Redundancy Elimination in Distributed Matrix Computation

Zihao Chen, Baokun Han, Chen Xu*, Weining Qian, Aoying Zhou
{zhchen,bkhan}@stu.ecnu.edu.cn,{cxu,wnqian,ayzhou}@dase.ecnu.edu.cn
East China Normal University†, Shanghai 200062, China

ABSTRACT

As matrix computation becomes increasingly prevalent in large-scale data analysis, distributed matrix computation solutions have emerged. These solutions support query interfaces of linear algebra expressions, which often contain redundant subexpressions, i.e., common and loop-constant subexpressions. Hence, existing compilers rewrite queries to eliminate such redundancy. However, due to the large search space, they fail to find all redundant subexpressions, especially for matrix multiplication chains. Furthermore, redundancy elimination may change the original execution order of operators, and have negative impacts. To reduce the large search space and avoid the negative impacts, we propose *automatic elimination* and *adaptive elimination*, respectively. In particular, automatic elimination adopts a block-wise search that exploits the properties of matrix computation for speed-up. Adaptive elimination employs a cost model and a dynamic programming-based method to generate efficient plans for redundancy elimination. Finally, we implement *ReMac* atop SystemDS, eliminating redundancy in distributed matrix computation. In our experiments, ReMac is able to generate efficient execution plans at affordable overhead costs, and outperforms state-of-the-art solutions by an order of magnitude.

CCS CONCEPTS

• **Information systems → MapReduce-based systems**.

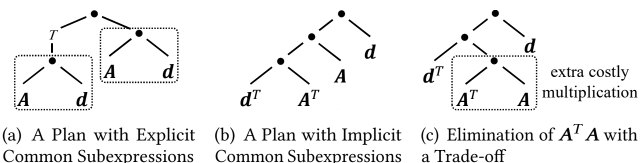
KEYWORDS

distributed system; matrix computation; redundancy elimination

ACM Reference Format:
Zihao Chen, Baokun Han, Chen Xu*, Weining Qian, Aoying Zhou. 2022. Redundancy Elimination in Distributed Matrix Computation. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3517877

1 INTRODUCTION

In the era of Big Data, techniques in the field of data analysis (e.g., machine learning and data science) are rapidly emerging. Among these techniques, matrix computing has become a fundamental and widely used utility. Examples of applications include linear regression, collaborative filtering, and dimensionality reduction. To apply matrix computation on large-scale datasets, numbers of systems and solutions have been proposed, such as SystemDS [7, 8], MLlib [11], ScaLAPACK [2], and SciDB [12].



(a) A Plan with Explicit Common Subexpressions (b) A Plan with Implicit Common Subexpressions (c) Elimination of $A^T A$ with a Trade-off

Figure 1: Different Execution Plans for $d^T A^T Ad$ in DFP

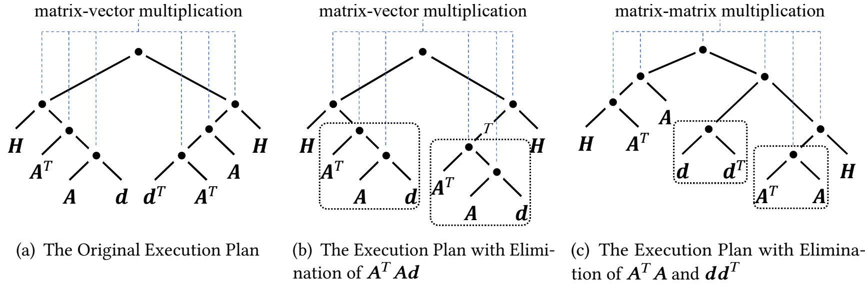
Based on these systems and solutions, a variety of research works have contributed to optimize distributed matrix computation (e.g., operator folding [14], matrix repartition [31], and multiplication acceleration [15, 17]). However, these contributions do not consider the redundancy in execution plans. For example, consider the Davidon-Fletcher-Powell (DFP) formulation. One can use the expression $d^T A^T Ad$ for $\|Ad\|_2^2$, where A represents a dataset, and d is a vector learnt in iterations. A typical execution plan is to perform three matrix multiplications, while an alternative is to reuse the result of Ad and eliminate one redundant matrix multiplication.

There are solutions for distributed matrix computation that provide limited support for redundancy elimination. SystemDS [10], MATFAST [32], and LIMA [24] support *explicit* common subexpression elimination (CSE). As an example, the identical subtrees in the execution plan in Figure 1(a) explicitly indicate the CSE of Ad. MATFAST and LIMA also support *explicit* loop-constant subexpression elimination (LSE), i.e., the elimination of subtrees with loop-constant outputs. Nonetheless, for an algorithm containing common or loop-constant subexpressions, the execution plan may not explicitly indicate CSE or LSE. In this case, CSE and LSE are considered to be *implicit*. For example, the execution plan in Figure 1(b) contains no identical subtree, although $d^T A^T Ad$ has a common subexpression Ad. Since the above solutions are oblivious to implicit CSE and LSE, they fail to make full advantage of redundancy elimination. In order to apply implicit CSE, SPORES [29] extends the equality saturation technique used in compilers. However, it adopts sampling to reduce the huge search space of multiplication chains, and its purpose is not to find and exploit all CSE. Hence, the issue of finding implicit CSE and LSE remains unsolved.

Furthermore, even if a plan optimizer manages to find implicit CSE and LSE, the optimizer needs to determine whether to apply them. For example, in DFP, the matrix A of $d^T A^T Ad$ is loop-constant, and so is $A^T A$. However, if we apply the LSE of $A^T A$, then we must first compute $A^T A$, as depicted in Figure 1(c). Given that A is a distributed matrix, this LSE may lead to expensive operations

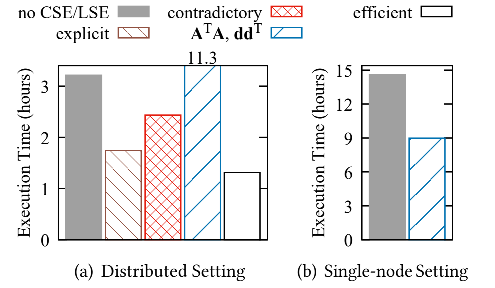
*Chen Xu is the corresponding author
†Shanghai Engineering Research Center of Big Data Management

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3517877



(a) The Original Execution Plan (b) The Execution Plan with Elimination of $\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}$ (c) The Execution Plan with Elimination of $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{d}\boldsymbol{d}^T$

Figure 2: Different Plans for $HA^TAdd^TA^TAH$ in DFP



(a) Distributed Setting (b) Single-node Setting

Figure 3: Performance of SystemDS on DFP

and become detrimental to performance, although it eliminates redundancy. In addition, there are contradictory CSE and LSE. For example, in $\boldsymbol{d}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}$, we cannot apply the elimination on both $\boldsymbol{A}\boldsymbol{d}$ and $\boldsymbol{A}^T\boldsymbol{A}$. In general, there is a trade-off in redundancy elimination.

In this paper, we implement ReMac to explore redundancy elimination in distributed matrix computation. First, we aim to find implicit CSE and LSE, to accomplish *automatic elimination*. Due to the complex properties of matrix computation (e.g., distributivity and associativity), it is impractical to traverse all execution plans. Interestingly, string matching is a simple yet fast approach to search matrix multiplication chains, where the internal order of operators does not matter. Hence, ReMac splits an execution plan into blocks of matrix multiplication chains, and applies a *block-wise* search to find implicit CSE and LSE. In our experiments, ReMac achieves a 41.5x speedup over SystemDS via automatic elimination in the best case, but it can also be 10.0x slower than SystemDS due to misused CSE or LSE. Then, we center on *adaptive elimination*, since the CSE and LSE found by automatic elimination may be detrimental or contradictory. In particular, we treat each CSE and LSE as an option, and the goal of adaptive elimination is to obtain the efficient combination of these options. To this end, we devise a *cost model* to evaluate the overheads of elimination options and, more importantly, a *dynamic programming*-based method to address the combinatorial explosion of options. Our experiments show that adaptive elimination prevents ReMac from abusing CSE or LSE and achieves a 14.4x speedup over SystemDS, ScaLAPACK, and SciDB.

In the rest, we highlight the motivation for automatic and adaptive elimination in Section 2, and make the following contributions.

- We propose a *block-wise* search for *automatic elimination* in Section 3, that exploits the properties of matrix computation to accelerate the search for implicit CSE and LSE.
- We contribute to *adaptive elimination* in Section 4, which adopts a *cost model* to evaluate CSE and LSE options as well as a *dynamic programming*-based method to address the combinatorial explosion of CSE and LSE options.
- We discuss the implementation of ReMac (based on SystemDS) in Section 5, and demonstrate its performance in Section 6.

In addition, we introduce related work in Section 7 and summarize our work in Section 8.

2 MOTIVATION

We elaborate the redundancy in matrix computation, motivating automatic elimination, in Section 2.1. Moreover, to take full advantage of redundancy elimination in varying cases, we need adaptive elimination, the benefits of which are described in Section 2.2.

2.1 Motivation for Automatic Elimination

There are two types of matrix computation redundancy, namely common subexpressions and loop-constant subexpressions.

Common Subexpression Elimination (CSE). CSE refers to the elimination of instances of identical subexpressions. Given an execution plan, its identical subtrees explicitly indicate CSE, termed as explicit CSE. Matrix computation solutions (e.g., SystemDS and TensorFlow [4]) apply explicit CSE to the execution plans they generate. For example, solving the least squares problem $\min_{\boldsymbol{x}\in\mathbb{R}^n} \|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}\|_2$ using DFP involves the following equations:

while *loop_condition* :

$$\boldsymbol{g} = \cdots$$

$$\boldsymbol{d} = \boldsymbol{H}\boldsymbol{g} \tag{1}$$

$$\boldsymbol{H} = \boldsymbol{H} - \frac{\boldsymbol{H}\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}\boldsymbol{d}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{H}}{\boldsymbol{d}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{H}\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}} + \frac{\boldsymbol{d}\boldsymbol{d}^T}{2\boldsymbol{d}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}} \tag{2}$$

Here, $\boldsymbol{H}$ is an inverse Hessian approximation (symmetric matrix), $\boldsymbol{g}$ is a gradient vector, and $\boldsymbol{A}$ represents an input dataset. For Equations 1 and 2, SystemDS is able to apply the explicit CSE of $\boldsymbol{H}\boldsymbol{g}$.

However, given that an expression typically has multiple equivalent execution plans, some CSE is explicit only in certain execution plans. In particular, we consider the CSE, not explicit in the execution plan, as implicit. For example, as shown in Figure 2(a), since the execution plan contains no identical subtree, the CSE for $\boldsymbol{d}^T\boldsymbol{A}^T\boldsymbol{A} = (\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d})^T$ is implicit. In order to find and apply this implicit CSE, we need to transform the execution plan into an equivalent plan that computes $\boldsymbol{d}^T\boldsymbol{A}^T\boldsymbol{A}$ in $(\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d})^T$, as depicted in Figure 2(b). As a result, to take full advantage of redundancy elimination, we need to search equivalent execution plans for implicit CSE.

