# ReMac: A Matrix Computation System with Redundancy Elimination

Zihao Chen
East China Normal University[†]
zhchen@stu.ecnu.edu.cn

Zhizhen Xu
East China Normal University[†]
zhizhxu@stu.ecnu.edu.cn

Baokun Han
East China Normal University[†]
bkhan@stu.ecnu.edu.cn

Chen Xu[*]
East China Normal University[†]
cxu@dase.ecnu.edu.cn

Weining Qian
East China Normal University[†]
wnqian@dase.ecnu.edu.cn

Aoying Zhou
East China Normal University[†]
ayzhou@dase.ecnu.edu.cn

## ABSTRACT

Distributed matrix computation solutions support query interfaces of linear algebra expressions, which often contain redundancy, i.e., common and loop-constant subexpressions. However, existing solutions fail to find all redundant subexpressions. Moreover, eliminating the found redundancy leads to new execution order of operators, which may have side effect. To exploit the benefits of redundancy elimination, we propose a new system called *ReMac*, which performs *automatic* and *adaptive elimination*. 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 with redundancy elimination. In this demonstration, attendees will have an opportunity to experience the effect that automatic and adaptive elimination have on distributed matrix computation.

**PVLDB Reference Format:**
Zihao Chen, Zhizhen Xu, Baokun Han, Chen Xu, Weining Qian, and Aoying Zhou. ReMac: A Matrix Computation System with Redundancy Elimination. PVLDB, 15(12): 3674 - 3677, 2022.
doi:10.14778/3554821.3554872

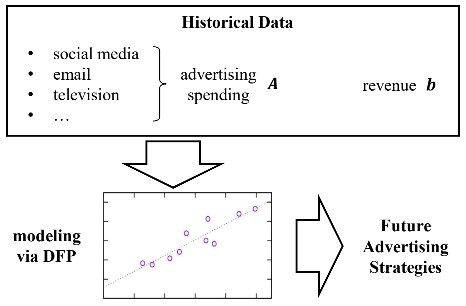
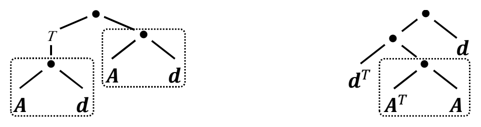


**Figure 1: Application of DFP**



(a) Common Subexpression (b) Loop-constant Subexpression

**Figure 2: Examples of Redundancy in DFP**

## 1 INTRODUCTION

Matrix computation is widely used in data science, machine learning, and statistical science. To perform matrix computation on large-scale datasets, numbers of systems and solutions have emerged, such as SystemDS [2], MLlib [3], ScaLAPACK [1], and SciDB [4]. To improve the performance of distributed matrix computation, one essential optimization is to eliminate the redundant computation and communication in execution plans.

To ease our discussion and illustrate redundancy elimination, consider using the Davidon-Fletcher-Powell (DFP) formulation to model the relationship between advertising spending and revenue for future advertising strategies, as depicted in Figure 1. For simplicity, the model uses linear regression on least squares, i.e., $\underset{\boldsymbol{x}\in\mathbb{R}^n}{min}\ \|\boldsymbol{Ax}-\boldsymbol{b}\|_2$. Here, $\boldsymbol{A}$ represents the historical advertising spending, and $\boldsymbol{b}$ represents the corresponding revenue. In particular, the DFP formulation involves an expression $\boldsymbol{d}^T\boldsymbol{A}^T\boldsymbol{Ad}$. There are two types of redundancy elimination in this expression: 1) common subexpression elimination (CSE), for the identical subtrees in the execution plan (e.g., $\boldsymbol{Ad}$ as depicted in Figure 2(a)), and 2) loop-constant subexpression elimination (LSE), for the subtrees with constant outputs in loops (e.g., $\boldsymbol{A}^T\boldsymbol{A}$ as depicted in Figure 2(b)).

In our recent work [5], we have found the execution plan may not explicitly indicate CSE or LSE, preventing the system from detecting redundancy. Hence, we aim to study a novel method to search for CSE and LSE, supporting automatic elimination. Furthermore, the found elimination options may be contradictory or detrimental to performance. This motivates us to discover an efficient method to combine those options, supporting adaptive elimination. We have developed a novel system called ReMac built atop SystemDS, which implements *automatic* and *adaptive elimination*, fully exploits CSE and LSE, and improves distributed computation over matrices. For automatic elimination, we propose a *block-wise* method that exploits the properties of matrix computation to speed up the search for CSE and LSE. For adaptive elimination, we adopts a *cost model* to

