Received: 24 December 2024 | Revised: 25 August 2025 | Accepted: 17 November 2025 | Published online: 11 August 2026

RESEARCH ARTICLE

Journal of Data Science and Intelligent Systems
2026, Vol. 00(00) 1–12
DOI: 10.47852/bonviewJDSIS62025078

BON VIEW PUBLISHING

Efficient Online Scheduling of Data Transfers in Multi-tiered Persistent Storage

Nan Noon Noon[1,*], Janusz R. Getta[1] and Tianbing Xia[1]

[1]*School of Computing and Information Technology, University of Wollongong, Australia*

Abstract: This paper addresses the problem of optimal online scheduling of data transfers among the levels of multi-tiered persistent storage systems. The paper describes the new algorithms for the optimization of data transfers and shows the important role played by multi-tiered persistent storage when processing large volumes of data. A notation of Extended Petri Nets (EPN) proposed in the paper is used to represent the data flows between the storage tiers. EPNs are also used to create efficient data transfer plans by integrating multiple queries. The paper presents new ways of online query processing with real-time constraints and shows the benefits of optimized serial data transfer plans. We conduct the evaluations of online scheduling algorithms in a simulated environment of multi-tiered persistent storage. The method was tested in a simulation that mimics real storage devices like hard disk drives (HDD), solid-state drives (SSD), and non-volatile memory (NVMe) drives, based on their usual read and write speeds. We performed extensive experiments on different storage setups, from single-tier to four-tier systems, using a variety of sample queries. The results showed that we achieved better online scheduling for data transfers in multi-tier storage systems.

Keywords: multi-tiered persistent storage, serial data processing, storage management, rule-based scheduling, data transfer process, data transfers, Petri nets

1. Introduction

The increasing amounts of operational and analytical data processing in modern database systems create the demand for higher capacity and faster persistent storage devices.

Because of financial constraints, many organizations cannot replace all their persistent storage devices at once. A commonly used phased approach results in a mixed storage environment that consists of various persistent storage devices with different speeds and capacities.

A multi-tiered storage (MTS) [1] logical model can address the challenges faced by data processing applications that work with many different persistent storage devices. The MTS logical model integrates different layers of persistent storage, including non-volatile memory (NVMe), solid-state drives (SSD), hard disk drives (HDD), and magnetic tape drives (MTD) [2]. Each tier (level) in MTS has distinct capacities and performance characteristics. Usually, the lower tiers consist of slower persistent storage devices than the higher tiers.

A very similar problem of logical integration of various persistent storage devices occurs in Cloud systems [3]. Typically, Cloud systems offer large volumes of data at different speeds and prices. The existence of data at the various storage devices adds one more dimension to the optimization of data processing. Like before, well-planned data transfers between the tiers of persistent storage are essential for improving efficiency while balancing expenses.

This work focuses on the optimization of data transfer scheduling within MTS. The goal is to develop a more efficient data transfer scheduler that can adapt to changing conditions.

The paper is organized as follows. Section 2 reviews the literature on query processing, MTS, and Petri Nets. Next, Section 3 defines a logical model of MTS and shows its impact on the efficiency of data processing. Section 4 defines a new Extended Petri Nets (EPNs) model for data flow management within MTS. Section 5 shows how the online serial data transfer plans can be optimized with EPNs. Finally, Section 6 summarizes key findings and suggests future research directions for online scheduling of data transfers in MTS.

2. Previous Works

The MTS model organizes data on different storage devices, which are of different types and performance levels [4]. Its organization depends on how business-critical the data is and how often the data will need to be accessed. One of the main goals of this model is to minimize total storage costs while enhancing the performance and availability of critical applications. Therefore, more and more organizations are adopting the logical management of persistent storage provided by MTS in order to manage large operation databases and historical data better.

Recent advances in machine learning (ML) present significant opportunities to improve data storage management, especially in systems with multiple storage levels. Researchers and industry professionals are exploring how to use ML to enhance data management practices. One key development is using reinforcement learning (RL) for organizing data in storage systems [5]. RL helps decide where to store data based on how often it is accessed and used. This approach improves performance and makes better use of resources, which can lower the costs of data storage.

The work by Shi et al. [6] is based on the multi-tier hybrid storage (MTHS) model . MTHS integrates SSDs, hybrid drives, and traditional hard disks to create the MTS system that utilizes the speed of SSDs while managing the costs. The objective of this work is to reduce random access latency without adaptation to the dynamic workload changes. Additionally, researchers are using ML to predict how people

*Corresponding author: Nan Noon Noon, School of Computing and Information Technology, University of Wollongong, Australia. Email: noon@uow.edu.au

will access data in the future. By looking at past usage patterns, these models can identify which data sets are likely to be used next. This allows for better planning of data movement and storage. As a result, response times can improve, making the user experience better overall.

A number of earlier published research papers addressed the importance of cloud computing and data analytics. For instance, the importance of efficient storage for the large volume of data created by cloud analytics was discussed in the works of Wang et al. [7] and Sivarajah et al. [8]. The tiered storage system adapts resources based on data characteristics and allocates them through various scheduling methods [9, 10]. Furthermore, the combination of big data analytics and artificial intelligence (AI) technologies offers a great opportunity to create scheduling models that adapt to changing workloads [11].

Yang et al. [12] and Lv and Huang [13] overview challenges in storage systems given the emerging SSD and NVMe technologies together with the creation of automatic data placement managers.

Moreover, classical scheduling methods like Shortest Job First (SJF) and Dynamic Priority Scheduling have been commonly used to improve response time and throughput in CPU and task scheduling [14]. However, these methods don't specifically tackle the challenges of latency and throughput variations in MTS systems. As a result, recent research has shifted toward more adaptive and storage-focused approaches. Noon et al. [15] have explored rule-based scheduling techniques based on Petri Nets for scheduling parallel data transfers in multi-tiered persistent storage, illustrating how formal models can help manage and generate efficient data transfer plans.

In summary, there is an increasing need for high-capacity and fast storage solutions that can handle evolving data processing tasks [16] in environments where the demands for persistent storage grow [17] and data is distributed across various devices with different capacities [18]. Such a situation led to the development of a logical model of multi-tiered persistent storage.

3. Multi-tiered Storage

In the recent years, the strong trends to search for information for effective management of business organizations significantly contributed to the increased amounts of data accrued by the organizations. It contributed to the increasing demand for storage devices of greater capacity and faster processing speeds to efficiently manage a huge amount of data.

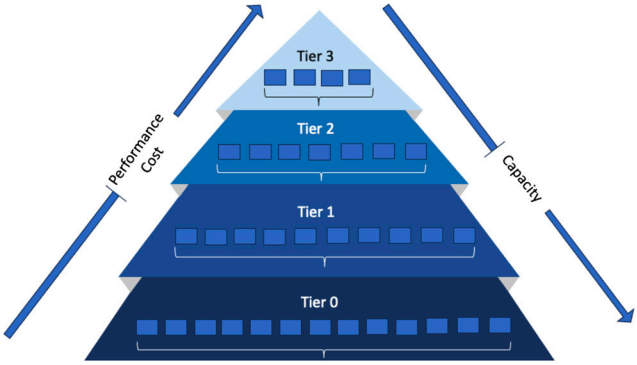
Due to the financial limitations, most of the organizations still use a mix of traditional and modern storage solutions for economic reasons. Also, the growing need to outsource data storage and management to the Cloud has equally motivated vendors to provide a set of persistent storage solutions. Synthesizing on-premises storage with cloud storage is critical for data usability across locations. Organizations would also like to allocate data effectively through the use of various persistent storage devices to maximize the time and expense of processing. Thus, managing diverse storage solutions and optimizing data processing in such a scenario became important issues in modern data management.

3.1. Logical and physical models

MTS systems consist of different types of storage devices in order to balance the costs and performance of data processing. MTS has a tiered (multi-level) structure where each tier has a given capacity and performance parameters. It contributes to a pyramid-like structure visualized in Figure 1.

At the top of the storage pyramid, the upper tiers offer faster access times and higher performance but have lower capacities. In contrast, the lower tiers have slower access times and reduced performance, but they make up for it with much larger capacities than the higher tiers.

Figure 1
Multi-tiered storage



3.1.1. Logical model of multi-tiered storage

MTS is a sequence of storage tiers $<l_0, \ldots, l_n>$, where the lowest, base, or backup tier is l_0 and the highest tier is l_n. The tiers are arranged in ascending order of data access speed. Tier l_0 data access speed is the lowest and tier l_n is the highest.

Each tier l_i, $i = 0 \ldots n$, is described by a quadruple $<r_i, w_i, s_i, u_i>$, where r_i is the read speed per data block, w_i is the write speed per data block, s_i is the total available storage, and u_i is the total used storage at tier l_i.

Each tier in MTS is a collection of persistent storage devices that have the same data access speed parameters. Persistent storage devices may include NVMe, SSD, and HDD technologies. Distributing data across many devices at the same tier can speed up overall data access through parallelization.

3.1.2. Physical model of multi-tiered storage

In MTS, each tier l_i for $i = 0 \ldots n$ is implemented as a collection of persistent storage devices, denoted as $D_i = \{d_1, \ldots, d_m\}$. Each device $d_j \in D_i$ for $j = 1 \ldots m$ is described by a quadruple $<r_j, w_j, s_j, u_j>$, where r_j represents the read speed of device d_j, w_j represents its write speed, s_j represents the available storage, and u_j represents the currently used storage at device d_j. All devices in a tier have the same read/write speed. Therefore, $r_1 = \ldots = r_m$ and $w_1 = \ldots = w_m$.

