

# Online aggregation query space optimization technology for streaming data

Jiaxuan Su, Xiaou Ding, Haibin Qin, and Hongzhi Wang<sup>(✉)</sup>

**Abstract** We present a novel subspace segment tree data structure designed for online aggregation queries on streaming data, addressing the challenge of efficient query processing under limited storage space in embedded systems. Traditional segment tree structures require several times the storage space of the original data, making them impractical for resource-constrained embedded devices. The proposed Subspace Segment Tree optimizes space utilization by partitioning the tree structure into three hierarchical layers, namely the solid, dynamic, and imaginary layers, and implementing a dynamic memory allocation and reclamation strategy based on a memory pool. This approach allocates memory only when nodes are actively used, significantly reducing space overhead. Experimental results demonstrate that the proposed method achieves over 90% space optimization compared to traditional segment trees while maintaining comparable time complexity, effectively resolving the space efficiency issue for aggregation queries in embedded systems.

**Keywords** streaming data; aggregation query; online algorithm; segment tree

## 1 Introduction

Embedded database systems on multimodal networks<sup>[1, 2]</sup> are frequently utilized in real-time domains, demanding instantaneous results rather than delayed or estimated outcomes. The system must respond promptly to requests. Embedded devices serve as the crucial components for data management and processing in these systems. Many embedded devices exhibit high mobility, portability, and operate in non-fixed environments, with limited access to on-site technical support from IT personnel. Therefore, embedded database systems must ensure high reliability within limited resources, optimizing for various devices while avoiding significant losses in efficiency or accuracy. In practical application domains

such as internet communications and financial markets, multimodal networks generate vast amounts of highly detailed streaming data. Modern data stream monitoring applications require complex real-time queries (often in an exploratory mode) to identify anomalies or abnormal activities based on complex relationships among data (e.g., network intrusion detection or telecommunication fraud detection). Operating on streaming data poses numerous technical challenges, as complex queries in traditional data sources can only be executed offline, while real-time queries are typically limited to simple calculations. These challenges are gradually being addressed within the database, systems, algorithms, networking, and other computer science communities, with research focusing on enabling complex queries over streaming data through techniques like SQL on streams<sup>[3]</sup>. Recent work highlights the critical importance of spatial optimization in embedded systems. For instance, Xu et al. demonstrate that memory-optimized B-tree variants can achieve up to  $30\times$  performance improvement on range-scan workloads<sup>[4]</sup>, while SINLK shows a  $2.33\times$  throughput boost by migrating hot paths to fast memory on Compute Express Link (CXL)-attached heterogeneous systems<sup>[5]</sup>. These results underscore that algorithmic spatial optimization, rather than hardware upgrades alone, is essential for achieving high performance under tight memory constraints.

This paper delves into the pertinent issues of spatial optimization for online aggregation queries on streaming data. Firstly, the issue of miniaturization of embedded data is discussed. The miniaturization issue arises from the limited storage space that various embedded database systems must take into account during application. Consequently, the crux of implementing embedded systems lies in achieving the miniaturization of embedded database systems and adopting appropriate memory management strategies<sup>[6]</sup>. Suitable miniaturization methods and memory management strategies can enhance the utilization of data storage space and increase the storage capacity of databases. The primary implementation methods involve optimizing compressed storage of data and optimizing relational schemas. Based on the specific application requirements of the system, the selection of essential component functionalities is made.

Memory in embedded systems is highly limited. Apart from

• Jiaxuan Su, Xiaou Ding, Haibin Qin, and Hongzhi Wang are with Harbin Institute of Technology, Harbin 150001, China. E-mail: leadovahs@outlook.com; dingxiaou.hit@163.com; qin\_haibin@hit.edu.cn; wangzh@hit.edu.cn.

<sup>(✉)</sup> To whom correspondence should be addressed.

Manuscript received: 2026-04-10; revised: 2026-06-03; accepted: 2026-07-14

the storage space occupied by the operating system and the database itself, data processing also incurs memory overhead. This includes the consumption of memory resources resulting from operations such as updating, querying, modifying, and deleting data in database tables after data is loaded. Consequently, in addition to streamlining the internal structure of the database, an essential approach to saving memory resources is to select an appropriate memory management strategy<sup>[7–12]</sup>.