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

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554872

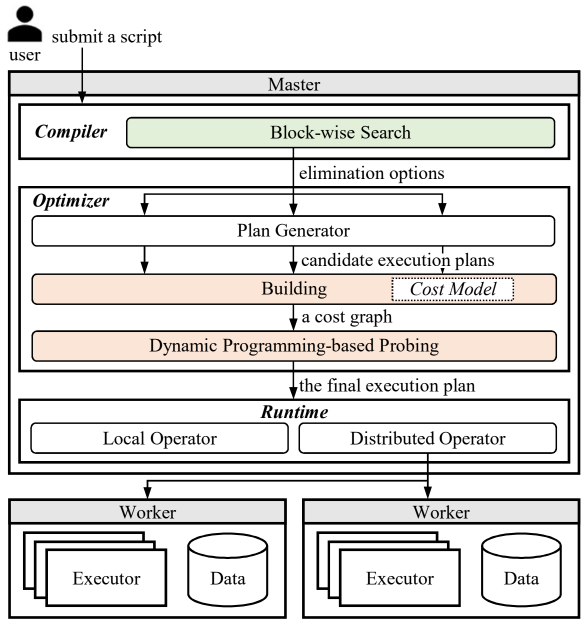


**Figure 3: System Architecture**

evaluate CSE and LSE options, as well as a *dynamic programming*-based method to address the combinatorial explosion of CSE and LSE options. In our earlier experiments, we determined that ReMac achieves a 14.4x speedup over SystemDS.

In this demonstration, we showcase two aspects: 1) how the *block-wise* method speeds up the search for CSE and LSE, 2) how ReMac evaluates the performance impact of CSE and LSE options via the *cost model*, and how the *dynamic programming*-based method overcomes the combinatorial explosion of those options.

## 2 REMAC OVERVIEW

In this section, we briefly introduce the goals of ReMac, and elaborate the design details.

### 2.1 System Goals

Redundancy elimination in matrix computation includes CSE and LSE. First, to make full advantage of CSE and LSE, ReMac automatically finds all elimination options, especially implicit ones, within negligible search time. Furthermore, we observe that the options may be contradictory, requiring trade-off. Meanwhile, some of them may even be detrimental to performance, and therefore have to be abandoned. Based on those observations, it is natural to employ adaptive elimination in ReMac, i.e., generating an efficient combination of CSE and LSE options to improve the final execution plan for matrix computation.

### 2.2 System Design

ReMac is designed to run on a cluster, which consists of two types of nodes: a master and multiple workers (as depicted in Figure 3). The master node generates an execution plan from an input script, and drives the execution, which is performed by the worker node.

In ReMac, the master node is comprised of three components: a *compiler*, an *optimizer*, and a *runtime*. First, according to a user-defined script, the compiler obtains a syntax tree, and finds CSE and LSE options through the block-wise search. Second, the role of the optimizer is to generate an efficient execution plan with redundancy elimination. In specific, the optimizer creates candidate plan trees to build a cost graph, and adapts the graph to the efficient combination of the options through a dynamic programming procedure. Finally, the runtime executes the operators of the plan in either local or distributed mode. In the following, we will dive into each of these three components.