3.2. Optimal data flow in MTS

In the MTS system, data is moved from one tier to another to enable the efficient storage transfers. Data can be moved from lower tiers to higher tiers for increasing the speed of read/write operations.

In a pipelined data processing model, the outputs of one operation become the inputs of the next operation in a pipeline. Transient memory is used as a buffer for the data transfers between tiers. Data read from permanent storage in a tier are temporarily stored in transient memory and, then, written from transient memory to another tier. Data can be transferred concurrently or sequentially with no concurrent access to the same storage device.

Transfer scheduling of data between devices is crucial because not all data can be stored in the topmost levels. Scheduling maximizes performance, efficiency, and usability by managing workload during transfers.

Moving data to higher levels speeds up future access to the same data. Efficient scheduling of data transfer between the levels minimizes idle time and enhances processing speed.

3.3. Enhanced efficiency with higher tiers

This part shows the performance gains from moving data to higher levels of MTS.

This part describes a system design where MTS is applied in data processing and shows how the duration for reading and writing data affects the total processing time. It also emphasizes the benefit of putting data into higher tiers to ensure maximum operations.

Let an operation v_i attempts to write a total of τ_n data blocks and the next operation in a pipeline, v_j, tries to read the same data. Let l_x is a lower tier and l_y is a higher tier. The writing time for an operation v_i to a tier l_x is calculated as follows:

$$Tw(v_i) = \tau_i * w_x \quad (1)$$

Here, $Tw(v_i)$ represents the total writing time for v_i, τ_j is the total number of data blocks written to a tier l_x, and w_x characterises the writing speed for tier l_x.

The writing time for an operation v_i to a tier l_y is calculated as follows:

$$Tw(v_i)' = \tau_i * w_y \quad (2)$$

Here, $Tw(v_i)'$ represents the total writing time for v_i, τ_j is the total number of data blocks written to a tier y, and w_y characterises the writing speed for tier l_y.

The reading time $Tr(v_i)$ for an operation v_i for reading from tier l_x can be calculated as:

$$Tr(v_i) = \tau_i * r_x \quad (3)$$

In Equation (3), $Tr(v_i)$ denotes the total reading time for v_i, τ_j corresponds to the total number of data blocks read from tier l_x, and r_x represents the reading speed for tier l_x.

The reading time $Tr(v_i)'$ for an operation v_i for reading from tier l_y can be calculated as:

$$Tr(v_i)' = \tau_i * r_y \quad (4)$$

In Equation (4), $Tr(v_i)'$ denotes the total reading time for v_i, where τ_j is the total number of data blocks read from tier l_y, and r_y is the reading speed for tier y.

Subsequently, $Tw(v_i)$, $Tw(v_i)'$, $Tr(v_j)$, and $Tr(v_j)'$ are used to calculate the benefit of the operation v_i when writing data to a higher tier, utilizing Equation (5). The total benefit of an operation v_i when writing, the output result on higher tiers is calculated as:

$$\beta(v_i) = [Tw(v_i) - Tw(v_i)'] + [Tr(v_j) - Tr(v_j)'] \quad (5)$$

When $\beta(v_i)$ yields a positive value, it is advisable to write the data to the highest tier available. This strategic choice ensures a swifter processing time, which, in turn, is ideal for optimizing the execution of complex tasks or managing substantial data volumes. If $\beta(v_i)$ equals zero or it is close to zero, then it implies that the result can be efficiently written on the same tier from which the operation v_j reads the data.

However, in the cases where $\beta(v_i)$ is negative, the operation should ideally wait for a higher tier to become available. This proactive approach enhances the efficiency of data processing.

3.3.1. Example

In this example, we consider two tiers $l_1 = (8, 10)$ and $l_2 = (5, 6)$. An operation v_i writes the total number of data blocks $\tau_x = 30$ to a tier and an operation v_j reads those data blocks.

Case 1: an operation v_i writes τ_x data blocks to tier l_1. Therefore, an operation v_j reads the data from the same tier.

1) From Equation (1), we have $Tw(v_i) = (30 * 10) = 300$.
2) From Equation (3), we have $Tr(v_j) = (30 * 8) = 240$.

Case 2: an operation v_i writes τ_x data blocks to tier l_2. Therefore, an operation v_j reads the data from the same tier.

1) From Equation (2), we have $Tw(v_i)' = (30 * 6) = 180$.
2) From Equation (4), we have $Tr(v_j)' = (30 * 5) = 150$.

Finally, by using Equation (5), the benefit of an operation v_i is $\beta(v_i) = [300 - 240] + [180 - 150] = 90$.

4. Petri Nets

In this section, we review a graphical notation of Petri Nets, and we define a notation of EPN for the representation of parallel query processing plans.

A Petri Net (PN) is a binary-directed graph that consists of two sets of vertices, *places*, and transitions connected with arcs [19–21]. Tokens are located at places. Places represent the distinct states or conditions within a system and are visualized as circles. Each place can hold zero or more tokens. Transitions represent events or activities that change the system's state and are typically represented as vertical bars. Arcs link places and transitions in a way that indicates the flow of tokens between them. Tokens represent data traversing the system in accordance with a given data processing plan, either moving from place to transition or vice versa. Tokens are represented as bold dots (·) in a Petri Net.

Interpretation of Petri Net is based on a fundamental principle: tokens can be transferred from a place to a transition. Then, for a transition to activate (fire), it must possess the required number of tokens within its input places. The arcs linking places to transitions are assigned numerical values, denoting the number of tokens required from the *input* places. For example, an arc marked with a '2' signifies the necessity of two tokens from the input place for the transition to activate (fire). Upon firing, a transition consumes tokens from its input places and generates tokens in each associated output place. Similarly, the arcs connecting transitions to output places are labeled with numbers, indicating the number of tokens generated and placed into the connected output places upon the transition's activation. However, if a place lacks tokens, the transition remains inactive until sufficient tokens accumulate in the input places to initiate it.

When an edge leads from a place to a transition, then the place is an input. When an edge leads from a transition to a place, then the place is considered the output of the transition.

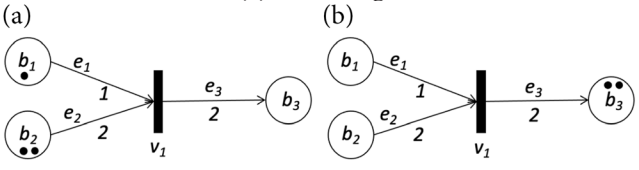
1) Definition (Petri Net)

A PN is a graph of quadruple (B, V, E, W), where $B=\{b_1, \ldots, b_n\}$ is a set of places represented by circles, $V =\{v_1, \ldots, v_m\}$ is a set of transitions represented by bars, E is a set of arcs or edges that link the places and transitions, and W is a weight function $W:E \longrightarrow \mathbb{N}^+$. E is denoted as $\{e_1, \ldots, e_y\}$, where $E \subseteq (B * V) \cup (V * B)$. Tokens (·) are entities located at each place.

2) Example

According to Figure 2, let the Petri Net be $PN = (B, V, E)$, where $B = \{b_4, b_2, b_3\}$, $V =\{v_1\}$, and $E = \{e_1, e_2, e_3\}$. In Figure 2, places b_1 and b_2 are input places for transition v_1, while b_3 is the output place. Figure 2(a) shows that the transition v_1 requires one token from b_1 and two tokens from b_2 to be fired. Conversely, the right-hand side of Figure 2(b) shows that the transition v_1 consumes input tokens from b_1 and b_2, and generates two output tokens for b_3.

Figure 2
Example of transition firing in a Petri net: (a) before firing v1; (b) after firing v



4.1. Extended Petri Nets

We extend Petri Nets with the annotations and with the more precise semantics (interpretation) directly related to the synchronization of parallel data transfers. An EPN is a directed, weighted, bipartite, and acyclic graph that consists of two sets of vertices: *data containers* (B) and *operations* (V). *A data container* is either an *input* or *output container* for operations. The *output data container* stores the results created by an operation.

A single EPN represents many of the different ways in which data can be processed in parallel. As a simple example, consider the EPN given in Figure 3. All input tokens are available for both operations v_1 and v_2 and because of that v_2 can be processed in parallel with v_1. When all input tokens are available in b_2, then an operation v_3 can start processing. Then, it is possible that v_1 can simultaneously process data with v_3. The number of tokens required to start processing an operation is determined by an operation type. For example, a relational join operation requires two input data containers. It can start only if at least one token from each data container is available.

In the EPN for query processing graph, data containers located on the left-hand side of an edge are input containers, whereas those on the right-hand side are output containers that store intermediate results of the operation. A set of operations, V, represents the operations in the EPN, and an edge, e, connects B and V and vice versa.

The implementation of data containers encompasses the creation of multiple transferable data, which can be distributed across various tiers for storage. These transferable data, referred to collectively as data transfers, may reside within the same or different tiers. Each data transfer can be assigned a token if it contains data ready for processing by an operation within the EPN architecture. Upon completion of an operation's processing of input data from each data transfer, a token is generated for the output data container. Consequently, each data container may be associated with one or more tokens.

Operations begin processing data transfers when the data transfers located in input data containers are associated with a required number of tokens. Such constraints are determined by the numerical range associated with the edge between the input container and the operation. If an input data container is linked to multiple operations, one or more operations may process a single data transfer. Once an operation completes processing on a data transfer, that data transfer with the token becomes immutable. Removal is only possible for a data transfer associated with a token if all connected operations have completed processing it.

