Holistic Sum-Product Optimization for Large-Scale Machine Learning

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

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

ABSTRACT

blah blah

1. INTRODUCTION

1. ML is often expressed in terms of LA.
2. Rewriting is important: LA is rich with identities that, when computed literally, drastically alter the computation runtime of training and inference.
3. Rewriting is a hard problem: the space of equivalent expressions of a ML script is large. No deterministic set of rules can find the optimal (fastest) expressions without overwhelming engineering and maintenance efforts.
4. Related work has addressed the problem of rewriting in limited ways: sum-product optimization frameworks (FAQ [1], AJAR, PANDA) provide a method to find the optimal expression for a limited class of expressions: sums over the product of distinct inputs, wherein the sum operations are semiring (or identical) to a unique product operation.
5. Real world ML scripts have more complexity: CSEs, selections, unary functions, multiple product types, and operator fusion within distributed execution.
6. Our work aims to expand the scope of sum-product optimization to tackle these additional issues present in real-world ML scripts. We present an addition to SystemML called SPOOF—sum-product optimization and operator fusion—that addresses these challenges and accelerates real-world ML scripts by XXX times.
7. SystemML already has an optimization framework that consists of a first phase of deterministic rewriting and a second phase of fusing operators together.

While SystemML does achieve speedup with this simplified framework, it leaves plenty of speedup on the table on account of the following factors:

(a) Most of SystemML's optimizations apply to a small, local sub-expression of operators. Optimizing at the local level may produce locally optimal yet globally suboptimal expressions. In SPOOF, we aim to widen the scope of optimization to entire DAGs at a time, avoiding expressions that are local minima in terms of runtime.

(b) SystemML uses a deterministic list of rules (ordered and guarded by conditions) for most optimizations. This list of rules may miss optimization opportunities and leads to a long-term maintenance burden: every time it is necessary to add a new rule, the rule list must be carefully engineered so as not to produce suboptimal rule interactions.

The fundamental issue is that deterministic rule lists do not establish *search space completeness*. We call a rewriting algorithm complete if for every query (initial expression) and input data, the optimal expression equivalent to the query for that data can be found via the rewriting algorithm.

Because the rule list is not search space complete, it must be updated every time a user presents a query that the rule list does not optimally rewrite. This issue is further compounded by the fact that SystemML uses compound operators such as `trace(A)`, which overlaps with rewrites that concern matrix sub-selection (of the diagonal, in this case) and summations.

SPOOF aims to achieve search space completeness by using a cost-based optimization method on an algera of elementary operators. These elementary operators include sums and products over scalar, and the rewrites we consider over them involve basic algebraic rewrites such as the commutative and distributive laws. We also expect a much lower long-term maintenance buden on account of using such a minimal set of operators and rewrites.

(c) SystemML rewrites expressions separately from fusing expressions together. While the separation between rewrite and fusion simplifies SystemML's architecture, it can lead to suboptimal plans: an expression that is more expensive to compute

*Work done during an internship at IBM Research – Almaden.

Proceedings of the VLDB Endowment, Vol. 10, No. 8

without fusion may be the optimal expression with fusion. SPOOF aims to holistically consider fusion and rewrites at the same time.

8. Use $\text{sum}((X - UV^T)^2)$ as a motivating example.

2. SYSTEM ARCHITECTURE

2.1 SystemML

2.2 SPlan Representation

1. Named attributes — concept of "plates" or "boundaries of for loops over attribtues, iterating over scalars"
2. Generator nodes (Read, DataGen, Ext) — 0 or 1-ary, propagate and possibly add new attribtues
3. Propagator nodes ($+, *$, log, exp, max, min, ...) — n-ary, propagate the union of attribtues of inputs (do not add new attributes)
4. Aggregator nodes (Write, $\Sigma, \Pi, \max, \min$, ...) — 1-ary, delete some attributes

3. SEARCH ALGORITHM

3.1 Normal Form

1. Figure with Σ on top, then $+$, then n $*$s with inputs.

3.2 Plan Enumeration

1. Review query hypergraph. It is a graph in the case of 2-D Linear Algebra.
2. Variable elimination order decision.
   (a) Partition into connected components.
   (b) Push Σ into $+$.
   (c) Multiply within groups first, for all edges with the same attributes.
   (d) Min-degree heuristic; no tensor intermediates.
   (e) Independent aggregated attributes can be factored right away. These are aggregated attributes that are not adjacent to other aggregated attributes.
3. Memoization structure (for determining CSEs) — E-DAG
4. Size of space (add statistics on how large the search space is in real-world ML scripts)
5. Pruning; upper and lower bounds

3.3 Plan Selection

1. The locally-optimal solution
2. Path share and root share opportunities; path shadowing considerations
3. Algorithm "SS-CSE" (Search Shares of Common Sub-Expressions)

4. EXPERIMENTS

5. RELATED WORK

Sum-product optimization frameworks: FAQ [1], AJAR, PANDA.

DB query compilation

PL comprehensions

HPC operator fusion

ML systems operator fusion

6. CONCLUSIONS

We have the best theory and the best system!

We have lots of future work!

7. REFERENCES

[1] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: Questions Asked Frequently. In *PODS*, 2016.