HyMAC: A Hybrid Matrix Computation System

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

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

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

Juan Soto
DIMA, Technische Universität Berlin
juan.soto@tu-berlin.de

Volker Markl
DIMA, Technische Universität Berlin
volker.markl@tu-berlin.de

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

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

ABSTRACT

Distributed matrix computation is common in large-scale data processing and machine learning applications. Iterative-convergent algorithms involving matrix computation share a common property: parameters converge non-uniformly. This property can be exploited to avoid redundant computation via *incremental evaluation*. Unfortunately, existing systems that support distributed matrix computation, like SystemML, do not employ incremental evaluation. Moreover, incremental evaluation does not always outperform classical matrix computation, which we refer to as a *full evaluation*. To leverage the benefit of increments, we propose a new system called *HyMAC*, which performs *hybrid plans* to balance the trade-off between full and incremental evaluation at each iteration. In this demonstration, attendees will have an opportunity to experience the effect that full, incremental, and hybrid plans have on iterative algorithms.

PVLDB Reference Format:
Zihao Chen, Zhizhen Xu, Chen Xu, Juan Soto, Volker Markl, Weining Qian, and Aoying Zhou. HyMAC: A Hybrid Matrix Computation System. PVLDB, 14(12): 2699 - 2702, 2021.
doi:10.14778/3476311.3476323

1 INTRODUCTION

Matrix computation is pervasive in data science, machine learning, and statistical science. As datasets grow in size, matrix computation should preferably be performed in a distributed computing environment using systems, such as SystemML [2], TensorFlow [1], and Spark MLlib [4]. Nonetheless, there are other strategies that may be employed to further improve the performance. Notably, iterative algorithms over matrices often exhibit the behavior that elements converge at different rates. Clearly, it is unnecessary to iterate over those elements that have already converged. Instead, we should leverage incremental evaluation, to exploit sparse computational dependencies and accelerate the execution of iterative algorithms.

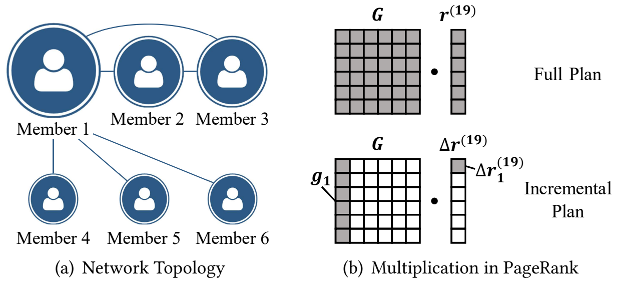


Figure 1: PageRank on Network *G*

To ease our discussion and illustrate incremental evaluation, consider the problem of determining the ranks of members in a tiny community network, as depicted in Figure 1(a). Under a full evaluation, the ranks $\boldsymbol{r}$ are computed iteratively as follows: $\boldsymbol{r}^{(k+1)} = d \times \boldsymbol{G} \cdot \boldsymbol{r}^{(k)} + \boldsymbol{\beta}$, where G is an $n \times n$ stochastic matrix of the network, and d and $\boldsymbol{\beta}$ are constant parameters. In the case of Figure 1(a), after nineteen iterations we observe that the only difference between $\boldsymbol{r}^{(18)}$ and $\boldsymbol{r}^{(19)}$ is the rank of the first member. Hence, there is no need to recompute the other ranks. Instead, we can re-express the formula as $\boldsymbol{r}^{(k+1)} = d \times \boldsymbol{G} \cdot (\boldsymbol{r}^{(k)} - \boldsymbol{r}^{(k-1)}) + \boldsymbol{r}^{(k)}$. In our example, $\boldsymbol{r}^{(19)} - \boldsymbol{r}^{(18)} = (0.03, 0, 0, 0, 0, 0)^T$ denoted by $\Delta \boldsymbol{r}^{(19)}$. Clearly, evaluating $\boldsymbol{G} \cdot \Delta \boldsymbol{r}^{(19)}$ can be simplified to calculate $\boldsymbol{g}_1 \times \Delta r_1^{(19)}$, as depicted in Figure 1(b). In this new form, we both optimize computation efficiency and, more importantly, reduce the communication costs due to performing multiplication in distributed environments.