When the processing of an operation is completed, it automatically generates the tokens for each data transfer originating from its output data containers. The total number of tokens generated depends on the numerical range specified on the edge between the operation and the output container. Moreover, only one token can be associated with a single data transfer.

The size and the total number of input and output data containers for each operation are estimated by a query optimizer from the histograms stored in a data dictionary of the database system. Database systems utilize histograms, constructed on certain columns or combinations thereof to estimate the size of output data created by each operation in EPN.

4.1.1. Definition (Extended Petri Net)

An Extended Petri Net is quadruple $\varepsilon = (B, V, E, W)$, where B and V are disjoint sets of **data containers** and **operations**, $\varepsilon \subseteq (B * V) \cup (V * B)$ is a set of **arcs** that connect the data containers and operations, and W is a **weight function** that attaches the weights to the **arcs**. The meanings of B, V, E, and W are the following.

1) B is a set of data containers denoted as $B = \{b_1, \ldots, b_n\}$. A **data container** $b_i \in B$ ($i = 1 \ldots n$) is associated with a set of data transfers $\{\delta_1, \ldots, \delta_i\}$.
2) A **data transfer** δ is a quadruple $<q_j, l_i.d_k, \tau_n, t>$ where q_i is a query that initiates the data transfer, $l_i.d_k$ represents a device d_k located in tier l_i, τ_n signifies the amount of data that will be transferred to/from the related device $l_i.d_k$, and t is either a token (·) or null (ε).
3) An **operation** $v \in V$ is a pair of sets $<\{b_1, \ldots, b_n\}, \{b_{n+1}, \ldots, b_m\}>.b_1, \ldots, b_n$ are the input data containers, and $b_{n+1}, \ldots, b_m$ are the output data containers used by an operation v.
4) An **edge** $e \in E$ is either a pair $<b, v>$, where b is an input **data container** of the **operation** v, or a pair $<v, b>$, where b is an output **data container** of the **operation** v.
5) A **weight function** W: $E \rightarrow \mathbb{N}^+$.
   a) $\forall\, e \in E$, $W(e)=x$ where $x \in \mathbb{N}^+$.
   b) If $e = (b, v)$, then an operation v can start reading data transfers when at least $W(e)$ data transfers are available, i.e. have a token (·).
   c) On the other hand, if $e = (v, b)$, then an operation v writes $W(e)$ data transfers to a container b.

4.1.2. Example

As a simple example, consider EPN $\varepsilon = (B, V, E, W)$ given in Figure 3. A set B consists of five data containers $\{b_1, b_2, b_3, b_4, b_5\}$. Three data transfers $\{(q_1, l_1.d_1, \tau_1, \cdot), (q_1, l_1.d_2, \tau_2, \cdot), (q_1, l_1.d_3, \tau_3, \cdot)\}$ are required to read all data from a container b_1. All data transfers in b_1 are marked as available now by a token (·). Three data transfers $\{(q_1, l_2.d_4, \tau_4, \cdot), (q_1, l_2.d_5, \tau_5, \cdot), (q_1, l_2, d_6, \tau_6, \varepsilon)\}$ are required to write data by an operation v_1 into a data container b_2. The same data transfers are required to read data by an operation v_3 from a data container b_2. One of the data transfers $\{(q_1, l_2, d_6, \tau_6, \varepsilon)\}$ is marked as unavailable now by a missing token (ε).

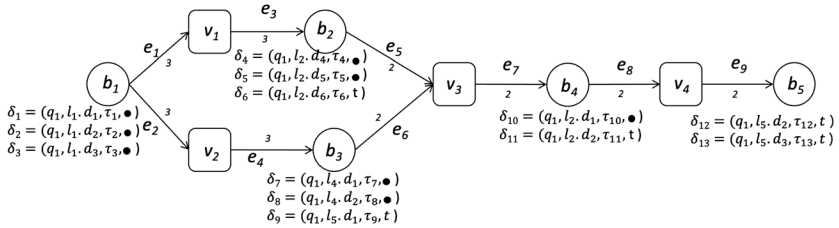
An operation v_1 can read data transfers when at least 3 data transfers in b_1 are marked with a token (·) because the weight value $W(e_1) = 3$.

An operation, v_1 performs three write data transfers into a data container b_2. This is because the weight value $W(e_3)$ is 3.

An operation v_3 can read data transfers when at least 2 data transfers in b_2 are marked with a token (·), $W(e_5) = 2$, and when at least 2 data transfers in b_3 are marked with a token (·), $W(e_6) = 2$.

The remaining components of EPN given in Figure 3 have the same meanings.

Figure 3
Example of Petri Net with tokens



4.2. Creating and processing of EPN

An EPN is created in the final stages of query optimization when a query optimizer estimates the amounts of data read and written by the operations implementing the queries. The data transfers required to access input data by a query are provided by an MTS manager. Then, a query optimizer estimates the total amounts of data written and read by each operation when a query is processed and sends such information to the MTS manager. The manager allocates the storage and returns the data transfers to a query optimizer. Information about the data transfers and the maximum number of simultaneous data transfers for each operation is used to create EPN.

At the beginning of query processing only the data transfers required to read a database are marked by a token (·), like, for example, data transfers in an EPN in Figure 3. All other data transfers are marked by a missing token (ε). Whenever an operation completes a write data transfer into a data container, then such transfer is re-marked from (ε) to (·).-Definition 4.2.1 determines the rules for the processing of EPNs.

4.2.1. Definition (processing of EPN)

The rules governing the execution of data transfers by an operation within the EPN framework are determined as follows:

1) An operation is eligible to execute a read data transfer under the condition that the data transfer is marked by a token (·) and it is not simultaneously accessed by another operation.
2) An operation can execute a write data transfer under the condition that the data transfer is marked by a missing token (ε), and it is not simultaneously accessed by another operation.

5. Scheduling on Online Data Transfer Processes

The conventional approach to data transfer planning, while simple, is inefficient. It often leads to extended transfer times, especially with large data volumes or multiple queries. Thus, there is a critical need for improved strategies in developing data transfer plans.

This section outlines a proficient approach for transferring data across multiple queries using a single process. This involves retrieving data from persistent storage to a temporary memory or vice versa. The development of a data transfer plan can be achieved through various approaches based on the number of transfer procedures available and the characteristics of the EPN and MTS. A data transfer plan is a structured sequence of transfers assigned to a data transfer process.

Assume that a stream of queries Q needs to be processed according to a given plan. The algorithm operates on the most up-to-date fragment of the stream visible as a window of size w. Initially, the algorithm checks the number of queries in Q. If it exceeds w, then it creates a query window of the first w queries. If there are fewer queries, all are stored in the window, and after processing, all are removed from Q.

A standard query optimizer transforms the queries in Q into the query processing plans and later into the EPNs. Then all EPNs are merged into a single EPN. The merging of EPNs is explained in Section 5.1.

In the following step, the merged EPN is converted into a sequence of data transfers $\Delta = < v_i : \delta_j, \ldots, v_x : \delta_y >$. The specifics of such conversion are described in Sections 5.2, 5.3 and 5.4.

In order to enable the processing of new queries, the transformation dynamically alters the incoming queries as data transfers that are requested by old queries are being processed. The transformation maintains a stream of new queries Q at all times. It re-computes the processing plans, and it alters the data transfer sequence as required.

If a queue of queries Q is not empty, the algorithm selects up to w queries, converts them into EPNs, and merges them into a single EPN. Then, the new EPN is merged with the existing sequence data transfers Δ, and the cycle repeats.

The entire procedure of online processing a stream of queries Q is shown in Algorithm 1.

Algorithm 1 Online data processing: serial execution

Input: A stream of queries $\boldsymbol{Q} = <\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m, \ldots>$ and window size w.

Output: Keep updating the sequence of data transfers $\boldsymbol{\Delta}$.

1. Create an empty data transfer plan $\boldsymbol{\Delta} = < >$.
2. Select at most first w number of queries from $\boldsymbol{Q}$ and save them into a temporary set of queries $temp_q$.
3. Remove selected queries from $\boldsymbol{Q}$.
4. Use **Algorithm 2** to generate a sequence of data transfers, data transfer plan $\boldsymbol{\Delta}$' for $temp_q$.
5. Update $\boldsymbol{\Delta} = \boldsymbol{\Delta} + \boldsymbol{\Delta}$'.
6. Iterate over the sequence of data transfers $\boldsymbol{\Delta}$; let the current data transfer be $\boldsymbol{v_i} : \boldsymbol{\delta_j}$.
   6.1. If processing on $\boldsymbol{v_i} : \boldsymbol{\delta_j}$ is completed, then remove it from $\boldsymbol{\Delta}$.
   6.2. If new queries are found or $\boldsymbol{Q}$ is not empty, then append new queries into $\boldsymbol{Q}$.
   6.3. Return to step 2.
7. If $\boldsymbol{\Delta}$ is empty, continuously monitor incoming queries until they are detected.
8. If queries are detected, append them into $\boldsymbol{Q}$ and return to step 2.

Algorithm 1 processes a sequence of data transfers $\boldsymbol{\Delta}$, and handles tasks like removing certain data transfers. As a result, the algorithm has a complexity of $O(n)$, where n is the size of $\boldsymbol{\Delta}$.