Loop-constant Subexpression Elimination (LSE). LSE refers to elimination of subexpressions having constant results in a loop. For example, in Equation 2, $\boldsymbol{A}^T\boldsymbol{A}$ is loop-constant, since the program in the loop does not update $\boldsymbol{A}$. Therefore, the LSE of $\boldsymbol{A}^T\boldsymbol{A}$ is to compute

$\boldsymbol{A}^T\boldsymbol{A}$ before the loop, and reuse the result in the loop. Similar to CSE, we classify LSE as explicit and implicit. We consider LSE as explicit, if there are subtrees in the execution plan explicitly indicating the loop-constant subexpression. Otherwise, we consider the LSE to be implicit. As depicted in Figure 2(a), the LSE of $\boldsymbol{A}^T\boldsymbol{A}$ is implicit, since the execution plan does not contain the multiplication of $\boldsymbol{A}^T$ and $\boldsymbol{A}$. Alternatively, we can find this LSE in the equivalent plan as shown in Figure 2(c). Hence, we need to transform the execution plan into equivalent plans, to find all redundant subexpressions.

Despite the complexity of searching for implicit CSE and LSE, there are two factors motivating our research on how to automatically apply CSE and LSE. First, both explicit and implicit elimination are critical to performance. As shown in Figure 3(a), for DFP, explicit elimination achieves a 1.8x speedup, while explicit and implicit elimination together achieve a 2.4x speedup in the best case. Second, it is non-trivial for users to handle all elimination. For illustration, to eliminate the loop constant subexpression $\boldsymbol{A}^T\boldsymbol{A}$ in DFP, a user adds a new line $\boldsymbol{T} = \boldsymbol{A}^T\boldsymbol{A}$ outside the loop and substitute $\boldsymbol{T}$ for $\boldsymbol{A}^T\boldsymbol{A}$ in Equation 2. However, the entire DFP algorithm has 1391 different CSE and LSE options, each of which eliminates a redundant subexpression. Hence, we propose automatic elimination, to both accelerate the execution and ease users' programming burden.

2.2 Motivation for Adaptive Elimination

Intuitively, we should apply all found elimination options to exploit the benefits of redundancy elimination. However, this is impractical, based on the following two observations. First, elimination options may *contradict* each other, which means we cannot apply them in one execution plan. Second, elimination options may change the original execution order and become *detrimental* to performance, in which case we should abandon those elimination options.

Contradictory Elimination Options. Multiple elimination options may adhere to different execution order, so that they do not fit in one execution plan, i.e., the options are contradictory. For example, in Equation 2, there are two contradictory elimination options of $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{A}\boldsymbol{d}$, since we cannot multiply $\boldsymbol{A}$ with both $\boldsymbol{A}^T$ and $\boldsymbol{d}$ when calculating $\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}$. Moreover, the contradiction has cascading effects on choosing other options. Specifically, the LSE option of $\boldsymbol{A}^T\boldsymbol{A}$ can be combined with the CSE option of $\boldsymbol{d}\boldsymbol{d}^T$, while the CSE option of $\boldsymbol{A}\boldsymbol{d}$ can be combined with the CSE option of $\boldsymbol{H}\boldsymbol{A}^T$. Hence, it is non-trivial to know which combination achieves the optimal performance. As depicted in Figure 3(a), the contradictions may lead to a suboptimal combination of elimination options, that performs worse than applying only explicit elimination options.

Detrimental Elimination Options. An elimination option is detrimental to performance, if it changes the original execution order, offsetting the benefits of the elimination. For illustration, as depicted in Figure 2(a), the original execution plan for $\boldsymbol{H}\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}\boldsymbol{d}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{H}$ contains six matrix-vector multiplications and one vector-vector multiplication. An alternative execution plan in Figure 2(c) is to eliminate $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{d}\boldsymbol{d}^T$. However, the elimination changes the original execution order, leading to six matrix-matrix multiplications. Furthermore, the communication overhead in distributed environments intensifies the side effect of CSE and LSE. For example, the implementations of multiplications (e.g., BMM for matrix-vector multiplications, and CPMM for matrix-matrix multiplications [8]) are extremely different in terms of communication overhead. As shown in Figure 3(b), in a single-node environment with sufficient memory, the elimination of $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{d}\boldsymbol{d}^T$ improves performance. Nonetheless, in a distributed environment, the same elimination switches the original BMM to CPMM for matrix-matrix multiplications and reduces performance by 400%, as depicted in Figure 3(a).

Due to contradictory elimination options, we have to pick elimination options to apply. Typically, we can conservatively pick the elimination options adhering the original execution order, or disregard the original execution order to make a more aggressive pick. However, neither of these strategies adapts to all cases. The conservative strategy may lose potential performance improvement attributed to unpicked options, while the aggressive strategy may suffer from detrimental options. It motivates us to explore an adaptive strategy, picking elimination options in a right way.

3 AUTOMATIC ELIMINATION

As described in Section 2.1, it is essential that a system automatically searches for CSE and LSE to improve performance. Next, we will elaborate the challenges to search for CSE and LSE in Section 3.1. Sections 3.2 and 3.3 will explain how ReMac overcomes these challenges to find CSE and LSE, respectively.

3.1 Challenges to Search for CSE and LSE

There are two challenges to search for CSE and LSE in practical.

Duplicated Search. A basic method to find redundant subexpressions is *tree-wise* search, which traverses all possible plan trees and detects whether there are common or loop-constant operators. Nonetheless, this traversal process involves duplicated search. For example, in Equation 2 with $\boldsymbol{d}$ substituted, the numerator of $\frac{\boldsymbol{H}\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}\boldsymbol{d}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{H}}{\boldsymbol{d}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{H}\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}}$ has two million possible plans. Tree-wise search simply regards the plan trees with different subtrees for the numerator as different, even though the trees share the same subtree for the denominator. Hence, tree-wise search would suffer from revisiting the same subtree for the denominator millions of times.

Large Search Space. Tree-wise search is strict with the internal execution order of subexpressions, enlarging the search space. Typically, a subexpression has equivalent plan subtrees representing different execution order (e.g., $(\boldsymbol{A}^T\boldsymbol{A})\boldsymbol{d}$ and $\boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{d})$ for the subexpression $\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}$). This complicates the matching of common subexpressions. For example, $(\boldsymbol{A}^T\boldsymbol{A})\boldsymbol{d}$ and $\boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{d})$ do not match because they have no common operators. As a result, tree-wise search has to traverse the plan subtrees for $(\boldsymbol{A}^T\boldsymbol{A})\boldsymbol{d}$ and $\boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{d})$ respectively, to identify the CSE of $\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}$. To mitigate this, SPORES [29] employs equality saturation. However, when handling a long multiplication chain, it still requires the sampling technique that sacrifices potential redundancy elimination for a smaller search space.

In general, the tree-wise search for CSE and LSE is impractical. Hence, we propose a block-wise search to avoid duplicate search and reduce the large search space, in finding all CSE and LSE.

3.2 Searching for CSE

We have the following rationales when designing a novel block-wise search method for CSE.

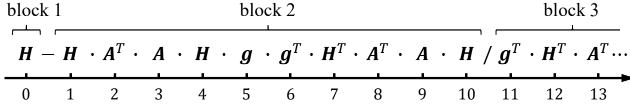


Figure 4: The Coordinate and Split Blocks for DFP

Rationale 1: We observe that transpositions significantly complicate the execution plans for multiplication chains. For example, $\boldsymbol{HA^TAHgg^TH^TA^TAH}$ has over two million different execution plans, whereas, without transposition, a multiplication chain of 10 matrices has only 4862 different plans, i.e., the 10th Catalan number. In order to reduce the search space incurred by transpositions, we push down transpositions, and deal with CSE related to transpositions exclusively.

Rationale 2: As illustrated in Section 3.1, due to duplicated search, we do not search a complex expression in a tree-wise manner. Instead, we split an expression into blocks, and search for CSE by blocks. Note that the divide-and-rule approach may exclude the CSE options across blocks. However, we can additionally find and include such options with a negligible overhead cost.

Rationale 3: To reduce the large search space, we disregard the internal execution order of matrix multiplication chains, according to the non-commutative and associative properties of matrix multiplication. For illustration, when we try to identify the common subexpression $\boldsymbol{A^TAd}$, it does not matter whether the subexpression is computed in the order of $(\boldsymbol{A^TA})\boldsymbol{d}$ or $\boldsymbol{A^T}(\boldsymbol{Ad})$.

Based on the rationales above, we propose our novel block-wise search for CSE, consisting of three steps as follows.

❶ **Push Down Transpositions.** This step reduces the search space via transposition push-down, as explained in Rationale 1. In specific, we bring forward the execution of transposition operators (e.g., from $(\boldsymbol{Ad})^T$ into $\boldsymbol{d^TA^T}$), following the property of matrix transpose. However, while the transposition push-down reduce large search space, it also prevents us from finding certain implicit common subexpressions (e.g., $\boldsymbol{Ad}$ in $\boldsymbol{Add^TA^T}$). To mitigate the side effect, we modify the identification for the common subexpressions hidden from transpositions, which will be explained in Step ❸.

❷ **Build Coordinates and Blocks.** As introduced in Rationale 2, in this step, we split expressions into blocks. As a preparation, we use the distributive law to expand a plan tree. Also, for convenience, we build coordinates and take the matrices as the scales on the coordinate axis (e.g., the coordinate built for DFP in Figure 4).

Subsequently, we split the coordinate into multiple blocks. According to Rationale 3, we try to make each of those blocks correspond to a matrix multiplication chain. In particular, we center on matrix multiplication, an operator with a higher cost [17] and a higher priority than the other major operators (e.g., element-wise addition). We split expressions at the operators which have a lower priority than matrix multiplication, and thereby acquire blocks of multiplication chains. For example, we split Equation 2 into five blocks with $\boldsymbol{d}$ substituted, $\boldsymbol{H}$, $\boldsymbol{HA^TAdd^TA^TAH}$, $\boldsymbol{d^TA^TAHA^TAd}$, $\boldsymbol{dd^T}$, and $2\boldsymbol{d^TA^TAd}$, as depicted in Figure 4.

❸ **Traverse Blocks upon the Coordinate.** According to Rationale 3, for the blocks of matrix multiplication chains, we employ sliding windows to detect CSE. Here, we try all window sizes (from one to the length of a block). Each time sliding a window, we capture the subexpression of the window via the associative law. Then, as shown in Figure 5, we record an entry in a hash table, where the key represents the subexpression and the value contains the location information, i.e., coordinates, of the window. After traversing all blocks, we will find CSE based on the conflicts in the hash table. Via sliding windows, we do not need to concern the internal execution order of the visited subexpression. Hence, the search space is significantly smaller than that of the tree-wise search.

In the hash table, the key of an entry is the subexpression string. However, as mentioned in Step ❶, this will cause $\boldsymbol{AH}$ and $\boldsymbol{HA^T}$ to have different keys, despite the fact that $(\boldsymbol{AH})^T = \boldsymbol{HA^T}$ (note that $\boldsymbol{H}$ is symmetric). Here, we compare the strings of the subexpression and its transposition on each character in alphabetical order, and then take the string with a smaller comparison result as the key. In this way, $\boldsymbol{AH}$ and $\boldsymbol{HA^T}$ would share the same key, $\boldsymbol{AH}$.

Discussion. Steps ❷ and ❸ focus on finding CSE inside blocks, leaving the CSE across blocks uncovered. In particular, since we expand expressions to split them into blocks, there may be implicit CSE across blocks hidden by the expansion. For example, $\boldsymbol{P \cdot XY + P \cdot YZ + XY \cdot Q + YZ \cdot Q}$ has a common subexpression $\boldsymbol{XY + YZ}$ across four blocks. However, we can discover such CSE via an extension of the aforementioned steps. Our intuition is to first reveal the subexpressions crossing blocks via reverting expansion, and then detect whether those subexpressions are common based on the CSE found inside blocks. Accordingly, in Step ❷, we extract common factors, including identity matrices, to revert expansion and group new blocks (e.g., $\boldsymbol{I} \cdot (\boldsymbol{PXY + XYQ})$, $\boldsymbol{P} \cdot (\boldsymbol{XY + YZ})$, $(\boldsymbol{XY + YZ}) \cdot \boldsymbol{Q}$, etc.). In Step ❸, we would search for common grouped parts (e.g., $\boldsymbol{XY + YZ}$) in newly grouped blocks. Since we have already known $\boldsymbol{XY}$ and $\boldsymbol{YZ}$ are common subexpressions, it is easy to find two addition operators taking $\boldsymbol{XY}$ and $\boldsymbol{YZ}$ as inputs, i.e., $\boldsymbol{XY + YZ}$ is common. Hence, this extension incurs a negligible overhead cost.