There are various approaches to address the challenges of aggregate queries<sup>[13]</sup>. The segment tree data structure<sup>[14]</sup> is often utilized to solve practical problems<sup>[15–18]</sup>. In this paper, we employ segment trees because they naturally support exact interval aggregation queries over ordered keys, including sum, count, minimum, maximum, and algebraic aggregates that can be composed from distributive summaries. This property is well aligned with streaming aggregation tasks in which new observations update indexed positions and users repeatedly ask for aggregate values over recent or semantically meaningful ranges. The specific functionalities and operations of segment trees will be elaborated in the subsequent sections. The problem we address concerns memory management in segment trees. In practical usage, a few nodes near the root are frequently accessed, requiring high query performance, while a large number of nodes near the leaves are rarely accessed, necessitating high space overhead. This access pattern dichotomy is a common challenge in hierarchical data structures, as highlighted by recent research on adaptive indexing and memory tiering. For example, the Adaptive and Robust Memory Tiering System (ARMS) achieves  $1.26\text{--}2.32\times$  speedup by dynamically adapting memory-tiering policies based on workload shifts<sup>[19]</sup>, while LITune demonstrates up to  $17\times$  throughput improvement by using reinforcement learning to auto-tune learned index parameters online<sup>[20]</sup>. These works reinforce that a one-size-fits-all approach is suboptimal, and that hierarchical structures benefit from workload-aware adaptation. Constraint-based anomaly detection methods have also been explored for multivariate time series data<sup>[21–23]</sup>. Therefore, a single metric cannot simultaneously meet these conflicting requirements.

Existing alternatives do not fully match this setting. Fenwick Trees (Binary Indexed Trees) offer a space-efficient alternative for prefix-sum queries<sup>[24]</sup>, but they are limited to invertible operations and cannot directly support non-invertible aggregates such as range minimum or maximum. Hash-based aggregation structures are efficient for key-based grouping or point lookup, yet they do not preserve ordered interval structure and often require additional memory for

maintaining hash tables. Approximate query processing can reduce latency or storage pressure, but it trades accuracy for estimation error, which is not suitable when embedded monitoring requires exact alarms or deterministic summaries. Wavelet-tree-like structures and other specialized indexes can be effective for value-range or compressed static workloads, but they introduce additional representation and maintenance costs for online updates. The research gap addressed in this paper is therefore not the design of a new aggregation operator, but the spatial optimization of an exact, interval-oriented, update-capable hierarchical index under embedded memory constraints.

Therefore, the problem we need to address is stated as follows: On embedded devices with limited storage space, where indexes that are multiples of the original data cannot be saved, we aim to implement a single-field indexing method for embedded databases to solve range aggregation and indexing queries that can be addressed using segment trees. This involves creating segment trees for single indexing fields to perform aggregation operations on other fields. Our work addresses a critical research gap in the efficient spatial optimization of hierarchical data structures for streaming data, particularly in resource-constrained environments. While recent advances in learned indexes<sup>[25]</sup> and adaptive systems<sup>[26]</sup> show promise, they often focus on specific data types or require significant computational overhead for model training and online tuning. Our approach specifically targets the core challenge of balancing time and space efficiency in segment trees for general streaming data aggregation, offering a specialized solution for embedded multimodal networks. Adopting the trade-off principle of time for space, we aim to accelerate aggregation while ensuring relatively low space consumption and maintaining a low occupancy rate. Ultimately, the accuracy of the proposed approach needs to be demonstrated and verified.

Based on the fundamental data structure of segment trees, this paper redesigns its memory implementation. Specifically, we address the hierarchical organization and memory management of segment trees. The contributions of this paper are as follows:

- (1) A subspace segment tree structure is proposed, which consists of three hierarchical layers.
- (2) Algorithms for construction, querying, and modification on the subspace segment tree are presented.
- (3) A memory management method for the subspace segment tree is introduced, including mechanisms for memory allocation and reclamation.

(4) Experiments are designed and implemented to validate the effectiveness of the proposed approaches.

## 2 Preliminaries

### 2.1 Aggregate Operations

In this section, we define the concept of aggregation operations. There exist numerous aggregation functions<sup>[27, 28]</sup>. We categorize these aggregation functions into three groups:

i. Distributed Aggregation.