Algorithm 2 Generating a sequence of data transfers for the first few queries from $\boldsymbol{Q}$.

Input: A set of queries $temp = < \boldsymbol{q}_i, \ldots, \boldsymbol{q}_j >$.

Output: Return a sequence data transfers $\boldsymbol{\Delta}$'.

1. For each query from temp, create a set of EPN $\boldsymbol{\xi} = \{\varepsilon_i, \ldots, \varepsilon_m\}$.
2. Use **Algorithm 3** to merge all EPNs from $\boldsymbol{\xi}$ to $\varepsilon = (\boldsymbol{B}, \boldsymbol{V}, \boldsymbol{E}, \boldsymbol{W})$.
3. Use **Algorithms 5, 6, 7** to generate a sequence of data transfers $\boldsymbol{\Delta}$' from ε.
4. Return the data transfer plan $\boldsymbol{\Delta}$'.

Algorithm 2 involves iterating over a sequence of query temp to create an EPN for each query. The primary loop runs n times, where n is the number of elements in temp. Therefore, the complexity of this algorithm is $O(n)$.

5.1. Merging EPNs

This section explains how a set of EPNs generated from a set of queries Q are merged into a single EPN. Merging of EPNs supports merging of common input data containers to encourage the efficiency of composite query processing.

Merging of EPNs recognizes the input data containers shared by EPNs to eliminate redundancies and group sub-diagrams into a shared structure for joining. Data containers read by different EPNs are joined into a single one for subsequent combining of other parts.

Algorithms 3 and 4 contain the description of this merging of EPNs.

Algorithm 3 Merging EPNs

Input: A stream of queries $\boldsymbol{Q} = <\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m, \ldots>$ and window size $\boldsymbol{w}$.

Output: Keep updating the sequence of data transfers $\boldsymbol{\Delta}$.

1. If a set of EPNs has more than one EPN, then take the first two EPNs $\mathcal{E}_1$, $\mathcal{E}_2$ and generate merged EPN $\mathcal{E}$' by using **Algorithm 4**.
2. Remove $\mathcal{E}_1$ and $\mathcal{E}_2$ from the set.
3. If the set is not empty, then iterate over the set and let the current EPN be $\mathcal{E}_i$.
   3.1. Use **Algorithm 4** to merge $\mathcal{E}$' with $\mathcal{E}_i$ and generate the updated merged EPN $\mathcal{E}$'.
   3.2. Remove $\mathcal{E}_i$ from the set.
4. Return the merged *EPN* $\mathcal{E}$'.

Algorithm 3 utilizes conditional logic (using if-else statements) based on an input stream of queries. However, it does not iterate over or manipulate the list elements. Because the control flow consists only of branching operations without loops or recursive calls, the complexity of this algorithm is $O(1)$.

Algorithm 4 Merging two EPNs

Input: Two EPN, $\mathcal{E}_i = (\boldsymbol{B}_i, \boldsymbol{V}_i, \boldsymbol{E}_i, \boldsymbol{W}_i)$ and $\varepsilon_j = (\boldsymbol{B}_j, \boldsymbol{V}_j, \boldsymbol{E}_j, \boldsymbol{W}_j)$.

Output: Merged EPN, $\boldsymbol{\mathcal{E}}' = (\boldsymbol{B}, \boldsymbol{V}, \boldsymbol{E}, \boldsymbol{W})$.

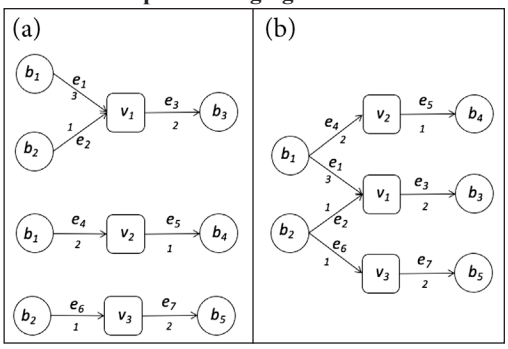
1. From both EPNs $\boldsymbol{\mathcal{E}}_i$ and $\boldsymbol{\mathcal{E}}_j$, get all common input data containers and put them into a set $\boldsymbol{B}_{temp} = \{\boldsymbol{b}_i, \ldots, \boldsymbol{b}_j\}$.
2. Set $\boldsymbol{B} = \boldsymbol{Bi} - \boldsymbol{B}_{temp}$.
3. Set $\boldsymbol{B} = \boldsymbol{B} \cup \boldsymbol{B}_j$.
4. Set $\boldsymbol{V} = \boldsymbol{V}_i \cup \boldsymbol{V}_j$.
5. Set $\boldsymbol{E} = \boldsymbol{E}_i \cup \boldsymbol{E}_j$.
6. Set $\boldsymbol{W} = \boldsymbol{W}_i \cup \boldsymbol{W}_j$.
7. Return merged $\boldsymbol{\mathcal{E}}' = (\boldsymbol{B}, \boldsymbol{V}, \boldsymbol{E}, \boldsymbol{W})$.

Similar to Algorithm 3, Algorithm 4 consists entirely of a series of if-else conditions. It does not rely on the size of any input and does not involve iterative or recursive processing. Consequently, its complexity is $O(1)$.

5.1.1. Example

The example below will use three EPNs ($\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{E}_3$) displayed in Figure 4(a). Where $\mathcal{E}_1 = (B_1, V_1, E_1, W_1)$, $\mathcal{E}_2 = (B_2, V_2, E_2, W_2)$, and $\mathcal{E}_3 = (B_3, V_3, E_3, W_3)$. First, Algorithm 3 takes $\mathcal{E}_1$ and $\mathcal{E}_2$ and generates a merged EPN, $\mathcal{E}' = (B, V, E, W)$, where $B = \{b_1, b_2, b_3, b_4\}$, $V = \{v_1, v_2\}$, $E = \{e_1, e_2, e_3, e_4, e_5\}$, and $W = \{W(e_1), W(e_2), W(e_3), W(e_4), W(e_5)\}$ by using Algorithm 5. Repeat the procedure and generate a merged EPN, as shown in Figure 4(b).

Figure 4
Example of Merging Three EPNs



5.2. Generating sequence of sets of operations

The present section explains how an integrated EPN $\varepsilon = (B, V, E, W)$ is converted into a sequence of sets of operations $\omega = < \{v_i, \ldots, v_j\}, \ldots, \{v_x, \ldots, v_y\} >$.The details can be found in Algorithm 5, which converts ε into a corresponding sequence of sets of operations, also referred to as ω.

In some cases, the operations within the EPN cannot be processed simultaneously. This is why we need to establish the order of processing for these operations. To find the best order, we calculate the profitability of each operation, and we assume the highest profit to be the priority. A "profit" is defined as the difference between the total amount of persistent storage that is allocated and that which is released for each operation from the temporary set. To estimate the profit for operation v_i, we use Equation (6).

The total amount of persistent storage released $\sum_{j=1}^{n} w_j$ is calculated by summing the weight values of the edges located on the left side of the operation v_i. The amount of persistent storage utilized in the output data containers of operation v_i is calculated as $\sum_{k=(n+1)}^{m} w_k$.

$$\rho(v_i) = \sum_{j=1}^{n} w_j - \sum_{k=n+1}^{m} w_k \tag{6}$$

If $\rho(v_i)$ is negative, then it means that the processing of the operation v_i does not contribute to any storage release. If $\rho(v_i)$ is positive, then after its processing, some storage can be released. Profit for root operations is always zero because root operations read data from persistent storage that cannot be released.

Algorithm 5 Generate a sequence of sets of operations

Input: An $\mathcal{E} = (B, V, E, W)$

Output: A sequence of sets $\boldsymbol{\omega} = < \{\boldsymbol{v_i}, \ldots, \boldsymbol{v_j}\}, \ldots, \{\boldsymbol{v_x}, \ldots, \boldsymbol{v_y}\} >$.

1. Create a new empty sequence of sets called $\boldsymbol{\omega} = <>$.
2. Finding all the root operations and adding them into a temporary set of operations $\boldsymbol{temp}_1 = \{\boldsymbol{v_i}, \ldots, \boldsymbol{v_j}\}$.
3. Finding operations which cannot be processed at the same time and put into the same group like $\boldsymbol{temp}_1 = \{\{\boldsymbol{v_i}, \ldots, \boldsymbol{v_k}\}, \ldots, \{\boldsymbol{v_j}, \ldots, \boldsymbol{v_y}\}\}$.
4. If more than one set is found in $\boldsymbol{temp}_1$, then, iterate over $\boldsymbol{temp}_1$ and let the current set of operations be $\{\boldsymbol{v_i}, \ldots, \boldsymbol{v_k}\}$.
   4.1. Compute profit for all operations from the current.
   4.2. Set the profit of root operations as zero because no temporary storage is required to release for root operations.
   4.3. Keep only one operation with the highest profit value and move the rest from $\boldsymbol{temp}_1$ to $\boldsymbol{temp}_2$.
5. Flatten $\boldsymbol{temp}_1$ as a set of sets of operations to a set of operations.
6. Append $\boldsymbol{temp}_1$ to $\boldsymbol{\omega}$, and operations from $\boldsymbol{temp}_1$ are removed from V.
7. Iterate over $\boldsymbol{temp}_1$ and let the current operation be $\boldsymbol{v_i}$.
   7.1. Select all the operations reading from $\boldsymbol{v_i}$'s output data containers.
   7.2. Append all selected operations into a temporary set of operations $\boldsymbol{temp}_2$.
8. Update $\boldsymbol{temp}_1 = \boldsymbol{temp}_2$ and set $\boldsymbol{temp}_2 = \{\}$.
9. Select all the operations from $\boldsymbol{temp}_1$ where the operations violate the processing order in EPN.
10. Move all selected operations from $\boldsymbol{temp}_1$ to $\boldsymbol{temp}_2$.
11. If V is not empty, go back to step 3.
12. Else return the resulting sequence of operations $\boldsymbol{\omega} = < \{\boldsymbol{v_i}, \ldots, \boldsymbol{v_j}\}, \ldots, \{\boldsymbol{v_x}, \ldots, \boldsymbol{v_y}\} >$.

