PreVision: An Out-of-Core Matrix Computation System with Optimal Buffer Replacement

KYOSEUNG KOO, Seoul National University, Korea
SOHYUN KIM, Seoul National University, Korea
WONHYEON KIM, Seoul National University, Korea
YOOJIN CHOI, Seoul National University, Korea
JUHEE HAN, Seoul National University, Korea
BOGYEONG KIM, Seoul National University, Korea
BONGKI MOON, Seoul National University, Korea

Large-scale matrix computations have become indispensable in artificial intelligence and scientific applications. It is of paramount importance to efficiently perform out-of-core computations that often entail an excessive amount of disk I/O. Unfortunately, however, most existing systems do not focus on disk I/O aspects and are vulnerable to performance degradation when the scale of input matrices and intermediate data grows large. To address this problem, we present a new out-of-core matrix computation system called *PreVision*. The *PreVision* system can achieve optimal buffer replacement by leveraging the deterministic characteristics of data access patterns, and it can also avoid redundant I/O operations by proactively evicting the pages that are no longer referenced. Through extensive evaluations, we demonstrate that *PreVision* outperforms the existing out-of-core matrix computation systems and significantly reduces disk I/O operations.

CCS Concepts: • **Information systems** → **Record and buffer management**; **Database query processing**.

Additional Key Words and Phrases: future log of references, array, matrix computation

ACM Reference Format:
Kyoseung Koo, Sohyun Kim, Wonhyeon Kim, Yoojin Choi, Juhee Han, Bogyeong Kim, and Bongki Moon. 2024. PreVision: An Out-of-Core Matrix Computation System with Optimal Buffer Replacement. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 42 (February 2024), 25 pages. https://doi.org/10.1145/3639297

1 INTRODUCTION

The ability to handle large-scale matrix computations becomes increasingly crucial for artificial intelligence and scientific applications. Many solutions have already been proposed for matrix computations, which range from easy-to-use packages to sophisticated distributed systems. While easy-to-use packages such as NumPy [16] and SciPy [44] are widely used for simple computational tasks on a standalone workstation, distributed systems such as SystemDS [9] and SciDB [39] focus mainly on parallelizing large-scale workloads over many computing nodes.

Large-scale matrix computation tasks such as PageRank [30] may suffer from an excessive amount of disk I/O operations, when large input matrices should be loaded into memory by each update iteration. The problem can become dire if a large quantity of intermediate data is produced,

Authors' addresses: Kyoseung Koo, koo@dbs.snu.ac.kr, Seoul National University, Seoul, Korea; Sohyun Kim, chloek409@dbs.snu.ac.kr, Seoul National University, Seoul, Korea; Wonhyeon Kim, dnjsgus@dbs.snu.ac.kr, Seoul National University, Seoul, Korea; Yoojin Choi, cyj@dbs.snu.ac.kr, Seoul National University, Seoul, Korea; Juhee Han, juheehan@dbs.snu.ac.kr, Seoul National University, Seoul, Korea; Bogyeong Kim, bgkim@dbs.snu.ac.kr, Seoul National University, Seoul, Korea; Bongki Moon, bkmoon@snu.ac.kr, Seoul National University, Seoul, Korea.

ACM 2836-6573/2024/2-ART42
https://doi.org/10.1145/3639297

spilled out to disk, and reloaded to memory for further processing repeatedly. A vicious pattern like this happens more often than not in matrix computations (*e.g.*, non-negative matrix factorization (NMF) [23]). Most of the existing distributed systems, however, do not address disk I/O challenges and are hardly optimized for out-of-core matrix computations, let alone those easy-to-use packages with limited support for out-of-core computations. For example, SystemDS and MLlib [28] rely on Spark [46], which spills data produced from a stage to disk for the next stage. NumPy, one of the popular easy-to-use packages, uses a naive data layout to store matrices on disk, contributing to inefficient data access for computations. Disk I/O is still a costly operation and accounts for a large portion of processing time for matrix computation tasks.

A new trend in query planning further aggravates the disk I/O problem. Traditionally, relational database management systems adopt a tree-structured query plan for its simplicity of expressing a relational algebra query. On the other hand, queries are now modeled as a directed acyclic graph (DAG) more often than before particularly for tasks requiring multiple accesses to the same data source [9, 46]. In a DAG query plan, the output produced by an operator can be fed into one or more operators as input. Consequently, the same output may have to be reproduced repeatedly for multiple consuming operators. A common approach to avoiding such redundant computation is to materialize the output data at least partially. For instance, Spark provides a function `persist()` to avoid the repeated computation of a Resilient Distributed Dataset (RDD). The materialized data will inevitably add to the working set of a DAG query plan, which lowers the buffer hit ratio and increases disk I/O operations.

Buffer replacement algorithms have been studied extensively for the past decades because they play a critical role in reducing disk I/O operations [21, 26, 27, 29]. Among those, an optimal buffer replacement algorithm OPT [26] is known to be infeasible to implement in most practical scenarios due to its reliance on the knowledge of future events. When it comes to an out-of-core matrix computation, however, it is not impossible to *accurately predict* the entire sequence of data chunks to be accessed. Consider a large disk-resident matrix stored as a group of tiles of varying shapes and sizes [38]. With those tiles as the unit of paging between memory and disk, the entire sequence of tile accesses can be predicted at the planning stage of a given query by taking advantage of the deterministic nature of matrix computation algorithms [42].

This paper presents *PreVision*, a data processing system designed for large out-of-core matrix computations. For a given matrix computation task modeled as a DAG query plan, *PreVision* can make use of the optimal buffer replacement algorithm by leveraging the entire page access pattern that can be accurately predicted for the DAG query plan. Besides, *PreVision* takes a proactive approach to buffer replacement by *preemptively* evicting data no longer referenced in the future. By immediately evicting data once they become known to be referenced no longer, *PreVision* can effectively increase the utilization and the hit ratio of the buffer pool.

The contributions of this work are as follows:

- We have developed an optimal buffer replacement algorithm for a DAG-structured query plan. Optimal buffer replacement becomes feasible at the nominal cost of run-time analysis that produces the *future log* of references for a given DAG query plan.
- We propose a *preemptive eviction* strategy that eagerly victimizes data no longer referenced. The future log is used to detect such data so that they can be evicted from the buffer pool immediately.
- We present the implementation details of *PreVision* focusing on the buffer management issues such as variable-length page allocation, a few design decisions for the buffer manager, and a disk-resident file format.

- We demonstrate experimentally that *PreVision* outperforms existing open-source solutions for large-scale matrix computation tasks. The experimental results corroborate the efficacy of the I/O mitigation methods of *PreVision*.

This paper is organized as follows. In Section 2, we describe how a DAG query plan is executed in detail and introduce the future log of references that can be created by analyzing a given query plan. Section 3 and Section 4, respectively, depict how the OPT buffer replacement algorithm can be applied and how data with no future reference can be evicted preemptively using the future log. The implementation details of *PreVision* are presented in Section 5, and the extensive evaluation with detailed analysis is presented in Section 6. The previous studies related to this paper are summarized in Section 7. Lastly, we conclude this work in Section 8.

2 LOOKING INTO THE FUTURE

In a matrix computation, the shapes of intermediate and output matrices can be predicted accurately. Besides, the block matrix algorithms [42] generally have deterministic data access patterns. Consider for example a matrix multiplication $M \times N$ that is performed by a block matrix multiplication algorithm. The shape of the output matrix is immediately determined by the number of rows in matrix M and the number of columns in matrix N. The matrix multiplication algorithm accesses the row tiles of the matrix M and the column tiles of the matrix N to produce output tiles. The order of tile pairs to multiply is fixed, and the entire sequence of tile accesses by this algorithm can be predicted accurately. In comparison with a relational join operation, the data access pattern by a block matrix algorithm can be more deterministic for input, intermediate, and output matrices.

A matrix computation task may have more than one operator in its DAG query plan. By analyzing the dependency among the operators in the query plan, *PreVision* selects an order of operators that will be carried out by the executor. By leveraging the fact that both the tile access pattern by an individual operator and the execution order of the operators are deterministic, the query planner of *PreVision* constructs a tile reference sequence by simulating the query plan. The constructed reference sequence is stored in a *future log* and passed to the buffer manager before the execution of the query plan begins.

This section presents the overview of query execution steps taken by *PreVision* for a matrix computation task (in Section 2.1). It then describes how individual operators in a DAG-structured query plan are executed and how a future log is constructed by simulating a given query plan in detail with concrete examples (in Section 2.2 and Section 2.3).

2.1 Overview of *PreVision*

The overall steps of query processing by *PreVision* is illustrated in Fig. 1. *PreVision* supports the `Array` data type and provides a set of Application Programming Interfaces (APIs) for linear algebra functions. The `Array` data type represents disk-resident arrays and temporary arrays produced by the linear algebra functions. A user task is typically expressed by a series of linear algebra functions, and it is activated by invoking the `compute()` function. The activated user task is submitted to *PreVision* as a query, for which the query planner constructs a DAG-structured query plan. The query planner performs query rewrites if necessary, builds a future log, and passes it to the buffer manager.

