American Journal of Information Science and Technology
2024; 8(3): 84-97
http://www.sciencepublishinggroup.com/j/ajist
doi: 10.11648/j.ajist.20240803.14
ISSN: 2640-057X (Print); ISSN: 2640-0588 (Online)

Optimization of Parallel Data Transfers in Multi-Tiered Persistent Storage

Nan Noon Noon*, Janusz Roman Getta, Tianbing Xia

School of Computing and Information Technology, University of Wollongong, Wollongong, Australia

Email address:

jrg@uow.edu.au (Janusz Roman Getta), txia@uow.edu.au (Tianbing Xia), nnn326@uowmail.edu.au (Nan Noon Noon)
*Corresponding author

To cite this article:

Nan Noon Noon, Janusz Roman Getta, Tianbing Xia. (2024). Optimization of Parallel Data Transfers in Multi-Tiered Persistent Storage. *American Journal of Information Science and Technology, 8*(3), 84-97. https://doi.org/10.11648/j.ajist.20240803.14

Received: 1 May 2024; **Accepted:** 27 May 2024; **Published:** 29 September 2024

Abstract: A logical model of multi-tiered persistent storage provides a view of data where all available storage resources are distributed over a number of levels depending on the data transfer parameters and capacities. The efficient parallelization of data transfers in multi-tiered persistent storage is a significant challenge for a pipelined data processing model. This work examines a category of database applications implemented as sequences of operations that transfer data between the levels of multi-tiered persistent storage. The concept of EPN: Extended Petri Nets represents how database applications can be processed in parallel. A proposed transformation involves converting EPN into sequences of parallel data transfers. Additionally, a method is demonstrated for partitioning these sequences of data transfers, with the goal of reducing the total number of conflicts when data transfers occur between the levels of multi-tiered persistent storage. The paper proposes new rule-based algorithms for scheduling parallel data transfers that minimize total data transfer time. The objectives of the new algorithms are to evenly distribute the workload among the data transfer processes and reduce their idle time. Several experiments have confirmed the effectiveness of the new algorithms in generating parallel data transfer plans.

Keywords: Multi-tiered Persistent Storage, Scheduling, Parallel Data Processing, Performance Tuning, Database Management Systems

1. Introduction

The progress in new data analysis techniques means that commercial organizations need reliable, high-capacity storage devices for the large volumes of data they generate. Different types of storage devices are available, both on-site and in the cloud, which together form the *multi-tiered view* of persistent storage [2, 3]. In this setup, data is spread across different storage levels, each with different capacities and performance. Typically, higher storage levels are more expensive but offer better performance. Data processing involves moving data between these levels, with applications competing for access to the best-performing levels. Optimizing parallel data transfers in multi-tiered storage is crucial for overall performance.

This study focuses on scheduling parallel data transfers between different storage levels. It looks at a model where data flows between operations in a graph. These operations read and process data, passing the results to other operations. The assumption is that data can be read from and written to different storage levels at the same time, enabling parallel transfers. However, due to the physical properties of multi-tiered storage, simultaneous access to the same level is not possible, leading to conflicts and delays when processing multiple applications at once. Therefore, careful scheduling of data transfers between storage levels is necessary.

The scheduling algorithm introduced in this study aims to minimize the total processing time for a given set of applications. It works to evenly balance the workloads of individual data transfer processors and reduce their idle times. The key research contributions of this paper are as follows:

1. It assumes that database applications are implemented as sequences of operations on data. It demonstrates how to convert these sequences of operations into sets of data transfers between the levels of multi-tiered storage.
2. The study shows how to partition sets of data transfers

to reduce the total number of conflicts between them.
3. New rule-based algorithms are proposed for scheduling data transfers. The goal is to minimize total processing time, evenly distribute the workload among data transfer processors, and reduce processor idle time.
4. The pipelined data processing model can be efficiently implemented in a multi-tiered persistent storage model.

The paper is organized as follows:

1. Section 2 provides an overview of previous research on scheduling data transfers.
2. Section 3 presents a model of multi-tiered persistent storage and explains the concepts utilized in this work.
3. Section 4 describes the generation of a processing plan using information from an Extended Petri Net (EPN).
4. The scheduling algorithms and resource allocation for parallel data processing are discussed in Section 5.
5. Section 6 includes the results of the experiments.
6. Finally, Section 7 concludes the paper.

2. Previous Works

A scheduling algorithm is crucial for optimizing task and job processing on multicomputer systems. Rule-based scheduling methods have been applied to parallel manufacturing machines and parallel computing systems [4, 5]. Priority scheduling rules algorithms, such as FCFS:First Come First Serve, SPT:Shortest Processing Time, LPT:Longest-processing-time-first, and random, were proposed in a previous study [6]. This research also investigated resource distribution across single and multiple processor systems and presented solutions to the task scheduling problem under the LogP model [7], featuring theoretical and experimental results.

Many organizations are currently implementing extensive relational databases to manage operational and historical information. Parallel processing of large data sets is a standard approach to address performance challenges. A significant body of research has been conducted on parallel data processing, covering topics such as automatic partitioning of databases [8], data clustering for parallel database systems [9], and the utilization of multidimensional data allocation for parallel database systems [10], among others.

The previous research [11] contributed to the development of automated performance tuning plans incorporating materializations and indices within a single layer of multi-tiered persistent storage. In a separate work, a novel resource allocation algorithm spanning multiple layers of multi-tiered persistent storage was introduced in [12, 13]. Additionally, the same publication presented a new approach for identifying query processing plans for predicted workloads using a new cost model presented in [12]. Furthermore, the development of an Extended Petri-net-based model and the optimization of query processing plans for multi-tiered persistent storage were outlined in [13].

3. Basic Concepts

In multi-tiered persistent storage, a list of persistent storage tiers is represented as a sequence of positive integer numbers denoted by $L = \langle l_0, \ldots, l_n \rangle$. The first number l_0 represents the lowest and slowest tier while the last number l_n represents the highest and fastest tier. Usually, $l_0 < l_1 < \ldots < l_{n-1} < l_n$. Each tier $l_i \in L$ is associated with a pair (r_i, w_i), where r_i is the total number of data blocks that can be read in a single time unit (for example, a millisecond) from a tier l_i, and w_i is the total number of data blocks that can be written to a single time unit to a tier l_i.

The processing of database queries involves transferring data between storage tiers L. These data transfers are identified by transforming an original query processing plan generated by a database query optimiser into an EPN, as described in the next section. The EPN is used to capture the possible parallelisations in the query processing.

Then, the EPN is transformed into a sequence of operations $E = \langle e_1, \ldots, e_n \rangle$. Input and output data for an operation $e_i \in E$ are represented by a pair of sets, $\{b_1, \ldots, b_m\}$ and $\{b_{m+1}, \ldots, b_n\}$, where the first set represents the input datasets, while the second set represents the output datasets. Each b_i contains pairs of data blocks represented as $\{(D_1, l_n), \ldots, (D_m, l_m)\}$, where D_i (for $i = 1, \ldots, m$) denotes the total number of data blocks, and l_j represents the location of the data blocks at tier l_j. For instance, $\langle\{(100, l_1), (250, l_2)\}, \{(200, l_3)\}\rangle$ represents 350 blocks of input data located at the levels l_1 and l_2 and 200 blocks of output data located at level l_3.

The operations in E read the data blocks from persistent storage into transient memory and write the data blocks from transient memory into persistent storage. Every single read or write operation engages a single tier in multi-tiered persistent storage. It is how the processing of a sequence of operations E contributes to a sequence of data transfers $Q = \langle (l_i, t_m), \ldots, (l_j, t_n) \rangle$, where $l_i \ldots l_j$ represent the tiers in persistent storage, and $t_m \ldots t_n$ represent the number of time units required to transfer data to or from persistent storage. Each pair (l, t) in the sequence is known as a *data transfer*.

In this work, simultaneous processing of multiple database queries submitted by users is considered, which are then transformed into a set of sequences of operations $\mathcal{E} = \{E_1, \ldots, E_n\}$ in the manner explained above.

The sequences of operations in $\mathcal{E}$ contribute to a set of sequences of data transfers $\mathcal{Q} = \{Q_1, \ldots, Q_i\}$.

The dedicated processes perform the bidirectional data transfers between the tiers of persistent storage and the transient data buffers. A single data transfer assigned to a process involves either reading data from transient memory and writing data to persistent storage or reading data from persistent storage and writing to the data buffer in transient memory.

Assuming there are m processes available for implementing data transfers in $\mathcal{Q}$, denoted as $\mathcal{P} = \{P_1, \ldots, P_m\}$, which are simultaneously transferring data between different levels

of multi-tiered persistent storage, an objective of data transfer scheduling proposed in this work is to allocate the data transfers in $\mathcal{Q}$ to the processes in $\mathcal{P}$ in a way that minimises the total data transfer time. For each process, $P_i \in \mathcal{P}$ is associated with a sequence of data transfers $\langle (l_i, t_j), \ldots, (l_n, t_m) \rangle$. The abbreviations and definitions are listed in Table 1.