In our recent work [3], we have found that incremental evaluation does not always outperform full evaluation. In fact, it can decrease the performance, due to the additional operations that are imposed (i.e., adding an increment term). In our experiments, we have found that incremental evaluation is typically detrimental when the incrementally updated term is a matrix rather than a vector. We aim to overcome these limitations by interleaving full and incremental evaluation.

We have developed a novel system called HyMAC [3] built atop SystemML, which implements *hybrid plans*, fully exploits both full

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

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476323

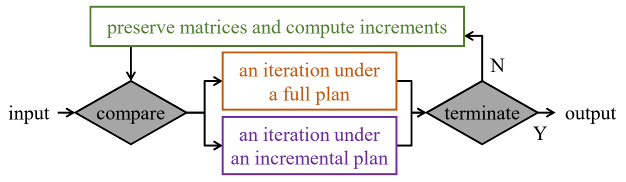

preserve matrices and compute increments
input
compare
an iteration under a full plan
an iteration under an incremental plan
terminate
N
Y
output

Figure 2: The Process of a Hybrid Plan

and incremental evaluations, and accelerates iterative computations over matrices. For each iteration, a hybrid plan chooses either a full plan or an incremental plan to execute. If the majority of the elements in the increments are zero, then a hybrid plan will revert to an incremental plan to reduce the computation costs. However, the elements in a distributed matrix are physically organized into blocks. Hence, the overall cost of an incremental plan might be comparable to the cost of a full plan.

To address this, we optimize the physical layout of matrices via *matrix reorganization*. Hybrid plans rely on the statistics of increments to decide which iteration to execute. However, obtaining these statistics requires the materialization of increments and introduces overhead. To avoid this, we propose an optimization approach called *selective comparison*. In our earlier experiments, we determined that HyMAC outperforms SystemML by 23%.

In this demonstration, we showcase three aspects: (1) how *hybrid plans* exploit both full and incremental plans to accelerate distributed matrix computation, (2) how *matrix reorganization* leverages increments to reduce communication costs, and (3) how *selective comparison* balances the benefits of incremental evaluation and the comparison overhead.

2 HYMAC OVERVIEW

In this section, we briefly introduce the goals of HyMAC, and explain the design details.

2.1 System Goals

Incremental plans work reasonably well when most of the elements do not change, whereas full plans typically outperform incremental plans when the majority of elements change. Based on these observations, it is natural to employ a hybrid (i.e., an optimal) plan, since it can dynamically choose between full and incremental plans for each iteration. As depicted in Figure 2, for an iterative algorithm, our hybrid plan repeatedly compares the performance of full and incremental plans for the next iteration until the algorithm terminates. By doing so, the hybrid plan will outperform both full and incremental plans over all of the iterations.

2.2 System Design

HyMAC is designed to run on a cluster, where there are two types of nodes: a master and a worker (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 HyMAC, the master node is comprised of three components: a *compiler*, an *optimizer*, and a *runtime*. Given a script, the compiler will produce the corresponding full, incremental, and hybrid plans. The role of the optimizer is to optimize each plan type. Finally, the

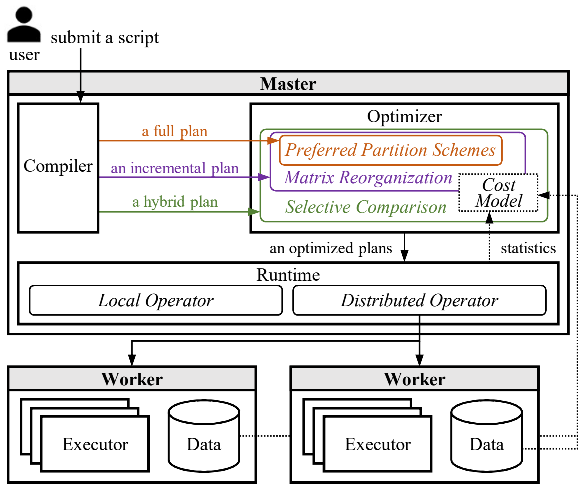


Figure 3: System Architecture

runtime will execute the hybrid plan in either local or distributed mode. Next, we dive into each of these three components.