The query plan is then executed by the query executor, which works with the buffer manager to access array data and perform computations on the array data. The buffer manager adopts the OPT buffer replacement algorithm by leveraging the future log, enables an $O(1)$ time victim selection when the buffer pool is full, and evicts obsolete data preemptively so that the buffer pool will not

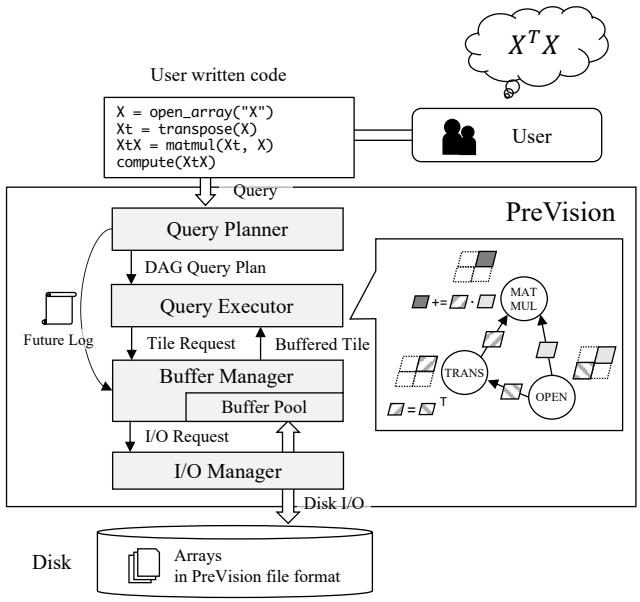


Fig. 1. Overview of *PreVision*

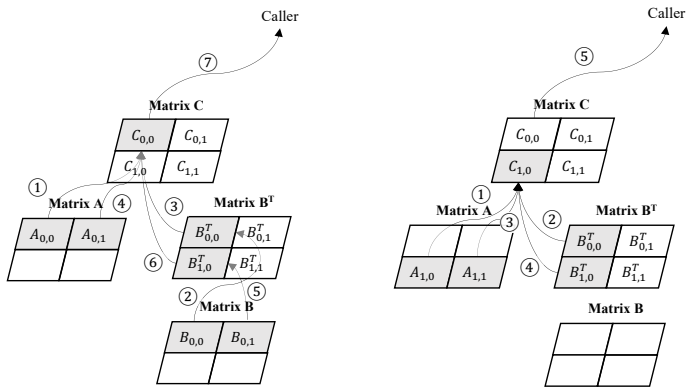
be polluted by them. The I/O manager deals with all read and write operations for disk-resident arrays stored in the file format defined by *PreVision*.

PreVision splits an array into a group of tiles so that it can take block matrix approaches [42] with those tiles to perform matrix computations. The buffer manager treats the tiles as the unit of buffering like pages. Unlike fixed-length pages, however, tiles can be of different shapes and lengths. Tiles of the same shape may differ in size due to different densities of the tiles.

2.2 Executing a DAG Query Plan

In *PreVision*, a query plan is represented as a directed acyclic graph (DAG). A DAG query plan is composed of nodes and edges, and each node represents an operation and each edge indicates the direction of data flow from one node to another. The source and destination of an edge are called a producer node and a consumer node, respectively. A node designated by a user with the `compute()` function is referred to as an output node. For example, in Fig. 1, the query plan of a matrix computation task $X^T X$ involves three nodes, namely `OPEN`, `TRANS`, and `MATMUL`. The `OPEN` node represents an operation that fetches a disk-resident array X. The `TRANS` and `MATMUL` nodes are responsible for a transposition operation and a matrix multiplication operation, respectively. The query plan also involves three directed edges, namely `OPEN` → `TRANS`, `OPEN` → `MATMUL`, and `TRANS` → `MATMUL`. The first two edges indicate that the same producer node `OPEN` feeds two consumer nodes `TRANS` and `MATMUL` with an array it fetches from disk. The `MATMUL` is designated as an output node as it returns the result of the task to the user.

The query executor of *PreVision* creates an iterator for each node in the query plan. Each iterator includes a common function called `getPos()` to fetch a tile. A tile request is made by invoking the `getPos()` function with the *coordinates* of the tile. While a query is being processed, each iterator maintains a Boolean state for each tile it is responsible for to indicate whether the tile has been computed or not. When a request is made for a tile already computed and marked as true, the



(a) Calculating $C_{0,0}$. (b) Calculating $C_{1,0}$.

Fig. 2. The tile access pattern of AB^T for 2×2 tiled matrices

iterator returns the computed tile without redoing the computation. Redundant computations can be avoided this way.

PreVision initiates the processing of a query by requesting tiles from the iterator of the output node in the query plan. The iterator of the output node then sends requests for all the necessary tiles to the iterators of corresponding producer nodes. This operation is propagated recursively down to all the relevant descendent nodes of the query plan.

Example 1. *Fig. 2 illustrates how PreVision performs a matrix computation task $C = AB^T$, where A, B and C are 2×2 tiled matrices. In the figure, each arrow shows the flow of data between nodes and is annotated with a circled number to indicate the order of execution.*

When the first output tile $C_{0,0}$ is requested by the executor of PreVision, the iterator of the output node makes tile requests to initiate a matrix multiplication as the requested tile $C_{0,0}$ is not computed yet. Since $C_{0,0} = A_{0,0}B^T_{0,0} + A_{0,1}B^T_{1,0}$, the iterator first attempts to fetch $A_{0,0}$ and $B^T_{0,0}$. Fetching $A_{0,0}$ and $B^T_{0,0}$ is done by sending a tile request to the iterator of A and sending another tile request to the iterator of B^T. While fetching $A_{0,0}$ can be done immediately by its iterator (①), the iterator of B^T has to send yet another tile request to the iterator of B (②) to initiate a matrix transposition. Upon completion of the matrix transposition, the iterator of B^T returns the $B^T_{0,0}$ tile to the iterator of C (③). The iterator of C then carries out a matrix multiplication with the $A_{0,0}$ and $B^T_{0,0}$ tiles. To perform the next matrix multiplication $A_{0,1}B^T_{1,0}$, the iterator of C requests the $A_{0,1}$ and $B^T_{1,0}$ tiles from its descendent iterators (④, ⑤, and ⑥) in the similar way. Finally, when both $A_{0,0}B^T_{0,0}$ and $A_{0,1}B^T_{1,0}$ become available, the iterator of C computes the $C_{0,0}$ tile and returns it to the user (⑦). All these steps are summarized in Fig. 2a.

The iterator of C repeats the same steps to compute $C_{1,0}$. Note, however, that the iterator of B may not have to be involved in the computation of $C_{1,0}$, because $B^T_{0,0}$ and $B^T_{1,0}$, which are necessary for $C_{1,0}$, have already been computed by the iterator of B^T. The two transposed tiles can simply be passed from it to the iterator of C without making any new request to the iterator of B (② and ④). These steps are summarized in Fig. 2b. □

2.3 Constructing a Future Log

The future log is implemented as a hash table. A hash element is created for each tile referenced by a given query. Each hash element contains (1) the metadata of a tile, (2) a list of reference events,

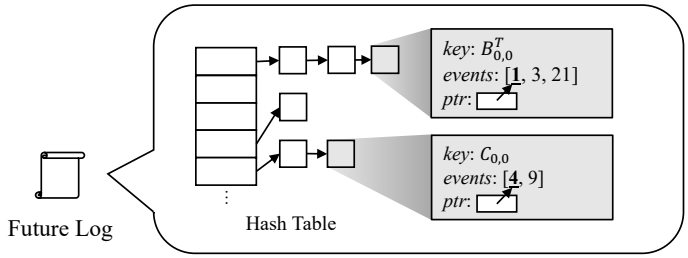


Fig. 3. An example of a future log

and (3) a pointer to the next reference event. The metadata of a tile is used as a hash key in the hash table. Since a tile can be referenced more than once, a list of reference events is stored in the hash element, and each of the reference events is essentially a logical timestamp of the corresponding reference. See Fig. 3 for an example of a future log.

The reference events can be obtained by the query planner of *PreVision* simulating the given query. The query planner initializes a logical clock to zero when it begins traversing the query plan. Following the prescribed execution procedure, when the query planner detects a buffer request for a tile, it appends the current timestamp of the logical clock to the list of events in the corresponding hash element. If the tile has not been referenced before and the hash element is not found in the future log, the planner creates a new hash element and adds it to the future log by using the metadata of the tile as a hash key. It then increments the logical clock by one. Since the logical clock never decrements, the list of events of a tile is sorted in the ascending order of timestamps.

Example 2. *Fig. 3 illustrates the future log entries of the $B^T_{0,0}$ and $C_{0,0}$ tiles described in Example 1. Suppose that the query planner begins simulating the query plan by applying a matrix transposition operation to the $B_{0,0}$ tile. The $B_{0,0}$ tile will be the first one to be requested, which is then followed by an empty tile for $B^T_{0,0}$ that will store the transposed tile. Thus, a hash element for $B_{0,0}$ is created and inserted into the future log with a timestamp zero, and another hash element for $B^T_{0,0}$ is created and inserted into the future log with a timestamp one. When $A_{0,0}$, $B^T_{0,0}$, and $C_{0,0}$ are referenced next by the matrix computation task in Example 1, two new hash elements are created for $A_{0,0}$ and $C_{0,0}$, and three new reference events with timestamps two, three, and four, respectively, are added to their corresponding hash elements in the future log. The simulation continues in this manner adding more hash elements and reference events to the future log until the query plan is traversed completely. Fig. 3 shows just a few reference events of the $B^T_{0,0}$ and $C_{0,0}$ tiles.* □

The future log is usually small enough to fit in memory. For example, when *PreVision* performed a logistic regression (LR) task for a 100×1 tall-skinny tiled matrix with three iterations, the future log was no more than 300 kilobytes in size. (See Section 6 for the details of the task.)

3 OPTIMAL TILE REPLACEMENT

The buffer manager of *PreVision* runs the OPT buffer replacement algorithm with a future log provided by the query planner. The optimal buffer replacement is achieved by evicting a buffer frame for replacement the next reference of which is the farthest into the future. Each tile cached in the buffer pool is associated with the timestamp of the next reference (denoted by *next_ts*). The next reference can be retrieved from the future log.