Table 1. *List of abbreviation and definition used in this paper*

Abbreviation	Definition
l	A tier in multi-tiered persistent storage
L	A sequence of tiers
r	Reading speed for a tier
w	Writing speed for a tier
EPN	Extended Petri Net
e	An operation
E	A sequence of operations
$\mathcal{E}$	A set of E
b	A dataset
D	Total number of data blocks
t	The amounts of time units required to transfer data between multi-tiered persistent storage
$\{b_i, \ldots, b_j\}$	Input or Output datasets
Q	A sequence of transfers
$\mathcal{Q}$	A set of Q
P	A process that transfers the data transfer to and from multi-tiered persistent storage
$\mathcal{P}$	A set of processes P

4. Generation of Sequences of Operations

This section presents the notation of EPN. It shows how to represent parallel query processing plans with an EPN and how to generate a sequence of operations implementing a query.

4.1. Extended Petri Nets

Consider a scenario where a relational database server receives a `SQL` query from a database application.

The query optimiser generates a query processing plan, which is represented as a directed bipartite graph that contains a list of operations. This graph is then transformed into an EPN to show the flow of data and the possible parallelisations when processing the query. The notation of EPNs proposed in this work is derived from a well-known notation of Petri Nets [14], which is commonly used in both sequential and concurrent systems to provide a clear understanding of parallel data processing.

An EPN is a quadruple (B, V, A, W), where input/output datasets (B) and operations (V) are visualised as circles and rectangles, respectively. The datasets and operations are connected by the arcs in $A \subseteq (B \times V) \cup (V \times B)$ representing the data flows in the processing of a query. An arc $a_i \in A$ is a pair (e, b) that connects an operation e to a dataset b or a pair (b, e) that connects a dataset b to an operation e. The total number of data blocks read and written by each operation is determined by a weight function $W : A \longrightarrow N^+$.

An operation can use multiple input and output datasets, and such datasets can also be used by multiple operations. Figure 1 provides a sample visualisation of an EPN.

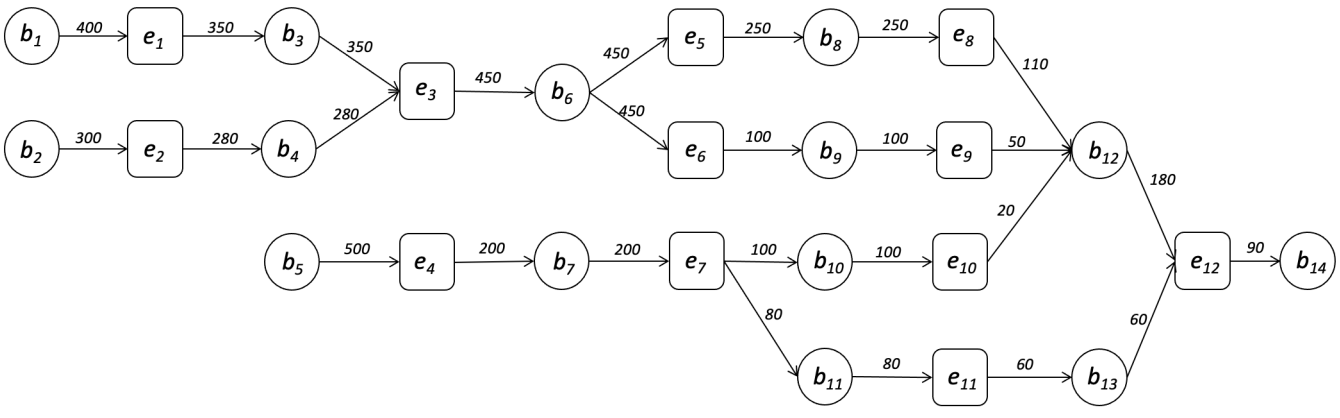

Figure 1. *Visualisation of Extended Petri Nets*

Example 4.1. An EPN (B, V, A, W) visualised in Figure 1 has the following components.

A set of datasets $B = \{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9, b_{10}, b_{11}, b_{12}, b_{13}, b_{14}\}$. $b_1 = \{(200, l_0), (200, l_1)\}$,
$b_2 = \{(200, l_1), (100, l_2)\}$,
$b_3 = \{(100, l_1), (250, l_2)\}$,
$b_4 = \{(100, l_3), (180, l_4)\}$,
$b_5 = \{(200, l_3), (300, l_4)\}$,
$b_6 = \{(200, l_2), (250, l_3)\}$,
$b_7 = \{(100, l_4), (100, l_5)\}$,

$b_8 = \{(100, l_2), (150, l_3)\}$,
$b_9 = \{(50, l_1), (50, l_2)\}$,
$b_{10} = \{(100, l_4)\}$,
$b_{11} = \{(80, l_5)\}$,
$b_{12} = \{(100, l_4), (80, l_5)\}$,
$b_{13} = \{(60, l_1)\}$,
$b_{14} = \{(90, l_5)\}$,

A set of operations $V = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}, e_{12}\}$

A set of arcs $A = \{a_1 : (b_1, e_1), a_2 : (e_1, b_3)$,
$a_3 : (b_2, e_2), a_4 : (e_2, b_4)$,
$a_5 : (b_3, e_3), a_6 : (b_4, e_3), a_7 : (e_3, b_6)$,
$a_8 : (b_5, e_4), a_9 : (e_4, b_7)$,
$a_{10} : (b_6, e_5), a_{11} : (e_5, b_8)$,
$a_{12} : (b_6, e_6), a_{13} : (e_6, b_9)$,
$a_{14} : (b_7, e_7), a_{15} : (e_7, b_{10}, a_{16} : (e_7, b_{11})$,
$a_{17} : (b_8, e_8), a_{18} : (e_8, b_{12})$,
$a_{19} : (b_9, e_9), a_{20} : (e_9, b_{12})$,
$a_{21} : (b_{10}, e_{10}), a_{22} : (e_{10}, b_{12})$,
$a_{23} : (b_{11}, e_{11}), a_{24} : (e_{11}, b_{13})$,
$a_{25} : (b_{12}, e_{12}), a_{26} : (b_{13}, e_{12}), a_{27} : (e_{12}, b_{14})\}$

The weight function, denoted as W, assigns a numerical value to each edge to represent the total number of data blocks read or written by its corresponding operation. For instance, the weight function assigns 400 data blocks to an edge $a_1 : (b_1, e_1)$. It means that operation e_1 reads 400 data blocks.

4.2. Transformation of Extended Petri Nets into Sequences of Operations

To discover a sequence of data transfers that is required to process a query, an EPN representing a query must first be converted into a sequence of operations. This section presents the algorithms that convert EPNs into sequences of operations.

The concepts of *root dataset* and *root operation*, *split dataset* and *split operation*, and *merge dataset* and *merge operation* are foundational in an EPN.

A *root dataset* is an input dataset that is not used as an output dataset by any operation. A *root operation* is an operation that is only connected to a *root dataset*. This means that an operation that reads from both a root dataset and a non-root dataset cannot be considered a root operation. A *top operation* refers to an operation that does not have any other operations reading its output data container.

A dataset is a *split dataset* when it is an input dataset for more than one operation. A dataset is a *merge dataset* when it is an output dataset for more than one operation.

Similarly, an operation is a *merge operation* when it has more than one input dataset, while an operation is a *split operation* when it has more than one output dataset.

Section 4.2.1 provides an algorithm for creating a sequence of sets of operations. Section 4.2.2 provides an algorithm for the conversion of a sequence of sets of operations into a sequence of operations.

4.2.1. Creating Sequences of Sets of Operations

Algorithm 4.1 transforms an EPN (B, V, A, W) into a sequence of sets of operations $S = \langle\{e_i, \ldots, e_j\}, \ldots, \{e_x, \ldots, e_y\}\rangle$. Each dataset in B is associated with the number of minimum tokens required to process it. For example, let the input dataset b_i for operation e_i be the output dataset for operations e_j and e_k. In this scenario, a minimum number of two tokens is required for b_i. As soon as e_j is written to dataset b_i, one token will be associated with it, and after e_k is written to b_i, another token will be associated with b_i. Once b_i has two tokens associated with it, operation e_i can read the data from b_i.

This work assumed that the EPN may have one or more root operations.

First, the algorithm associates tokens to all the root input datasets. All the operations become ready to process once tokens are assigned to their input datasets.

Then, the algorithm identifies all the operations that can be processed and groups them into a set. Following that, it removes operations that cannot be processed together from the selected operations if they are merge operations or their input/output dataset is a merge/split dataset. This implies that operations that read from common datasets cannot be located in the same set or that operations that write to a shared dataset cannot be located in the same set. In that case, the algorithm randomly selects one operation and eliminates another that cannot process concurrently from the set. Afterwards, the algorithm appends that set of operations to a sequence of sets. The procedure repeats until the top operation is added to S. Finally, the algorithm returns the outcome $S = \langle\{\langle e_i, \ldots, e_j\rangle, \ldots, \langle e_x, \ldots, e_y\rangle\}, \ldots, \{\langle e_n, \ldots, e_m\rangle, \ldots, \langle e_k, \ldots, e_z\rangle\}\rangle$.

Algorithm 4.1. Generate a sequence of sets of operations
Input: An Extended Petri Net (B, V, A, W)
Output: A sequence of sets $S = \langle\{e_i, \ldots, e_j\}, \ldots, \{e_x, \ldots, e_y\}\rangle$.