Initially, the algorithm starts from all root operations and includes them in a temporary set of operations $\{v_i, \ldots, v_j\}$. Operations that cannot be processed simultaneously are included in the same set. It implies that operations accessing the same data container can be grouped together. The same holds for the operations that write to a shared data container. At the end of the first step, the temporary set of operations is restructured into a set composed of sets of operations.

Next, the algorithm eliminates from each set the operations with minimal profits. It retains only the operation that calculates the highest profit. Equation (6) is used to compute the profit of each operation. In the next step, the algorithm finds the operation with the highest profit. All remaining operations are eliminated from the temporary set. At the end of the present step, the temporary set of operations is appended to a sequence of sets ω.

In the next steps, the Algorithm adds to the temporary set the operations that retrieve data from the output data containers associated with operations already present in ω. Such process continues until all high-level operations have been successfully integrated into the sequence of sets ω. At the end, the algorithm returns the sequence of sets of operations $\omega = < \{v_i, \ldots, v_j\}, \ldots, \{v_x, \ldots, v_y\} >$.

Algorithm 5, like Algorithm 3 and Algorithm 4, uses if-else statements that run in constant time. Therefore, its complexity is $O(1)$.

5.2.1. Example

The present example considers the merged EPN shown in Figure 5. All weight values in the merged EPN are equal to 1 because the EPN is related to serial processing only, i.e. one data transfer at a time.

At the beginning, the algorithm creates a set $temp_1 = \{v_1, v_2, v_3, v_4\}$ that consists of all root operations. A set of operations, namely $\{v_1, v_2\}$ and $\{v_3, v_4\}$ cannot be executed simultaneously. It is because v_1 and v_2 are reading from the same input data container b_1 and v_3 and v_4 are writing to the same data container b_6. Information about the operations that cannot be processed simultaneously is used to transform the set $temp_1$ into a set of sets of operations $temp_1 = \{\{v_1, v_2\}, \{v_3, v_4\}\}$.

Next, the algorithm iterates over the sets included in $temp_1$.Let the current set be $\{v_1, v_2\}$. The algorithm calculates the profits $\rho(v_1) = 0$ and $\rho(v_2) = 0$. At this point, only one operation is retained, the updated set becomes $temp_1 = \{\{v_1\}, \{v_3, v_4\}\}$. The removed operation, v_2, is then moved to $temp_2 = \{v_2\}$. This procedure is repeated for a set $\{v_3, v_4\}$. After all iterations are completed, $temp_1 = \{\{v_1\}, \{v_3\}\}$ and $temp_2 = \{v_2, v_4\}$.

Then, the algorithm flattens the set to form $temp_1 = \{v_1, v_3\}$. Next, it appends $temp_1$ to ω, resulting in $\omega = <\{v_1, v_3\}>$. Then, the algorithm processes the operations in $temp_1$. Let the current operation be v_1. It reads v_1's output data container, b_4, and adds the operation v_5 to $temp_2 = \{v_2, v_4, v_5\}$. Such steps are repeated for the subsequent operation v_3. At the end of the iterations, $temp_2 = \{v_2, v_4, v_5, v_6\}$.

The next step moves all operations from $temp_2$ to $temp_1$ creating $temp_1 = \{v_2, v_4, v_5, v_6\}$. All further operations that create violations in the order of processing in EPN are removed. Operation v_6 has to be removed from $temp_1$ because v_4 is not in ω. So, v_6 can be processed only after v_2, v_3, and v_4 are processed. That is why, v_2, v_3, and v_4 have to be ω before v_6 can be processed. This is repeated until all operations of the merged EPN are included in ω. Finally, the update $\omega = <\{v_1, v_3\}, \{v_2, v_4, v_5\}, \{v_7, v_8\}, \{v_6\}, \{v_9\}>$.

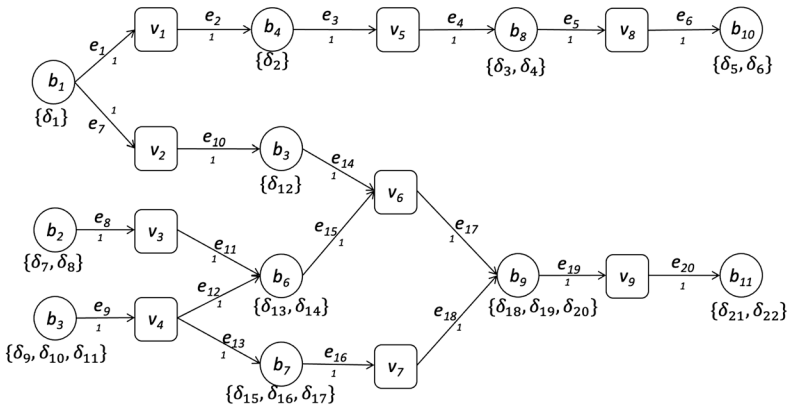
5.3. Generating sequence of operations

A sequence of operations generated by Algorithm 6 is created by prioritizing the operations that release the larger amounts of storage first. The operations that release the smaller amounts of storage are appended later to the sequence. Processing of the operations that release amounts of storage must be done first to earlier reuse such storage for subsequent operations.

At the beginning, Algorithm 6 calculates the profit for each operation in each set of operations within ω, using Equation (6). Then the calculated profits are used to organize the operations in descending order of profitability and save them in a sequence ψ. We assume that releasing larger volumes of faster storage earlier improves performance later. Such process is repeated for each set of operations in ω, the sets of operations into a single sequence of sets of operations.

Algorithm 6 processes a sequence of set operations, denoted as $\boldsymbol{\omega}$, where each element in the sequence may represent a set of varying sizes. For each set, the algorithm checks a condition, and if satisfied, iterates over the elements of that set. In the worst-case scenario where the condition is always met and each set contains m elements, the algorithm performs $O(n * m)$ operations, where n is the number of sets in $\boldsymbol{\omega}$.

Figure 5
Illustration of merged EPN



$\delta_1 = (q_1, l_0, d_1, 100, \bullet)$ $\delta_7 = (q_2, l_0, d_2, 80, \bullet)$ $\delta_{13} = (q_2, l_3, d_2, 60, \varepsilon)$ $\delta_{19} = (q_2, l_3, d_3, 40, \varepsilon)$
$\delta_2 = (q_1, l_2, d_3, 100, \varepsilon)$ $\delta_8 = (q_2, l_0, d_3, 120, \bullet)$ $\delta_{14} = (q_2, l_3, d_2, 40, \varepsilon)$ $\delta_{20} = (q_2, l_2, d_3, 60, \varepsilon)$
$\delta_3 = (q_1, l_2, d_1, 40, \varepsilon)$ $\delta_9 = (q_2, l_1, d_1, 100, \bullet)$ $\delta_{15} = (q_2, l_3, d_1, 80, \varepsilon)$ $\delta_{21} = (q_2, l_3, d_3, 150, \varepsilon)$
$\delta_4 = (q_1, l_2, d_2, 20, \varepsilon)$ $\delta_{10} = (q_2, l_1, d_3, 130, \bullet)$ $\delta_{16} = (q_2, l_3, d_2, 70, \varepsilon)$ $\delta_{22} = (q_2, l_3, d_1, 50, \varepsilon)$
$\delta_5 = (q_1, l_2, d_1, 10, \varepsilon)$ $\delta_{11} = (q_2, l_1, d_2, 70, \bullet)$ $\delta_{17} = (q_2, l_2, d_1, 50, \varepsilon)$
$\delta_6 = (q_1, l_2, d_1, 10, \varepsilon)$ $\delta_{12} = (q_2, l_3, d_1, 60, \varepsilon)$ $\delta_{18} = (q_2, l_3, d_1, 100, \varepsilon)$

Algorithm 6 Generate a processing plan $\boldsymbol{\psi}$.

Input: A sequence of sets of operations $\boldsymbol{\omega}$ from Algorithm 5

Output: A sequence of operations $\boldsymbol{\psi} = < \boldsymbol{v_i}, \ldots, \boldsymbol{v_y} >$.

1. Initiate an empty sequence called $\boldsymbol{\psi} = < >$.
2. Iterate over $\boldsymbol{\omega}$ and let the current set be $\{\boldsymbol{v_x}, \ldots, \boldsymbol{v_y}\}$.
   - 2.1. If the current set has more than one operation.
     - 2.1.1. Iterate over the current set $\{\boldsymbol{v_x}, \ldots, \boldsymbol{v_y}\}$ and let the current operation be $\boldsymbol{v_x}$.
     - 2.1.2. Compute the profit $\boldsymbol{\rho}(\boldsymbol{v_x})$ by using Equation (4).
     - 2.1.3. Arrange the descending order of operations in the current set according to their profit value.
     - 2.1.4. Append the operations to $\boldsymbol{\psi}$.
   - 2.2. Else append an operation from the current set into $\boldsymbol{\psi}$.
3. Return the result $\boldsymbol{\psi} = < \boldsymbol{v_i}, \ldots, \boldsymbol{v_y} >$.