For the fast selection of a victim, all the cached tiles are maintained in a doubly linked skip list [32] in the increasing order of their *next_ts* values. A victim selection can be done in $O(1)$ time because an unpinned tile with the greatest *next_ts* value is always found at the tail end of the skip

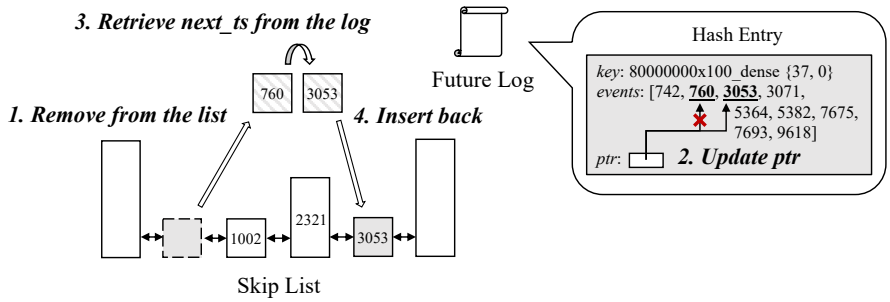


Fig. 4. An example of list update process for a buffer hit

list. In general, if a large number of pinned tiles had to be skipped, the victim selection time could indeed exceed $O(1)$ time. However, this does not happen with *PreVision* which executes only a single operation at a time. The number of pinned tiles is always a constant, which is the number of operand and output tiles for an operator being executed.

The timestamps of future references are determined by a logical clock that ticks once every tile reference. Thus the *next_ts* values of all cached tiles are greater than or equal to the current clock at any moment in time. This also implies that, on a buffer hit, the next tile to reference is always at the head of the skip list. When this tile is actually referenced, it is removed from the skip list and is inserted back with a renewed *next_ts* value. On a buffer miss, the buffer manager loads the referenced tile from disk, sets the *next_ts* value of the tile by accessing the future log, and inserts the tile to the skip list. Fig. 4 illustrates the steps of a list update operation when a buffer hit occurs.

In the future log, there is a hash element for each tile to be referenced. The hash element of a tile stores a list of reference events in chronological order. To determine the *next_ts* value of a tile quickly, the hash element of the title includes ptr pointing to the next reference event. The pointer ptr advances by a slot within the list after each reference. When there is no more event left in the hash element, the pointer ptr is set to NULL, and the *next_ts* is set to infinity.

Whenever a tile is referenced, the buffer manager updates the *next_ts* value for the referenced tile, and the new *next_ts* value can be obtained in $O(1)$ time by following the *ptr* in the event list of the hash element. The buffer manager then inserts the tile into the skip list, which takes time logarithmic to the size of the skip list. Overall, each tile reference requires $O(logN)$ time for maintaining the skip list, where N is the number of tiles in the skip list. Although the overhead may seem high, it is negligible considering the dominant I/O cost required for out-of-core matrix computations and N being typically quite small. In our experiment, N did not exceed 100 while an LR task was carried out. (See Section 6 for details.)

Example 3. *Fig. 4 shows how the skip list and the future log are updated when a buffer hit occurs by a tile reference. The tile reference can be considered a replay of a query simulation preserved in the future log. Thus, for example, when the logical clock of the buffer pool strikes 760, a tile with next_ts value 760 is referenced. Assuming that the tile is cached in the buffer pool, the buffer manager temporarily removes the tile from the skip list and searches for the next reference event from the future log by advancing the ptr in the tile's hash element. Since the next event (or its timestamp) is 3053, the ptr advances from the current event 760 to the next event 3053. Then, the buffer manager updates the next_ts value of the tile to 3053 and inserts the tile back into the skip list. In the case of a buffer miss where the tile with next_ts value 760 does not exist in the skip list, the procedure of removing the tile from the skip list is replaced with the procedure of loading the tile from the disk.* □

Optimality. The OPT buffer replacement algorithm is known to be optimal for a fixed number of fixed-length buffer frames [6]. However, the optimality of the algorithm is not guaranteed when dealing with variable-length buffer frames. Since the tiles of *PreVision* are not of the same length, adopting the OPT algorithm does not always guarantee the optimal buffer replacement. Nonetheless, the OPT algorithm still yields higher buffer hit ratios than LRU-K [29] and Most Recently Used (MRU) algorithms do. The impact of the OPT algorithm will be discussed more in Section 6.4.

Conditional Control Flow. One of the challenges arising in query processing is the potential data dependency. For instance, the termination condition of an iterative algorithm needs to be evaluated every iteration, and the subsequent operations will remain undecided until the evaluation of the termination condition is complete. *PreVision* attempts to break the data dependency by separating a query into two subqueries, one for the computation of prerequisite values and the other for the rest of the computation. While this strategy may lead to an increased query planning overhead, the increased overhead has only a negligible effect on the query performance. We will discuss the performance impact of the query planning in Section 6.4.

Applicability. The proposed method for tile replacement is based on the future log and tailored for processing a single query at a time. Following the sequential execution model described in Section 2, the tile access pattern is predicted at the planning stage of an individual query. If multiple queries are processed concurrently, the optimality of tile replacement is not guaranteed because the chronological order of the reference events of a certain tile is not deterministic. Consequently, a list of reference events saved in the future log may not be the same as the sequence of actual references, and the optimal selection protocol can be violated. Note that, however, out-of-core matrix computations can still take advantage of the optimal tile replacement of *PreVision*, because they are often adopted in a long-running data analytic application administered as a sole task.

4 PREEMPTIVE EVICTION

In a DAG-structured query plan, intermediate data produced by an operator may be consumed by one or more operators. Unless all the consuming operators are perfectly synchronized with a producing operator, some or all intermediate data from the producing operator will have to be materialized and preserved on disk until they are no longer needed. Once the intermediate data are materialized, they can be accessed by any operator. They can be cached in the buffer pool in the same way as the disk-resident matrices are done in the database. In the presence of multiple consuming operators accessing the same intermediate data, it is difficult for the buffer manager to predict when the data will no longer be accessed and need not be cached in the buffer pool. Consequently, the intermediate data tend to stay in the buffer pool longer than they should and lower the buffer hit ratio.

To address this problem of obsolete intermediate data lingering in the buffer pool, *PreVision* adopts a preemptive eviction strategy. As tiles are the unit of paging, the buffer manager evicts tiles eagerly as soon as it determines they are of no more use. The buffer manager can identify obsolete tiles in the buffer pool by analyzing the reference events stored in the future log. When a tile is referenced by the last event in the future log, the *next_ts* of the tile is set to infinity. When such a tile is unpinned, it can be immediately evicted from the buffer pool (without being flushed to disk).

Example 4. *Consider the computation of* $C_{1,0} = A_{1,0}B^T_{0,0} + A_{1,1}B^T_{1,0}$ *for the matrix computation task described in Example 1. Since the* $B^T_{0,0}$ *and* $B^T_{1,0}$ *tiles are obtained while* $C_{0,0}$ *and* $C_{0,1}$ *are computed, the executor does not need to access the* B *matrix to compute* $C_{1,0}$*. Thus, the computation of* $C_{1,0}$ *only requires four tiles, namely,* $A_{1,0}$*,* $B^T_{0,0}$*,* $A_{1,1}$*, and* $B^T_{1,0}$*. The left and right sides of Fig. 5, respectively, show the future log records of the* $A_{1,0}$*,* $B^T_{0,0}$*, and* $C_{1,0}$ *tiles before and after these tiles are referenced. When*

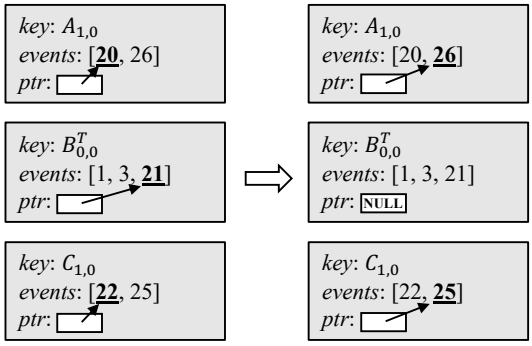


Fig. 5. Changes of *future log* entries after buffer requests

$A_{1,0}$ is referenced at the logical clock 20, the ptr field of its future log record advances to the next slot in the list of reference events so that ptr points to 26 instead of 20. The next_ts value of the tile is also updated from 20 to 26. When $B^T_{0,0}$ is referenced at the logical clock 21, the ptr field of its future log record is set to `NULL` *because there is no more future reference left in the list. Its next_ts value is set to infinity to indicate that $B^T_{0,0}$ will no longer be referenced. Similarly to $A_{1,0}$, when $C_{1,0}$ is referenced at the logical clock 22, its ptr field advances from 22 to 25, and the next_ts value is set to 25.*

The $A_{1,0}$ and $B^T_{0,0}$ tiles are unpinned when the matrix multiplication operation is completed. Since the next_ts value of $B^T_{0,0}$ is infinity, the buffer manager immediately evicts the tile without flushing the content to disk. As a result, $B^T_{0,0}$ causes a disk write only if it is evicted (or materialized) after the transposition of $B_{0,0}$. □

5 IMPLEMENTATION DETAILS

This section presents the rationale behind the design choices and the implementation details of *PreVision*. We begin with memory management in Section 5.1 and then discuss I/O management in Section 5.2.

5.1 Memory Management

5.1.1 Dynamic Memory Allocator. The buffer manager of *PreVision* deals with tiles of varying shapes and sizes, each of which is loaded into a contiguous memory chunk for higher memory bandwidth. The buffer manager relies on its dedicated dynamic memory allocator to manage the system memory effectively for the variable-length memory chunks. The allocator is capable of allocating memory chunks of different sizes and supports a few primitives similar to the standard `malloc()` and `free()` functions for the buffer manager.

The memory allocator sets aside all the memory available to itself at the initialization. On a request from the buffer manager, the memory allocator attempts to find a contiguous memory chunk from its memory pool by applying the best-fit allocation algorithm. The $O(N)$ time complexity of the algorithm is relatively high, but it helps minimize fragmentation by finding the most suitable chunk in the memory pool. Note that the operating system can be entrusted with the task of handling memory fragmentation as well as memory allocation. However, that approach would incur a large number of system call invocations and hence degradation in performance. Thus, we opt for a dedicated memory allocator for the *PreVision* buffer manager.