(1) Create a new empty sequence of sets called $S = \langle\rangle$.
(2) Assign tokens to all root datasets and copy V to V_{temp}.
(3) Identify all the operations from V_{temp} where all their input datasets have the minimum number of tokens. And add these operations to a temporary set called $S_{temp} = \{e_i, \ldots, e_j\}$.
(4) Remove all the operations from S_{temp} that are already in S.
(5) Partition the S_{temp} using Algorithm 4.2.
(6) Create the empty set $temp = \{\}$.
(7) If S_{temp} is not empty, then iterate over S_{temp}, and let current set be $\{e_i, \ldots, e_j\}$.
    (a) Iterate over the current set, and let the current operation be e_i.
        (i) If e_i is not in $temp$ and is not in the same set as one of the operations in $temp$, then append e_i to the $temp$ and remove and exit from the iteration.
(8) Else, go to Step (11).
(9) Append $temp$ to $S = \langle\ldots, \{e_i, \ldots, e_j\}\rangle$ and remove all the operations from V_{temp} which are included in $temp$.
(10) Next, associate all output datasets of each operation from $temp$ with a token and go back to Step (3).

(11) Return the resulting set $S = \langle\{e_i, \ldots, e_j\}, \ldots, \{e_x, \ldots, e_y\}\rangle$.

Algorithm 4.2. Generate a set of sets conflicted operations

Input: A set of operations $S_{temp} = \{e_i, \ldots, e_j\}$.

Output: Partitioned S_{temp}, a set of sets of operations $S_{temp} = \{\{e_i, \ldots, e_j\}, \ldots, \{e_x, \ldots, e_y\}\}$.

(1) Create a temporary set with the first operation from S_{temp} like $temp = \{\{e_i\}\}$, remove e_i form S_{temp} and set the boolean variable check to false, $check = false$.

(2) Iterate over S_{temp} , let the current operation be e_i.

(a) Iterate over $temp$, let the current set be $\{e_j, \ldots, e_k\}$.

(i) If e_i is reading from the same input dataset from one of the operations in the current set or e_i is writing to the same output datasets from one of the operations in the current set, then append e_i to that set and set the boolean variable check to true, $check = true$.

(b) If $check = false$, then create a new set with e_i like $\{e_i\}$ and append it to $temp$.

(3) After the iteration, set $S_{temp} = temp$.

(4) Return partitioned $S_{temp} = \{\{e_i, \ldots, e_j\}, \ldots, \{e_x, \ldots, e_y\}\}$.

Example 4.2. In this example, the EPN given in Figure 1 is transformed into a sequence of sets of operations. The root input datasets b_1, b_2 and b_5 are associated with tokens. The next step is to identify all the operations that are ready to process. In this case, the set of operations $\{e_1, e_2, e_4\}$ can be processed, since all their input datasets have associated tokens. Additionally, the operations in this set do not read from the same input datasets or write to the same output datasets, so they can be appended to the sequence of sets $S = \langle\{e_1, e_2, e_4\}\rangle$. The output datasets of each operation in this set are then associated with tokens.

Next, all operations ready for processing are identified.

In this step, a set of operations $\{e_3, e_7\}$ is ready to process. These operations do not read from the split input dataset or write to the merge output dataset. The output datasets of each operation in the set are then associated with tokens.

This set is appended to S, and S is updated as $S = \langle\{e_1, e_2, e_4\}, \{e_3, e_7\}\rangle$.

Next, all operations ready for processing are identified.

In this step, a set of operations $\{e_5, e_6, e_{10}, e_{11}\}$ can be processed.

Given that operations e_5 and e_6 access the same dataset b_6, one operation is selected, and the other is excluded from the group. Consequently, the revised group becomes $\{e_5, e_{10}, e_{11}\}$, which is then added to $S = \langle\{e_1, e_2, e_4\}, \{e_3, e_7\}, \{e_5, e_{10}, e_{11}\}\rangle$.

Finally, the output datasets of each operation in this set are associated with tokens.

The same procedure is repeated until all operations from V are appended to S. After completing the process, the algorithm generates a sequence of sets S, as shown below.

$\langle$
$\{e_1, e_2, e_4\}$,
$\{e_3, e_7\}$,
$\{e_5, e_{10}, e_{11}\}$,
$\{e_6, e_8\}$,
$\{e_9\}$,
$\{e_{12}\}$
$\rangle$

4.2.2. Creating Sequences of Operations

To process a single query, it is necessary to determine a sequence of data transfers. This can be achieved by transforming a sequence of sets of operations into a sequence of operations. To accomplish this, the optimal order for processing the operations within each set of operations must be identified. This section explains how to convert the sequence of sets of operations S obtained from Algorithm 4.1 into a sequence of operations.

Initially the algorithm calculates the profit for each operation in each set of operations in S. To calculate the profit, the algorithm subtracts the weighted amount of storage that must be allocated while processing an operation from the weighted total amount of storage that can be released after the processing of that operation. Let e_i represent an operation in S and $\rho(e_i)$ represent the profit from processing e_i. To calculate the value of $\rho(e_i)$, the algorithm must determine the weighted amounts of storage read ω_r and written ω_w. To compute ω_r for the input datasets, the algorithm multiplies the size of each input dataset by the reading speed at the tier where the dataset is located. If an operation e_i reads the datasets $\{(D_i, l_x), \ldots, (D_j, l_y)\}$, then Equation (1) is used to calculate ω_r.

$$\omega_i = (D_i * r_x) + \cdots + (D_j * r_y) \tag{1}$$

If an operation e_i writes the datasets $\{(D_k, l_i), \ldots, (D_n, l_j)\}$, then Equation 2 is used to compute ω_w.

$$\omega_o = (D_k * w_i) + \cdots + (D_n * w_j) \tag{2}$$

Finally, Equation (3) is used to compute profit $\rho(e_i)$ by subtracting ω_w from ω_r.

$$\rho(e_i) = \omega_i - \omega_o \tag{3}$$

Once the profit for each operation in a set of operations is calculated, the algorithm sorts the operations in the descending order of profits, replacing a set of operations with a sequence of operations.

Higher amounts of faster storage released as early as possible are anticipated to enhance the performance of future processing.

This procedure is repeated for each set of operations in S. Algorithm 4.3 below transforms a sequence of sets of operations into a sequence of operations.

Algorithm 4.3. Generate a sequence of operations

Input: An Extended Petri Net (B, V, A, W), a sequence of tiers $L = \langle l_0, \ldots, l_n\rangle$, and a sequence of sets $S = \langle\{e_i, \ldots, e_j\}, \ldots, \{e_x, \ldots, e_y\}\rangle$.

Output: A processing plan denoted by a sequence of

operations $E = \langle e_i, \ldots, e_j \rangle$.

(1) Create an empty sequence called $E = \langle\rangle$ and an empty set of pairs called $temp = \{\}$.
(2) Iterate over S and let current set be $\{e_i, \ldots, e_j\}$.
  (a) If the current set has more than one operation, then
    (i) Iterate over the current set $\{e_i, \ldots, e_j\}$ and let the current operation be e_i.
      - Use the Algorithm 4.4 with input parameters (B, V, A, W), L, and e_i to compute the profit $\rho(e_i)$ for current operation e_i.
      - Append $\rho(e_i)$ to $temp = \{\ldots, \rho(e_i)\}$.
    (ii) From $temp$, sort the profit value in descending order and gets the corresponding list of operations $\langle e_i, \ldots, e_j \rangle$.
    (iii) Append list of operations $\langle e_i, \ldots, e_j \rangle$ to $E = \langle \ldots, e_i, \ldots, e_j \rangle$ and go back to step (2).
  (b) Else If the current set has only one operation, then appen that operation to E and make empty $temp$ by setting $temp = \{\}$.
(3) Return E.

Algorithm 4.4. Calculate a profit ρ for an operation e

Input: An Extended Petri Net (B, V, A, W), a sequence of tiers $L = \langle l_0, \ldots, l_n \rangle$, and an operation e.

Output: A profit $\rho(e)$ for an operation e.

(1) If e_i is a root operation then set $\rho(e) = 0$.
(2) Else, compute ω_i by using equation (1) and ω_o by using equation (2).
  The value of input/output data blocks and read/write speed of each tier can be extracted from *Extended Petri Net* and L.
  (a) Use the value of ω_i and ω_o and calculate $\rho(e_i)$ using euqation (3).
(3) Return the profit $\rho(e)$ for an operation e.

Example 4.3. In this example, a sequence of sets of operations $S = \langle \{e_1, e_2, e_4\}, \{e_3, e_7\}, \{e_5, e_{10}, e_{11}\}, \{e_6, e_8\}, \{e_9\}, \{e_{12}\} \rangle$ is transformed as illustrated in Example 4.2, along with the EPN detailed in Example 4.1. Multi-tiered persistent storage $L = \langle l_0, l_1, l_2, l_3, l_4, l_5 \rangle$ is applied with the following read/write parameters: reading speeds of 20 blocks per time unit at tier l_0, 30 blocks per time unit at l_1, 40 data blocks per time unit at l_2, 50 data blocks per time unit at l_3, 60 data blocks per time unit at l_4, and 70 data blocks per time unit at l_5.

Also, assume writing speeds of 10 blocks per time unit at tier l_0, 20 blocks per time unit at l_1, 30 blocks per time unit at l_2, 40 blocks per time unit at l_3, 50 blocks per time unit at l_4 and 60 per data block at l_5.