We define a distributed aggregation function  $f$  as satisfying the following conditions:

(1) For a sequence  $S$  of length 1, containing a single element  $x$  (i.e.,  $S = \{x\}$ ), we define an initial quantity computation function  $g$ , such that:

$$f(S) = g(x) \quad (2.1)$$

(2) For two sequences  $S_1$  and  $S_2$ , each with a length of at least 1, that are concatenated to form a new sequence  $S = S_1.S_2$ , there exists a function  $h$  such that:

$$h(S) = h(S_1.S_2) = h(f(S_1).f(S_2)) \quad (2.2)$$

In this context, the symbol “.” represents the concatenation operation of sequences, where  $A.B$  signifies the new sequence created by appending sequence  $B$  to the end of sequence  $A$ .

(3) There exists an element  $z$  such that for any element  $x$ , when:

$$f(S.\{z\}) = f(\{z\}.S) = f(S) \quad (2.3)$$

We refer to  $z$  as the zero element. Distributed aggregation includes aggregation methods such as counting, summation, minimum, maximum, product, and sum of squares. Table 1 summarizes representative definitions of the corresponding functions  $g$  and  $h$  for these aggregations.

**Table 1 The Aggregation Function**

| $f$            | $g(x)$ | $h(S_1.S_2)$           | $z$       |
|----------------|--------|------------------------|-----------|
| sum            | $x$    | $f(S_1) + f(S_2)$      | 0         |
| max            | $x$    | $\max(f(S_1), f(S_2))$ | $-\infty$ |
| count          | 1      | $f(S_1) + f(S_2)$      | 0         |
| sum_of_squares | $x^2$  | $f(S_1) + f(S_2)$      | 0         |

For example, let's consider *count*1, which is defined as counting the number of elements in sequence  $S$  that have a value of 1. In this case,  $g(x)$  is defined as 1, regardless of the value of  $x$ , as each element counts as 1. Subsequently,  $h(S_1.S_2) = f(S_1) + f(S_2)$  is used to accumulate the results, producing the final aggregated value.

ii. Algebraic Aggregation.

We define algebraic aggregation as the combined aggregation of two or more different distributed aggregations. For

instance, the average of a series of values can be calculated by combining the sum and count. Applications of algebraic aggregation include mean, standard deviation, geometric mean, and so on.

iii. Holistic Aggregation.

All other types of aggregation fall into this category.

The focus of this work lies in distributed and algebraic aggregation, while holistic aggregation is beyond our discussion. For the sake of brevity and generality, the remainder of this paper will discuss the algorithm construction and operation based on the distributed aggregation max operation. This discussion applies to the general handling of all distributed and algebraic aggregations.

The remaining part of this paper will be discussed based on a sequence  $S$  of length  $N$ , where each element is a real number. Let's define its elements as  $S_1, S_2, \dots, S_N$ .

### 2.2 Definitions Related to Segment Tree Structure

A segment tree is a commonly used data structure for maintaining information about intervals. It enables operations such as single-point modification, interval modification, and interval aggregate queries (e.g., interval summation, finding the maximum or minimum value within an interval) with a time complexity of  $O(\log N)$ . This makes it suitable for streaming aggregation problems in which the data order is meaningful and the query workload repeatedly requests exact aggregates over contiguous ranges, such as time windows, ordered sensor identifiers, or ranked positions. Compared with a linear scan, the segment tree avoids recomputing the whole interval after each update. Compared with prefix-only structures, it preserves a general merge tree that can support both invertible and non-invertible distributive aggregates. The application of segment trees to streaming data aggregation has been explored in recent research. For instance, TUCKET generalizes the segment tree concept to streaming tensor data, achieving  $3\times$  lower latency for range aggregation through precomputed decompositions<sup>[26]</sup>.

The basic construction method of a segment tree involves recursively dividing each interval of length greater than 1 into left and right subintervals, forming a binary tree-like structure. The information of an interval is obtained by merging the information of its left and right subintervals.

Regarding the structure of a segment tree, we provide the following definitions:

**Leaf Node:** A node in the tree representing a single value. The information recorded in each leaf node includes the identifier  $I$ , value  $V$ , and parent node  $F$ . Leaf nodes contain information about only one value and do not describe

information or aggregate values for an interval.