The memory allocator is tightly integrated with the buffer manager. A tile eviction from the buffer pool is triggered by the memory allocator when a chunk request from the buffer manager

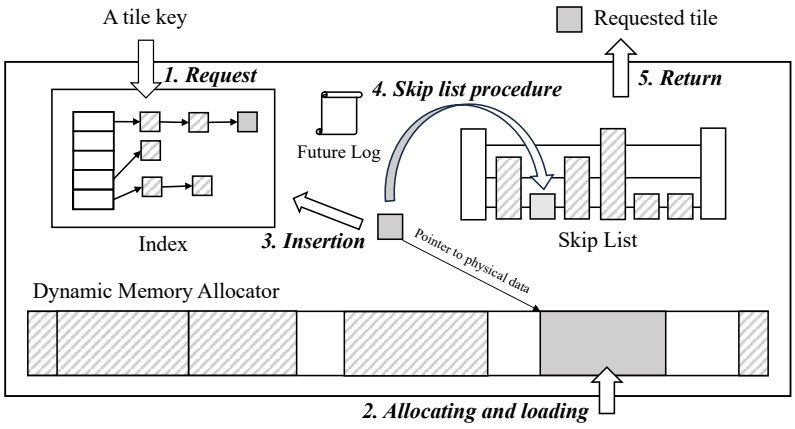


Fig. 6. Processing a tile request on a buffer miss

cannot be fulfilled because there is not enough free space in the memory pool. This eviction process may have to be repeated by the buffer manager until the allocator secures a contiguous memory chunk large enough to accommodate the request.

5.1.2 Unified Buffer Pool. Most database systems manage buffer frames in multiple pools. For example, Oracle keeps cached pages in a few separate buffer pools by the sizes of pages, and SciDB maintains a buffer pool for persistent arrays separately from another buffer pool for temporary arrays [39]. In contrast, *PreVision* uses a single unified buffer pool to manage tiles of all sizes for higher utilization of memory. It is particularly relevant in matrix computation tasks, where the volume of intermediate arrays cached in the buffer pool tends to fluctuate rapidly over time. Consequently, a separate buffer pool for the intermediate arrays would be underutilized, and the overall utilization of memory would also be degraded.

5.1.3 Tile Identification. The buffer manager uses tile keys as a means of identification of tiles. A tile key consists of the name of an array and the coordinates of a tile, and it is included in each tile as metadata. When the buffer manager receives a request for a tile, it can retrieve a future log record of the tile using its tile key. Note that tile keys are created not only for persistent arrays but also for temporary (or intermediate) arrays. The names of temporary arrays are coined internally and used in their tile keys.

5.1.4 Handling a Tile Request. When it receives a tile request, the buffer manager searches the buffer pool index for the tile key to determine whether the requested tile is cached in the buffer pool. If the tile exists in the buffer pool, the buffer manager applies the update procedure to the skip list as described in Section 3. Otherwise, the buffer manager determines whether the request is for a new tile or a persistent copy of the requested tile on disk. In both cases, the buffer manager passes the request to the memory allocator so that an empty memory chunk is allocated for the tile. Additionally, in the latter case, the buffer manager works with the I/O manager to load the disk-resident copy into the empty memory chunk. When the copy of the tile is loaded, the buffer manager adds the memory chunk with loaded data into the buffer pool, updates the buffer pool index, and applies the update procedure to the skip list. Fig. 6 illustrates processing a tile request on a buffer miss.

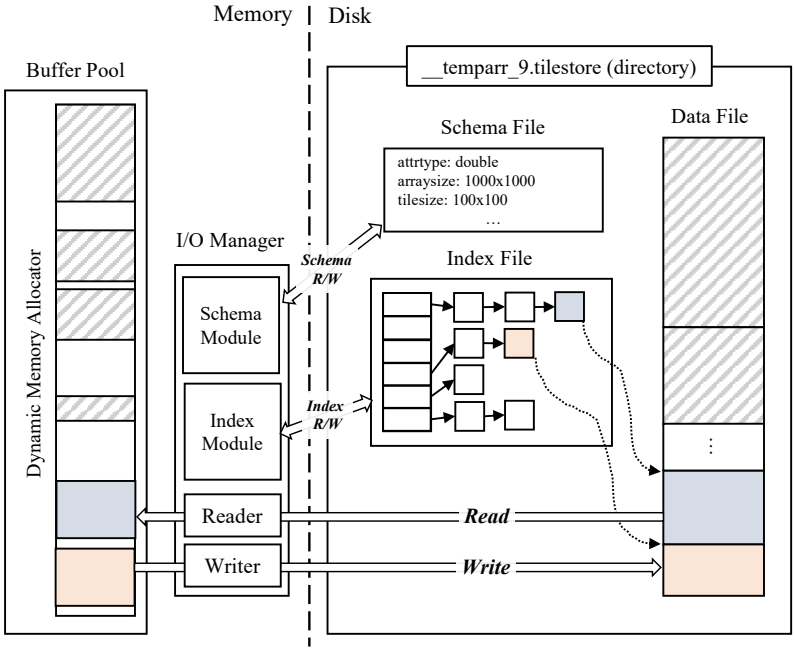


Fig. 7. An overview of the I/O management

5.1.5 Tile Format. Depending on its sparsity, a tile, either memory-resident or disk-resident, is represented by a single vector or a group of three vectors. A dense tile includes a single vector that stores the cell values sequentially in row-major order. A sparse tile includes three vectors, namely, a *data vector*, an *index vector*, and a *pointer vector*, following the compressed sparse row (CSR) format [37]. The data vector stores non-zero cell values, and the index vector stores column indices of these cell values. The pointer vector stores the offsets of rows to the data vector and the index vector.

5.2 I/O Management

5.2.1 Array File Format. A disk-resident array is composed of a *schema file*, a *data file*, and an *index file*. A schema file stores metadata of an array such as the dimension and size of the array itself and the individual tiles in the array. A data file is segmented such that each tile is stored separately and contiguously in a segment. By storing tiles contiguously in segments, *PreVision* can take advantage of sequential disk I/O and achieve higher disk throughput. An index file contains a hash table that stores an element for each tile, and the hash element of a tile contains the offset and length of the corresponding segment. The index file is responsible for retrieving the metadata of a given tile quickly.

The data file is prone to having a bloated size because it may suffer from fragmentation caused by variable-length segments. *PreVision* performs a *file-collapsing* operation to return fragmented disk spaces to the operating system when the fragmentation ratio of a file exceeds a certain threshold. The operation keeps the physical size of an array to an appropriate level without copying the tile data.

5.2.2 I/O Operations. The I/O manager supports simple read, write, and file-collapsing operations that are performed as follows. Fig. 7 shows the overview of the I/O operations as well as the array file format of *PreVision*.

Read. A read operation is initiated by checking the index file to determine whether the requested tile is in the array. If the tile exists in the array, the I/O manager obtains the offset and length of the tile from the metadata stored in the index file and loads the tile data from the segment to the buffer pool.

Write. When a new tile is added to an array, the I/O manager simply appends a new segment created for the new tile to the end of the data file. A new hash element is also created for the new tile and added to the index file. When an existing tile is updated, the I/O manager compares the size of the existing tile and the size of the updated one. If the updated tile is larger than the existing one, then the segment of the existing tile is invalidated and a new segment for the updated tile is appended to the data file. If the updated tile is no larger than the existing one, then the updated tile is overwritten to the existing segment. The index is updated accordingly to reflect the changes. When an existing tile is updated, file fragmentation may occur due to the invalidated segment in the former case and the unused portion of the segment in the latter case. Thus, the file-collapsing operations need to be performed occasionally. Note that the addresses of buffer frames are aligned to 512-byte offsets for all the tiles so that every disk I/O operation can bypass the operating system page cache via Direct IO.

File Collapsing. When the degree of file fragmentation exceeds a certain threshold, the I/O manager initiates a file-collapsing operation. The I/O manager examines the index file to locate the region of fragmentation within each array file. If fragmentation is detected, the fragmented region is returned to the operating system by invoking a tool for allocating space for a file (*e.g.*, `fallocate()` of Linux with the `FALLOC_FL_COLLAPSE_RANGE` flag up). A file-collapsing operation may change the offsets of segments, so the I/O manager updates the index entries of the affected segments if necessary.

6 EVALUATION

To demonstrate the effectiveness of *PreVision*, we conducted extensive experiments that were focused on answering the following questions.

- How well does *PreVision* perform matrix computation workloads in comparison with other solutions? (Section 6.2)
- How effective is the preemptive eviction in reducing disk I/O? (Section 6.3)
- How well does the OPT buffer replacement algorithm work? What is the overhead associated with that? (Section 6.4)

We first outline the workloads and the computing platform used for the evaluation in Section 6.1. Then, we present the evaluation results and address the questions in Section 6.2 through Section 6.4.

6.1 Experimental Settings

To assess the target systems for matrix computation, three representative workloads were chosen: logistic regression (LR) with batch gradient descent [41], non-negative matrix factorization (NMF) [23], and PageRank [30].[1] We used the SLAB linear algebra benchmark [41] for the LR and NMF tasks. We also generated several synthetic datasets for both in-memory and out-of-core computation scenarios to better understand the effect of disk I/O on the overall performance of the target systems. Specifically, all the synthetic matrices were *tall-and-skinny* with 100 columns

[1]The parameters chosen for the tasks are α=0.0000001 for LR, rank=10 for NMF, and damping factor=0.85 for PageRank.

Label	Dimension	Size in GB
10M	10M × 100	8GB
20M	20M × 100	16GB
40M	40M × 100	32GB
80M	80M × 100	64GB

(a) Synthetic Dense

Label	Dimension	Non-zeros	Size in GB
0.0125	400M × 100	500M	11GB
0.025	400M × 100	1B	19GB
0.05	400M × 100	2B	35GB
0.1	400M × 100	4B	67GB

(b) Synthetic Sparse

Label	Dimension	Non-zeros	Size in GB
Enron	37K × 37K	368K	0.006GB
Epinions	76K × 76K	509K	0.009GB
LiveJournal	4.8M × 4.8M	69M	1.1GB
Twitter	62M × 62M	1.5B	24GB

(c) PageRank

Table 1. Dataset Description

System	Spark (MLlib)	SystemDS	PostgreSQL	MADlib	NumPy	Dask	SciDB	OpenBLAS
Version	3.3.2	3.1.0	12.14	1.12.0	1.22.4	2022.6.0	19.11	0.3.0

Table 2. System Versions

containing double-precision values. The number of rows in the dense matrices was one of 10 million, 20 million, 40 million, and 80 million. That is, the dimension of a dense matrix was one of $10^7 \times 100$, $(2 \cdot 10^7) \times 100$, $(4 \cdot 10^7) \times 100$, and $(8 \cdot 10^7) \times 100$. For sparse matrices, the number of rows was fixed to 400 million with varying densities. The population of non-zero cells in the sparse matrices follows a uniform distribution. Table 1a and Table 1b summarize the specifics of the dense and sparse synthetic matrices, respectively.