The initial step involves iterating over S to select the current set $\{e_1, e_2, e_4\}$. Following, compute the profit for each operation from the current set. Since all operations are root operations, the profit $\rho(e_1)$ value becomes 0. Append all operations to E, which becomes $E = \langle e_4, e_2, e_1 \rangle$. Moving on to the next iteration, select the second set from S, which is $\{e_3, e_7\}$. Compute the profit for operation e_3. Compute ω_o for operation e_3 by using Equation (1) as ω_i = (100 * 30) + (250 * 40) + (100 * 50) + (180 * 60) = 28,800. Next, use Equation (2) and compute ω_o as ω_o = (200 * 30) + (250 * 40) = 16,000. Those input-output data sizes can be obtained from Example 4.1, and the read and write speeds can be obtained from L. Using Equation (3), obtain $\rho(e_3)$ = 13,000. Append $\rho(e_3)$ to $temp$, which then becomes $\rho(e_3)$. Repeat this same procedure for e_7, resulting in a set of profits, such as $temp = \{\rho(e_3), \rho(e_7)\}$. Since $\rho(e_3)$ is larger than $\rho(e_7)$ in this step, put e_3 in the sequence first, followed by e_7, resulting in $E = \langle e_4, e_2, e_1, e_3, e_7 \rangle$.

The same procedure is repeated, resulting in the sequence of operations as follows:

$E = \langle e_4, e_2, e_1, e_4, e_7, e_5, e_{10}, e_{11}, e_6, e_8, e_9, e_{12} \rangle$.

5. Optimisation of Parallel Data Transfers

5.1. An Overview

To create a plan for parallel data transfers, the sequences of operations on the datasets need conversion into data transfers, and the total number of data transfer processes must be determined.

A data transfer is represented as a pair (l_i, t), where l_i is the level of multi-tiered storage involved in the transfer, and t is the estimated total number of time units required for the transfer. A data transfer represents the movement of data between multi-tiered persistent storage and transient memory.

The directions of data transfers between transient memory and multi-tiered persistent storage are not distinguished.

A *data transfer process* implements the data transfers to and from multi-tiered persistent storage. A *data transfer plan* is a sequence of data transfers assigned to a data transfer process.

The following steps are taken to allocate the data transfers to the processes. Each sequence of operations $E_i \in \mathcal{E}$ obtained from Algorithm 4.3 is transformed into a sequence of data transfers $Q_i = \langle (l_i, t_x), \ldots, (l_j, t_y) \rangle$. The transformation process is explained in Section 5.2.

In the next step, the set of sequences $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ obtained from the previous step is analysed to create a sequence of sets based on the total number of transfers to or from a specific level in multi-tiered persistent storage within a total number of time units. The first set in a sequence includes the transfers to or from the most frequently utilised levels. The aim of this step is to prioritise the data transfers that may have a significant number of conflicts with other data transfers when accessing the same level in multi-tiered storage. The classification of data transfers is explained in Section 5.3.

The scheduling of data transfers over data transfer processes is performed in a number of iterations. At the beginning of each iteration, the first data transfers from each sequence of transfers are used to form *a candidate set of data transfers*. The candidate data transfers are selected such that none of them conflicts with any other transfers that have already been assigned to data transfer processes.

In cases where multiple candidate transfers exist, a set of

scheduling rules is applied to determine the selection. In the instance of a singular candidate transfer, it gets allocated to the processor with the minimal current workload. The scheduling rules are detailed in Section 5.4. The partitioning of transfer plans is adjusted upon each transfer assignment to a processor, generating a fresh set of candidate transfers.

5.2. Creating Data Transfer Plans

A set of sequences of operations $\mathcal{E} = \{E_1, \ldots, E_m\}$ and the descriptions of the storage tiers in multi-tiered persistent storage $L = \langle l_0, \ldots, l_n \rangle$ are used to create the transfer plans.

For every sequence $E_i = \langle e_1, \ldots, e_n \rangle$ and every operation e_i within the sequence, all input datasets $\{B_1, \ldots, B_m\}$ and all output datasets $\{B_{m+1}, \ldots, B_n\}$ are identified.

Both input and output datasets are used to find all the data transfers.

To transfer dataset B, the process involves performing a series of transfers.

Let (D_i, l_j) be one such transfer, where D_i denotes the total number of data blocks to be transferred, and l_j denotes a level in multi-tiered persistent storage to be engaged.

Given a (D_i, l_j) pair, the estimated total number of time units t_i required to read/write data blocks in D_i at tier l_j is computed, resulting in the creation of a transfer (l_j, t_i). Subsequently, a transfer (l_j, t_i) is added to a transfer plan Q_i associated with processing an operation e_i. This procedure is then repeated for the subsequent operation $e_{i+1} \in E$, with the next transfer (l_k, t_{i+1}) appended to a transfer plan for Q_i. After processing the current set E, progression is made to the next sequence from $\mathcal{E}$ to generate a transfer plan Q_{i+1}.

Algorithm 5.1 demonstrates the details of the steps to generate a data transfer plan Q from a sequence of E.

Algorithm 5.1. Generate a data transfer plan

Input: A processing plan E and a sequence of multi-tiered persistent storage L.

Output: A data transfer plan represented by a sequence of data transfers $Q = \langle (l_i, t_j), \ldots, (l_j, t_n) \rangle$.

(1) Create an empty sequence $Q = \langle\rangle$.

(2) Iterate over E and let the current operation be $e_i \in E$.

(a) Find a set of input datasets $\{(D_i, l_j), \ldots, (D_j, l_k)\}$ for e_i and put it into a temporary set $temp$ = $\{(D_i, l_j), \ldots, (D_j, l_k)\}$.

(b) If e_i is the first read/write operation in a set, then create a new empty set $\{\}$.

(c) Iterate over $temp$ and let the current pair be (D_i, l_j).

(i) To process the data size of D_i, calculate the total time units required based on the read speed (r_j) and write speed (w_j) of l_j.

(ii) If e_i is a write operation, then calculate the total time units as $t_i = D_i/r_j$.

(iii) Else calculate the total time units as $t_i = D_i/w_j$.

(iv) Next, create a transfer (l_i, t_j).

(v) If e_i is the first read/write operation, then append it into a set created in step (b) like $\{\ldots, (l_i, t_j)\}$.

(vi) Else if e_i is the read operation then iterate over Q_{temp}.

- If all the data transfers from the current set are not accessing the level l_j, then append a transfer (l_i, t_j) to the current set like $\{\ldots, (l_i, t_j)\}$.

(vii Else if e_i is a write operation, then iterate over Q_{temp} but skip all the sets which included data transfers related to e_i's input datasets.

- If all the data transfers from the current set are not accessing the level l_i, then append a transfer (l_i, t_j) to the current set like $\{\ldots, (l_i, t_j)\}$.

(viii) Else, create a new set and append a transfer (l_i, t_j) to that set like $\{(l_i, t_j)\}$ and add that set to Q_{temp}.

(d) If step (b) and step (c)(viii) created a new set, then append the new set to the last $Q_{temp} = \langle \ldots, \{(l_i, t_j), \ldots, (l_j, t_k)\}\rangle$.

(e) Find a set of the output datasets $\{(D_k, l_i), \ldots, (D_n, l_m)\}$ for e_i and put it into temporary set $temp$ = $\{(D_k, l_i), \ldots, (D_n, l_m)\}$.

(f) If the data transfers for the e_i's output datasets are not generated, then repeat steps (b) to step (d).

(g) Else, append each operation from Q_{temp} to Q, set $Q_{temp} = \{\}$, and go back to the iteration in step (b) until all the operations from E are transformed into data transfers.

(3) Finally, return the result $Q = \langle (l_i, t_j), \ldots, (l_j, t_n)\}\rangle$.

Example 5.1. The example uses a set $\mathcal{E}$ containing two processing plans E_1 and E_2. E_1 consists of a single element $e_1 = \langle\{(l_0, 10), (l_1, 20)\}, \{(l_2, 30)\}\rangle$, while E_2 consists of two elements e_2 and e_3 and can be represented as $E_2 = \langle e_2, e_3\rangle$, where $e_2 = \langle\{(l_1, 15)\}, \{(l_2, 10)\}\rangle$ and $e_3 = \langle\{(l_2, 20)\}, \{(l_2, 10)\}\rangle$. Suppose the data buffer can hold up to 30 data blocks from tiers l_0 and l_1 and up to 25 data blocks from tier l_2. The data transfer speeds to/from tiers l_0, l_1 and l_2 are 2, 5 and 10 data blocks per time unit, respectively.

Based on this calculation, it is determined that it will take 5 time units to read 10 data blocks from l_0, 4 time units to read 20 data blocks from l_1, and 3 time units to write 30 data blocks to l_2. Using this information, a transfer plan Q_1 can be created as $\langle (l_0, 5), (l_1, 4), (l_2, 3)\rangle$ for E_1. Similarly, for E_2, a transfer plan Q_2 can be generated as $\langle (l_1, 3), (l_2, 2), (l_2, 2), (l_2, 2)\rangle$.

To better understand the results $\mathcal{Q} = \{Q_1, Q_2\}$, refer to Figure 2.

Q_1:	(l_0, 5)					(l_1, 4)				(l_2, 3)		
Q_2:	(l_1, 3)			(l_2, 2)		(l_2, 2)		(l_2, 2)				
	1	2	3	4	5	6	7	8	9	10	11	12

Figure 2. *Visualisation of $\mathcal{Q}$ from Example 5.1*

5.3. Analysing Transfer Statistics for Conflict Minimisation

To reduce the total number of conflicts when allocating transfers to transfer processes, Algorithm 5.2 finds the most frequently used tiers. The algorithm uses $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$ obtained from Algorithm 5.1 to collect statistics on the total number of transfers and the time spent accessing each level in multi-tiered storage. The algorithm creates a set of triples, such as (l_i, c_{l_i}, c_{t_i}), where l_i is a device level, c_{l_i} is the total number of transfers accessing tier l_i, and c_{t_i} is the total number of time units spent accessing tier l_i.