5.3.1. Example

In this example, we consider a set sequence $\omega = <\{v_1, v_3\}, \{v_2, v_4, v_5\}, \{v_7, v_8\}, \{v_6\}, \{v_9\}>$ taken from Example 5.2.1. Algorithm 6 iterates over the sets in ω. Let the current set be $\{v_1, v_3\}$. Since v_1 and v_3 are root operations with zero profit. Both operations are appended to $\psi = <v_1, v_3>$.

In the second iteration, the current set $\{v_2, v_4, v_5\}$ includes more than one operation. Algorithm 6 uses Equation (6) to compute the profit of each operation. The results are $\rho(v_2) = 0$, $\rho(v_4) = 0$, and $\rho(v_5) = 40$. Then, the operations are arranged in a sequence $<v_5, v_2, v_4>$ on the descending order of their profit values. The sequence is appended to $\psi = <v_1, v_3, v_5, v_2, v_4>$.

In the third iteration, the current set is $\{v_7, v_8\}$. Algorithm 6 calculates the profit of each operation. The results are $\rho(v_7) = 100$ and $\rho(v_8) = 40$. After sorting the operations in descending order by their profits, a sequence of operations $<v_7, v_8>$ is appended to the sequence $\psi = <v_1, v_3, v_5, v_2, v_4, v_7, v_8>$.

The fourth and fifth iterations do not compute profit because the last two sets in ω are single-element sets. The operations are appended to the sequence $\psi = <v_1, v_3, v_5, v_2, v_4, v_7, v_8, v_6, v_9>$.

5.4. Generating sequence of data transfers

In this section, we present Algorithm 7 that constructs a sequence of data transfers $\Delta = < v_i : \delta_j, \ldots, v_x : \delta_y >$. Observe, that each data transfer in Δ must be associated with the corresponding operation as multiple operations can process on the same data transfer. The Algorithm 7 is based on a sequence of operations $\psi = < v_i, \ldots, v_j >$ and EPN $\mathcal{E} = (B, V, E, W)$.

In order to construct the sequence of data transfers Δ, the algorithm identifies the input and output data containers for each operation in ψ. It extracts data transfers from the input containers and adds them to the transfer plan and repeats this process for the output containers. The systematic applications to all operations in ψ, guarantees a full derivation of the sequence of data transfers. The result is the data transfer plan Δ for a single EPN via a single data transfer process.

Algorithm 7 executes a straightforward iteration over a sequence of operations represented by $\boldsymbol{\psi}$. Therefore, the complexity of this algorithm is $O(n)$, where n is the number of operations in the input sequence.

5.4.1. Example

The present example uses a sequence of operations $\psi = <v_1, v_3, v_5, v_2, v_4, v_7, v_8, v_6, v_9>$ created in Example 5.3.1 and the EPN from Figure 5.

Algorithm 7 Generating sequence of data transfer plan $\boldsymbol{\Delta}$

Input: A sequence of operations $\boldsymbol{\psi}$ from **Algorithm 6**, and a merged $\mathcal{E} = (\boldsymbol{B}, \boldsymbol{V}, \boldsymbol{E}, \boldsymbol{W})$ from **Algorithm 3**.

Output: A data transfer plan $\boldsymbol{\Delta} = < \boldsymbol{v_1} : \boldsymbol{\delta_x}, \ldots, \boldsymbol{v_j} : \boldsymbol{\delta_y} >$.

1. Initialize an empty sequence for the data transfer plan, denoted by $\boldsymbol{\Delta} = < >$.
2. Iterate over $\boldsymbol{\psi}$ and let the current operation be $\boldsymbol{v_i}$.
   - 2.1. Use as set of edges $\boldsymbol{E}$ in the merged $\mathcal{E}$ to find all the input data containers related to operation $\boldsymbol{v_i}$. Retrieve all the data transfers associated with those data containers. For each data transfer, create a pair in the format $\boldsymbol{v_i} : \boldsymbol{\delta_x}$ and append it to the sequence of data transfers $\boldsymbol{\Delta}$.
   - 2.2. Use as set of edges $\boldsymbol{E}$ in the merged $\mathcal{E}$ to find all the output data containers related to operation $\boldsymbol{v_i}$. Retrieve all the data transfers associated with those data containers. For each data transfer, create a pair in the format $\boldsymbol{v_i} : \boldsymbol{\delta_y}$ and append it to the sequence of data transfers $\boldsymbol{\Delta}$.
3. Finally, return the data transfer plan as $\boldsymbol{\Delta} = < \boldsymbol{v_1} : \boldsymbol{\delta_x}, \ldots, \boldsymbol{v_j} : \boldsymbol{\delta_y} >$

At the beginning, Algorithm 7 creates an empty, $\Delta = < >$. Then, the algorithm iterates over the sequence of operations ψ.

Let the current operation be v_1.The operation has exactly one input data container b_1. The data container is associated with a set of data transfers, which include δ_1. Therefore, all data transfers from b_1 are added to $\Delta = <v_1:\delta_1>$.

The operation v_1 has one output data container b_4, which is supposed to be filled with a data transfer δ_2. After δ_2 is added to Δ, it becomes $<v_1:\delta_1, v_1:\delta_2>$. The same procedure is repeated for all operations in the sequence ψ. At the end, an updated data transfer plan is a sequence of the single data transfers $\Delta = <v_1:\delta_1, v_1:\delta_2, v_3:\delta_7, v_3:\delta_8, v_3:\delta_{13}, v_5:\delta_2, v_5:\delta_3, v_5:\delta_4, v_2:\delta_1, v_2:\delta_{12}, v_4:\delta_9, v_4:\delta_{10}, v_4:\delta_{11}, v_4:\delta_{15}, v_4:\delta_{16}, v_4:\delta_{17}, v_4:\delta_{14}, v_7:\delta_{15}, v_7:\delta_{16}, v_7:\delta_{17}, v_7:\delta_{19}, v_7:\delta_{20}, v_8:\delta_3, v_8:\delta_4, v_8:\delta_5, v_8:\delta_6, v_6:\delta_{12}, v_6:\delta_{13}, v_6:\delta_{14}, v_6:\delta_{18}, v_9:\delta_{18}, v_9:\delta_{19}, v_9:\delta_{20}, v_9:\delta_{21}, v_9:\delta_{22}>$.

6. Experiments

We run the experiments within a simulated scenario of a four-tiered persistent storage architecture. The performance parameters of reads and writes as operations on persistent storage reflect characteristics of real-life market devices, see Appendix A.1.

We discussed three different scenarios presented in Experiment 1, Experiment 2, and Experiment 3. In each experiment, we employ sequential data transfer processes between the persistent storage devices to study the efficacy of different scheduling mechanisms.

The TPC-H benchmark database [22] is utilized to generate the queries. We construct 15 query templates out of 22 query templates available in this work [23]. The data is stored across various storage tiers, as shown in Table 5 of Appendix A.2.

This experiment measures how long it takes to process different queries with various storage setups using a custom-made simulator. Although we didn't use actual storage systems, the simulator was designed to closely reflect real-world performance by including realistic storage and processor features.

Experiment 1 runs a stream of queries $<Q_1, Q_2, Q_3, Q_4, Q_5>$. Experiment 2 has a stream of queries consisting of $<Q_6, Q_7, Q_8, Q_9, Q_{10}>$, and Experiment 3 has $<Q_{11}, Q_{12}, Q_{13}, Q_{14}, Q_{15}>$.

The experiments examine the impact of one- to four-tiered persistent storage infrastructure on query processing when random allocation and serial allocation plans proposed in the paper are employed.

Every query is executed first in a single-tiered infrastructure, then in two-, three-, and four-tiered infrastructures. We measure the times of

query processing at each step. We examine an optimized serial allocation plan for the four-tiered infrastructure too. The data processing times of the data are listed in Tables 1, 2, and 3. The total processing time of all queries in each stream is listed in Table 4.

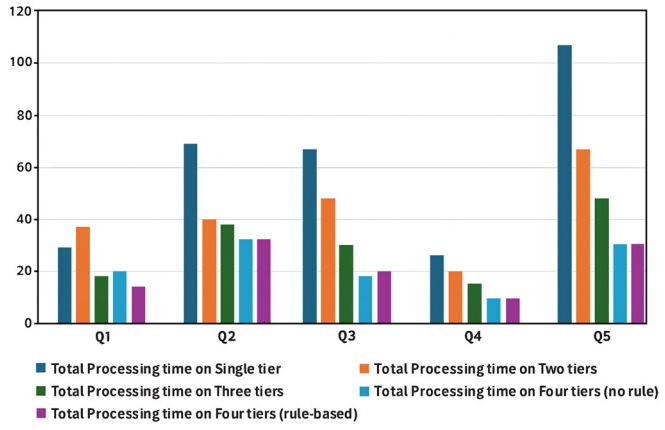
6.1. Experiment results

Experiment 1: The queries Q_1, Q_2, Q_3, Q_4, and Q_5 are processed across different tiers, with varying time units. The optimized allocation plan, or serial execution plan, consistently outperforms setups without allocation rules, and performance generally improves with more tiers. While processing time decreases with additional tiers, some exceptions exist. The results are presented in Table 1 and Figure 6, where the Y-axis spans 0 to 120 time units and the X-axis lists queries Q_1 to Q_5.