For the PageRank task, we selected four real-world graph datasets: Enron, Epinions, and LiveJournal from Stanford Network Analysis Project [24], and Twitter [22]. Each dataset is transformed into a sparse adjacency matrix. Table 1c shows the specifics of the sparse matrices. The first three real-world datasets are relatively small, and the target systems can perform the PageRank task with the entire dataset loaded in memory. On the other hand, the Twitter dataset is too large to fit in memory, and the memory consumed by the task exceeds the memory budget slightly, which results in disk spills during computation. Note that the target systems may adopt different metadata or formats for sparse matrices (*e.g.*, Compressed Sparse Column format for MLlib and Coordinate List format for MADlib), so they may differ from one another in the actual memory consumption.

All experiments were carried out with native binary input data for each target system. For the systems using tiled arrays, the SLAB and PageRank matrices were partitioned into 100×1 tiles and 10×10 tiles, respectively. In the case of SciDB, each side of a chunk for each array was configured to 1000 since SciDB did not allow for a size greater than 1024 in the matrix multiplication. When executing the LR and NMF tasks, the same synthetic matrices were fed to the target systems so that interference from using input matrices with different random values could be eliminated.

We used a machine running Ubuntu 18.04.5 equipped with an Intel i7-9700K CPU, 32GB DRAM memory and a 1TB Samsung 860 Pro SSD. We conducted every run for evaluation following the steps below.

(1) Clean the OS page cache and the buffer pools of each target system to ensure a cold start.
(2) Run a task with disk-resident arrays on each target system; input arrays are loaded to memory and part or all of them may be spilled to disk.

(3) Write the output to disk.

Each run was allowed for ten hours and was terminated forcefully at timeout.

6.2 Comparison with Existing Systems

We selected the following six target systems for comparsion with *PreVision*: SystemDS [9], MLlib [28], MADlib [17], SciDB [39], NumPy [16], and Dask [36]. We did not include SciPy [44], Tensorflow [1] and PyTorch [31] because they did not support out-of-core computation. The versions of target systems for evaluation are provided in Table 2. Every system capable of leveraging OpenBLAS routines was configured to make use of those routines. Except for the parallelism evaluation described in Section 6.2.3, each target system was configured to be single-threaded with a single worker for a fair evaluation. The memory budget for each system was configured to 30GB, which was sufficient to keep many tiles simultaneously.

For SystemDS and MLlib, Spark was configured to run with a driver process and a single executor process. We did not isolate JVM processes to a specific core in order to prevent performance degradation caused by garbage collection or just-in-time compilation. We tested a few memory configurations to discover a specific memory budget for each driver and executor that delivers the best performance. PostgreSQL, which MADlib runs on, recommends configuring the shared buffer size to 40% of the total memory [14]. So, we allocated 12GB to the shared buffer and reserved 18GB for MADlib and the OS page cache. We also created indexes for all the tables used by MADlib.

NumPy was set up to utilize memory-mapped arrays to enable out-of-core computation. Since a NumPy operation returns an in-memory array by default, we created a memory-mapped array manually (with the `out` keyword argument) for each intermediate array and returned it as output from each operation.

6.2.1 Varying Data Size. The scalability of the target systems was evaluated with both synthetic and real-world datasets. All three matrix computation tasks are based on iterative algorithms, and the number of iterations was fixed to three in this experiment. This was a very small number of iterations but was still sufficient to observe meaningful differences among the target systems. Sparse NMF tasks were omitted in this experiment because the intermediate matrices of NMF tasks were dense and the dense matrix computation became a dominant factor in the processing time.

Dense LR and NMF. Dense LR and NMF experiments are presented in Fig. 8a and Fig. 8b, respectively. *PreVision* outperformed the other systems in most cases, except for LR results with the 10M and 20M datasets. In such cases, *PreVision* showed slightly slower performance than NumPy. This was due to the small-sized intermediate arrays generated in each iteration of the LR task, which prevented any disk spills. Consequently, *PreVision* did not have a significant advantage over NumPy in this scenario. Meanwhile, *PreVision* incurred some overheads such as future log generation and skip list update, contributing to the slower execution times.

SystemDS presented comparable performances in both the LR and NMF with the 10M and 20M datasets. Although *PreVision* had better performance, both SystemDS and *PreVision* exhibited similar tendencies regarding the matrix size. However, the SystemDS's elapsed time spiked in both LR and NMF tasks with the 40M and 80M datasets. This was due to SystemDS starting to use Spark instruction when the dataset size exceeded the memory budget. The system inserted a checkpoint instruction that makes an RDD for the input matrix persist, which consumed a significant amount of time.

MLlib exhibited poor performance since it caused significant disk I/O. Whenever a Spark stage was completed, MLlib spilled processed data into the disk and loaded spilled data for proceeding with another stage. Dask had slower performance compared to NumPy in LR experiments, while

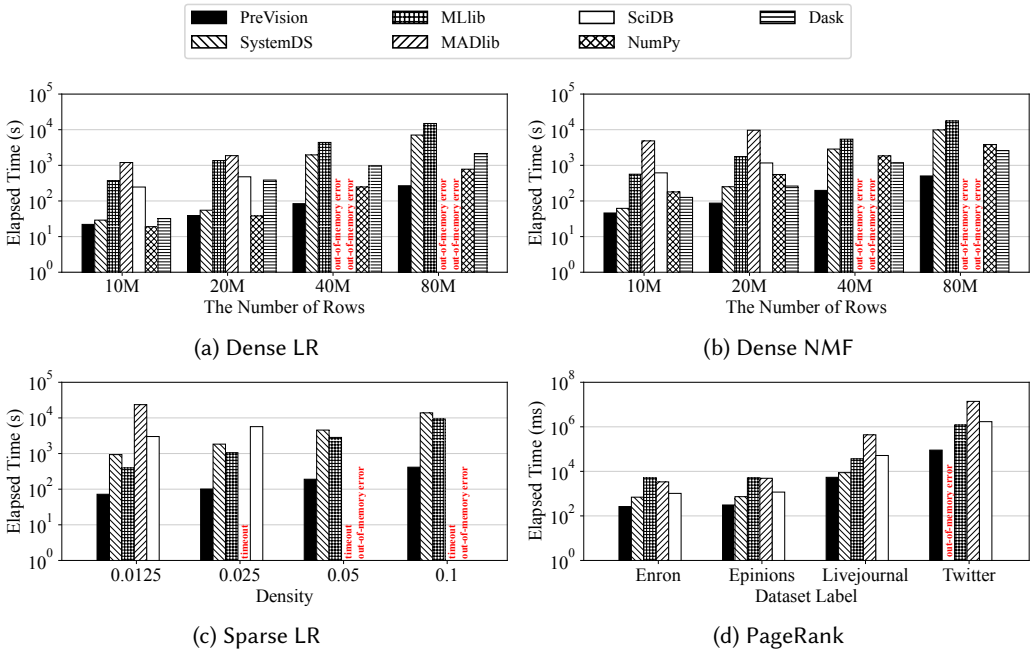
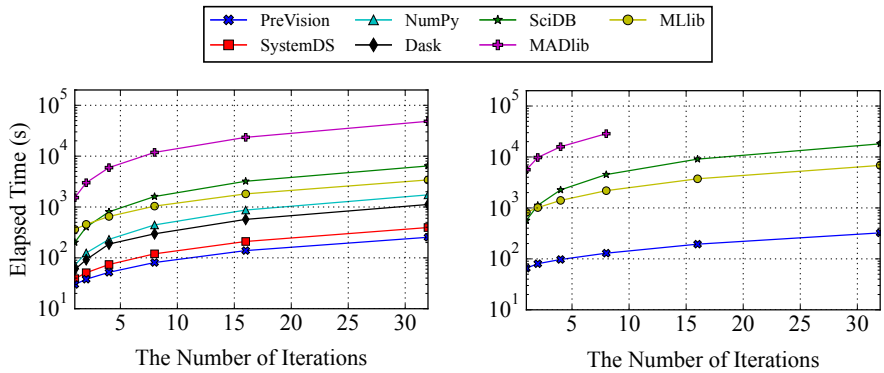


Fig. 8. Performance of the target systems

it slightly outperformed NumPy in NMF experiments. The main reason for this was the Dask scheduling. Dask employed its own scheduling method which tends to use last-used chunks, giving Dask a better buffer hit ratio. Meanwhile, the NMF tasks caused a huge volume of intermediate data. The better hit ratio in these experiments saved a huge volume of disk I/O for intermediate data. On the other hand, Dask showed worse performances in LR experiments because of its scheduling overhead as well as much smaller volumes of intermediate data.