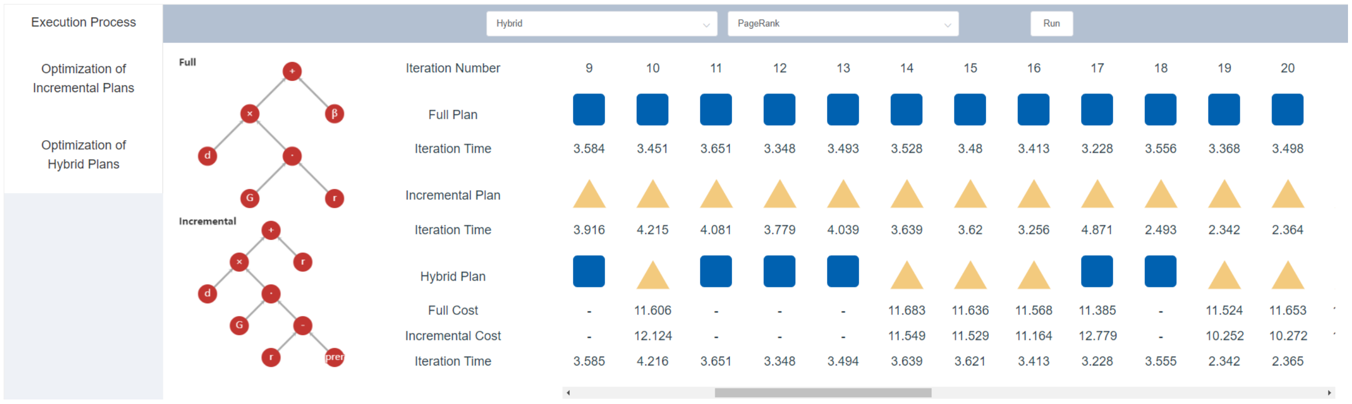
Compiler. The compiler will initially parse a script and create a full plan. Subsequently, it will transform the full plan to an incremental plan, to more easily exploit incremental evaluation.

Due to the fact that incremental plans do not always outperform full plans and vice versa, the compiler will combine the two types of plans to generate a hybrid plan, as depicted in Figure 2. To determine which plan is faster (at a given iteration), we require auxiliary information, i.e., the number of non-zero increments, in order to analyze the execution costs. Consequently, we store matrices and compute increments after each iteration.

Optimizer. The optimizer is concerned with the optimization of each of the plan types. For the full plan, the optimizer employs *preferred partition schemes*, a state-of-art technique proposed in DMac [5], to speedup distributed matrix multiplication.

Similarly, for the incremental plan, the optimizer also employs preferred partition schemes. However, a key difference is the use of increments to reduce computational costs. Logically, this occurs when the majority of the elements in a vector or matrix increment term are zero. Unfortunately, matrix elements are physically organized into blocks. Thus, the overall costs due to the incremental and full plans might be the same, when the non-zero elements are scattered across blocks. In particular, incremental plans may fail to reduce the overwhelming communication costs (associated with distributed matrix multiplication), when they are not optimized.

To address this issue, we propose *matrix reorganization*. The main idea is to gather the increments into as few blocks as possible. For $G \cdot \Delta \boldsymbol{r}^{(k)}$, we first extract the rows containing increments from $\Delta \boldsymbol{r}^{(k)}$, and then permute the corresponding columns of G. Since the permutation of the large matrix G may be costly, we utilize the optimization provided by the preferred partition schemes to achieve an efficient permutation. As a consequence, we can reduce the communication cost associated with the increment involved in multiplication ($G \cdot \Delta \boldsymbol{r}^{(k)}$), and thus fully exploit the benefit of incremental evaluation.