Table 1
Experiment 1 – processing time for each query

	Total processing time on				
Queries	Single tier	Two tiers	Three tiers	Four tiers (no rule)	Four tiers (rule-based)
Q_1	29	37	18	20	14
Q_2	69	40	38	32	32
Q_3	67	48	30	18	20
Q_4	26	20	15	9	9
Q_5	107	67	48	30	30
Total	298	212	149	109	105

Figure 6
Experiment 1 – processing time for each query



Experiment 2: The queries Q_6, Q_7, Q_8, Q_9, and Q_{10} are processed across different tiers. As in Experiment 1, the optimized allocation plan (serial execution) outperforms scenarios without allocation rules, with improved performance as tiers increase. Although processing time generally decreases with more tiers, there are exceptions. Results are shown in Table 2 and Figure 7, where the Y-axis ranges from 0 to 250 time units and the X-axis covers queries Q_6 to Q_{10}.

Experiment 3: The queries Q_{11}, Q_{12}, Q_{13}, Q_{14}, and Q_{15} are processed across different tiers. An optimized allocation plan outperforms random allocation, with processing time decreasing as the number of tiers increases. The results are displayed in Table 3 and Figure 8, where the Y-axis ranges from 0 to 140 time units.

Table 2
Experiment 2 – processing time for each query

	Total processing time on				
Queries	Single tier	Two tiers	Three tiers	Four tiers (no rule)	Four tiers (rule-based)
Q_6	48	29	20	15	11
Q_7	62	42	31	27	23
Q_8	199	80	46	34	34
Q_9	150	58	39	27	28
Q_{10}	83	37	38	30	18
Total	542	246	174	133	114

Figure 7
Experiment 2 – processing time for each query

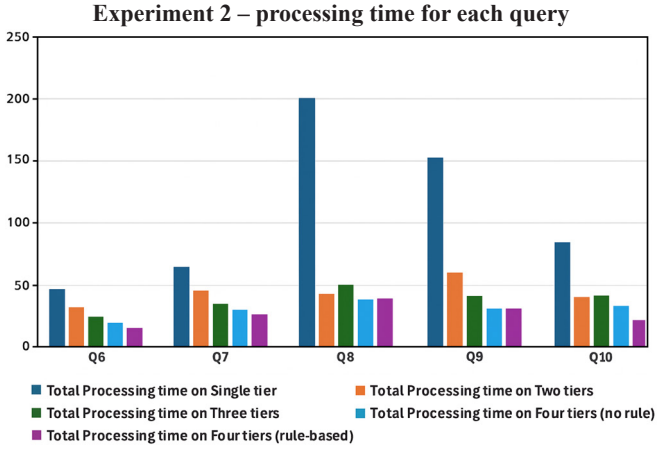
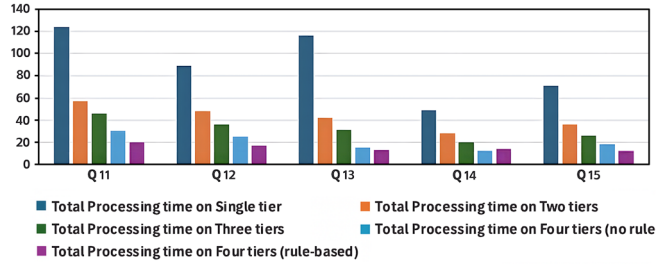


Table 3
Experiment 3 – processing time for each query

	Total processing time on				
Queries	Single tier	Two tiers	Three tiers	Four tiers (no rule)	Four tiers (rule-based)
Q_{11}	124	57	46	30	20
Q_{12}	89	48	36	25	17
Q_{13}	116	42	31	15	13
Q_{14}	49	28	20	12	14
Q_{15}	71	36	26	18	12
Total	449	211	159	100	76

Figure 8
Experiment 3 – processing time for each query



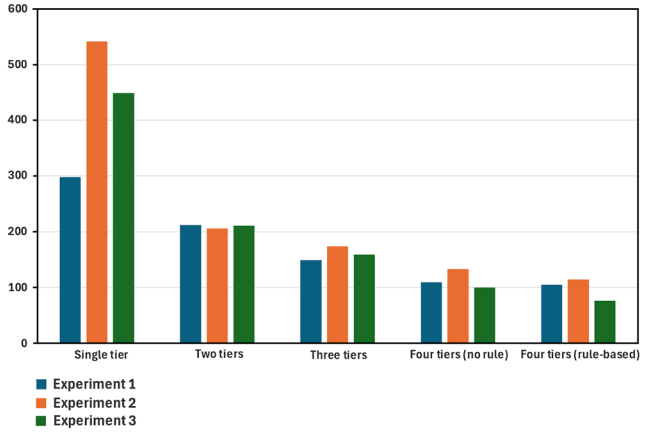
Total processing time for each experiment: The total processing time for each experiment is shown in Table 4, using serial processing. An optimized allocation plan consistently outperforms random allocation, and processing time generally decreases with more tiers. Figure 9 illustrates this, with the Y-axis ranging from 0 to 600 time units and the X-axis showing processing times for one to four tiers in different configurations across three experiments. It is important to note that lower time units mean better performance, showing how efficient each configuration is. This means that specific queries are processed faster.

The experimental results show that a rule-based online scheduling system for data transfers in multi-tiered persistent storage is effective. When compared to a single-tier setup, a four-tier architecture with rule-based scheduling led to total processing times that were up to 83.1% shorter for certain queries. The shift from one tier to two tiers usually results in significant performance improvements. Although the gains from two to three tiers vary, three tiers typically offer better performance than two. Overall, the serial allocation plan consistently outperforms other strategies.

Table 4
Total processing time for three experiments

	Total processing time on				
Experiments	**Single tier**	**Two tiers**	**Three tiers**	**Four tiers (no rule)**	**Four tiers (rule-based)**
Experiment 1	298	212	149	109	105
Experiment 2	542	246	174	133	114
Experiment 3	449	211	159	100	76

Figure 9
Overall processing time for each experiment



7. Summary, Conclusions and Future Work

7.1. Summary

The main objective of this research is to improve the efficiency of processing the streams of queries in the environments of MTS. The main objective is to find the new online algorithms for optimization of data transfer plans between the tiers of persistent storage. We propose a new notation of EPNs to improve the understanding of data flows and operational dependencies. Then, we concentrate on creating sequential processing plans that use the unique features of MTS architectures such as data distributed across multiple devices at the different persistent storage levels. Such plans ensure the efficient management of data in MTS based on access patterns and operational needs of individual queries. We propose an algorithm that dynamically updates the processing plan and adjusts data transfer sequences in real time.

The outcomes of experiments confirm that optimized data placement strategies significantly outperform random data allocation strategies in terms of processing and responsiveness. The processing times decrease with an increase in the number of tiers, and the proposed serial allocation plan outperforms the other strategies.

7.2. Conclusions

This research presents a novel approach to optimizing processing plan in multi-tier storage with a rule-based scheduling decision. Unlike traditional static tiering or manual allocation methods, the proposed model enables dynamic, formalized scheduling that adapts to access patterns and system characteristics in real time. Dynamic adaptation in data processing and efficient scheduling significantly reduce processing time. The solutions outlined in this work improve data processing in MTS environments.

The practical solutions provided in this work enhance data processing effectiveness in MTS environments. We also find that utilization of EPNs makes it simpler to identify serialization opportunities across MTS environments. This capability improves how we process data, making the system work better and faster.

Through a simulation-based evaluation using realistic device parameters, the study demonstrates that the model can significantly reduce query processing time up to 72% compared to baseline configurations while maintaining scalability and adaptability. Experimental results show that online data processing methods must dynamically adapt to fit the growing and diverse data landscape. As the data continues to grow in complexity and diversity, it becomes increasingly critical for processing systems to develop in response to these changes. The new scheduling algorithms proposed in the paper react more appropriately to evolving data dynamics, reducing resource consumption and processing time. The proposed online scheduling methods prove that optimized allocation plans consistently beat random allocation.

7.3. Future work

One of the most significant problems in online data movement scheduling in multi-tiered persistent storage is a way how to divide large amounts of data into smaller high-level partitions. It is because storage allocations need to change while query processing is being performed. A question as to which allocations are to be prioritized at the higher-levels is a very crucial consideration.

Future research would enhance real-world MTS methodologies, including optimizing query processing prioritization and storage allocation. Pre-processing techniques to pre-optimize queries before scheduling data transfer could be the solutions that improve efficiency and scalability.

The scheduling algorithms proposed in the paper address some efficiency issues, but opportunities remain to reduce idle time and tailor storage based on query requirements. Integrating ML into scheduling process may further improve decision-making by analyzing historical data in real-time.

Moreover, in the future, the current method of scheduling storage by adding ML techniques and real-time data management can be improved. Recent developments have shown that using predictive models and reinforcement learning can help in deciding where to place data across different types of storage [22]. These methods can analyze past access patterns and system workloads to figure out the best storage tier for each data object, which will enhance performance and make better use of resources.

Ethical Statement

This study does not contain any studies with human or animal subjects performed by any of the authors.

Conflicts of Interest

The authors declare that they have no conflicts of interest to this work.