The algorithm groups the triples by the values of c_l and sorts the groups in descending order. Next, it further sorts the triples in the descending order of the c_t value within each category of c_l. The result is a sequence of sets of triples denoted by $\langle\{(l_i, c_{l_i}, c_{t_i}), \ldots, (l_j, c_{l_j}, c_{t_j})\}, \ldots, \{(l_m, c_{l_m}, c_{t_m}), \ldots, (l_n, c_{l_n}, c_{t_n})\}\rangle$.

From each triple, the algorithm extracts tiers and creates a sequence of sets of tiers denoted by $\alpha = \langle\{l_i, \ldots, l_j\}, \ldots, \{l_m, \ldots, l_n\}\rangle$. This process helps to identify the busiest tier, which has the highest c_l and highest c_t value and is most likely to result in conflicts.

Algorithm 5.2. Generate analyzed transfers

Input: A set of sequences of data transfers $\mathcal{Q} = \{Q_1, \ldots, Q_n\}$.

Output: Statistics on data transfers $\alpha = \langle\{l_i, \ldots, l_j\}, \ldots, \{l_m, \ldots, l_n\rangle\}$.

(1) Create an empty sequence called $\alpha = \langle\rangle$ and an empty set called $temp = \{\}$.

(2) Iterate over $\mathcal{Q}$ and let current sequence be Q_i.

  (a) Iterate over Q_i and let current transfer be (l_i, t_j).

    (i) If l_i is already included in a triple from $temp$, then retrieve that triple and update c_{l_i} by adding 1 and c_{t_i} by adding t_j.

    (ii) Else, create a new triple like (l_i, c_{l_i}, c_{t_i}), where $c_{l_i} = 1$ and $c_{t_i} = t_j$. Then, add the new triple to $temp = \{\ldots, (l_i, c_{l_i}, c_{t_i})\}$.

(3) Sort $temp$ in descending order based on the value of c_l and group the same value of c_l.

(4) Within the same group of c_l, sort the triples in descending order based on the value of c_t.

(5) Group the triples with the same value of c_t into sets and add them to a sequence of a set of triples like $\langle\{(l_i, c_{l_i}, c_{t_i}), \ldots, (l_j, c_{l_j}, c_{t_j})\}, \ldots, \{(l_m, c_{l_m}, c_{t_m}), \ldots, (l_n, c_{l_n}, c_{t_n})\rangle\}$.

(6) Next, extract the tier and create a sequence of sets of tiers $\alpha = \langle\{(l_i, \ldots, l_j\}, \ldots, \{l_m, \ldots, l_n\}\rangle\}$.

(7) Return the α.

Example 5.2. A set $\mathcal{Q} = \{Q_1, Q_2\}$ is utilized from Example 5.1.

Algorithm 5.2 iterates over $\mathcal{Q}$ and lets the current sequence be Q_1. Then, the algorithm iterates over the transfers in Q_1 and lets the first transfer be $(5, l_0)$. The first transfer contributes to a triple $(l_0, 1, 5)$ and appends it to $temp$ as $\{(l_0, 1, 5)\}$.

The same process is repeated, yielding the updated $temp = \{(l_0, 1, 5), (l_1, 1, 4), (l_2, 1, 3)\}$

After the iterations over Q_1 are completed, the algorithm processes the following sequence Q_2 in the same way.

The same process is repeated, and eventually, the algorithm will have $temp = \{(l_0, 1, 5), (l_1, 2, 7), (l_2, 4, 9)\}$.

Next, the algorithm groups the same values of c_l and arranges the groups in the descending order of c_l to get a sequence of triples $temp = \{(l_2, 4, 9), (l_1, 2, 7), (l_0, 1, 5)\}$.

Since there are no triples with the same values of c_l, there is no need for sorting over the secondary key c_t. Finally, the algorithm generates $\langle\{(l_2, 4, 9)\}, \{(l_1, 2, 7)\}, \{(l_0, 1, 5)\}\rangle$, and it extracts the tiers and creates $\alpha = \langle\{l_2\}, \{l_1\}, \{(l_0\}\rangle$.

5.4. Applying Scheduling Rules

To assign a data transfer to a transfer process with the lowest workload, a candidate transfer from $\mathcal{Q}$ must be selected. First, an algorithm identifies all *candidate transfers* in $\mathcal{Q}$. If all candidate transfers are eliminated, a single idle time unit is assigned to the process. If only one *candidate transfer* is found, it is assigned to the process with the lowest current workload. If multiple *candidate transfers* are found, the *scheduling rules* listed below are applied until a single transfer is left.

Rule 0: Select the candidate transfers that do not conflict with the others.

Eliminate any candidate transfers that do not satisfy the following conditions. Two data transfers (t_i, l_i) and (t_j, l_j) assigned to a data transfer process *conflict* when

1. both transfers try to access the same level of multi-tiered storage within the same time frame and/or
2. both transfers are part of the same sequence of transfers, and their order in the data transfer processes is opposite to their order in their transfer plan.

Rule 1 is applied when more than one candidate data transfer remains after the elimination of conflicting candidate transfers.

Rule 1: Select the candidate transfers included in the longest sequence of transfers Q.

The length of a sequence of transfers is defined as the total number of time units needed for processing the transfers. This rule aims to minimise the overall transfer time and to balance the workload among the data transfer processes. For example, when a conflict over access to the same tier is found, the transfers included in the shorter sequences can be more efficiently allocated.

Rule 2 is applied when more than one candidate data transfer remains after applying *Rule 1*.

Rule 2: *Prioritise data transfers that have the potential to cause conflicts in the future.*

To identify the relevant transfers, the rule uses a value of α obtained from Algorithm 5.2 and focuses on the data transfers accessing the same tier as the first set in α. When no suitable transfers are found within the first set in α, the next set is examined, and the process continues until one or more potential transfers are identified.

Rule 3 is applied when more than one candidate data transfer remains after applying *Rule 2*.

Rule 3: *Select the candidate transfers included in the sequence of transfers that consist of the largest number of transfers from the set of transfers returned by* Rule 2.

This rule minimises the idle time of the processors. A sequence with many shorter transfers can quickly fill the gaps when a conflict occurs.

Rule 4 is applied when more than one candidate data transfer remains after applying *Rule 3*.

Rule 4: *Select the shortest transfers.*

This rule is based on the observation that the smaller size of the candidate transfers reduces the waiting time for future allocations for other transfers, and it allows for more efficient allocation of the remaining sequences of transfers.

Rule 5 is applied when more than one candidate data transfer remains after applying *Rule 4*.

Rule 5: If more than one transfer is left after applying *Rule 4*, randomly select and assign one of the transfers to a processor with the lowest current workload.

The *scheduling rules* always provide a single candidate transfer that is assigned to a data transfer process with the lowest current workload. The algorithms in the next section provide a formal description of the process described above.

5.5. Enhancing Data Transfer Allocation Through Effective Scheduling Rules

An input to Algorithm 5.3 is a set of sequences of transfers $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ obtained from the transformation of query processing plans and a set of processors $\mathcal{P} = \{P_1, \ldots, P_m\}$. The algorithm returns updated allocations of the data transfers to the processor $\mathcal{P} = \{P_1, \ldots, P_m\}$.

Algorithm 5.3. Generate allocation plan

Input: A set of sequences of transfers $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ and a set of processes $\{P_1, \ldots, P_m\}$.

Output: Updated set of processes $\mathcal{P} = \{P_1, \ldots, P_m\}$.

(1) Copy a set of input sequences of transfers $\mathcal{Q}$ into a temporary variable $Temp_1 = \{Q_1, \ldots, Q_m\}$.

(2) While $Temp_1$ is not empty, iterate over the following steps:

   (a) Get a value of α from Algorithm 5.2 using $Temp_1$ as the input.

   (b) Get two sets of data transfer processes: $\rho_l = \{P_i, \ldots, P_j\}$ with the lowest workload, and $\rho_h = \{P_x, \ldots, P_y\}$ with the highest workload. Let $\mathcal{P} = \rho_l \cup \rho_h$.

   (c) Choose a data transfer process P_i from the lower workload processes, where $P_i \in \rho_l$, and its allocation plan is a set of transfers, which is denoted $P_i = \{(l_j, t_k), \ldots, (l_x, t_y)\}$.

   (d) Collect all the candidate transfers from $Temp_1$ and put them into a set of candidate transfers $Q_c = \{(l_i, t_j), \ldots, (l_n, t_m)\}$.

   (e) Use *Algorithm 5.4* with input Q_c and ρ_h to apply *Rule 0* and update Q_c accordingly.

   (f) If Q_c contains only one transfer, then get a transfer from Q_c and which is denoted (l_i, t_j).

   (g) Else, if Q_c contains more than one transfers, then use *Algorithm 5.5* with input a set of sequences $Temp_1$, a set of candidate transfers Q_c, and a sequence of sets α to select a candidate transfer (l_i, t_j).