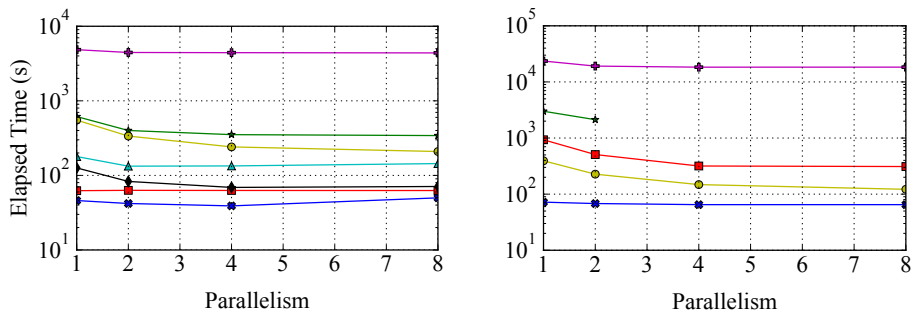
A notable observation was that both SciDB and MADlib failed with the 40M and 80M datasets. The SciDB's `gemm()` operator relied on ScaLAPACK [7], assuming that all data were loaded on distributed memory. When the `gemm()` started, SciDB attempted to convert operand matrices to ScaLAPACK data format. In the failed cases, the sizes of such data were larger than the memory budget, causing an out-of-memory error. In the case of MADlib, the PostgreSQL executor crashed when attempting matrix multiplication. MADlib operations consumed lots of memory, so the operating system killed the executor. We observed the same result even when we reduced the buffer size of PostgreSQL.

Sparse LR. As shown in Fig. 8c, *PreVision* outperformed the other systems in LR experiments with sparse datasets. MADlib was time-outed with the 0.025, 0.05 and 0.1 matrices since running times exceeded 10 hours. SciDB was killed in both the 0.05 and 0.1 experiments by the operating system due to its memory overuse. The sparse matrix multiplication operator of SciDB (*i.e.*, `spgemm()`) attempted to load the whole row chunks of a left-handed side array and the whole column chunks of a right-handed side array for computing a single result chunk. When SciDB computes the multiplication result of a wide-short matrix and tall-skinny matrix in LR tasks, the operator used lots of memory, activating the out of memory killer.



(a) NMF 10M (b) PageRank Twitter

Fig. 9. Elapsed times with a varying number of iterations



(a) NMF 10M (b) Sparse LR 0.0125

Fig. 10. Elapsed times with a varying degree of parallelism

PageRank. Fig. 8d presents elapsed times for running PageRank in each system. *PreVision* outperformed the others in every case. SystemDS running on the Twitter dataset encountered an out-of-memory error due to its overuse of memory. MADlib showed comparable performances to MLlib in the Enron and Epinions experiments, while it had slower performances with the LiveJournal and Twitter datasets.

6.2.2 Varying Number of Iterations. To observe the performance trends on a long-term basis, we ran the tasks on the target systems with a varying number of iterations from one to 32. The elapsed times taken by each system for the NMF (10M) and PageRank (Twitter) tasks are shown in Fig. 9. As is shown in the figure, the elapsed times of all systems increased linearly to the number of iterations. The differences among them were in the rate (or slope) of changes as well as the absolute amount of elapsed times. *PreVision* was the best performer with respect to both the rate and absolute amount. SystemDS experienced the same out-of-memory error as before in the PageRank experiment. MADlib was timed out and terminated in the PageRank experiment with nine or more iterations. Similar trends were observed in experiments with other datasets.

6.2.3 Varying Degree of Parallelism. We evaluated the target systems with different degrees of parallelism. The NMF 10M and LR 0.0125 tasks were chosen for this experiment because they were more CPU-intensive than the other tasks. The parallel execution of the target systems was

configured as follows. We ran SciDB and MADlib with a varying number of workers. For NumPy and *PreVision*, we changed the number of threads for a single operation because these systems ran in the operator-at-a-time mode. For SystemDS and MLlib, we changed the number of threads for a single worker because this setup yielded the best performance.

Fig. 10 shows the elapsed times of the target systems. Although *PreVision* did not achieve enough speedup after the degree of parallelism reached four, it still outperformed the other target systems consistently in all test cases. SciDB experienced an out-of-memory error while processing some of the LR 0.0125 tasks.

6.3 Effects of Preemptive Eviction

We conducted another experiment to demonstrate the effectiveness of preemptive eviction. *PreVision* with the `getPos` function executes a matrix computation task on a per-tile basis. For the matrix computation task $C = AB^T$, given in Example 1, a matrix multiplication for an output tile can be interleaved with a matrix transposition for another output tile. To accurately evaluate the effect of preemptive eviction on the I/O performance, we ran *PreVision* with and without preemptive eviction in the interleaved execution mode and non-interleaved (or *blocking*) execution mode. In the blocking execution mode, an operator waits until all the child operators complete their execution and produce the entire sets of output tiles. Note that NumPy works this way in the blocking mode.

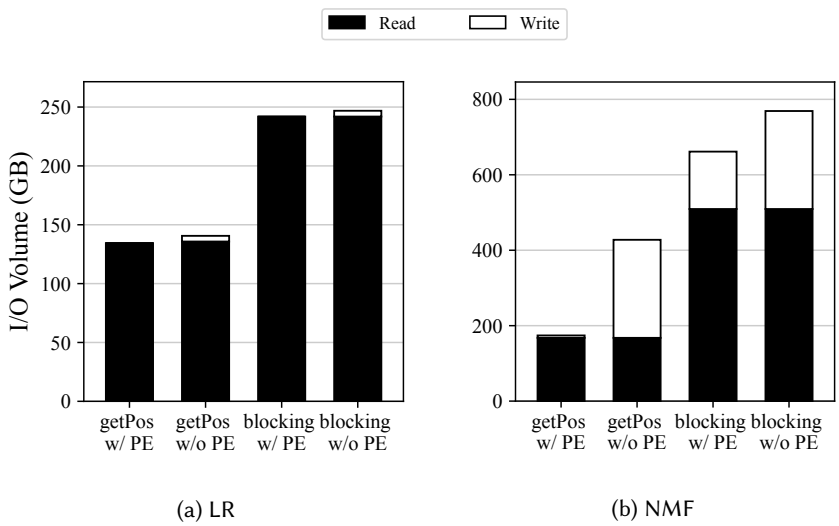
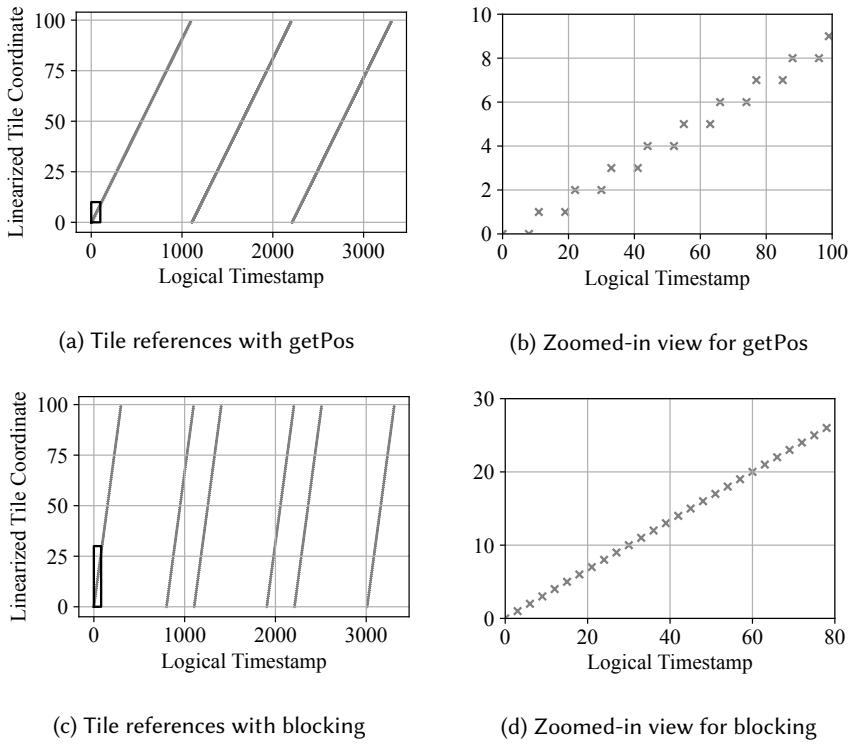


Fig. 11. The I/O volumes with execution modes and preemptive eviction (PE)

Fig. 11 presents I/O volumes for each version when running LR and NMF tasks with the 80M dense matrix. The getPos with preemptive eviction showed the minimum disk I/O volumes compared to the other versions. We observed two interesting points. First, the versions with preemptive eviction showed significantly lower disk writes compared to the versions without preemptive eviction. Especially, in cases except for the blocking with preemptive eviction in NMF, disk writes occurred only for the output arrays. Second, the getPos versions presented tremendously lower disk read volumes than the blocking versions. This was mainly due to the better *temporal locality* of the getPos execution mode.

Fig. 12 illustrates the tile access patterns of the input matrix X when running LR with the getPos and blocking modes. Fig. 12a and Fig. 12c represent the access patterns during the entire



(a) Tile references with getPos

(b) Zoomed-in view for getPos

(c) Tile references with blocking

(d) Zoomed-in view for blocking

Fig. 12. Tile references in the LR task

computation, while Fig. 12b and Fig. 12d show zoomed-in views within the black rectangles on the left. The blocking mode accessed tiles in sequential order, whereas the getPos mode repeatedly accessed the same tile coordinates before accessing the other coordinates. Since the experiment with the 80M input matrix X was out-of-core computation, this temporal locality provided the buffer manager with more opportunities for the buffer hit, resulting in reduced read I/O volume for the getPos mode.

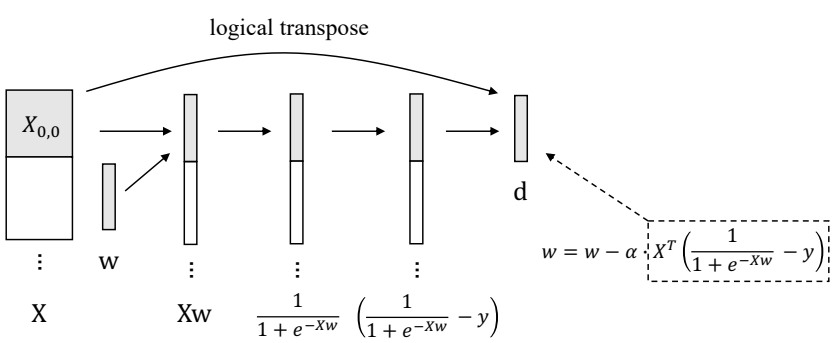


Fig. 13. Temporal locality in the LR task

Fig. 13 illustrates the situation in which an input tile was accessed repeatedly. To compute d in the LR equation, the d iterator requested the $X_{0,0}$ tile. Next, the d iterator requested the right-hand operand. Since this tile had not been computed yet, its iterator requested its operand tile. These recursive requests continued until the Xw iterator requested the $X_{0,0}$ tile. That is, a buffer request for $X_{0,0}$ was made to compute $Xw_{0,0}$, and it was repeatedly made to compute d after the recursive requests were completed. The time between these tile requests was short, increasing the likelihood that the $X_{0,0}$ tile would remain in the buffer pool.