Data Availability Statement

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

Author Contribution Statement

Nan Noon Noon: Conceptualization, Methodology, Validation, Formal analysis, Investigation, Writing – original draft, Writing – review & editing, Visualization. **Janusz R. Getta:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Writing – original draft, Writing – review & editing, Visualization, Supervision. **Tianbing Xia:** Conceptualization, Methodology, Validation, Formal analysis, Investigation, Writing – original draft, Writing – review & editing, Visualization, Supervision.

References

[1] Chang, H. P., Su, W. M., & Chang, D. W. (2021). A cross-layered readahead architecture for multi-tiered storage systems. In *2021 IEEE Symposium on Computers and Communications*, 1–3. https://doi.org/10.1109/ISCC53001.2021.9631462

[2] Shu, J. (2024). *Data storage architectures and technologies*. Singapore: Springer. https://doi.org/10.1007/978-981-97-3534-1

[3] Li, J., Yan, H., & Zhang, Y. (2022). Efficient identity-based provable multi-copy data possession in multi-cloud storage. *IEEE Transactions on Cloud Computing*, *10*(1), 356–365. https://doi.org/10.1109/TCC.2019.2929045

[4] Batrouni, M. (2020). A hybrid architecture for tiered storage with fuzzy logic and AutoML. In *Cooperative Design, Visualization, and Engineering: 17th International Conference*, 67–74. https://doi.org/10.1007/978-3-030-60816-3_8

[5] Yi, X., Du, H., Wang, Y., Zhang, J., Li, Q., & Xue, C. J. (2025). ArtMem: Adaptive migration in reinforcement learning-enabled tiered memory. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, 405–418. https://doi.org/10.1145/3695053.3731001

[6] Shi, H., Arumugam, R. V., Foh, C. H., & Khaing, K. K. (2013). Optimal disk storage allocation for multi-tier storage system. *IEEE Transactions on Magnetics*, *49*(6), 2603–2609. https://doi.org/10.1109/TMAG.2013.2250936

[7] Wang, F., Wu, Y., & Huang, F. (2020). Rio: A personal storage system in multi-device and cloud. *The Journal of Supercomputing*, *76*(9), 7519. https://doi.org/10.1007/s11227-020-03309-8

[8] Sivarajah, U., Kamal, M. M., Irani, Z., & Weerakkody, V. (2017). Critical analysis of big data challenges and analytical methods. *Journal of business research*, *70*, 263–286. https://doi.org/10.1016/j.jbusres.2016.08.001

[9] Chen, N., Kang, W., Kang, N., Qi, Y., & Hu, H. (2022). Order processing task allocation and scheduling for e-order fulfilment. *International Journal of Production Research*, *60*(13), 4253–4267. https://doi.org/10.1080/00207543.2021.2018140

[10] Khan, Z. I., Khan, M., & Shah, S. N. M. (2025). Agent-based Adaptive Dynamic Round Robin (AADRR) scheduling algorithm. *IEEE Access*, *13*, 18308–18324. https://doi.org/10.1109/ACCESS.2025.3534031

[11] Zhang, C., & Lu, Y. (2021). Study on artificial intelligence: The state of the art and future prospects. *Journal of Industrial Information Integration*, *23*, 100224. https://doi.org/10.1016/j.jii.2021.100224

[12] Yang, Z., Hoseinzadeh, M., Andrews, A., Mayers, C., Evans, D. T., Bolt, R. T., ..., & Swanson, S. (2017). AutoTiering: Automatic data placement manager in multi-tier all-flash datacenter. In *2017 IEEE 36th International Performance Computing and Communications Conference*, 1–8. https://doi.org/10.1109/PCCC.2017.8280433

[13] Lv, J., & Huang, B. (2024). A Petri-net-based anytime A* search for scheduling resource allocation systems. *IEEE Transactions on Industrial Informatics*, *20*(2), 2865–2872. https://doi.org/10.1109/TII.2023.3296909

[14] Silberschatz, A., Galvin, P. B., & Gagne, G. (2006). *Operating system concepts: Windows XP update*. USA: John Wiley & Sons.

[15] Noon, N. N., Getta, J. R., & Xia, T. (2022). Scheduling parallel data transfers in multi-tiered persistent storage. In *Recent Challenges in Intelligent Information and Database Systems: 14th Asian Conference*, 437–449. https://doi.org/10.1007/978-981-19-8234-7_34

[16] Peng, Y., Xu, W., & Xiong, Y. (2021). Critical analysis of big data challenges on similar sequential algorithm based data search. *Journal of Applied Science and Engineering*, *24*(4), 653–659.

[17] Spiceworks. (2020). *Data storage trends in 2020 and beyond*. https://www.spiceworks.com/marketing/reports/storage-trends-in-2020-and-beyond/

[18] Wang, S., Lu, Z., Cao, Q., Jiang, H., Yao, J., Dong, Y., ..., & Xie, C. (2022). Exploration and exploitation for buffer-controlled HDD-writes for SSD-HDD hybrid storage server. *ACM Transactions on Storage*, *18*(1), 6. https://doi.org/10.1145/3465410

[19] Qin, H., Ding, W., Xu, L., & Ruan, C. (2024). Petri-net-based charging scheduling optimization in rechargeable sensor networks. *Sensors*, *24*(19), 6316. https://doi.org/10.3390/s24196316

[20] Jiang, W., Zhou, K.-Q., Sarkheyli-Hägele, A., & Zain, A. M. (2022). Modeling, reasoning, and application of fuzzy Petri net model: A survey. *Artificial Intelligence Review*, *55*(8), 6567–6605. https://doi.org/10.1007/s10462-022-10161-0

[21] Janicki, R., Kleijn, J., Koutny, M., & Mikulski, Ł. (2022). *Paradigms of concurrency: Observations, behaviours, and systems—A Petri net view*. Germany: Springer. https://doi.org/10.1007/978-3-662-64821-6

[22] Nambiar, R., & Poess, M. (Eds.). (2024). *Performance evaluation and benchmarking: 15th TPC Technology Conference*. Switzerland: Springer Nature. https://doi.org/10.1007/978-3-031-68031-1

[23] UserBenchmark. (2025). *UserBenchmark – Storage device benchmarks*. https://www.userbenchmark.com/

[24] TPC. (2025). *TPC download current specs*. https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp

How to Cite: Noon, N. N., Getta, J. R., & Xia, T. (2026). Efficient Online Scheduling of Data Transfers in Multi-Tiered Persistent Storage. *Journal of Data Science and Intelligent Systems*. https://doi.org/10.47852/bonviewJDSIS62025078

Appendix A

1. Multi-tiered Persistent Storage

A sequence of tiers $\mathcal{L} = <l_0, l_1, l_2, l_3>$ is used in the experiments described in this paper.

The tiers l_0, l_1, l_2, and l_3 are comprised of multiple devices, as outlined below:

1) l_0: The tier is implemented as three identical HDD devices denoted as $D_0 = \{d_1, d_2, d_3\}$. All devices provide the same performance level, with an estimated write speed of 230 MB/s and a read speed of 240 MB/s.
2) l_1: The tier is implemented as three SSD devices, denoted as $D_1 = \{d_1, d_2, d_3\}$, all of which have identical performance levels, with an estimated write speed of 530 MB/s and a read speed of 560 MB/s.
3) l_2: The tier is implemented as three high-speed SSD devices denoted as $D_2 = \{d_1, d_2, d_3\}$, all of which have identical performance, with an estimated write speed of 950 MB/s and a read speed of 1000 MB/s.
4) l_3: The tier is implemented as one NVMe device denoted as $D_3 = \{d_1\}$, with an estimated write speed of 4500 MB/s and a read speed of 5000 MB/s.

The read and write speeds used in the simulator for HDD, SSD, high-speed SSD, and NVMe devices were selected to reflect typical performance values observed in real-world consumer and enterprise-grade hardware. Although no specific model was used as a reference, these values closely align with benchmark ranges reported by well-known testing platforms like UserBenchmark [23].

2. Sample Database

The TPC-H benchmark database serves as the foundation for a sample database used in the experiments and query configuration. Of the 22 available queries in TPC [24], 15 queries are used in the experiments. Table 5 shows how the size, number of rows, and storage location of each table affect the time it takes for queries to run. Large tables like Lineitem (up to 24 million rows, 2,563 MB) and Orders (up to 10 million rows, 992 MB) are stored in $l_1.d_1$ and $l_1.d_2$, which are slower storage areas. This increases the input/output (I/O) costs for queries that need to scan or join data. In contrast, small reference tables like Nation and Region (less than 1 MB) are stored in l_0, which uses faster HDD storage. How table partitions are spread across devices, such as Lineitem being in d_1, d_2, and d_3, helps access data more quickly.

Table 5
Relational table locations and estimated sizes

Table name	Location	Number of rows	Size (MB)	Row size (bytes)
Lineitem	$l_1.d_1$	24,000,000	2,563	112
Lineitem	$l_1.d_2$	24,000,000	2,563	112
Lineitem	$l_1.d_3$	12,012,150	1,283	112
Orders	$l_1.d_2$	10,000,000	992	104
Orders	$l_1.d_3$	5,000,000	495	104
Orders	$l_1.d_2$	10,000,000	992	104
Customer	$l_0.d_3$	1,500,000	256	179
Nation	$l_0.d_1$	250	<1	128
Region	$l_0.d_2$	50	<1	124
Part	$l_0.d_1$	2,000,000	296	155
Partsupp	$l_0.d_2$	8,000,000	1099	144
Supplier	$l_0.d_2$	100,000	15	159