3.3 Searching for LSE

The search for explicit LSE is typically simple, since we only need to determine whether the inputs of an operator is loop-constant in an execution plan. However, the challenges to find implicit LSE are similar to those of implicit CSE. Hence, we divide the search into two steps to find explicit and implicit LSE, and embed the steps into the block-wise search for CSE.

❶* **Label Explicit LSE.** In Step ❶, we also parse the loop body to find explicit loop-constant symbols and label them to prepare for finding implicit LSE.

❸* **Find implicit LSE.** Since the search for CSE already addresses duplicated search and large search space, we search for implicit LSE along with the search for CSE in Step ❸. In specific, if all matrices in the sliding window are labeled loop-constant, we record the subexpression corresponding to the window as an LSE option.

Discussion. Due to the expansion in Step ❷, there may be implicit LSE across blocks as well. Similar to the aforementioned extension of the search for CSE, we group blocks in Step ❷ and detect whether the grouped part is loop-constant in Step ❸.

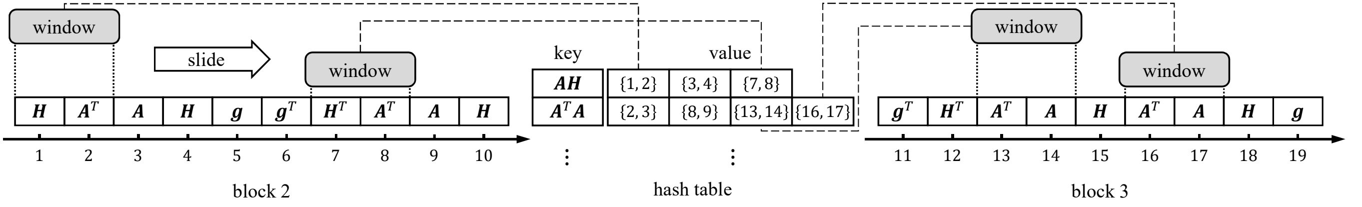


Figure 5: Traverse the Split Blocks of DFP

Via the block-wise search, we can find both explicit and implicit elimination options. Note that, since all transformations to expressions in the search steps follow the algebraic equivalence, the found options would not affect the expression results.

4 ADAPTIVE ELIMINATION

As shown in Section 2.2, contradictory and detrimental elimination options motivate us to explore an adaptive strategy to find the efficient *combination* of CSE and LSE *options*. Due to the massive number of options and the overhead cost of evaluation, we cannot enumerate and evaluate each combination, which will be discussed in Section 4.1. Hence, we propose adaptive elimination, which involves a cost model for evaluation in Section 4.2, and a dynamic programming-based method for combinations in Section 4.3.

4.1 Curses in Combining Elimination Options

An intuitive method of finding the efficient combination is to enumerate and compare combinations via cost evaluation. However, this may lead to an extreme overhead cost.

Efficiency vs. Accuracy in Cost Evaluation. The prerequisite of finding the efficient combination is an accurate cost model. Since the matrix sparsity is critical to the cost of an execution plan [20], the prerequisite, in turn, becomes an accurate sparsity estimator. That is, an inaccurate sparsity estimator may result in a wrong cost evaluation and thereby a suboptimal execution plan. However, on the other hand, an accurate estimator inevitably causes inefficient cost evaluation, extremely prolonging compilation time. Hence, we have to carefully choose a sparsity estimator, which will be discussed in Section 4.2.

Combinatorial Explosion in Enumeration. Due to the large number of elimination options and the combinatorial explosion, enumerating and evaluating elimination combinations would lead to unaffordable overhead costs. For example, the DFP algorithm has 1391 options of CSE and LSE, ending up with millions of possible combinations to be enumerated and evaluated. Hence, we have to avoid brute-force enumeration when finding the efficient combination of CSE and LSE, which will be discussed in Section 4.3.

4.2 The Cost Model

ReMac employs a cost model to evaluate operators, which involves choosing the troublesome sparsity estimator. For an operator O in a plan tree, the cost model regards its cost, c_O, as two parts: the computation cost, $compute_O$, and the transmission cost, $transmit_O$.

$$c_O = compute_O + transmit_O. \tag{3}$$

Here, $transmit_O$ considers the impacts of matrix partition schemes.

Computation Cost. $compute_O$ indicates the computation cost of O, which is related to the number of floating point operations (FLOP) [14, 22]. In specific, ReMac computes $compute_O$ as follows:

$$compute_O = w_{flop}\, FLOP_O, \tag{4}$$

where w_{flop} is a constant representing the reciprocal of the peak floating point performance of a cluster, and $FLOP_O$ represents the number of FLOP attributed to O. Note that the cost model assumes only one algorithm is running, not considering the resource competition in multi-tenant environments. However, the goal of the model is not to calculate exactly the real execution time, but to compare the magnitude of the costs of various plans for adaptive elimination. Hence, despite the complexity of real-world environments, our simplified model is valid in generating an efficient plan.

To illustrate the computation of $FLOP_O$, we take a certain operator, matrix multiplication of $\boldsymbol{U}$ and $\boldsymbol{V}$, as an example. $\boldsymbol{U}$ is a $R_U \times C_U$ matrix with sparsity of S_U, and $\boldsymbol{V}$ is a $C_U \times C_V$ matrix with sparsity of S_V. Then, ReMac computes $FLOP_O$ by $3\,(R_U\, C_U\, C_V\, S_U\, S_V)$, where multiplication and addition accounts for $2\,(R_U\, C_U\, C_V\, S_U\, S_V)$ and $R_U\, C_U\, C_V\, S_U\, S_V$, respectively. Clearly, the sparsity directly decides the value of $FLOP_O$.

Transmission Cost. $transmit_O$ indicates the costs attributed to four transmission primitives: *collection* refers to the collection of data from a cluster, *broadcast* refers to the broadcast of data to a cluster, *shuffle* refers to the exchange of data among nodes in a cluster, and *dfs* refers to the data transmission incurred by interacting with a distributed file system. In particular, *broadcast* and *shuffle* cover the transmission of partitioning matrices. ReMac computes $transmit_O$ by accumulating these costs, i.e.,

$$transmit_O = \sum_{pr \in PR} w_{pr}\, D_{pr}, \tag{5}$$

where PR is the set of transmission primitives, w_{pr} is the reciprocal of the transmission speed of pr, and D_{pr} is the data volume in pr.

For example, O is a broadcast-based matrix multiplication (BMM) [8, 17] of $\boldsymbol{U}$ and $\boldsymbol{V}$, where $\boldsymbol{U}$ is distributed across a cluster and $\boldsymbol{V}$ is in local memory. In this case, the transmission of O involves broadcasting $\boldsymbol{V}$ to join $\boldsymbol{U}$ and $\boldsymbol{V}$, and aggregating the products of matrix blocks by rows with a shuffle. Here, the broadcast partitions $\boldsymbol{V}$, and the shuffle partitions the product of $\boldsymbol{U} \cdot \boldsymbol{V}$.

First, for the broadcast process, $D_{broadcast} = size(\boldsymbol{V})$. Here, the value of $size(\boldsymbol{V})$ depends on the storage format of $\boldsymbol{V}$, which relies on the sparsity S_V. In specific, following SystemDS, we use a dense format if $S_V > 0.4$. Moreover, when using a sparse format, $size(\boldsymbol{V})$ is linearly related to S_V. For example, when $0.0004 < S_V \leqslant 0.4$, we use compressed sparse rows to store $\boldsymbol{V}$. Then, $size(\boldsymbol{V}) = \alpha S_V + \beta$, where αS_V is the size of the arrays to store the non-zeros and

column indexes, and β is the size of the row pointers and other meta data fields.

Second, BMM shuffles the products between the blocks of $\boldsymbol{U}$ and $\boldsymbol{V}$, the average size of which is $size$(one block of $\boldsymbol{U}\cdot$one block of $\boldsymbol{V}$). If $\boldsymbol{U}$ is split into $B_{\boldsymbol{U}}$ blocks, then the size of shuffled data is $B_{\boldsymbol{U}}$ times of $size$(one block of $\boldsymbol{U}\cdot$one block of $\boldsymbol{V}$). However, in each partition, ReMac aggregates the blocks of the same rows before the shuffle, so as to reduce the shuffled data. Hence, if in one partition, there are $P_{\boldsymbol{U}}$ blocks having the same rows, then we have

$$D_{shuffle} = size(\text{one block of } \boldsymbol{U} \cdot \text{one block of } \boldsymbol{V}) \times B_{\boldsymbol{U}}/P_{\boldsymbol{U}}. \quad (6)$$

It is worth noting that ReMac inherits the hash partition scheme of matrices exploited in SystemDS. However, the cost model is not coupled with a fixed partition scheme. That is, for a different partition scheme of $\boldsymbol{U}$, the terms in Equation 6 will change. For example, if $\boldsymbol{U}$ is split into larger blocks, then we will have a bigger $size$(one block of $\boldsymbol{U}\cdot$one block of $\boldsymbol{V}$) as well as smaller $B_{\boldsymbol{U}}$ and $P_{\boldsymbol{U}}$. Similarly, if the partition scheme of $\boldsymbol{U}$ changes, then $P_{\boldsymbol{U}}$ should be changed accordingly.

Sparsity Estimator. As aforementioned, the matrix sparsity directly decides $FLOP_O$ in $compute_O$ and D_{pr} in $transmit_O$. Hence, the accuracy of the sparsity estimator plays a key role in the cost model. Nonetheless, we also require the sparsity estimator not to incur an overwhelming overhead cost. We investigated existing estimators, from the efficient ones (e.g., metadata-based [10] and sampling-based [32]) to the accurate ones (e.g., MNC [27] and density map [19]). Among them, we chose two typical estimators. The first estimator is metadata-based, which assumes uniformly distributed non-zeros and derives sparsity solely from the sparsity of input matrices [10]. In general, it sacrifices accuracy for estimation efficiency. According to our experimental results in Section 6.3.2, the metadata-based cost evaluation incurs a negligible overhead cost, but may mislead ReMac to a suboptimal combination of elimination options. On the other hand, the state-of-the-art sparsity estimator, MNC, exploits structural properties of matrices for accurate sparsity estimates. According to experiments, MNC[1] helps ReMac to apply the efficient combination of elimination options. Nonetheless, MNC performs additional operations to collect necessary statistics, which may be costly.

4.3 Dynamic Programming for Combining Elimination Options

Due to the curses in combining elimination options, it is impractical to enumerate and evaluate each combination to find the efficient one. Our basic idea is to evaluate each elimination option solely and form the efficient combination based on those evaluation results. As shown in Algorithm 1, we propose a dynamic programming-based method, consisting of two phases, namely *building* and *probing*.

The building phase uses the cost model to evaluate the plan tree of each elimination option (Line 4), and collates the results into a *cost graph* (Line 5). Here, we build a directed acyclic graph, namely cost graph, that collects sufficient information for the probing phase to form the efficient elimination combination. The details of building a cost graph will be explained in Subsection 4.3.1.