6.4 Effects of Buffer Replacement

To evaluate the effectiveness of the OPT replacement algorithm and the overheads associated with the skip list and future log, we compared OPT with two additional replacement algorithms, LRU-K [29] and MRU. For LRU-K, K was set to two for all tasks (*i.e.*, LRU-2). Since the number of buffer frames in the pool was relatively small compared with the traditional relational database systems adopting small fixed-length pages (*e.g.*, 8KB page size), the retained information period of LRU-2 was ignored. Preemptive eviction was enabled for each replacement algorithm.

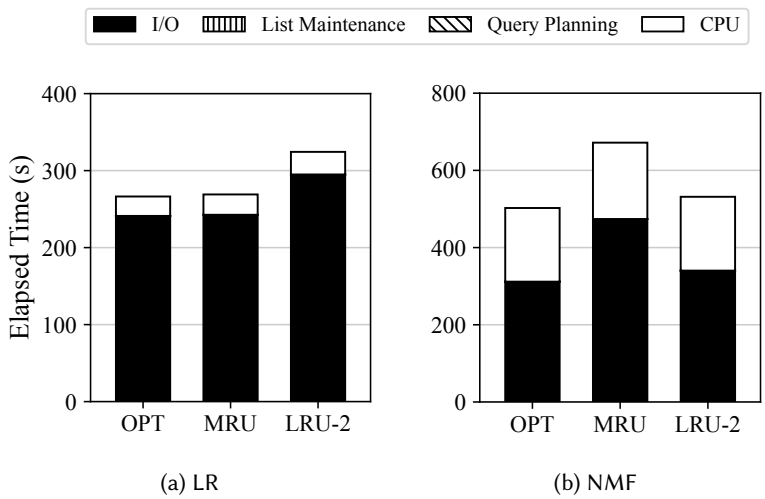


Fig. 14. Performance of buffer replacement algorithms

Fig. 14 shows the elapsed times of executing the LR and NMF tasks on the 80M dense matrix with each replacement algorithm. The I/O, List Maintenance, Query Planning, and CPU mean the sum of read and write times, the time for skip list update and future log retrieval, the query planning time including future log creation, and the computation time, respectively.

PreVision operating under the OPT replacement algorithm took the shortest elapsed times in both the LR and NMF computations. Notably, the MRU algorithm showed almost the same performance as OPT in the LR experiment. In this case, the input matrix X required the most time to read. As the input matrix was sequentially accessed as shown in Fig. 12a, *sequential flooding* of the matrix occurred. In this situation, the best buffer replacement algorithm for the matrix is MRU [34], leading to its near-comparable performance to OPT. In the NMF experiment, the elapsed time of LRU-2 followed OPT's elapsed time, whereas MRU showed poor performance. The time for list maintenance and query planning was negligible in every case. In the LR task under OPT, for instance, the overhead time was accounted for 13 milliseconds and 14 milliseconds, respectively.

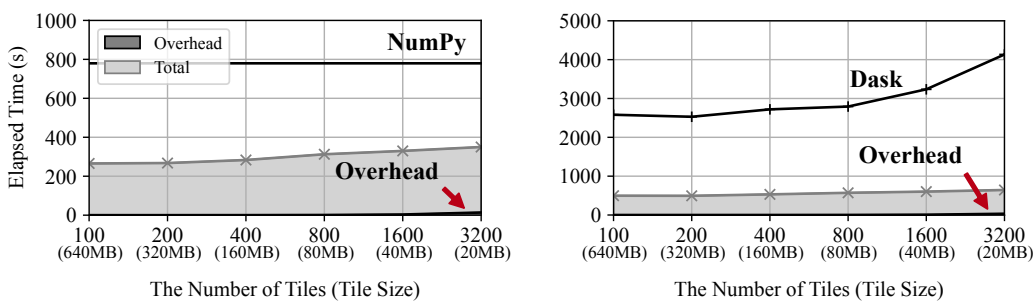
Table 3 presents buffer hit ratios for each task under the different replacement algorithms. In the LR experiment, OPT achieved a lower hit ratio than MRU. This discrepancy was because *PreVision* used variable-length buffer size to hold a tile, meaning that both buffer hits for a small tile and a large tile were counted as one hit equally. In the NMF experiment, OPT demonstrated the best hit ratio among the replacement algorithms.

Tasks	OPT	MRU	LRU-2
LR	0.6	**0.611**	0.572
NMF	**0.687**	0.664	0.636

Table 3. Buffer hit ratios with different replacement algorithms

We also investigated the impact of the number of tiles and its associated overheads. The overheads incurred by the future log and skip list are influenced by the number of tiles. Fig. 15 shows the relationship between smaller tile sizes and the resulting elapsed times. The elapsed times of the second-best performers mentioned in Section 6.2 are also displayed with black lines. Numbers in the parentheses of the x-axis indicate the size of a single tile in the input matrix. The results indicate that increasing the number of tiles led to longer elapsed times. However, *PreVision* consistently outperformed the second-best performers.

Our experiment presents that the overheads of the future log and skip list had a trivial impact on total elapsed times, regardless of the number of tiles. We argue that these overheads are negligible since the default tile size of many data processing systems is typically larger than 20MB. Notably, Spark defaults to a block size of 128 megabytes [13] and SciDB emphasizes that the block size should be several megabytes at least [39].



(a) LR tasks (b) NMF tasks

Fig. 15. Elapsed times with varying tile sizes

7 RELATED WORK

Buffer Replacement Algorithm. The MIN buffer replacement algorithm (or Belady's algorithm) [5] and the OPT buffer replacement algorithm [26] are theoretically optimal but they are known to be infeasible to implement because they require knowledge of the future. Jain and Lin show how a cache replacement algorithm can learn from Belady's algorithm by applying it to past references to inform future replacement decisions [20]. Simulating the OPT buffer replacement algorithm has been studied to determine the performance measures and to characterize the buffer misses. Mattson *et al.* introduce a stack algorithm that simulates the OPT buffer replacement by two-pass

buffer trace scans [26]. Sugumar *et al.* propose a more efficient MIN simulation scheme employing a limited look-ahead scan and tree-based stack maintenance [40].

LRU, which selects the least recently used buffer frame as a victim for replacement, has been the de facto standard buffer replacement algorithm for many database systems. In contrast, MRU selects the most recently used buffer frame as a victim for replacement. MRU may not be effective for random references but it can perform better than LRU for certain database workloads by preventing sequential flooding [34]. The LRU-K algorithm selects a buffer frame having the greatest backward-K distance as a victim for replacement [29]. The distance metric allows the algorithm to consider the frequency of references rather than the recency of references. Such buffer replacement algorithms as 2Q [21], ARC [27], and CAR [3] have also been proposed to overcome the disadvantages of LRU.

While the studies mentioned above assume that all buffer frames in the buffer pool have the same size, another branch of research has considered variable-length buffer frames. The OPT algorithm achieves optimal buffer replacement for fixed-length buffer frames by adopting a simple eviction strategy that selects a buffer frame with the greatest forward distance. For variable-length buffer frames, however, optimal caching is proven to be NP-hard [12]. Many heuristics have been suggested to improve buffer replacement for variable-length buffer frames. One of the popular heuristics is Belady-Size, which evicts a buffer frame with the greatest distance weighted by its size [6]. The GreedyDual-Size-Frequency [2], Hyperbolic [8], and LHD [4] replacement algorithms resort to multiple metrics, variable decaying priorities, and hit density, respectively, for victim selection.

Systems for Matrix Computation. Spark [46] is one of the most widely used data processing frameworks. A Spark query plan is split into stages, each of which is executed by Spark executors. Spark spills every output produced from a stage to disk and reads spilled data back when it starts processing another stage. MLlib [28] is a machine learning library built on top of Spark, and operations in MLlib are composed of Spark operations. SystemDS [9] is another machine-learning framework that leverages various back-ends. Some tasks are delegated to Spark when the standalone mode of SystemDS cannot deal with the data volume. The buffer pool of SystemDS is enhanced to link between its host program and Spark. This feature poses the challenge of detecting a buffered object that is no longer needed due to the lazy evaluation of Spark. To work around it, SystemDS recursively checks descendant RDDs or broadcasts in its lineage on an object cleanup. It determines whether the object is still referenced and performs a cleanup of the unreferenced object [10].

SciDB [39] is a database system for large-scale arrays. It is known for using tile pipelining for its query execution, which means a tile computed by an operator can subsequently be used by another operator. MADlib [17] is a machine learning system running on a relational database system. A tuple in the system consists of a row vector or a cell value with coordinates depending on the type of a matrix.

NumPy [43] is a numerical computation package focusing on ease of use. It can access disk-resident arrays via memory-mapped I/O. Likewise, MATLAB [19] supports memory-mapped files to access arrays on disk. Dask [36] supports parallel matrix computations by splitting an array into tiles of a NumPy matrix. SciPy [44] offers scientific computing functionalities such as sparse matrix computations, complex linear algebra functions, and signal processing. TensorFlow [1] and PyTorch [31] are among the most popular machine-learning libraries in diverse fields. TensorFlow aims at distributing large-scale machine learning tasks, while PyTorch enables a high level of flexibility in modeling. Unfortunately, these libraries do not support out-of-core computation for large matrices. Marques *et al.* exploit cache memory to enable out-of-core computations, but they focus on exploiting parallelism rather than minimizing I/O [25].