   (h) Else, if Q_c is empty, then add an idle time unit on all processors in ρ_l and go back to step (2)(b).

   (i) Assign a selected candidate transfer (l_i, t_j) to the allocation plan P_i.

   (j) Remove the transfer (l_i, t_j) from $Temp_1$ and go to step (2)(a).

(3) Once all transfers have been assigned, the updated set of allocation plans for processors $\mathcal{P} = \{P_1, \ldots, P_m\}$ is returned.

The first step *Algorithm 5.4* gets the candidate transfers $Q_c = \{(l_i, t_j), \ldots, (l_n, t_m)\}$ and selects a group of high workload processors known as $\rho_h = \{P_x, \ldots, P_y\}$. The next step removes any transfers that conflict with each other from the set of candidate transfers, denoted as Q_c. Finally, the modified set of candidate transfers Q_c is returned as output.

Algorithm 5.4. Apply *Rule 0* to eliminate conflicted candidate transfers.

Input: A set of candidate transfers $Q_c = \{(l_i, t_j), \ldots, (l_n, t_m)\}$, and $\rho_h = \{P_x, \ldots, P_y\}$.

Output: Updated set of candidate transfers $Q_c = \{(l_x, t_y), \ldots, (l_o, t_p)\}$

(1) If ρ_h is empty, then no elimination is needed because there is no conflict, and go to step (3).

(2) Else eliminate the conflicted transfers from Q_c.

   (a) Next, remove all the transfers from Q_c if the transfers come from the same sequences that are allocated at the end of each processor in ρ_h.

   (b) Next, get all the levels $\mathcal{L} = \{l_i, \ldots, l_j\}$ from transfers that are allocated at the end of each set of data transfers from high-workload data transfer processes in ρ_h.

   (c) Next, remove all the sequences from Q_c that the level of the transfer is in $\mathcal{L}$.

(3) Return $Q_c = \{(l_x, t_y), \ldots, (l_o, t_p)\}$

Algorithm 5.5 gets a set of sequences $Temp_1 = \{Q_i, \ldots, Q_j\}$, a set of candidate transfers $Q_c = \{(l_i, t_j), \ldots, (l_n, t_m)\}$ and a sequence of sets of tiers $\alpha = \langle\{l_i, \ldots, l_j\}, \ldots, \{l_m, \ldots, l_n\}\rangle$. The next step removes candidate transfers from the Q_c according to the schedule rules. Finally, the modified set of candidate transfers Q_c is returned as output.

Algorithm 5.5. To assign the best candidate transfer to a data transfer process, apply the Rules 1 through 5.

Input: A set of sequences $Temp_1 = \{Q_i, \ldots, Q_j\}$, a set of candidate transfers $Q_c = \{(l_i, t_j), \ldots, (l_n, t_m)\}$, and a sequence of sets of tiers $\alpha = \langle\{l_i, \ldots, l_j\}, \ldots, \{l_m, \ldots, l_n\}\rangle$.

Output: Selected a candidate transfers (l_x, t_y)

(1) According to *Rule 1*, select all transfers from Q_c that belong to the longest sequence in $Temp_1$, and remove any that do not meet this criteria.

(2) If Q_c has more than one transfer, then, apply *Rule 2*, itereate over α and let the currnet set be $\{l_i, \ldots, l_j\}$.

  (a) Select all the transfers from Q_c that access the same tier as one of the transfers in the current set and place them in Q_{temp}.

  (b) If Q_{temp} is not empty, then update $Q_c = Q_{temp}$ and exit from iteration and go to step (2).

(3) If Q_c has more than one transfer, then apply *Rule 3*, select all transfers from Q_c that appear in the sequences with the highest number of transfers in $Temp_1$, and remove all the transfers from Q_c which are not selected by Rule 3.

(4) If Q_c has more than one transfer, then apply *Rule 4*, select all transfers with the smallest number of t values from Q_c, and remove all the transfers from Q_c which are not selected by *Rule 4*.

(5) If Q_c has more than one transfer, then apply *Rule 5*, select one transfer randomly from Q_c, and let the selected transfer be (l_x, t_y).

(6) Return the selected transfer as (l_x, t_y).

Example 5.3. In this example, three sequences of data transfers $\mathcal{Q} = \{Q_1, Q_2, Q_3\}$ are assigned to two data transfer processes P_1 and P_2.

$Q_1 = \langle(l_0, 5), (l_4, 3), (l_0, 2), (l_3, 2), (l_0, 5), (l_2, 3)\rangle$

$Q_2 = \langle(l_2, 3), (l_1, 2), (l_4, 2), (l_3, 2), (l_2, 3), (l_4, 3)\rangle$

$Q_3 = \langle(l_1, 2), (l_3, 2), (l_4, 2), (l_2, 5), (l_1, 4), (l_3, 2)\rangle$

In the beginning, the algorithm chooses a data transfer process with a lower workload, specifically P_1 and P_2, which have no transfers assigned yet. This means that both processes have no workload, and the algorithm randomly selects P_1 to begin with. The next step is to gather candidate transfers from $\mathcal{Q}$ and to put the candidates in Q_c, which consists of $\{(l_0, 5), (l_2, 3), (l_1, 2)\}$. Based on *Rule 0* from *Algorithm 5.4*, there are no conflicts, and there is no need to eliminate any transfers from a set of candidate transfers Q_c. Since there is more than one transfer in Q_c, *Algorithm 5.5* is used to select a transfer from Q_c and assign it to P_1. Applying *Rule 1* updates Q_c to $\{(l_2, 3)\}$. Since Q_c has only one transfer at this point, it can be assigned to processor $P_1 = \langle(l_2, 3)\rangle$.

In the next iteration, P_2 is chosen as the processor with the lowest workload. The candidate transfers from $\mathcal{Q}$ are collected and saved in Q_c, which consists of $\{(l_0, 5), (l_1, 2), (l_1, 2)\}$. Based on *Rule 0* from *Algorithm 5.4*, there are no conflicts, and there is no need to eliminate any transfers from Q_c. Since there are multiple transfers in Q_c, *Algorithm 5.5* is used to select a transfer from Q_c and allocate it to P_2. Applying *Rule 1* updates Q_c to $\{(l_0, 5)\}$. A single transfer in Q_c is assigned to processor $P_2 = \langle(l_0, 5)\rangle$.

The total length of transfers assigned to P_2 is 5 time units long, and the total length of transfers assigned to P_1 is 3 time units. Therefore, in the next iteration, P_1 is chosen again as the data transfer process with the lowest workload. The candidate transfers are collected and saved in $Q_c = \{(l_4, 3), (l_1, 2), (l_1, 2)\}$. Based on *Rule 0* in *Algorithm 5.4*, there appears to be a conflict that requires the removal of one transfer: $(l_4, 3)$ from Q_c.

To allocate a transfer from Q_c to P_1, *Algorithm 5.5* is used, as there are multiple transfers in Q_c. Applying *Rule 1* updates $Q_c = \{(l_1, 2), (l_1, 2)\}$, and applying *Rule 2* updates $Q_c = \{(l_1, 2)\}$. Since Q_c only has one transfer now, it can be assigned to processor $P_1 = \langle(l_2, 3), (l_1, 2)\rangle$.

The same iterations are repeated until all data transfers are assigned to either P_1 or P_2. The complete data transfer plans for P_1 and P_2 are the following.

P_1 : $\langle(l_2, 3)\ (l_1, 2), (l_4, 3), (l_4, 2), (l_2, 5), (l_3, 2), (l_2, 3), (l_1, 4), (l_3, 2)\rangle$

P_2 : $\langle(l_0, 5), (l_3, 2), (l_1, 2), (l_0, 2), (l_3, 2), (l_4, 2), (l_0, 5), (l_2, 3), (l_4, 3)\rangle$

Figure 3 illustrates the complete data transfer plans.

For comparison, the complete data transfer plans for the same data transfer $\mathcal{Q} = \{Q_1, Q_2, Q_3\}$ are created through the random assignment of the data transfer to the processes, i.e. by applying *Rule 5* only. The results are illustrated in Figure 4. Upon comparison, it is clear that the results shown in Figure 3, which incorporates multiple rules, are superior to the results based solely on *Rule 5*, as seen in Figure 4.

P_1:	$(l_2, 3)$			$(l_1, 2)$		$(l_4, 3)$			$(l_4, 2)$		$(l_2, 5)$				
P_2:	$(l_0, 5)$					$(l_3, 2)$		$(l_1, 2)$		$(l_0, 2)$		$(l_3, 2)$		$(l_4, 2)$	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

P_1:	$(l_3, 2)$		$(l_2, 3)$			$(l_1, 4)$			$(l_3, 2)$		
P_2:	$(l_0, 5)$					$(l_2, 3)$			$(l_4, 3)$		
	16	17	18	19	20	21	22	23	24	25	26

Figure 3. *Allocated data transfers to the processors through the application of the scheduling rules proposed in section 5.5.*

P_1:	(l_2, 3)			(l_1, 2)		(l_2, 2)		3 idle time units			(l_4, 2)		(l_3, 2)	
P_2:	(l_1, 2)		(l_0, 5)					(l_4, 3)			(l_0, 2)		(l_4, 2)	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

P_1:	(l_3, 2)		(l_0, 5)					4 idle time units				(l_2, 3)			
P_2:	(l_2, 3)			(l_4, 3)				(l_2, 5)				(l_1, 4)			
	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

Figure 4. *Allocated transfers over processors by using Rule 5.*

5.6. Comparative Analysis and Complexity Evaluation

The implementation of the scheduling rules presented in Section 5 leads to a more balanced assignment of data transfers to data transfer processes.