**Interval Node:** A node in the tree representing an interval. The information recorded in each interval node includes the left boundary  $L$ , right boundary  $R$ , left child  $C_L$ , right child  $C_R$ , parent node  $F$ , and aggregate value  $A$ .

**Subtrees of Interval Nodes:** Each interval node satisfies  $L < R$  and has both left and right subtrees. The dividing position  $M$  is defined as:

$$M = \frac{L + R}{2} \quad (2.4)$$

The left subtree has a left boundary of  $L$  and a right boundary of  $M$ , while the right subtree has a left boundary of  $M + 1$  and a right boundary of  $R$ . The dividing position  $M$  splits the interval  $[L, R]$  evenly into two closed intervals,  $[L, M]$  and  $[M + 1, R]$ . Although the sizes of these two split intervals are not equal when the interval length is odd, it does not affect our calculations or complexity estimation.

**Segment Tree:** A segment tree structure is either:

- (1) A tree structure consisting of a single leaf node;
- (2) A binary tree structure with an interval node as the root, and both left and right subtrees being segment trees.

We denote the root node of the segment tree as ROOT.

### 2.3 Basic Operations of Segment Tree

The segment tree mentioned in this article will include the following three basic operations:

**Initialization( $S$ ):** The operation of converting the sequence  $S$  into a segment tree structure. The root node's interval is recorded as  $[1, N]$ , and it is divided into two subintervals,  $[1, \frac{N}{2}]$  and  $[\frac{N}{2} + 1, N]$ , for top-down interval decomposition and tree construction. This process continues until the intervals are broken down into single-value nodes. Then, the aggregation values are calculated from the bottom up, layer by layer, and stored in the interval nodes.

**Query( $[L, R]$ ):** The operation of querying the aggregate value of the interval  $[L, R]$ . The interval is decomposed into several intervals that exist in the segment tree. The aggregate values of the corresponding interval nodes of these intervals are aggregated to achieve faster query results. Compared to general linear structures, the data merging strategy of the segment tree allows aggregate queries to be completed in logarithmic time. The pre-stored data in the nodes significantly reduces the time consumed for aggregate queries.

**Modify( $i, v$ ):** The operation of modifying the value of the  $i$ -th element  $S_i$  in sequence  $S$  to  $v$ .

This operation affects not only the leaf node corresponding to that element in the segment tree but also all interval nodes

in the tree that include this element. After modifying the leaf node, all involved interval nodes need to be re-aggregated from the bottom up. This update process has a worst-case time complexity of  $O(\log N)$ , which is efficient for streaming data workloads.

The above definitions directly lead to the design problem considered in this paper. The standard segment tree offers a useful query/update abstraction for exact range aggregation, but its complete materialization produces a large number of interval nodes. In embedded streaming systems, this memory overhead competes with the operating system, the database engine, buffering, and application logic. The proposed method therefore keeps the segment tree query model while changing how nodes are materialized and reclaimed in memory.

### 2.4 Problem Overview

This article aims to optimize aggregation queries for indexed columns. Since aggregation queries and modifications can be accomplished using segment trees, we have developed a data structure and algorithm based on the segment tree structure.

Conventional segment trees require additional space several times greater than the original sequence  $S$ , making it unfeasible for some sensor devices to bear such memory overhead. Meanwhile, we have observed that most nodes in the segment tree structure are not utilized during queries, and the storage space occupied by these nodes represents a significant waste.

To address these issues, we propose a segment tree structure based on a comprehensive memory management strategy that combines memory pooling and tree depth balancing. We define this structure as a subspace segment tree. The core idea is to modify the existing segment tree structure such that not all nodes are allocated space in memory during tree construction. Instead, we dynamically allocate space for tree nodes in a memory pool based on actual usage. This approach enables us to perform aggregation operations and related query operations with similar performance to a complete segment tree while using less memory space. When expansion and computation are required, we perform on-the-spot computations for specific nodes while simultaneously constructing and generating the tree structure, ensuring that the complexity of consumption remains close to the original  $\log N$  complexity.