[1]We choose the MNC version using $\mathbf{E}_{\text{dm}}$ over $\mathbf{h}^r_{\mathbf{A}}$ and $\mathbf{h}^c_{\mathbf{B}}$ for accurate estimates [27].

Algorithm 1 The Process of Adaptive Elimination

Input: A set of CSE and LSE options, *OptionSet*
Output: A cost graph indicates the efficient combination of options

```
1:  CostGraph ← an empty graph
2:  // Building Phase
3:  for option in OptionSet do
4:      EvalResults ← costModel(option.plan)
5:      CostGraph ← collate(CostGraph, EvalResults)
6:  // Probing Phase
7:  PROBE(CostGraph.topOperator)
8:  return CostGraph
9:  // construct the efficient plan tree where operator is the top operator
10: function PROBE(operator)
11:     // remove costs of operator
12:     operator.costs ← pick(operator.costs)
13:     // remove downstream operators of operator
14:     for input in operator.inputs do
15:         for downstreamOp in input.downstreamOps do
16:             PROBE(downstreamOp)
17:         input.downstreamOps ← pick(input.downstreamOps)
18:     return operator
```

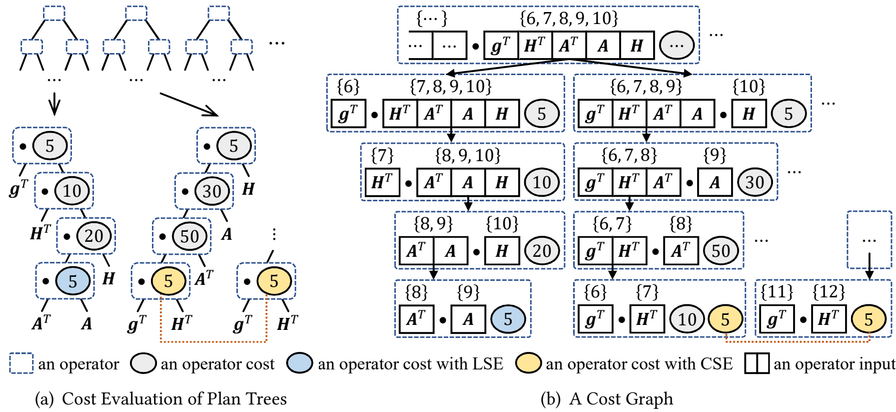
Based on the cost graph, the probing phase constructs the plan tree with the efficient elimination combination through a dynamic programming process. In particular, ReMac recursively removes operator costs (Line 12) and operators (Lines 14 - 17) from a cost graph, and eventually achieves a pruned cost graph indicating a plan tree. We will elaborate the process in Subsection 4.3.2.

4.3.1 Building Phase. In the building phase, we generate the efficient plan trees for each CSE or LSE option, where the operators are candidates for our final plan tree. Particularly, there may be multiple plan trees equivalent in performance for one option. Then, we evaluate operators of plan trees via the cost model illustrated in Section 4.2 and collate the operators along with their costs into a cost graph. In the following, we will explain the structure of a cost graph and its building process, respectively.

The Structure of a Cost Graph. We follow the structure of a plan tree to collate the evaluation results, as depicted in Figure 6(a). After collating multiple plan trees, the tree that records evaluation results becomes a cost graph, because the same operator may have different upstream operators in different plans.

As depicted in Figure 6(b), a cost graph consists of multiple dashed rectangles, each of which represent an operator. In a dashed rectangle, the solid rectangles indicate the inputs of the operator, and the ellipse represents the cost of the operator. Note that an operator may have multiple values of cost, if there are CSE or LSE options reusing the output of the operator. The blue and yellow ellipses represent the cost reduced by LSE and CSE, respectively.

Table 1 lists the notations of operators and operator costs. Here, we reuse the coordinates introduced in Section 3 to indicate the inputs of operators. For example, the multiplication of $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{H}$ has two inputs, $I_l = \{8, 9\}$ and $I_r = \{10\}$. Hence, we use $O(\{8,9\},\{10\})$ to represent the operator of this multiplication.



(a) Cost Evaluation of Plan Trees (b) A Cost Graph

Figure 6: An Example of the Dynamic Programming-based Method Processing DFP

Table 1: Notations of Dynamic Programming-based method

Symbol	Meaning
$O(I_l, I_r)$	An operator O with inputs, I_l and I_r
$c_{O(I_l,I_r)}$	A cost of $O(I_l, I_r)$
$\mathbb{C}_{O(I_l,I_r)}$	An accumulated cost of $O(I_l, I_r)$
$\mathbb{C}_{I_i}$	An accumulated cost of I_i
$c'_{O(I_l,I_r)}$	A candidate cost of $O(I_l, I_r)$
$\mathbb{C}'_{O(I_l,I_r)}$	An accumulated cost of $O(I_l, I_r)$ containing candidate costs
$\mathbb{C}'_{I_i}$	An accumulated cost of I_i containing candidate costs

Building with LSE. We start with the building phase that involves an LSE option (e.g., of $\boldsymbol{A}^T\boldsymbol{A}$). First, we evaluate the plan tree of the LSE option, as depicted in the left tree in Figure 6(a). Then, we visit the operator of the plan (e.g., the multiplication of $\boldsymbol{g}^T$ and $\boldsymbol{H}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{H}$ with a cost of 5). Accordingly, to build a cost graph, we create $O(\{6\}, \{7, 8, 9, 10\})$ with two inputs, $\boldsymbol{g}^T$ and $\boldsymbol{H}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{H}$, and the cost of 5, as shown in Figure 6(b). Recursively, we visit the downstream operator, the multiplication of $\boldsymbol{H}^T$ and $\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{H}$. Similar to $O(\{6\}, \{7, 8, 9, 10\})$, we create $O(\{7\}, \{8, 9, 10\})$ underneath the input, {7,8,9,10}.

In particular, the LSE option reuses the output of $O(\{8\}, \{9\})$ during the iterative process. Hence, we divide $c_{O(\{8\},\{9\})}$ by the number of iterations. If the cost of $O(\{8\}, \{9\})$ without LSE is 500 and the algorithm requires 100 iterations, then $c_{O(\{6\},\{7\})} = 500/100 = 5$.

Building with CSE. The right tree in Figure 6(a) illustrates the evaluation result of the plan tree that applies a CSE option of $\boldsymbol{g}^T \cdot \boldsymbol{H}^T$. Since the CSE option reuses the output of $O(\{6\}, \{7\})$ in $O(\{11\}, \{12\})$, we apportion the original cost of $\boldsymbol{g}^T \cdot \boldsymbol{H}^T$ (which is 10) to $c_{O(\{6\},\{7\})}$ and $c_{O(\{11\},\{12\})}$ (which becomes $10/2 = 5$). As depicted in Figure 6(b), we append the reduced cost of 5 after the original cost of 10, in $O(\{6\}, \{7\})$.

4.3.2 Probing Phase. In the probing phase, we remove operators and operator costs from a cost graph, to convert the graph into a tree. In this converted tree, each operator input has only one downstream operator, determining how to compute the operator input, and each operator has only one cost, determining whether to apply CSE or LSE. Furthermore, in order to obtain a tree having the lowest overall cost, we devise a dynamic programming process. In the following, we start with a simple case involving LSE only, and then extend to the cases involving both CSE and LSE.

Probing with LSE. In each step, the dynamic programming process visits an operator O with inputs I_l and I_r. Particularly, the process minimizes the *accumulated cost* of $O(I_l, I_r)$, based on $c_{O(I_l,I_r)}$ and the minimized *accumulated cost* of the downstream operators.

Definition 1. *The* ***accumulated cost of an operator*** $O(I_l, I_r)$*, consists of its own cost and the accumulated costs of* I_l *and* I_r*, i.e.,* $\mathbb{C}_{O(I_l,I_r)} = c_{O(I_l,I_r)} + \mathbb{C}_{I_l} + \mathbb{C}_{I_r}$.

Definition 2. *The* ***accumulated cost of an operator input*** I_i *is the accumulated cost of an operator* O *underneath* I_i*, i.e.,* $\mathbb{C}_{I_i} = \mathbb{C}_O$.

According to Definition 2, computing the accumulated cost of an operator input involves picking one of the downstream operators. Combined with Definition 1, this computation is a recursive process, which would eventually pick a plan subtree from the graph, and add up all operator costs of the subtree as the accumulated cost. Clearly, for the *top* operator with no upstream operator, its accumulated cost represents the overall cost of the entire plan tree. Hence, generating the efficient plan with redundancy elimination is equivalent to computing and minimizing the accumulated cost of the top operator. In specific, when minimizing $\mathbb{C}_{O(I_l,I_r)}$, we have to minimize $\mathbb{C}_{I_l}$ and $\mathbb{C}_{I_r}$, related to the accumulated costs of downstream operators.

In the case involving LSE only, we minimize $\mathbb{C}_{O(I_l,I_r)}$ as follows.

$$\begin{cases} min\, \mathbb{C}_{O(I_l,I_r)} = min\, c_{O(I_l,I_r)} + min\, \mathbb{C}_{I_l} + min\, \mathbb{C}_{I_r}, & (7) \\ min\, \mathbb{C}_{I_i} = \min(\{min\, \mathbb{C}_O \mid O \text{ is underneath } I_i\}). & (8) \end{cases}$$

In Equation 7, $min\, c_{O(I_l,I_r)}$ means picking the minimal cost for $O(I_l, I_r)$ (Line 12 in Algorithm 1), and $min\, \mathbb{C}_{I_i}$ means picking the downstream operator with the minimal accumulated cost (Line 17 in Algorithm 1). Recursively, Equation 8 uses Equation 7 to calculate $min\, \mathbb{C}_O$ for a downstream operator O (Line 16 in Algorithm 1).

Equations 7 and 8 compose the dynamic programming process to minimize the accumulated cost of an operator. Eventually, after we minimize the accumulated cost of the top operator, the cost graph will indicate the efficient plan tree with redundancy elimination.

Example 4.1. In Figure 6(b), there are $O(\{6\}, \{7, 8, 9, 10\})$ and $O(\{6, 7, 8, 9\}, \{10\})$ underneath the operator input $\{6, 7, 8, 9, 10\}$. In order to minimize $\mathbb{C}_{\{6,7,8,9,10\}}$, we have to minimize $\mathbb{C}_{O(\{6\},\{7,8,9,10\})}$ and $\mathbb{C}_{O(\{6,7,8,9\},\{10\})}$, respectively. Suppose there is no CSE of $\boldsymbol{g}^T\boldsymbol{H}^T$, which leads to $min\, \mathbb{C}_{O(\{6\},\{7\})} = 10$. In addition, we suppose $min\, \mathbb{C}_{\{8\}} = 0$, since there is no $O(\{8\})$. Then, according to Equations 7 and 8,

$$\begin{aligned} min\, \mathbb{C}_{O(\{6,7\},\{8\})} &= min\, c_{O(\{6,7\},\{8\})} + min\, \mathbb{C}_{\{6,7\}} + min\, \mathbb{C}_{\{8\}} \\ &= min\, c_{O(\{6,7\},\{8\})} + min\, \mathbb{C}_{O(\{6\},\{7\})} + min\, \mathbb{C}_{\{8\}} \\ &= 50 + 10 + 0 = 60. \end{aligned}$$

Eventually, through the recursion of Equations 7 and 8, we have $min\,\mathbb{C}_{O(\{6,7,8,9\},\{10\})} = 95$ for $O(\{6,7,8,9\},\{10\})$. Similarly, we also have $min\,\mathbb{C}_{O(\{6\},\{7,8,9,10\})} = 40$ for $O(\{6\},\{7,8,9,10\})$. Since $min\,\mathbb{C}_{O(\{6\},\{7,8,9,10\})}$ is smaller, we pick $O(\{6\},\{7,8,9,10\})$ for the operator input $\{6,7,8,9,10\}$. That is, $min\,\mathbb{C}_{\{6,7,8,9,10\}} = 40$.