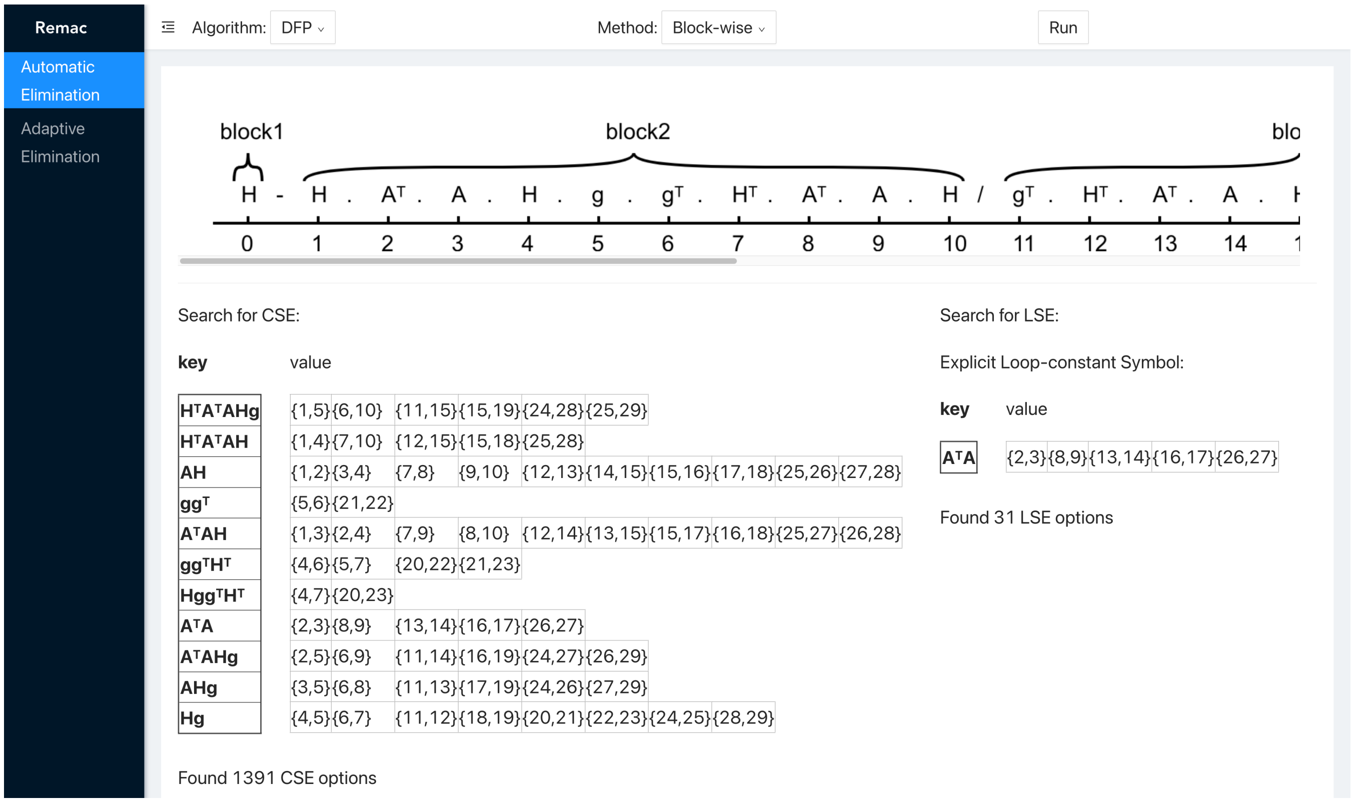
**Compiler.** The compiler takes charge of automatic elimination. It initially parses a script into a syntax tree. Based on the tree, the compiler searches for common or loop-constant subexpressions, preparing for automatic elimination. However, to find all implicit redundancy from the tree directly, we have to traverse all equivalent syntax trees. This tree-wise search method involves a duplicated search procedure and suffers from a large search space as well. 1) The method needs to scan the whole tree for redundancy, after transforming a syntax tree into an equivalent tree by changing a certain subtree. That means revisiting the other unchanged subtrees. 2) The method is strict with the internal execution order of subexpressions. However, it may be unnecessary to traverse all execution order to find out redundancy. For example, to detect the common subexpression $\boldsymbol{A}^T\boldsymbol{A}\boldsymbol{d}$, we do not need to concern whether its internal execution order is $(\boldsymbol{A}^T\boldsymbol{A})\boldsymbol{d}$ or $\boldsymbol{A}^T(\boldsymbol{A}\boldsymbol{d})$.

As a result, to overcome the search issues, we propose a *block-wise search* method for ReMac. First, we mitigate duplicated search via employing a divide-and-rule approach. That is, we split a given expression into blocks, and search for redundancy by blocks, to avoid revisiting unchanged parts during traversal. Second, 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. In particular, instead of traversing equivalent subtrees, we employ sliding windows to accomplish the search on the chains.

**Optimizer.** The optimizer is responsible for adaptive elimination. It generates an efficient execution plan along with the elimination options fed by the compiler. That means, the optimizer has to combine the options properly. To do so, we employs a cost model to evaluate operators in terms of computation and transmission. The cost model allows the optimizer to predict the impacts of different elimination options and thus choose the more efficient options.

However, due to the large number of elimination options, enumerating and evaluating elimination combinations would lead to the combinatorial explosion and thus unaffordable overhead costs. Hence, we propose a *dynamic programming-based* method to mitigate this. The main idea of this method is to evaluate each elimination option solely and form the efficient combination based on those evaluation results. Accordingly, the method consists of two phases, namely building and probing.