Iteration Number	9	10	11	12	13	14	15	16	17	18	19	20
Full Plan	■	■	■	■	■	■	■	■	■	■	■	■
Iteration Time	3.584	3.451	3.651	3.348	3.493	3.528	3.48	3.413	3.228	3.556	3.368	3.498
Incremental Plan	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲
Iteration Time	3.916	4.215	4.081	3.779	4.039	3.639	3.62	3.256	4.871	2.493	2.342	2.364
Hybrid Plan	■	▲	■	■	■	▲	▲	▲	■	■	▲	▲
Full Cost	-	11.606	-	-	-	11.683	11.636	11.568	11.385	-	11.524	11.653
Incremental Cost	-	12.124	-	-	-	11.549	11.529	11.164	12.779	-	10.252	10.272
Iteration Time	3.585	4.216	3.651	3.348	3.494	3.639	3.621	3.413	3.228	3.555	2.342	2.365

Figure 4: Execution Process

For the hybrid plan, the optimizer seeks to optimize the cost-based comparison. Since we have to store matrices and compute increments for cost evaluation, the comparison itself is costly. Hence, it is infeasible to compare a full plan versus an incremental plan at each iteration. A naive solution would be to require users to specify a static step φ, and then HyMAC would *selectively* perform a *comparison* every φ iterations. However, a short step leads to a large overhead due to frequent comparisons, whereas a long step does not maximize the benefits of incremental evaluation. As an alternative, we propose to dynamically adjust the step at runtime. First, we estimate the costs of full plans, incremental plans, and the comparison via matrix statistics, respectively. Then, based on the estimates, we decide whether applying the comparison is worthwhile, and let the step sequence gradually converge to a finite value and thereby adapt to different cases. However, in general, the dynamic step for selective comparison is initially large and then is gradually reduced to 1. In this manner, sidestepping the need to perform unnecessary comparisons at the beginning, and exploiting incremental evaluation from the middle iterations.

Runtime. The runtime component drives the optimized execution plan fed by the optimizer. An execution plan consists of two kinds of operators: local operators (that fit in memory on a single machine) and distributed operators (that are performed in a distributed environment). For distributed operators, the runtime component drives the executors in the workers to perform distributed matrix computation. Besides executing operators, the runtime component collects matrix statistics (e.g., the number of non-zero increments) for the cost-based comparison employed by the optimizer.

3 DEMONSTRATION

In our demonstration, we deploy HyMAC 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 execution process of three different plans, i.e., full, incremental and hybrid plan, on four algorithms, including PageRank (PR), Hyperlink-Induced Topic Search (HITS), Alternating Least Squares (ALS), and Gaussian Non-Negative Matrix Factorization (GNMF). In addition, we demonstrate the optimization of incremental and hybrid plans.