Probing with CSE and LSE. Next, we extend the probing phase to the cases involving CSE and LSE. For simplicity, we term an operator cost reduced by CSE as a *CSE cost*. The most difference here is that a group of CSE costs are relevant (e.g., the CSE costs of $O(\{6\},\{7\})$ and $O(\{11\},\{12\})$). That is, due to CSE, an operator cost is apportioned among those CSE costs. Hence, we pick either the whole group of relevant CSE costs or none of them. However, the aforementioned dynamic programming process does not handle such cases. For example, when minimizing $\mathbb{C}_{O(\{6,7,8,9\},\{10\})}$, we would have picked $min\,\mathbb{C}_{O(\{6\},\{7\})} = 5$. Nonetheless, at that time, it is unknown whether we would also pick $min\,\mathbb{C}_{O(\{11\},\{12\})} = 5$ to minimize the accumulated costs in the upstream of $O(\{11\},\{12\})$. Hence, we have to pass the accumulated costs containing CSE costs (e.g., $\mathbb{C}_{O(\{6\},\{7\})} = \mathbb{C}_{O(\{11\},\{12\})} = 5$) to upstream as candidates, until we can determine whether to pick those CSE costs. Accordingly, we define *candidate* costs as follows.

Definition 3. *For any operator $O(I_l, I_r)$, if there is a CSE cost $c_{O(I_l,I_r)}$, then the cost is a* ***candidate*** *for $O(I_l, I_r)$ and its upstream operators, denoted as $c'_{O(I_l,I_r)}$.*

To probe with CSE and LSE, we extend the dynamic programming process with candidate costs. The main idea is to maintain a candidate set of accumulated costs and gradually reduce this set, so as to finally minimize the accumulated cost of the top operator, during the dynamic programming process. In specific, when visiting an operator $O(I_l, I_r)$, we first minimize $\mathbb{C}_{O(I_l,I_r)}$ with no candidate cost, following Equations 7 and 8. Then, we calculate a candidate set $\{\mathbb{C}'_{O(I_l,I_r)}\}$, where each element contains at least one candidate cost for $O(I_l, I_r)$, i.e.,

$$
\begin{cases}
\{\mathbb{C}'_{O(I_l,I_r)}\} = \{c_{O(I_l,I_r)} + \mathbb{C}_{I_l} + \mathbb{C}_{I_r} \mid \\
\qquad c_O = min\,c_O \text{ or } c'_O, \mathbb{C}_I = min\,\mathbb{C}_I \text{ or } \mathbb{C}'_I, & (9)\\
\qquad \text{except } c_O = min\,c_O \text{ and both } \mathbb{C}_I = min\,\mathbb{C}_I\}, \\
\{\mathbb{C}'_{I_i}\} = \{\mathbb{C}'_O \mid O \text{ is underneath } I_i\}. & (10)
\end{cases}
$$

Example 4.2. For the operator input $\{6,7,8,9,10\}$, we first minimize $\mathbb{C}_{\{6,7,8,9,10\}}$ with no candidate costs, so that $min\,\mathbb{C}_{\{6,7,8,9,10\}} = 40$ as shown in Example 4.1. Then, we consider the CSE of $\boldsymbol{g}^T\boldsymbol{H}^T$. Due to the candidate cost $c'_{O(\{6\},\{7\})} = 5$, we have $\{\mathbb{C}'_{O(\{6\},\{7\})}\} = \{5\}$. For the upstream operator $O(\{6,7\},\{8\})$, we use Equations 9 and 10 to maintain a candidate set of accumulated costs.

$$
\begin{aligned}
\{\mathbb{C}'_{O(\{6,7\},\{8\})}\} &= \{min\,c_{O(\{6,7\},\{8\})} + \mathbb{C}'_{\{6,7\}} + min\,\mathbb{C}_{\{8\}}\} \\
&= \{min\,c_{O(\{6,7\},\{8\})} + \mathbb{C}'_{O(\{6\},\{7\})} + min\,\mathbb{C}_{\{8\}}\} \\
&= \{50 + 5 + 0\} = \{55\}.
\end{aligned}
$$

Here, $\{\mathbb{C}'_{O(\{6,7\},\{8\})}\}$ has only one element, because there is solely one candidate cost. Recursively, we have $\{\mathbb{C}'_{O(\{6,7,8,9\},\{10\})}\} = \{90\}$ for $O(\{6,7,8,9\},\{10\})$, and thereby $\{\mathbb{C}'_{\{6,7,8,9,10\}}\} = \{90\}$.

After obtaining a candidate set of costs, we have to gradually withdraw elements from the set, until it becomes empty. This withdrawal means, we either pick an element from the candidate set to minimize accumulated costs, or discard an element. In particular,

- We pick a group of relevant candidate costs, only if, in the joint upstream of their operators, we find that the CSE improve the performance. For example, when minimizing $\mathbb{C}_{\{6\},\{7\}}$, we cannot pick $c'_{O(\{6\},\{7\})} = 5$, because the performance impact of the CSE of $\boldsymbol{g}^T\boldsymbol{H}^T$ is unknown. That is, the relevant cost $c'_{O(\{11\},\{12\})}$ is not underneath $\{6,7\}$, such that we do not know whether to pick $c'_{O(\{11\},\{12\})}$ at that time. Hence, we keep the candidate costs, until, in the joint upstream of their operators, we have the accumulated costs containing both $c'_{O(\{6\},\{7\})}$ and $c'_{O(\{11\},\{12\})}$, which indicates the impact of the CSE. Typically, we compare $\mathbb{C}'_{O(I_l,I_r)}$ containing a group of relevant candidate costs with $min\,\mathbb{C}_{O(I_l,I_r)}$, to determine whether to pick those candidates.
- We discard a group of relevant candidate costs, if each of them cannot contribute to performance improvement. In specific, we regard a candidate cost as useless, if the candidate cannot contribute to performance improvement, i.e., minimize the accumulated costs of the upstream operators. Further, if a whole group of relevant candidate costs are useless, then the performance impact of the CSE is negative. Hence, we discard those costs from the candidate set. As illustrated in Example 4.2, with $c'_{O(\{6\},\{7\})}$ accumulated, $\mathbb{C}'_{O(\{6,7,8,9\},\{10\})} = 90$, larger than $min\,\mathbb{C}_{\{6,7,8,9,10\}}$ calculated in Example 4.1. Therefore, we can conclude that $c'_{O(\{6\},\{7\})}$ is useless for $\boldsymbol{g}^T\boldsymbol{H}^T\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{H}$. Suppose we find both $c'_{O(\{6\},\{7\})}$ $c'_{O(\{11\},\{12\})}$ useless for their upstream operators. We would discard $c'_{O(\{6\},\{7\})}$ and $c'_{O(\{11\},\{12\})}$ at once, since the CSE of $\boldsymbol{g}^T\boldsymbol{H}^T$ clearly does not improve performance.

According to the withdrawal of candidate costs, we remove operator costs and operators from the cost graph. If we withdraw $c'_{O(I_l,I_r)}$ from the candidate set, then we recalculate $min\,c_{O(I_l,I_r)}$ and remove the other $c_{O(I_l,I_r)}$ (extension of Line 12 in Algorithm 1). Additionally, in an operator underneath $O(I_l, I_r)$, if we withdraw a candidate cost for $O(I_l, I_r)$, then we recheck and remove the edge between $O(I_l, I_r)$ and the downstream operator (extension of Line 17 in Algorithm 1). This downstream operator has no candidate cost and its costs are not picked in $min\,\mathbb{C}_{O(I_l,I_r)}$.

In general, considering both CSE and LSE, we use Equations 7 and 8 to determine the efficient execution plan as well. The difference is, we maintain a candidate set of accumulated costs through Equations 9 and 10 to handle CSE costs, and recalculate the minimal accumulated costs step by step based on the candidate set.

5 SYSTEM IMPLEMENTATION

For the implementation of ReMac, we explored open-source distributed solutions that produce high abstraction level for algorithm specifications, including SystemDS [7], pbdR [23] (based on ScaLAPACK [2]), and SciDB [12]. According to a comparative evaluation research for scalable linear algebra-based analytics [28], SystemDS, pbdR, and SciDB are comparable on processing dense matrices. However, pbdR treats sparse matrices as dense ones, and SciDB does not widely support for sparse distributed matrices [31, 32].

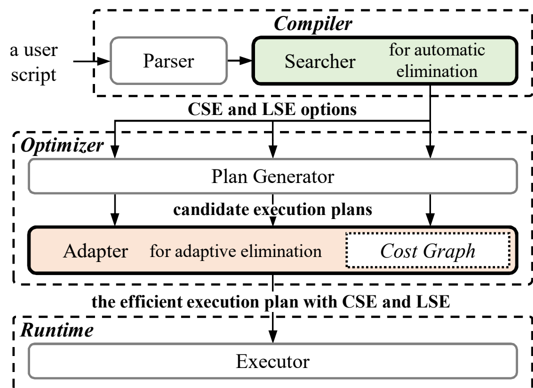


Figure 7: System Architecture

Therefore, when processing sparse matrices, SystemDS outperforms the others. Moreover, as shown in our experiments in Section 6.4, SystemDS automatically switches the execution between local and distributed mode, to avoid heavy communication cost. Hence, we chose to implement ReMac on top of SystemDS.

Basically, ReMac performs automatic elimination to find elimination options at first. However, due to contradictory and detrimental options, ReMac performs adaptive elimination to improve performance further. In specific, the architecture of ReMac consists of a *compiler*, an *optimizer*, and a *runtime* component, as depicted in Figure 7. 1) The compiler parses a user script into a syntax tree, and finds CSE and LSE options through our block-wise search. 2) For each option, the optimizer generates candidate plan trees to build a cost graph. Based on the cost graph, the optimizer adapts the execution plan to the efficient combination of the options through a dynamic programming process. 3) The runtime component takes charge of the execution of the optimized plan. We can also switch the components (e.g., the executor of SystemDS) to migrate ReMac to other solutions. Moreover, with the customizable components, ReMac is compatible to other optimization techniques. For example, the plan generator employs operator fusion [9] of SystemDS. Similarly, we can integrate other techniques into the plan generator.

6 EXPERIMENTAL STUDIES

Next, we will demonstrate the performance of ReMac employing automatic elimination and adaptive elimination, respectively. In addition, we evaluate other state-of-the-art solutions as references.

6.1 Experimental Setup

We conducted the performance analysis of five solutions, including ReMac, SystemDS 2.0.0, SPORES [29], SciDB 19.3.5, and pbdR (based on ScaLAPACK 2.0.2), on a seven-node cluster. Here, we use all internal nodes in our research group to let the baselines have the best performance. Each node has two Intel(R) Xeon(R) E5-2620 0 @ 2.00GHz six-core processors, 32GB DRAM, a 4TB hard disk and 1Gbps Ethernet. In addition, we deployed ReMac, SystemDS, and SPORES upon Spark 3.0.1.

Algorithms. In our experiments, we chose three algorithms resolving linear regression problems. They were Gradient Descent (GD), Davidon-Fletcher-Powell (DFP), and Broyden–Fletcher–Goldfarb–Shanno (BFGS). In particular, GD involves loop-constant subexpressions, and DFP as well as BFGS involve both common and loop-constant subexpressions.