In the building phase, the optimizer generates optimized execution plan for each elimination options, regarded as candidates for our final execution plan. Subsequently, the optimizer evaluates those candidate plans via the cost model and collates the results into a cost graph. Here, the cost graph includes the topologies of



Algorithm: DFP

Method: Block-wise

Search for CSE:

| key | value |
|---|---|
| $H^TA^TAHg$ | {1,5}{6,10} {11,15}{15,19}{24,28}{25,29} |
| $H^TA^TAH$ | {1,4}{7,10} {12,15}{15,18}{25,28} |
| $AH$ | {1,2}{3,4} {7,8} {9,10} {12,13}{14,15}{15,16}{17,18}{25,26}{27,28} |
| $gg^T$ | {5,6}{21,22} |
| $A^TAH$ | {1,3}{2,4} {7,9} {8,10} {12,14}{13,15}{15,17}{16,18}{25,27}{26,28} |
| $gg^TH^T$ | {4,6}{5,7} {20,22}{21,23} |
| $Hgg^TH^T$ | {4,7}{20,23} |
| $A^TA$ | {2,3}{8,9} {13,14}{16,17}{26,27} |
| $A^TAHg$ | {2,5}{6,9} {11,14}{16,19}{24,27}{26,29} |
| $AHg$ | {3,5}{6,8} {11,13}{17,19}{24,26}{27,29} |
| $Hg$ | {4,5}{6,7} {11,12}{18,19}{20,21}{22,23}{24,25}{28,29} |

Found 1391 CSE options

Search for LSE:

Explicit Loop-constant Symbol:

| key | value |
|---|---|
| $A^TA$ | {2,3}{8,9}{13,14}{16,17}{26,27} |

Found 31 LSE options

Figure 4: Block-wise Search

candidates and the costs of their operators, which are sufficient enough to derive our final plan later.

In the probing phase, the optimizer prunes the cost graph through a dynamic programming procedure. In specific, the procedure is to compare and choose operators based on their costs from the bottom up, and finally convert the graph into a tree, representing our final execution plan.

**Runtime.** The runtime component runs the efficient execution plan with redundancy elimination provided by the optimizer. An execution plan contains two types of operators: local operators performed on a single machine and distributed operators performed in a distributed environment. For distributed operators, the runtime component drives the executors in the workers to perform distributed matrix computation.

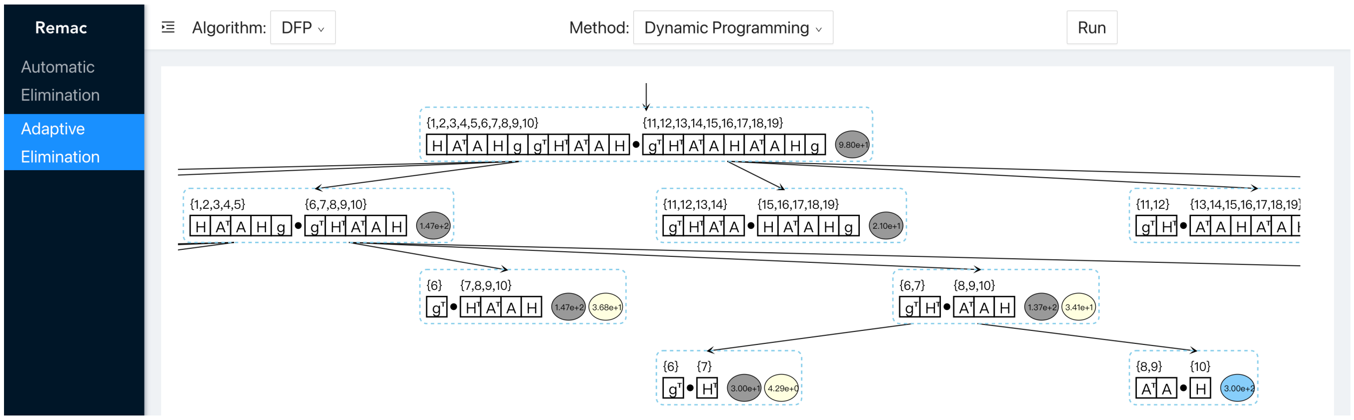
## 3 DEMONSTRATION

In our demonstration, we deploy ReMac on a seven-node cluster, where 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 particular, we implement a GUI to visualize the procedure of automatic elimination, i.e., the block-wise search, and adaptive elimination, i.e., the dynamic programming-based method. There are three algorithms provided in the GUI, including 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.

### 3.1 Automatic Elimination

Attendees will initially see a UI for the "Automatic Elimination" as depicted in Figure 4. On top of the UI, attendees will choose among three algorithms: GD, DFP, and BFGS, and the method to search for elimination options. Particularly, in addition to our block-wise search, the demonstration will have the tree-wise search as a baseline. To perform the demonstration, attendees will click on the "run" button. For the block-wise search, the UI will show how an expression is split into blocks along with their coordinates, and the hash tables recording the search results. Here, the keys in the tables indicates common or loop-constant subexpressions. Attendees can tell the genesis of the redundancy in the expression via the values, i.e., coordinates of matrices, recorded in the table. Once the search is completed, the UI records the search time at the bottom of the UI so that, after running both two search methods, attendees can distinguish among the differences in the performance. Due to space limitations, we do not include this part in Figure 4.