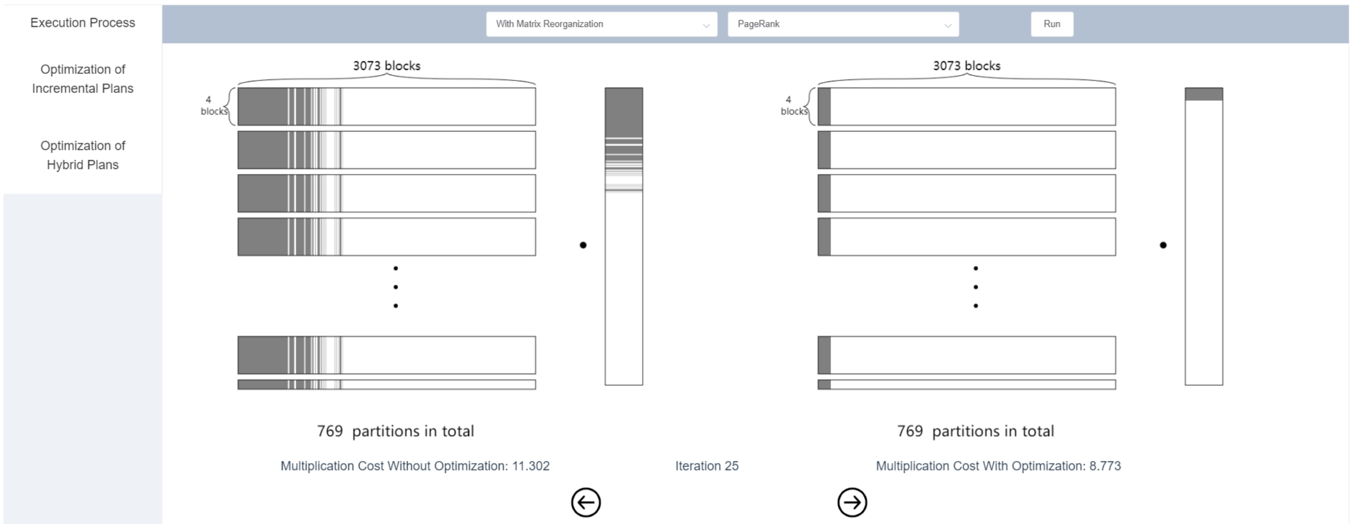
3.1 Execution Process

Attendees will initially see a UI for the "Execution Process" as depicted in Figure 4. Users will choose among four algorithms: PR (PageRank), HITS (Hyperlinked-Induced Topic Search), ALS (Alternating Least Squares), and GNMF (Gaussian Non-Negative Matrix Factorization). The UI includes the plan tree of one iteration for both full and incremental evaluation, respectively, as well as depicting the execution process (represented as a blue square or a yellow triangle) for the full, incremental and hybrid plans, accordingly. To execute a particular plan, a user would click on the "run" button. This enables them to visualize the plan tree and the execution time per iteration, corresponding to their chosen plan. For hybrid plans, they can also observe the cost of either full plans or incremental plans. As depicted in Figure 4, the hybrid plan chooses the plan tree with the lower cost. Once the execution of a certain plan is completed, the UI records the entire execution time at the bottom of the UI so that, after running all of the plans, attendees can distinguish among the differences in the performance. Due to space limitations, we do not include this part in Figure 4.

Figure 4 illustrates the execution process for PR. In the figure we can see the moment, i.e., iteration, when HyMAC detects that the incremental plan outperforms the full plan, and switches to the iteration under the incremental plan (e.g., the 19th iteration).

3.2 Optimization of Incremental Plans

Upon clicking on the left hand side, attendees will see a UI for the "Optimization of Incremental Plans," which provides insight into the matrix reorganization, as depicted in Figure 5. The left and right hand sides of the UI reflect the increment involved in multiplication ($G \cdot \Delta \boldsymbol{r}^{(k)}$) with and without matrix reorganization, respectively. Specifically, the UI demonstrates the dispersion of increments in matrix multiplication and the corresponding costs at different iterations. Here, the black bordered square represents a partition of a distributed matrix, and the gray area represents the portion corresponding to the non-zero increments, i.e., the blocks involved in the multiplication. In addition, the UI records the execution time of hybrid plans with and without optimization for attendees to compare.