A comparison of the results has been conducted, revealing that utilizing more rules, as depicted in Figure 3, is more effective than relying on random assignments, particularly in the application of *Rule 5* in Figure 4.

In Figure 3, the total time units required to complete the data transfers is 26, while Figure 4 shows 29. Algorithm 5.3 does not include idle time units across the transfer processes during the transfer allocation process.

Algorithm 4.1 produces a sequence of sets of operations (S) by analysing the input EPN. The algorithm's complexity is linear and is denoted as $O(N)$, where N represents the number of operations in V from the EPN.

Algorithm 4.2 creates sets of operations by processing a given set of operations. It partitions the input set of operations into various sets. The complexity of this algorithm is $O(N - 1)$, where N represents the number of operations in the input set of operations.

Algorithm 4.3 creates a set of operations called E by processing a sequence of sets of operations called S. The complexity of this algorithm is linear and is denoted as $O(N)$, where N represents the number of operations in S.

Algorithm 4.4 calculates the profit for a single input operation. Its complexity is linear, indicated by $O(N)$, where N represents the operation.

A sequence of transfers Q is generated by Algorithm 5.1 using a sequence of operations E. The algorithm has a complexity of $O(NM)$, where N is the sequence of E, and M is the set of input datasets for each operation in E.

This process, referred to as Algorithm 5.2, produces a sequence of sets of tiers denoted as α. It accomplishes this by analysing a group of sequences of transfers known as (Q). The algorithm's complexity is $O(N)$, where N is determined by the number of transfers in $\mathcal{Q}$.

The process plan for a set of processes is generated by Algorithm 5.3 using a sequence of transfers known as $\mathcal{Q}$. The algorithm's complexity is $O(N)$, where N refers to the number of transfer in $\mathcal{Q}$.

The process outlined in Algorithm 5.4 updates the set of candidate transfers Q_c by applying Rule 0 and removing certain transfers from Q_c. The complexity of this algorithm is $O(N)$, with N representing the number of input datasets of each candidate transfers from Q_c.

The set of candidate transfers Q_c is updated by Algorithm 5.5, which uses Rules 1 to 5 and removes some transfers from Q_c. The complexity of this algorithm is $O(N)$, where N represents the number of transfers in the input set of candidate transfers.

6. Experiments

The experiments included 29 queries transformed into the following series of data transfers Q. $Q_1 = \langle(l_0, 3), (l_1, 2), (l_3, 4), (l_2, 4), (l_4, 2)\rangle$

$Q_2 = \langle(l_3, 3), (l_2, 5), (l_0, 2), (l_4, 3), (l_1, 3)\rangle$
$Q_3 = \langle(l_2, 2), (l_4, 3), (l_0, 4), (l_1, 3)\rangle$
$Q_4 = \langle(l_3, 3), (l_0, 4), (l_1, 4), (l_2, 3), (l_0, 2)\rangle$
$Q_5 = \langle(l_1, 4), (l_2, 3), (l_3, 2), (l_4, 2), (l_0, 3), (l_4, 2)\rangle$
$Q_6 = \langle(l_2, 3), (l_1, 5), (l_3, 4)\rangle$
$Q_7 = \langle(l_4, 4), (l_3, 5)\rangle$
$Q_8 = \langle(l_0, 4), (l_3, 2), (l_2, 2)\rangle$
$Q_9 = \langle(l_3, 3), (l_4, 2)\rangle$
$Q_{10} = \langle(l_2, 3), (l_1, 2)\rangle$
$Q_{11} = \langle(l_3, 2), (l_1, 2)\rangle$
$Q_{12} = \langle(l_2, 3), (l_4, 3), (l_2, 4)\rangle$
$Q_{13} = \langle(l_1, 3), (l_0, 5)\rangle$
$Q_{14} = \langle(l_0, 4), (l_4, 3), (l_2, 2), (l_0, 3)\rangle$
$Q_{15} = \langle(l_1, 3), (l_2, 3), (l_0, 5)\rangle$
$Q_{16} = \langle(l_2, 3), (l_1, 2), (l_3, 3), (l_2, 2)\rangle$
$Q_{17} = \langle(l_1, 4), (l_4, 3), (l_3, 2)\rangle$
$Q_{18} = \langle(l_2, 3), (l_1, 5), (l_5, 1)\rangle$
$Q_{19} = \langle(l_4, 3), (l_2, 3)\rangle$
$Q_{20} = \langle(l_3, 2), (l_2, 3), (l_5, 2)\rangle$
$Q_{21} = \langle(l_1, 3), (l_0, 3)\rangle$
$Q_{22} = \langle(l_3, 3), (l_1, 3)\rangle$
$Q_{23} = \langle(l_2, 2), (l_1, 3)\rangle$
$Q_{24} = \langle(l_1, 8)\rangle$
$Q_{25} = \langle(l_2, 3), (l_3, 3), (l_4, 3), (l_5, 2), (l_1, 3)\rangle$
$Q_{26} = \langle(l_6, 3), (l_1, 3), (l_4, 3), (l_7, 3)\rangle$
$Q_{27} = \langle(l_1, 9)\rangle$
$Q_{28} = \langle(l_1, 5), (l_2, 3), (l_3, 2), (l_5, 2), (l_4, 2)\rangle$
$Q_{29} = \langle(l_1, 5), (l_6, 4), (l_4, 3), (l_7, 3)\rangle$

Twelve experiments were conducted in total. In the first three experiments, seven sequences $\mathcal{Q}$ were utilized: $\{Q_1, Q_2, Q_3, Q_4, Q_5, Q_6, Q_7\}$, with four, three, and two data transfer processes, respectively. The subsequent three experiments employed six sequences $\mathcal{Q}$: $\{Q_8, Q_9, Q_{10}, Q_{11}, Q_{12}, Q_{13}\}$, with four, three, and two data transfer processes, respectively.

For the ensuing two experiments, four sequences $\mathcal{Q}$ were used: $\{Q_{14}, Q_{15}, Q_{16}, Q_{17}\}$, with three and two data transfer processes, respectively. Another experiment involved three sequences $\mathcal{Q}$: $\{Q_{18}, Q_{19}, Q_{20}\}$, with two data transfer processes. Subsequent experiments employed a different set of three sequences $\mathcal{Q}$: $\{Q_{21}, Q_{22}, Q_{23}\}$, with two data transfer processes. Additionally, a separate experiment utilized another set of three sequences $\mathcal{Q}$: $\{Q_{24}, Q_{25}, Q_{26}\}$, with two data transfer processes. Following these, another set of three sequences $\mathcal{Q}$ was employed: $\{Q_{27}, Q_{28}, Q_{29}\}$, with two data transfer processes. The experiment plans are displayed in Table 2.

Table 2. *Experiment Plans.*

Experiment	Number of sequences	Total number of time units	Number of Processors
1	7	97	4
2	7	97	3
3	7	97	2
4	6	40	4
5	6	40	3
6	6	40	2
7	4	42	3
8	4	42	2
9	3	22	2
10	3	17	2
11	3	34	2
12	3	38	2

The various rules were applied to each experiment, and the processing times of different scheduling methods were compared. The following scheduling rules were used to assign data transfers to data transfer processes in each experiment:

1. *Shortest processing time (SPT)*: Select the transfer with the shortest length, and if more than one transfer is found, select the sequence with the smallest total time units.
2. *First come, first served (FCFS)*: Select the transfer that belongs to the first sequence in the queue.
3. *Combination of scheduling rules (CSR)*: This combines the scheduling rules proposed in this paper.
4. *Random scheduling (R5)*: Select the transfer randomly chosen by the system.

Finally, all solutions were compared with the optimal solution denoted as the *optimal resource allocation plan* $(ORAP)$.

Table 2 aggregates the information related to the datasets and the scheduling rules used in each experiment. For instance, the first three experiments involved seven queries that took a total of 97 time units. Experiment 1 used these queries with four processes, while Experiments 2 and 3 used the same queries with three and two processes, respectively.

Four different scheduling rules were applied for each experiment, and the execution time was recorded in Table 3. The processing times were measured in time units, which were also applied to each transfer. Despite the same dataset being used for Experiments 1, 2, and 3, the results varied due to the different scheduling rules employed in each experiment. Finally, the scheduling rules and experiment plans outlined in Table 2 were utilized to calculate 48 test cases, and the execution time was recorded in Table 3.

Table 3. *Experiment results.*

Experiment	*ORAP*	*CSR*	*SPT*	*FCFS*	*R5*
1	25	25	30	30	33
2	33	33	34	34	37
3	49	49	55	52	54
4	10	10	15	14	16
5	14	14	16	15	19
6	20	20	21	20	23
7	14	14	16	15	18
8	21	21	23	23	25
9	11	11	17	13	19
10	9	9	14	11	16
11	17	17	17	18	17
12	21	21	23	28	30
Total	244	244	283	275	307

In summary, the total result of *CSR* was 244 time units, and comparing it with the *ORAP* result showed the same output results. As seen in column *SPT*, the total result was 283 time units, with 39 idle time units, and in column *FCFS*, the total result was 275, which had 31 idle time units when processing the 12 experiments. In the last rule, *R5*, the total processing time was 307, with 69 idle time units occurring in the experiments.