Table 2: Dataset Statistics

Dataset	Abbr.	Rows#	Columns#	Sparsity	Footprint
criteo1	*cri1*	116.8M	47	6.0×10^{-1}	40.9GB
criteo2	*cri2*	58.4M	8.7K	4.5×10^{-3}	30.0GB
criteo3	*cri3*	58.4M	15.0K	2.6×10^{-3}	30.0GB
reddit1	*red1*	120.0M	34	5.1×10^{-1}	30.4GB
reddit2	*red2*	104.5M	5.0K	3.9×10^{-3}	31.5GB
reddit3	*red3*	104.5M	20.0K	9.6×10^{-4}	31.5GB

Datasets. We conducted experiments based on two real-world datasets, *criteo*[2] and *reddit*[3], akin to [21, 22, 28]. We conducted the benchmark on differently sized samples of both datasets. Specifically, we used the benchmark script[4] to convert the *criteo* click logs, and varied the lower bound of frequency for a raw feature to have its own dimension [18]. Eventually, we obtained a dense matrix, *cri1*, generated from the logs of the first two days, and two sparse matrices, *cri2* and *cri3*, generated from the logs of the first day. For the *reddit* dataset, we performed feature-hashing [30] to vectorize it into three different size of matrices by varying the number of output features. The matrices include one dense matrix, *red1*, generated from the data of September to October 2018, and two sparse matrices, *red2* and *red3*, generated from the data of September 2018. Table 2 lists the numbers of rows and columns, sparsity, and the memory footprint of these datasets.

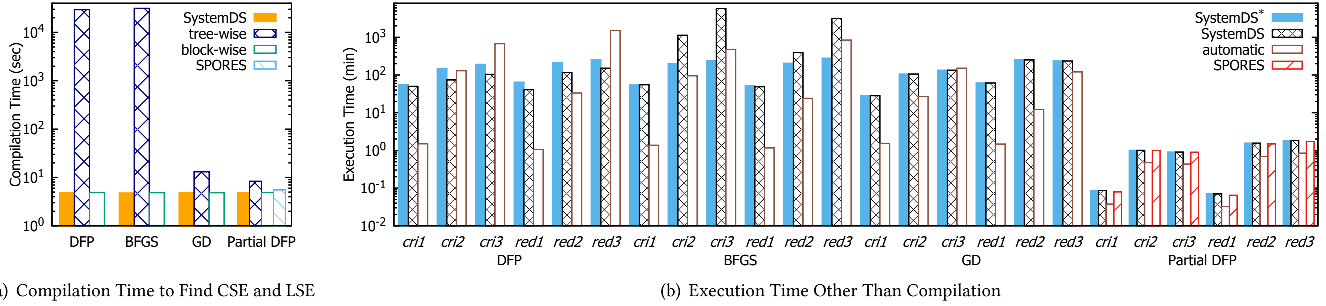
6.2 Performance of Automatic Elimination

In this section, we evaluate the overheads of different methods searching for CSE and LSE. Moreover, we show the inefficiency of blindly applying CSE and LSE.

6.2.1 Searching for CSE and LSE. Since the overhead of searching for CSE and LSE is irrelevant to datasets, here we demonstrate the experiments of all algorithms on one dataset, *cri2*. We compare the compilation time, that SystemDS and ReMac, i.e., block-wise search, take to find CSE and LSE. In addition, we implement a tree-wise search upon SystemDS. This method traverses all plan trees in multi-threading parallelism to find implicit CSE and LSE. We also evaluate the compilation time of SPORES, which supports implicit CSE. However, given that the current implementation of SPORES does not support running DFP or BFGS entirely, we take the longest subexpression that SPORES supports in DFP, $\boldsymbol{d}^T \boldsymbol{A}^T \boldsymbol{A} \boldsymbol{H} \boldsymbol{A}^T \boldsymbol{A} \boldsymbol{d}$ (denoted as partial DFP), to conduct experiments.

As depicted in Figure 8(a), in comparison to SystemDS, the block-wise search requires additional 61ms on average to find implicit CSE and LSE for DFP, BFGS, and GD, which is insignificant against the entire compilation. In contrast, due to the extremely high complexity of the traversal, the tree-wise search requires additional 8.3 seconds on GD, and more than 8 hours on both DFP and BFGS. For

[2]http://labs.criteo.com/2013/12/download-terabyte-click-logs-2/
[3]http://files.pushshift.io/reddit/comments/
[4]https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets



(a) Compilation Time to Find CSE and LSE

(b) Execution Time Other Than Compilation

Figure 8: Performance of Automatic Elimination

the partial DFP, SPORES and the block-wise search achieve comparable compilation time, whereas the tree-wise search requires additional 2.8 seconds.

In general, while the tree-wise search is impractical for complex algorithms, the block-wise search is able to find implicit CSE and LSE with a minor overhead.

6.2.2 Applying CSE and LSE. We conducted experiments to study the performance impact of CSE and LSE found in Subsection 6.2.1. As depicted in Figure 8(b), we evaluated the execution time with no CSE or LSE, i.e., disabling elimination for SystemDS (denoted as SystemDS*), the explicit CSE that SystemDS found, the CSE and LSE found through the block-wise search, i.e., automatic elimination, and the CSE that SPORES found. Note that the bars of automatic elimination also represent the execution time along with the tree-wise search, since the block-wise and tree-wise searches output the same results. In addition, due to contradiction, automatic elimination cannot apply all CSE and LSE options. Instead, it applies as many options as possible.

DFP. For DFP, via explicit CSE, SystemDS achieves a 1.1x speedup over SystemDS* on *cri1* and 1.8x speedups on average on the other datasets. Further, automatic elimination applies more implicit CSE and LSE to achieves speedups over SystemDS, which are 36.0x on *cri1* and *red1*, and 3.5x on *red2*. Nonetheless, automatic elimination is also 1.8x slower than SystemDS on *cri2*, and 8.3x slower on *cri3* and *red3*, illustrating implicit CSE and LSE may either improve or decrease performance. In particular, automatic elimination applies the implicit LSE of $\boldsymbol{A}^T\boldsymbol{A}$, which accelerates each iteration, yet incurs extra overhead cost outside the loop. The extra overhead cost is positively correlated with the size of $\boldsymbol{A}^T\boldsymbol{A}$, which essentially relies on the column number of the input dataset. Hence, the LSE becomes more costly, causing performance degradation (e.g., on *cri2*, *cri3*, and *red3*), as the column number of the dataset increases.

BFGS. As depicted in Figure 8(b), SystemDS does not benefit from explicit CSE when running BFGS. Especially, SystemDS is over 1.9x slower than SystemDS* on *cri2* and *red2*, and 11.4x on *cri3* and *red3*. That is, SystemDS employs the optimization rule of CSE first, but then the applied CSE prevents SystemDS from the subsequent optimization rules to improve the execution order of operators, and decreases performance instead. Based on SystemDS, automatic elimination applies more implicit CSE and LSE. The performance improvement attributed to implicit CSE and LSE offsets the side effect of the explicit CSE, achieving 42.1x speedups over SystemDS* on *cri1* and *red1*, and 5.3x speedups on *cri2* and *red2*. However, on *cri3* and *red3*, due to the extremely negative influence of explicit CSE, even automatic elimination is 2.5x slower than SystemDS*.

GD. Since there is no CSE in GD and SystemDS does not support LSE, the execution of SystemDS and SystemDS* are identical for GD. Different from SystemDS, automatic elimination applies LSE, which computes $\boldsymbol{A}^T\boldsymbol{A}$ ($\boldsymbol{A}$ is an input dataset) and changes the original execution order. As depicted in Figure 8(b), the LSE helps automatic elimination become 25.8x faster than SystemDS on *cri1*, *red1*, and *red2*, and 2.7x faster on *cri2* and *red3*. However, because of the change on the execution order, the LSE may also incur an overwhelming overhead. Especially, the overhead of the LSE offsets the performance improvement on *cri3* and *red3*, resulting in comparable performance between SystemDS and automatic elimination. This is because, both *cri3* and *red3* are "fat" datasets with smaller ratios of rows to columns than other datasets, in which case the execution of $\boldsymbol{A}^T\boldsymbol{A}$ becomes costlier.

Partial DFP. We also evaluate SystemDS*, SystemDS, SPORES, and automatic elimination for partial DFP. Here, the execution of SystemDS* and SystemDS is identical, since there is no explicit CSE in partial DFP. As depicted in Figure 8(b), by applying implicit CSE, automatic elimination achieves a 2.2x speedup over SystemDS on average. Furthermore, SPORES explores implicit CSE as well. However, the performance of SPORES and SystemDS are comparable, because SPORES only permutes a multiplication chain in a limited number of attempts, and fails to find all implicit CSE. As a remedy, SPORES depends on the fused mmchain operator to accelerate the execution of multiplication chains. Nonetheless, mmchain only supports a multiplication chain of three matrices, and has constraints on the column number of the second matrix (less than 1K in default). For example, SPORES fails to use mmchain for partial DFP on *cri3*, since the dataset matrix has 15K columns. As a result, SPORES is not suitable for long multiplication chains.

In general, although SystemDS and SPORES support applying CSE to improve performance, there still remain implicit CSE and LSE not utilized. In the best case, automatic elimination achieves a 41.5x speedup by applying implicit CSE and LSE. However, automatic elimination may also be extremely slower than SystemDS and SPORES (e.g., 10.0x slower for DFP on *cri3*). This motivates us to study adaptive elimination next.

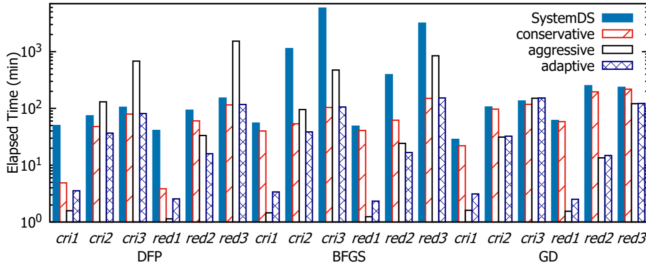


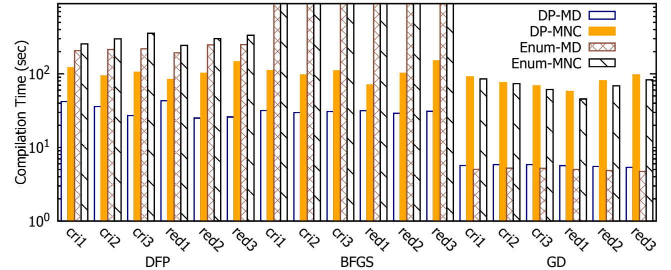
Figure 9: Overall Performance with Different CSE and LSE

6.3 Performance of Adaptive Elimination

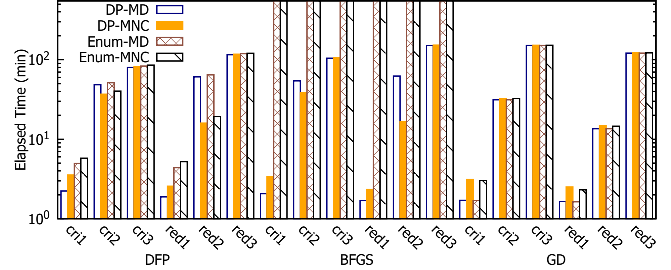
Since it is essential to adapt CSE and LSE to different cases, we further evaluate ReMac performing adaptive elimination. In addition, to show the insights of the performance, we demonstrate the efficiency and accuracy of the cost model and the effect of the dynamic programming-based method used for adaptive elimination.