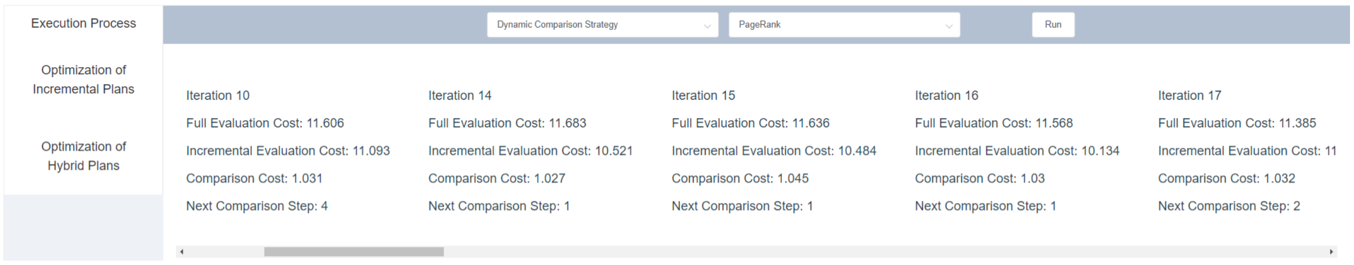


Multiplication Cost Without Optimization: 11.302

Iteration 25

Multiplication Cost With Optimization: 8.773

Figure 5: Optimization of Incremental Plans


Execution Process
Optimization of Incremental Plans
Optimization of Hybrid Plans
Dynamic Comparison Strategy
PageRank
Run

Iteration 10
Full Evaluation Cost: 11.606
Incremental Evaluation Cost: 11.093
Comparison Cost: 1.031
Next Comparison Step: 4

Iteration 14
Full Evaluation Cost: 11.683
Incremental Evaluation Cost: 10.521
Comparison Cost: 1.027
Next Comparison Step: 1

Iteration 15
Full Evaluation Cost: 11.636
Incremental Evaluation Cost: 10.484
Comparison Cost: 1.045
Next Comparison Step: 1

Iteration 16
Full Evaluation Cost: 11.568
Incremental Evaluation Cost: 10.134
Comparison Cost: 1.03
Next Comparison Step: 1

Iteration 17
Full Evaluation Cost: 11.385
Incremental Evaluation Cost: 11
Comparison Cost: 1.032
Next Comparison Step: 2

Figure 6: Optimization of Hybrid Plans

For example, Figure 5 illustrates PR at certain iterations. At the 25th iteration, matrix reorganization will extremely reduce the number of blocks participating in the computation and thus the cost of multiplication. Consequently, our optimization accelerates the execution by reducing the involved matrix blocks.

3.3 Optimization of Hybrid Plans

Upon clicking on the left hand side, attendees will see a UI for the "Optimization of Hybrid Plans," as depicted in Figure 6. Users will pick an algorithm and a comparison strategy that is either static or dynamic. Then, they will experience the process of selective comparison. The UI will show the iteration number and the next step for each comparison. Since the cost of full plans, incremental plans, and the comparison is essential to the dynamic step strategy, the UI also shows these values. Finally, the UI records the execution time of the static strategy and the dynamic strategy to demonstrate the effect of selective comparison.

For ALS, shown in Figure 6, the cost of the full plan is lower than the overall cost of the incremental plan and the comparison at the 14th iteration. Since it is unlikely that the incremental plan will become immediately faster than the full plan, the dynamic strategy sets the step to four to avoid further comparisons. As the incremental plan becomes more efficient, the dynamic strategy reduces the step to one, to utilize incremental evaluation.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (No. 61902128), Shanghai Sailing Program (No. 19YF1414200), and the German Federal Ministry of Education and Research (BMBF) as BIFOLD (funding mark 01IS18025A).

REFERENCES

[1] Martín Abadi et al. 2016. TensorFlow: A System for Large-scale Machine Learning. In *OSDI*. 265–283.
[2] Matthias Boehm et al. 2016. SystemML: Declarative Machine Learning on Spark. *PVLDB* 9, 13 (2016), 1425–1436.
[3] Zihao Chen, Chen Xu, Juan Soto, Volker Markl, Weining Qian, and Aoying Zhou. 2021. Hybrid Evaluation for Distributed Iterative Matrix Computation. In *SIGMOD*. 300–312.
[4] Xiangrui Meng et al. 2016. MLlib: Machine Learning in Apache Spark. *J. Mach. Learn. Res.* 17, 1 (2016), 1235–1241.
[5] Lele Yu et al. 2015. Exploiting Matrix Dependency for Efficient Distributed Matrix Computation. In *SIGMOD*. 93–105.