Tiling. Tiling is a common method for partitioning an array into smaller subarrays such that data locality is preserved [38]. It has been adopted by numerous systems such as SciDB [39], SystemDS [9], MLlib [28], and Dask [36]. In addition, many studies have been conducted focusing on reducing disk I/O for processing tiled arrays. For instance, RIOT [47] examines the I/O patterns present in a given program, identifying transformations that minimize I/O operations.

Reducing the Memory Pressure. SuperNeurons introduces liveness analysis to evict obsolete variables from memory [45]. It analyzes the input and output variables of each layer in deep learning tasks to identify objects that will not be used in the next layer, and evicts those obsolete objects after processing the current layer. The liveness analysis is similar to analyzing the future log of *PreVision*. The eviction unit of *PreVision* is a tile, whereas that of SuperNeurons is a tensor object. Operator fusion [11, 15] is a method for enhancing the performance by merging one operator into another, enabling their joint execution. This fusion technique reduces intermediate data by avoiding unnecessary materialization of data, and it improves query performance by eliminating redundant scans and computations through sparsity exploitation.

GPU Memory Management. GPU memory management in deep learning tasks is a highlighted subject in the GPU research community as an increasing number of model states are required by the tasks. One strategy is GPU-CPU data swapping, which involves offloading GPU variables to CPU memory when the deep learning training states overrun the memory capacity of the GPU. There are many studies focused primarily on overlapping communication costs between the GPU and other components [33, 35, 45].

It is worth mentioning that SwapAdvisor [18] takes a similar approach as *PreVision*. Both of them select a victim for replacement based on future references. However, a key difference lies in the timing of the victim selection. While SwapAdvisor selects a victim during an optimization phase performed before runtime, *PreVision* makes this decision dynamically during runtime. *PreVision* possesses the capability of predicting access patterns based on the future log of a given plan, and focuses on the management of variable-length buffer frames. This is due to the inherent unpredictability of the buffer pool content, and variable-length tiles and memory fragmentation making it more challenging to plan evictions. In contrast, SwapAdvisor uses fixed-sized pages, ensuring predictability in the buffer pool content and allowing for the generation of eviction plans in advance.

8 CONCLUSION

We present an out-of-core matrix computation system called *PreVision* with optimal buffer replacement. A matrix larger than the available memory budget is inevitably split into smaller chunks or tiles so that they can be separately loaded into memory for further computations or query processing. The chunk-by-chunk matrix computation will incur potentially a large number of I/O operations, which would become the dominant cost of any task involving an out-of-core matrix computation. To minimize the I/O overhead, *PreVision* predicts the entire pattern of tile accesses for a given task by exploiting the deterministic nature of matrix computation algorithms. By referring to the predicted access patterns collected in the *future log*, the buffer manager can handle buffer replacement optimally. It also adopts the preemptive eviction strategy to remove obsolete tiles from the buffer pool eagerly without causing unnecessary flushing to disk.

We demonstrate that *PreVision* outperforms the other competing systems in all the out-of-core tasks tested in our experimental evaluations. The significant performance gain of *PreVision* is attributed to the following factors. First, *PreVision* enables the optimal buffer replacement and thereby reduces the overall I/O overhead. Second, the preemptive eviction strategy of *PreVision* avoids unnecessary disk write operations. Third, *PreVision* reduces I/O operations further by taking

advantage of access locality. By leveraging the approaches introduced by *PreVision*, we believe that the disk I/O and the memory pressure challenges arising from many large-scale matrix computations can be addressed more effectively.

The buffer replacement of *PreVision* is considered optimal only on the assumption that all the tiles are given the same weight. Thus, if a larger tile is given a heavier weight than a smaller one, then the buffer replacement of *PreVision* will no longer be considered optimal. For the wider application of *PreVision*, we need to address such problems as synchronizing the timestamps of tiles across multiple concurrent tasks, processing operators in parallel, and planning multiple queries globally. We leave these challenges as future work.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (grant no. 2020R1A2C1010358). The authors assume all responsibility for the content of the paper.

REFERENCES

[1] 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 G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, Xiaoqiang Zheng, and Google Brain. 2016. TensorFlow: a system for Large-Scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. Savannah, GA, USA.

[2] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. 2000. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review* 27, 4 (March 2000), 3–11.

[3] Sorav Bansal and Dharmendra S. Modha. 2004. CAR: Clock with Adaptive Replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*. San Francisco, CA, USA.

[4] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA, USA.

[5] L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966), 78–101.

[6] Daniel. S. Berger, Nathan Beckmann, and Mor Harchol-Balter. 2018. Practical bounds on optimal caching with variable object sizes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2 (June 2018), 1–38.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 1987. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA.

[8] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA, USA, 499–511.

[9] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie 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 *10th Conference on Innovative Data Systems Research*. Amsterdam, The Netherlands.

[10] Matthias Boehm, Michael 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. *Proceedings of the VLDB Endowment* 9, 13 (Sept. 2016), 1425–1436.

[11] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Alexandre V. Evfimievski, and Prithviraj Sen. 2018. On Optimizing Operator Fusion Plans for Large-Scale Machine Learning in SystemML. *Proceedings of the VLDB Endowment* (Aug. 2018), 1755–1768.

[12] Marek Chrobak, Gerhard J. Woeginger, Kazuhisa Makino, and Haifeng Xu. 2012. Caching is hard—even in the fault model. *Algorithmica* 63, 4 (Aug. 2012), 781–794.

[13] Apache Software Foundation. 2023. RDD Programming Guide. https://spark.apache.org/docs/3.3.2/rdd-programming-guide.html.

[14] The PostgreSQL Global Development Group. 2023. PostgreSQL: Documentation: 12: 19.4. Resource Consumption. https://www.postgresql.org/docs/12/runtime-config-resource.html

[15] Donghyoung Han, Jongwuk Lee, and Min-Soo Kim. 2022. FuseME: Distributed Matrix Computation Engine based on Cuboid-based Fused Operator and Plan Generation. In *Proceedings of the 2022 International Conference on Management of Data*. Philadelphia, PA, USA, 1891–1904.

[16] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362.

[17] Joseph Hellerstein, Florian Ré, Christopherand Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library. *Proceedings of the VLDB Endowment* 5, 12 (Aug. 2012), 1700–1711.

[18] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. Lausanne, Switzerland, 1341–1355.

[19] The MathWorks Inc. 2022. *MATLAB version: 9.13.0 (R2022b)*. Natick, Massachusetts, United States. https://www.mathworks.com

[20] Akanksha Jain and Calvin Lin. 2016. Back to the future: Leveraging Belady's algorithm for improved cache replacement. *ACM SIGARCH Computer Architecture News* 44, 3 (June 2016), 78–89.

[21] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. *Proceedings of the 20th International Conference on Very Large Data Bases* (Sept. 1994), 439–450.

[22] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10: Proceedings of the 19th international conference on World wide web* (Raleigh, North Carolina, USA). ACM, New York, NY, USA, 591–600.

[23] Daniel D. Lee and H. Sebastian Seung. 2000. Algorithms for non-negative matrix factorization. *Advances in neural information processing systems* 13 (2000).

[24] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[25] Mercedes Marqués, Gregorio Quintana-Orti, Enrique. S. Quintana-Orti, and Robert. A. van de Geijn. 2009. Solving "large" dense matrix problems on multi-core processors. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. Rome, Italy.

[26] Richard. L. Mattson, Jan. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems journal* 9, 2 (1970), 78–117.

[27] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. San Francisco, CA, USA.

[28] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. Mllib: Machine learning in apache spark. 17, 1 (Jan. 2016), 1235–1241.

[29] Elizabeth. J. O'neil, Patrick E. O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record* 22, 2 (June 1993), 297–306.

[30] Lawrence Page, S. Brin, R. Motwani, and T. Winograd. 1998. *The pagerank citation ranking: Bring order to the web*. Technical Report. Technical report, Stanford University.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*.

[32] William Pugh. 1990. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (June 1990), 668–676.

[33] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: Breaking the gpu memory wall for extreme scale deep learning. In *SC '21: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. St. Louis, MO, USA, 1–14.

[34] Raghu Ramakrishnan and Johannes Gehrke. 2002. *Database management systems* (3rd ed.). McGraw-Hill.

[35] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Taipei, Taiwan, 1–13.

[36] Matthew Rocklin. 2015. Dask: Parallel computation with blocked algorithms and task scheduling. In *Proceedings of the 14th python in science conference*. Austin, TX, USA.

[37] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[38] Sunita Sarawagi and Michael Stonebraker. 1994. Efficient organization of large multidimensional arrays. In *Proceedings of 1994 IEEE 10th International conference on data engineering*. IEEE, Houston, TX, USA, 328–336.

[39] Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. 2011. The architecture of SciDB. In *Proceedings of the 23rd International Conference on Scientific and Statistical Database Management*. Portland, OR, USA.

[40] Rabin A. Sugumar and Santosh G. Abraham. 1993. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. Santa Clara, CA, USA.

[41] Anthony Thomas and Arun Kumar. 2018. A comparative evaluation of systems for scalable linear algebra-based analytics. *Proceedings of the VLDB Endowment* 11, 13 (Sept. 2018), 2168–2182.

[42] Sivan Toledo. 1999. A survey of out-of-core algorithms in numerical linear algebra. *External memory algorithms* 50 (Dec. 1999), 161–179.

[43] San van der Walt, S. Chris Colbert, and Gaël Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering* 13, 2 (March 2011), 22–30.

[44] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, Ilhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* 17, 3 (Feb. 2020), 261–272.

[45] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. 2018. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. New York, NY, USA, 41–53.

[46] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA, USA.

[47] Yi Zhang and Jun Yang. 2012. Optimizing I/O for Big Array Analytics. *Proceedings of the VLDB Endowment* 5, 8 (Aug. 2012).

Received July 2023; revised October 2023; accepted November 2023