Based on the results obtained from the experiments, a comparison was made among the different scheduling methods: *ORAP*, *CSR*, *SPT*, *FCFS*, and *R5*.

1. *ORAP* and *CSR*: These methods consistently performed equally well, achieving the lowest time units in all the experiments. They were the most efficient and dependable in reducing time units.
2. *ORAP* and *SPT*: In most of the experiments, *ORAP* performed better than *SPT* by consistently achieving lower time units. Nevertheless, there were instances, such as Experiments 4 and 5, where *SPT* came close to matching *ORAP*'s performance.
3. *ORAP* and *FCFS*: In most of the experiments, *ORAP* tended to perform better than *FCFS* by maintaining lower time units. However, *FCFS* seemed to have higher time units than *ORAP* in several scenarios.
4. *ORAP* and *R5*: In most of the experiments, *ORAP* performed better than *R5* and consistently had lower time units. *R5*, by contrast, tended to have higher time units in comparison to *ORAP*.

After conducting several experiments, it was found that the *ORAP* and *CSR* scheduling methods consistently outperformed other methods in terms of minimizing time units.

CSR was found to be the most efficient and reliable scheduling method. Although the *SPT* method sometimes came close to *ORAP* and *CSR*, it generally took longer time

units in most of the experiments. Similarly, the *FCFS* and *R5* methods also had variable performance, but they consistently took longer time units compared to *ORAP* and *CSR*. Although the *SPT*, *FCFS* and *R5* methods performed moderately well, they generally took longer time units compared to *ORAP* and *CSR*. Across all the experiments, scheduling *ORAP* and *CSR* consistently had the lowest time units, ranging from 9 to 49 time units. Scheduling *SPT* and *FCFS* generally performed slightly worse, with time units ranging from 14 to 55. Scheduling *R5* had the highest time units among all the methods, ranging from 16 to 30.

Overall, *ORAP* and *CSR* were found to be the most efficient in minimising time units, while *R5* was the least efficient in most cases. *SPT* and *FCFS* fell in between, with varying performance depending on the experiment. In conclusion, *ORAP* and *CSR* are the top choices for minimising time units and ensuring efficient scheduling. Although *SPT*, *FCFS* and *R5* are viable alternatives, they may not consistently achieve the same level of efficiency as *ORAP* and *CSR*.

7. Summary and Conclusions

This work presented new algorithms that create efficient processing plans for parallel data transfers between the levels of multi-tiered persistent storage. The optimisation concentrates on balancing the workload among the data transfer processes and reducing the idle time of the processes. In particular, the paper addressed the problem of scheduling parallel data transfers between the storage levels in a pipelined data processing model. The proposed algorithms can discover the data transfers needed to process the operations of database applications, perform the partitioning of data transfers to reduce the number of conflicts in the access to persistent storage and apply rule-based scheduling to minimise the processing time.

The process starts from the conversion of query processing plans into EPNs and then into parallel data transfer plans. The scheduling algorithm minimises the total processing time for a given set of applications through the elimination of conflicts and delays during parallel data processing. The proposed rule-based algorithms aim at the even distribution of the workload among the data transfer processes.

This work offers a practical approach to the efficient scheduling of parallel data transfers in multi-tiered persistent storage. While the scheduling algorithms may not always yield the optimal solution, they balance the quality of the data transfer plans and the time taken to generate the plans. The approach is based on reducing the initial set of candidate data transfers through the application of elimination rules, allowing for flexibility in the trade-offs between the processing time and the quality of the solution. The experiments showed that rule-based scheduling is highly effective and produces the optimal results in most cases. In general, utilising multiple rules in combination yields superior results compared to using fewer scheduling rules. However, it is worth noting that employing more rule combinations requires more time for computation than using a single scheduling rule. As a result, the algorithms presented in this paper aim to select a limited number of rules and create the most effective combination to improve resource allocation for parallel processing within a shorter timeframe.

In conclusion, the paper presented practical algorithms for creating efficient processing plans that facilitate parallel data transfers in multi-tiered persistent storage systems. These algorithms strike a balance between optimisation and computational efficiency, making them suitable for real-world applications where practicality and time constraints often outweigh the pursuit of absolute optimality. The research contributions shed light on the challenges and solutions related to parallel data transfers in multi-tiered persistent storage, enable organisations to effectively handle large-scale data processing tasks and enhance performance and resource utilisation in commercial data analysis applications.

Note

This journal article is an extended version of a previously published article, *Scheduling Parallel Data Transfers in Multi-tiered Persistent Storage* [1] presented at the ACIIDS 2022 conference. The aim is to provide a more comprehensive analysis and include additional data and insights not covered in the initial article. Moreover, the extended journal article introduces innovative methodologies for generating operation sequences and expands upon the scheduling rules by incorporating more algorithms. Additionally, it presents new experimental results. These enhancements greatly improve the optimization, concentrating on balancing the workload among the data transfer processes and reducing their idle time.

ORCID

0000-0003-3985-5455 (Nan Noon Noon)
0000-0001-6492-5641 (Janusz Roman Getta)
0000-0002-4520-5021 (Tianbing Xia)

Author Contributions

Nan Noon Noon: Conceptualization, Formal Analysis, Funding acquisition, Investigation, Methodology, Resources, Validation, Visualization, Writing - original draft, Writing - review & editing

Janusz R. Getta: Resources, Supervision, Validation, Visualization, Writing - review & editing

Tianbing Xia: Resources, Supervision, Validation, Visualization, Writing - review & editing

Conflicts of Interest

The authors declare no conflicts of interest. The authors' funding supported the research. This statement indicates that the research is free from external conflicts or influences.

References

[1] N. N. Noon, J. R. Getta and T. Xia, *Scheduling Parallel Data Transfers in Multi-tiered Persistent Storage*, in Intelligent Information and Database Systems, ACIIDS 2022, Communications in Computer and Information Science, vol 1716, Springer, Singapore. https://doi.org/10.1007/978-981-19-8234-7_34

[2] P. Tsai, Spiceworks, *Spiceworks Research Examines Storage Trends in 2020 and Beyond*, (2020), https://community.spiceworks.com/blog/3240-spiceworks-research-examines-storage-trends-in-2020-and-beyond, last accessed 09 June 2023.

[3] R. Sheldon, G. Kranz and D. Raffo, Evaluator Group, *Tiered Storage*, (2021), https://searchstorage.techtarget.com/definition/tiered-storage, last accessed on 09 June 2023.

[4] E. Frachtenberg, G. Feitelson, F. Petrini, and J. Fernandez, *Adaptive parallel job scheduling with flexible coscheduling*, in IEEE Transactions on Parallel and Distributed Systems, vol. 16, no. 11, pp. 1066-1077, Nov. 2005, https://doi.org/10.1109/TPDS.2005.130.

[5] Y. Zhang, H. Franke, J. Moreira and A. Sivasubramaniam, *An integrated approach to parallel scheduling using gang-scheduling, backfilling, and migration*, in IEEE Transactions on Parallel and Distributed Systems, vol. 14, no. 3, pp. 236-247, March 2003, https://doi.org/10.1109/TPDS.2003.1189582.

[6] J. Blazewicz, Klaus H. Ecker, E. Pesh, G. Schmidt, M. Sterna and J. Weglarz, *Handbook on Scheduling From Theory to Practice*, 2nd edn. Springer, Cham, (2019). https://doi.org/10.1007/978-3-319-99849-7

[7] K. Wang, S. H. Choi and H. Qin, *A cluster-based scheduling model using SPT and SA for dynamic hybrid flow shop problems*, International journal of advanced manufacturing technology, 67, 2243-2258 (2013). https://doi.org/10.1007/s00170-012-4645-7

[8] R. Nehme and N. Bruno, N, *Automated partitioning design in parallel database systems*, in: SIGMOD, Association for Computing Machinery, New York, NY, USA, 1137–1148 (2011). https://doi.org/10.1145/1989323.1989444

[9] J. Li, J. F. Naughton and R.V. Nehme, *Resource bricolage and resource selection for parallel database systems*, The VLDB Journal, vol. 26, no. 1, pp. 31–54 (2017). https://doi.org/10.1007/s00778-016-0435-4

[10] T. Sthr, H. Mrtens and E. Rahm, *Multi-dimensional database allocation for parallel data warehouses*, In: Proceedings of the 26th International Conference on Very Large Databases, pp. 273–284, (2000).

[11] N. N. Noon, and J. R. Getta, *Automated Performance Tuning of Data Management Systems with Materializations and Indices*, In: Journal of Computer and Communications, 4, pp. 46–52 (2016). https://doi.org/10.4236/jcc.2016.45007

[12] N. N. Noon, and J. R. Getta, *Optimisation of query processing with multilevel storage*, In: Lecture Notes in Computer Science, 691–700. Da Nang, Vietnam Proceedings of the 8th Asian Conference, ACIIDS (2016). https://doi.org/10.1007/978-3-662-49390-8_67

[13] N. N. Noon, J. R. Getta and T. Xia, *Optimization Query Processing for Multi-tiered Persistent Storage*, 2021 IEEE 4th International Conference on Computer and Communication Engineering Technology (CCET), 2021, pp. 131–135. https://doi.org/10.1109/CCET52649.2021.9544285

[14] W. Reisig, *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*, Springer Publishing Company, Incorporated, 2013. https://doi.org/10.1007/978-3-642-33278-4