Large Join Queries. In *SIGMOD*, pages 677–692, 2018.

[74] T. Neumann and G. Weikum. Scalable Join Processing on Very Large RDF Graphs. In *SIGMOD*, pages 627–640, 2009.

[75] NVIDIA. TensorRT - Programmable Inference Accelerator. `developer.nvidia.com/tensorrt`.

[76] S. Palkar et al. A Common Runtime for High Performance Data Analysis. In *CIDR*, 2017.

[77] S. Palkar et al. Weld: Rethinking the Interface Between Data-Intensive Applications. *CoRR*, 2017.

[78] L. Passing et al. SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In *EDBT*, pages 84–95, 2017.

[79] A. Pavlo et al. Self-Driving Database Management Systems. In *CIDR*, 2017.

[80] H. Pirk et al. Voodoo - A Vector Algebra for Portable Database Performance on Modern Hardware. *PVLDB*, 9(14):1707–1718, 2016.

[81] L. Pouchet et al. Loop Transformations: Convexity, Pruning and Optimization. In *POPL*, pages 549–562, 2011.

[82] J. Rao et al. Compiled Query Execution Engine using JVM. In *ICDE*, 2006.

[83] T. Rohrmann et al. Gilbert: Declarative Sparse Linear Algebra on Massively Parallel Dataflow Systems. In *BTW*, pages 269–288, 2017.

[84] T. Rompf et al. Optimizing Data Structures in High-Level Programs: New Directions for Extensible Compilers based on Staging. In *POPL*, pages 497–510, 2013.

[85] H. Rong et al. Sparso: Context-driven Optimizations of Sparse Linear Algebra. In *PACT*, pages 247–259, 2016.

[86] V. Rosenfeld et al. Processing Java UDFs in a C++ environment. In *SoCC*, pages 419–431, 2017.

[87] S. Schelter et al. Samsara: Declarative Machine Learning on Distributed Dataflow Systems. *NIPS Workshop MLSystems*, 2016.

[88] A. Shaikhha et al. How to Architect a Query Compiler. In *SIGMOD*, pages 1907–1922, 2016.

[89] A. Shaikhha et al. Building Efficient Query Engines in a High-Level Language. *ACM Trans. Database Syst.*, 43(1):4:1–4:45, 2018.

[90] N. Shamgunov. MemSQL 5 Ships with LLVM-based Code Generation for SQL Queries. `blog.memsql.com/memsql-5-ships`.

[91] L. D. Shapiro et al. Exploiting Upper and Lower Bounds In Top-Down Query Optimization. In *IDEAS*, pages 20–33, 2001.

[92] Y. N. Silva et al. Exploiting Common Subexpressions for Cloud Query Processing. In *ICDE*, pages 1337–1348, 2012.

[93] U. Sirin et al. Micro-architectural Analysis of In-memory OLTP. In *SIGMOD*, pages 387–402, 2016.

[94] E. R. Sparks et al. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *ICDE*, pages 535–546, 2017.

[95] M. Steinbrunn et al. Bypassing Joins in Disjunctive Queries. In *VLDB*, pages 228–238, 1995.

[96] M. Stonebraker et al. The Architecture of SciDB. In *SSDBM*, pages 1–16, 2011.

[97] A. K. Sujeeth et al. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*, pages 609–616, 2011.

[98] C. Szegedy et al. Rethinking the Inception Architecture for Computer Vision. In *CVPR*, pages 2818–2826, 2016.

[99] A. Unkrig. Janino: A super-small, super-fast Java compiler. `janino-compiler.github.io/janino/`.

[100] N. Vasilache et al. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. In *CoRR*, 2018.

[101] S. Wanderman-Milne and N. Li. Runtime Code Generation in Cloudera Impala. *IEEE Data Eng. Bull.*, 37(1):31–37, 2014.

[102] L. Yu et al. Exploiting Matrix Dependency for Efficient Distributed Matrix Computation. In *SIGMOD*, pages 93–105, 2015.

[103] Y. Yu et al. In-Memory Distributed Matrix Computation Processing and Optimization. In *ICDE*, pages 1047–1058, 2017.

[104] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, pages 15–28, 2012.

[105] M. Zhang et al. Measuring and Optimizing Distributed Array Programs. *PVLDB*, 9(12):912–923, 2016.

[106] J. Zhou et al. Efficient Exploitation of Similar Subexpressions for Query Processing. In *SIGMOD*, pages 533–544, 2007.