For example, the block-wise search splits the expression of DFP into blocks (e.g., $HA^TAHgg^THA^TAH$) and multiple common subexpressions and loop-constant subexpressions. Eventually, ReMac finds 1391 CSE options as well as 31 LSE options.



Algorithm: DFP

Method: Dynamic Programming

**Figure 5: Dynamic Programming-based Method of Combining Options**

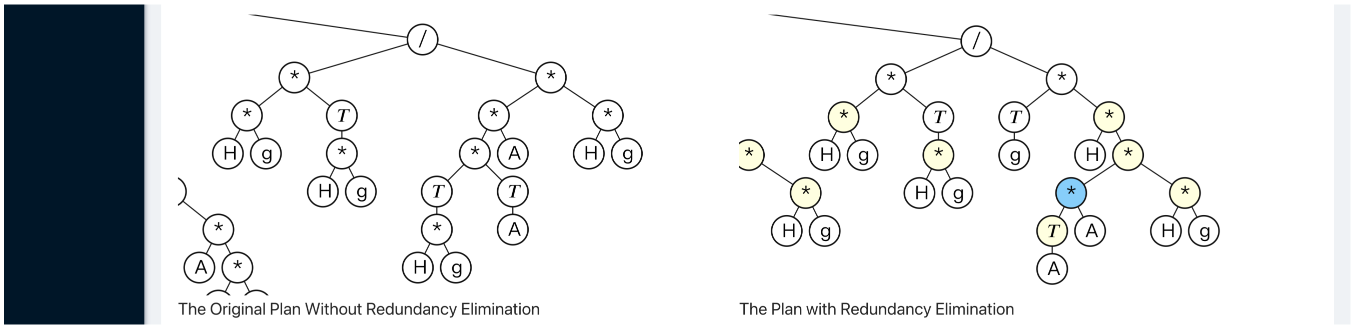

The Original Plan Without Redundancy Elimination
The Plan with Redundancy Elimination

**Figure 6: Execution Plan**

## 3.2 Adaptive Elimination

Upon clicking on the left hand side, attendees will see a UI for the "Adaptive Elimination", which provides insight on how to combine elimination options efficiently. Attendees will pick an algorithm and a combining method, either our dynamic programming-based method or enumeration. For the dynamic programming-based method, they will experience the procedure of building and probing a cost graph. Initially, the UI demonstrates a empty graph. By gradually raising the step number underneath the graph, attendees will see the cost graph growing. This demonstrate how ReMac collates candidate execution plans with different elimination options into the cost graph, i.e., the building phase. Afterwards, at some point, the cost graph will begin to narrow down, meaning ReMac enters the probing phase. In this phase, attendees will see how ReMac prunes the cost graph into a tree step by step.

As demonstrated in Figure 5, the cost graph involves the topologies of candidate plans, where each dotted box represents an operator. Also, the cost graph maintains the evaluation results of the cost model. That is, the ellipses in dotted boxes represent the costs of operators. Particularly, the yellow and blue ellipses are the costs related to CSE and LSE options, respectively.

After combining the found options, ReMac will eventually generates the final execution plan. Accordingly, attendees will see two execution plans underneath the cost graph. One plan is optimized with no redundancy elimination, and the other plan applies the efficient combination of elimination options where the yellow and blue vertices indicates the eliminated redundancy. For example, by comparing the two plans of DFP demonstrated in Figure 6, attendees can see ReMac eliminates a loop-constant subexpression $\boldsymbol{A}^T\boldsymbol{A}$ and multiple constant subexpressions such as $\boldsymbol{Hg}$.

Finally, to showcase the performance impact of adaptive elimination, the UI will record both the compilation time (that generates the final execution plan with redundancy elimination) and the execution time (that completes the algorithm) after each running.

## ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (No. 61902128), and the ECNU-OceanBase Joint Research Lab.

## REFERENCES

[1] ScaLAPACK. http://www.netlib.org/scalapack/.
[2] Matthias Boehm et al. 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).*
[3] Reza Bosagh Zadeh et al. 2016. Matrix Computations and Optimization in Apache Spark. In *SIGKDD*. 31–38.
[4] Paul G. Brown. 2010. Overview of SciDB: Large Scale Array Storage, Processing and Analysis. In *SIGMOD*. 963–968.
[5] Zihao Chen et al. 2022. Redundancy Elimination in Distributed Matrix Computation. In *SIGMOD*. 573–586.