6.3.1 Overall Performance. Figure 9 shows the overall performance of SystemDS, as a reference, and adaptive elimination on DFP, BFGS, and GD. Furthermore, we take two alternative strategies that apply different elimination options. The "conservative strategy" only applies the elimination options following the original execution order of operators. Particularly, the conservative strategy applies CSE after all optimizations improving the operator order, whereas SystemDS applies CSE before certain optimizations, at the risk of performance reduction (e.g., BFGS in Figure 8(b)). In addition, the "aggressive strategy" applies the elimination options changing the original execution order in prior and the other options in posterior.
DFP. The conservative strategy outperforms SystemDS by following the original execution order of operators (10.4x faster on *cri1* and *red1*, and 1.4x faster on the other datasets, in specific). However, the conservative strategy also fails to exploit the elimination options changing the original execution order. On the other hand, those options help the aggressive strategy achieve 3.2x speedups over the conservative strategy on *cri1* and *red1*, and a 1.8x speedup on *red2*. However, there is a side effect of changing the execution order (e.g., the elimination of $\boldsymbol{A}^T\boldsymbol{A}$ and $\boldsymbol{dd}^T$), which may cause an extreme performance degradation. Hence, the aggressive strategy is 2.7x slower than the conservative strategy on *cir2*, 8.5x slower on *cri3*, and 13.3x slower on *red3*.

Unlike the conservative or aggressive strategy, adaptive elimination applies different combinations of CSE and LSE on different datasets. On *cri1* and *red1*, since adaptive elimination is not bound to the original execution order, it achieves 1.7x speedups over the conservative strategy by applying more elimination options. On *cri3* and *red3*, adaptive elimination finds the CSE and LSE applied by the aggressive strategy to be detrimental. Hence, adaptive elimination follows the conservative strategy, achieving 1.26x speedups over SystemDS. Moreover, via picking elimination options, adaptive elimination outperforms both conservative and aggressive strategies by 2.5x on *cri2* and *red2* on average. For example, adaptive elimination picks the option of $\boldsymbol{A}^T\boldsymbol{A}$, but not the detrimental option of $\boldsymbol{dd}^T$. However, the conservative strategy applies neither of the two options, and the aggressive strategy applies both of them.



(a) Compilation Time to Generate the Efficient Execution Plan



(b) Elapsed Time of Compilation and Execution

Figure 10: Adaptive Elimination Using Different Methods

Even though adaptive elimination incurs an extra overhead cost to pick CSE and LSE, the cost is affordable and worthy. For example, although adaptive elimination is slower than the aggressive strategy on *cri1* and *red1*, it still achieves significant speedups over SystemDS and outperforms the conservative strategy. More importantly, on the other datasets, adaptive elimination avoids the performance degradation incurred by the aggressive strategy.
BFGS. As depicted in Figure 9, the conservative strategy becomes 1.3x faster than SystemDS on *cri1* and *red1*, 6.3x faster on *red2*, and over 21.0x faster on the other datasets. Furthermore, the aggressive strategy applies more CSE and LSE changing the original execution order, to achieve 30.3x speedups over the conservative strategy on *cri1* and *red1*, and a 2.6x speedup on *red2*. Nonetheless, similar to DFP, those CSE and LSE may also decrease performance for BFGS. For example, the aggressive strategy is 1.8x slower than the conservative strategy on *cri2*, and 5.1x slower on *cri3* and *red3*. Hence, either the conservative or aggressive strategy has its shortcomings.

To mitigate this, ReMac employs adaptive elimination. On *cri3* and *red3*, in comparison to the aggressive strategy, adaptive elimination ignores detrimental CSE and LSE, and achieves 5.0x speedups. On the other datasets, adaptive elimination does not follow the restrictions of the conservative strategy, and achieves 16.9x speedups over the conservative strategy on *cri1* and *red1*, and 2.2x and 3.1x speedups on *cri2* and *red2*.
GD. Different from SystemDS, the conservative strategy applies the additional LSE of a matrix-vector multiplication. However, with this LSE only, the conservative strategy fails to resolve the performance bottleneck and achieves merely 1.2x speedups over SystemDS on average. Different from the conservative strategy, the aggressive strategy also applies the LSE of a matrix-matrix multiplication, to further improve performance. Consequently, adaptive elimination follows the aggressive strategy, and achieves 6.7x speedups over the conservative strategy on average.

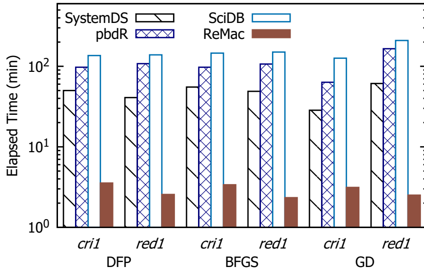


Figure 11: Alternative Solutions

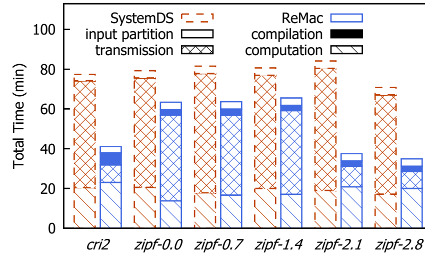


Figure 12: Performance Analysis for DFP

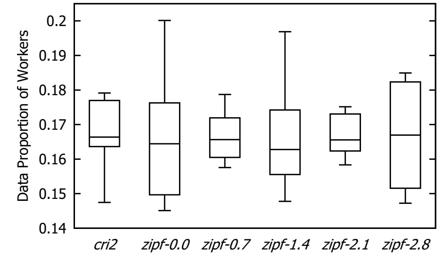


Figure 13: Work Balance for DFP

In summary, since the conservative strategy follows the original execution order, it certainly outperforms SystemDS. Nonetheless, this strategy is too conservative in applying CSE and LSE, and thereby cannot fully explore the benefits of redundancy elimination. On the other hand, the aggressive strategy applies more CSE and LSE, trying to achieve higher speedups, but it may suffer from contradictory and detrimental elimination options. Hence, to address this, ReMac adaptively applies elimination options, especially those change the original execution order. Although adaptive elimination incurs an extra overhead cost, it manages to outperform conservative and aggressive strategies, and eventually achieves a 13.3x speedup over SystemDS on average.

6.3.2 Efficiency and Accuracy of Cost Model. Further, we examine the efficiency and accuracy of our cost model in adaptive elimination, as shown in Figures 10. Here, we focus on the cost model using different sparsity estimators with the dynamic programming-based method (denoted as DP). DP-MD and DP-MNC represent using a metadata-based sparsity estimator and MNC, respectively.

In terms of compilation time, i.e., the efficiency of cost evaluation, DP-MD outperforms DP-MNC in each case, as shown in Figure 10(a). From the perspective of elapsed time, the metadata-based sparsity estimator helps ReMac become 1.6x faster than DP-MNC on *cri1* and *red1* on average, as shown in Figure 10(b). However, there are cases DP-MD generates suboptimal execution plans, getting greater performance degradation than cost evaluation, whereas DP-MNC is able to generate more efficient plans. For example, for BFGS on *red2*, DP-MNC is 3.7x faster than DP-MD.

In general, DP-MNC is 3.8x faster than DP-MD at most (for DFP on *red2*), whereas DP-MD is 1.8x faster than DP-MNC at most (for GD on *cri1*). Given that, we choose MNC as sparsity estimator for adaptive elimination in ReMac to report the results in Section 6.3.1.

6.3.3 Effect of Dynamic Programming-based Method. We also evaluated the dynamic programming-based method used for adaptive elimination and the brute-force enumeration (denoted as Enum) as a reference. Here, we implemented two Enum methods that enumerate elimination combinations in the depth-first and breadth-first manner, respectively. Due to limited space, we show the faster of the two Enum methods with respect to compilation time.

As depicted in Figure 10(a), we investigated the compilation time, i.e., time for generating the efficient execution plan with redundancy elimination, of DP and Enum methods. For GD, since the expressions is simple enough, the two methods achieve comparable performance. For DFP, the DP method addresses the combinatorial explosion and outperforms the Enum method by 5.0x on average, whatever sparsity estimator ReMac uses. Moreover, due to the complexity of GNMF, the combinatorial explosion becomes more critical, resulting in the Enum method taking over three days. Whereas the DP method costs less than 150 seconds.

In terms of elapsed time as depicted in Figure 10(b), the DP method also outperforms the Enum method. Note that the execution time may dilute the advantages of the DP method. For example, for DFP on *cri2*, *cri3*, *red2*, and *red3*, the DP method is only 1.1x faster than the Enum method. Nonetheless, the cost of combinatorial explosion remains overwhelming for DFP on *cri1* and *red1*, and GNMF. The DP method manages to mitigate this, and achieves 2.1x speedups for DFP on *cri1* and *red1* and over 30x speedups for GNMF.

6.4 Comparison with Other Solutions

We showcase the advantages of ReMac in comparison to the state-of-the-art solutions, including SystemDS, pbdR based on ScaLAPACK, and SciDB, corresponding to dataflow, HPC, and database systems, respectively. Here, we conducted experiments on two dense datasets, *cri1* and *red1*, because ScaLAPACK and SciDB does not widely support sparse matrices [31, 32]. In particular, ScaLAPACK treats sparse matrices as dense ones, and SciDB does not support multiplying a sparse matrix by a dense matrix.

As depicted in Figure 11, SystemDS is 2.8x faster than pbdR and SciDB, because SystemDS dynamically chooses to execute operators in either local mode or distributed mode. In this way, SystemDS saves significant communication overheads, whereas pbdR and SciDB keep running in distributed mode. However, the shortcoming of SystemDS is that it does not fully utilize CSE and LSE. As a result, we built ReMac atop SystemDS, performing automatic elimination and adaptive elimination. As shown in Figure 11, ReMac benefits from the superiority of SystemDS and further achieves 14.4x speedups over SystemDS.

6.5 Discussion

In this section, we provide further insights on the performance of distributed matrix computation, and how ReMac improves that. Moreover, since sparse datasets are common to be skewed, we discuss the impact of skewness on ReMac.

Input Partition Time. For fair comparisons, we conduct the experiments in Sections 6.2 - 6.4 after partitioning data into systems. That is, the input data has already been partitioned in HDFS for SystemDS and ReMac, distributed across the cluster for pbdR, and stored in a distributed table for SciDB. In general, we find the input partition time is minor in SystemDS and ReMac, but significant in

pbdR and SciDB. We run DFP on *cri2* without partitioning input data in advance, and Figure 12 shows that SystemDS takes 3 minutes to partition *cri2*. By inheriting from SystemDS, ReMac achieves the same input partition time. Here, ReMac still achieves a 1.9x speedup over SystemDS, close to the 2.0x speedup with input partitioned in advance as shown in Figure 9. Certainly, the input partition time is minor and does not affect how ReMac applies CSE and LSE options. Nonetheless, pbdR and SciDB take hours for input partition. In particular, both pbdR and SciDB do not support automatically splitting and partitioning a dataset in parallel. Moreover, pbdR builds a dense distributed matrix even if the data is sparse, and SciDB requires a redimension operation to build a sparse matrix [3].

Performance Bottleneck. We analyse the performance of distributed matrix computation in the example of DFP on *cri2*. Except for the aforementioned input partition, the total running time consists of the compilation time as well as the computation time and the time blocked by transmission during execution. According to the results in Figure 12, the performance bottleneck is transmission, which occupies 70% of the total time. However, ReMac reduces it to 22% via redundancy elimination. In particular, ReMac applies the LSE of $\boldsymbol{A}^T\boldsymbol{A}$, moving the distributed computation of Equation 2 out of the loop. Hence, ReMac becomes 1.9x faster than SystemDS.

Skewness. Next, we evaluate the performance of ReMac on skewed datasets. We generate five synthetic datasets skewed with Zipf distributions, namely *zipf-0.0*, *zipf-0.7*, *zipf-1.4*, *zipf-2.1*, and *zipf-2.8*. They have the same row and column numbers as well as the sparsity of *cri2*, but their rows and columns skewed with Zipf exponents from 0 to 2.8. Particularly, the non-zeros in *zipf-0.0* follow a uniform distribution, whereas in *zipf-2.8*, more than 95% of the non-zeros gather in 5% of the rows and columns.

Figure 12 shows how ReMac handles the differently skewed data. On average, ReMac is 1.7x faster than SystemDS. In specific, ReMac reduces the transmission time by 27% on *zipf-0.0*, *zipf-0.7*, and *zipf-1.4*, and 83% on *zipf-2.1* and *zipf-2.8*. Here, a vital factor is, the sparsity of intermediate matrices varies with the distributions of the non-zeros in datasets. ReMac senses this difference via the sparsity estimator of the cost model, and generates different execution plans on those datasets. For DFP, ReMac finds the fact that the LSE of $\boldsymbol{A}^T\boldsymbol{A}$ is actually detrimental on *zipf-0.0*, *zipf-0.7*, and *zipf-1.4*, but efficient on *zipf-2.1* and *zipf-2.8*. This leads to the different execution plans and the jump in performance from *zipf-1.4* to *zipf-2.1*.

Moreover, to explore the impact of skewness on work balance, we measure the data size processed by Spark workers in proportion. As depicted in Figure 13, whether the data is skewed, the proportions are close to 1/6. Since there are six workers, the results mean the workload is balanced, which is actually attributed to SystemDS and Spark. In specific, ReMac follows SystemDS to split a matrix into 1000×1000 blocks and partition them via hashing, and also exploits the load balancing strategy of Spark. In general, ReMac mitigates work imbalance with existing techniques of SystemDS and Spark.

7 RELATED WORK

This section describes the related work on redundancy elimination and distributed matrix computation.

Redundancy Elimination in Databases. The elimination of common subqueries has been widely studied in databases and data analysis systems (e.g., PostgreSQL [1], Oracle Database [6], HP Vertica [5], and Microsoft's PDW [25]). The problem is also known as Common Table Expressions (CTEs). Similar to CSE, the CTE optimization reuses the results of common subqueries in multiple places to improve performance. However, those systems only optimize CTEs manually defined in SQL scripts (e.g., with "WITH" clauses), unlike ReMac automatically searching for and applying CSE. In addition, Mitos [16] supports loop-invariant hoisting, i.e., the elimination of explicit loop-constant subqueries, whereas ReMac supports both explicit and implicit LSE.

Furthermore, there are works focusing on achieving globally optimal plans with CTE. Silva et al. [26] observe that queries with CTEs may require different partition schemes of the CTE outputs, and accordingly propose SCOPE trading off such conflicting requirements in a cost-based manner. Based on SCOPE, El-Helw et al. [13] further contribute to pruning the plan space for globally optimal plans. They propose a plan enumeration technique with specified transformation rules, which compute a lower bound on the cost of an optimization, and depend on the lower bound to cut off unnecessary CTE optimizations early on. The idea of achieving globally optimal plans with CTE is similar to that of our adaptive elimination. However, instead of plan enumeration, ReMac employs a dynamic programming-based method to prune the plan space.

Distributed Matrix Computation. Distributed matrix computation is typically optimized from two aspects: the physical layouts of distributed matrices and the execution plans. In particular, our work focuses on optimizing execution plans, where the optimization decisions also involves the existing works on physical layouts.

For physical layouts, the well-known large-scale solutions (e.g., MLlib [11] and SciDB [12]) partition distributed matrices into fixed-size blocks, so as to accelerate the physical processing of binary operators. Based on those works, we analyse the costs of operators to help make decisions on redundancy elimination.

For execution plans, with respect to redundant subexpressions, SystemDS [8] and MATFAST [32] only have limited support for explicit CSE or LSE, whereas ReMac exploits both explicit and implicit CSE and LSE. In addition, SPORES [29] centers on the study of operator fusion [9, 14] considering CSE. However, it uses a sampling technique to handle the large search space, particularly for multiplication chains, which has no guarantee to find all CSE. Differently, ReMac is able to automatically find all CSE and LSE with a negligible overhead. Furthermore, ReMac employs adaptive elimination to achieve the efficient plan.

8 CONCLUSION

In this paper, we propose a new system, ReMac. In order to automatically exploit both explicit and implicit redundancy elimination, ReMac employs a block-wise search that finds CSE and LSE with negligible overheads. Moreover, given that multiple elimination options may be related to each other, ReMac adaptively chooses the efficient combination of elimination options. To prevent the combinatorial explosion in adaptive elimination, ReMac employs a dynamic programming-based method. Presently, the ReMac prototype is built on top of SystemDS. Compared to state-of-the-art solutions including SystemDS, SciDB, and ScaLAPACK, ReMac is over 13.3x faster across a range of algorithms. In addition, since the

techniques are independent with execution engines, it is possible to integrate our work into other systems, like SciDB and Flink.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (No. 61902128), Shanghai Sailing Program (No. 19YF1414200), and CCF-Tencent Open Fund (RAGR20210110).

REFERENCES

[1] PostgreSQL. http://www.postgresql.org/.

[2] ScaLAPACK. http://www.netlib.org/scalapack/.

[3] SciDB Document. https://paradigm4.atlassian.net/wiki/spaces/scidb/.

[4] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 265–283.

[5] Chuck Bear, Andrew Lamb, and Nga Tran. 2012. The Vertica Database: SQL RDBMS for Managing Big Data. In *Proceedings of the 2012 Workshop on Management of Big Data Systems (MBDS)*. 37–38.

[6] Srikanth Bellamkonda, Rafi Ahmed, Andrew Witkowski, Angela Amor, Mohamed Zait, and Chun-Chieh Lin. 2009. Enhanced Subquery Optimizations in Oracle. *Proc. VLDB Endow. (PVLDB)* 2, 2 (2009), 1366–1377.

[7] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*.

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

[9] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Prithviraj Sen, Alexandre V. Evfimievski, and Niketan Pansare. 2018. On Optimizing Operator Fusion Plans for Large-scale Machine Learning in SystemML. *Proc. VLDB Endow. (PVLDB)* 11, 12 (2018), 1755–1768.

[10] Matthias Böhm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian. 2014. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *IEEE Data Eng. Bull.* 37, 3 (2014), 52–62.

[11] Reza Bosagh Zadeh, Xiangrui Meng, Alexander Ulanov, Burak Yavuz, Li Pu, Shivaram Venkataraman, Evan Sparks, Aaron Staple, and Matei Zaharia. 2016. Matrix Computations and Optimization in Apache Spark. In *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 31–38.

[12] Paul G. Brown. 2010. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *Proceedings of the 2010 ACM International Conference on Management of Data (SIGMOD)*. 963–968.

[13] Amr El-Helw, Venkatesh Raghavan, Mohamed A. Soliman, George Caragea, Zhongxian Gu, and Michalis Petropoulos. 2015. Optimization of Common Table Expressions in MPP Database Systems. *Proc. VLDB Endow. (PVLDB)* 8, 12 (2015), 1704–1715.

[14] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. 2017. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR)*.

[15] Rong Gu, Yun Tang, Chen Tian, Hucheng Zhou, Guanru Li, Xudong Zheng, and Yihua Huang. 2017. Improving Execution Concurrency of Large-Scale Matrix Multiplication on Distributed Data-Parallel Platforms. *IEEE Trans. Parallel Distributed Syst. (TPDS)* 28, 9 (2017), 2539–2552.

[16] Gábor E. Gévay, Tilmann Rabl, Sebastian Breß, Loránd Madai-Tahy, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Efficient Control Flow in Dataflow Systems: When Ease-of-Use Meets High Performance. In *Proceedings of the 37th IEEE International Conference on Data Engineering (ICDE)*.

[17] Donghyoung Han, Yoon-Min Nam, Jihye Lee, Kyongseok Park, Hyunwoo Kim, and Min-Soo Kim. 2019. DistME: A Fast and Elastic Distributed Matrix Computation Engine Using GPUs. In *Proceedings of the 2019 ACM International Conference on Management of Data (SIGMOD)*. 759–774.

[18] Yuchin Juan, Yong Zhuang, Wei-Sheng Chin, and Chih-Jen Lin. 2016. Field-Aware Factorization Machines for CTR Prediction. In *Proceedings of the 10th ACM Conference on Recommender Systems (RecSys)*. 43–50.

[19] David Kernert, Frank Köhler, and Wolfgang Lehner. 2015. SpMacho - Optimizing Sparse Linear Algebra Expressions with Probabilistic Density Estimation. In *Proceedings of the 18th International Conference on Extending Database Technology (EDBT)*. 289–300.

[20] Arun Kumar, Matthias Boehm, and Jun Yang. 2017. Data Management in Machine Learning: Challenges, Techniques, and Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*. 1717–1722.

[21] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *Proc. VLDB Endow. (PVLDB)* 12, 11 (2019), 1553–1567.

[22] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A Data System for Optimized Deep Learning Model Selection. *Proc. VLDB Endow. (PVLDB)* 13, 12 (2020), 2159–2173.

[23] G. Ostrouchov, W.-C. Chen, D. Schmidt, and P. Patel. 2012. *Programming with Big Data in R.* http://r-pbd.org/

[24] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-Grained Lineage Tracing and Reuse in Machine Learning Systems. In *Proceedings of the 2021 ACM International Conference on Management of Data (SIGMOD)*. 1426–1439.

[25] Srinath Shankar, Rimma Nehme, Josep Aguilar-Saborit, Andrew Chung, Mostafa Elhemali, Alan Halverson, Eric Robinson, Mahadevan Sankara Subramanian, David DeWitt, and César Galindo-Legaria. 2012. Query Optimization in Microsoft SQL Server PDW. In *Proceedings of the 2012 ACM International Conference on Management of Data (SIGMOD)*. 767–776.

[26] Yasin N. Silva, Per-Åke Larson, and Jingren Zhou. 2012. Exploiting Common Subexpressions for Cloud Query Processing. In *Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE)*. 1337–1348.

[27] Johanna Sommer, Matthias Boehm, Alexandre V. Evfimievski, Berthold Reinwald, and Peter J. Haas. 2019. MNC: Structure-Exploiting Sparsity Estimation for Matrix Expressions. In *Proceedings of the 2019 ACM International Conference on Management of Data (SIGMOD)*. 1607–1623.

[28] Anthony Thomas and Arun Kumar. 2018. A Comparative Evaluation of Systems for Scalable Linear Algebra-Based Analytics. *Proc. VLDB Endow. (PVLDB)* 11, 13 (2018), 2168–2182.

[29] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *Proc. VLDB Endow. (PVLDB)* 13, 12 (2020), 1919–1932.

[30] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. 2009. *Feature Hashing for Large Scale Multitask Learning.* 1113–1120.

[31] Lele Yu, Yingxia Shao, and Bin Cui. 2015. Exploiting Matrix Dependency for Efficient Distributed Matrix Computation. In *Proceedings of the 2015 ACM International Conference on Management of Data (SIGMOD)*. 93–105.

[32] Yongyang Yu, MingJie Tang, Walid G. Aref, Qutaibah M. Malluhi, Mostafa M. Abbas, and Mourad Ouzzani. 2017. In-Memory Distributed Matrix Computation Processing and Optimization. In *Proceedings of the 33rd IEEE International Conference on Data Engineering (ICDE)*. 1047–1058.