Regarding time complexity, the worst-case theoretical time overhead for a single operation (query or modification) remains at the  $O(\log N)$  level, consistent with that of a standard segment tree. This is because, in the worst case, the algorithm may need to traverse from the root to a leaf node, which

involves  $O(\log N)$  steps. The specific constant factors in the time cost are influenced by parameters  $D_1$  and  $D_2$ , as well as the device's performance characteristics, but they do not affect the fundamental algorithmic complexity. Here,  $D_1$  and  $D_2$  are depth thresholds that partition the tree levels into the solid, dynamic, and imaginary layers; detailed definitions are given later. For space complexity, while the algorithm achieves significant optimization in practice, the asymptotic space complexity remains  $O(N)$ , as the number of nodes in a segment tree is linearly related to the size of the input sequence  $N$ . The space optimization is primarily a constant-factor improvement determined by the parameters  $D_1$  and  $D_2$ , rather than a reduction in the order of complexity.

In the context of our problem, we do not consider transaction management issues and focus the aggregation method on distributed aggregation. We assume that modification operations occur relatively infrequently with high costs, while query operations are more frequent. For a single query, we design the following theoretical process:

- (1) Split the queried intervals into two categories: currently active nodes (i.e., nodes that are already present in the tree structure) and inactive nodes (i.e., nodes that have not been constructed or have been destructed).
- (2) Update the weights of active nodes and maintain the priority queue structure.
- (3) Calculate the amount of space required to activate all inactive nodes.
- (4) If the remaining memory pool is insufficient, attempt to identify memory pool nodes to release based on the priority queue of weight and depth, popping out enough memory pool nodes until there is sufficient space for the active nodes.
- (5) Attempt to construct all inactive nodes from bottom to top. The above space release process ensures that the space required for activating nodes can be satisfied.

The subspace segment tree is not intended as a standalone database but as an optimized indexing component compatible with existing embedded DBMSs such as SQLite or Berkeley DB. It can be integrated as a specialized index module for range aggregation queries, coexisting with traditional B+ trees or hash indexes. For example, in SQLite, it could be implemented as a virtual table extension, allowing SQL queries to leverage its efficiency for specific aggregation workloads without altering the core database engine.

### 3 Proposed Approach

#### 3.1 The State of Interval Nodes

Based on our design of the subspace segment tree structure, the construction of the segment tree formed by interval nodes exists only as a theoretical concept, and it does not necessarily occupy actual memory space. We define two states for each interval node: active state and inactive state.

Nodes in the active state are allocated storage space from the memory pool to store node data, including the interval endpoints, interval size, aggregated results within the interval, and the node's memory pool weight. Nodes in the inactive state are those that have not been constructed and only exist theoretically, not occupying actual memory space.

#### 3.2 Hierarchical Classification of the Segment Tree Structure

Assuming the depth of the segment tree is  $D$ , we define the root node depth as 1 and the leaf node depth as  $D$ .

We set two depth thresholds:  $D_1$  and  $D_2$  (where  $0 < D_1 < D_2 < D$ ). These parameters divide the segment tree into three hierarchical layers from top to bottom: the solid layer, the dynamic layer, and the imaginary layer. For any node, if its depth  $d$  satisfies  $0 < d \leq D_1$ , it is considered to be in the solid layer; if  $D_1 < d \leq D_2$ , it is in the dynamic layer; and if  $D_2 < d \leq D$ , it is in the imaginary layer.

The node information in the solid layer will always be recorded in memory. The node information in the imaginary layer will never be recorded in memory, and only on-the-spot computations will be performed when needed. For nodes in the dynamic layer, there are two operations: "activation" and "destruction", which allow them to be continuously activated and released in memory usage. A fixed-size memory pool is provided for these nodes, and the priority of each node occupying the memory pool is determined based on the last time it was used. When a node is put into use, it must be in an active state. When the memory pool is depleted, the least recently used (LRU) node in the dynamic layer will be prioritized for destruction, releasing its space for newly constructed dynamic layer nodes. This allows the available memory pool to be managed based on actual usage.

The choice of  $D_1$  and  $D_2$  controls the main trade-off of the proposed structure. A larger  $D_1$  keeps more upper-level nodes permanently resident, which reduces activation overhead for frequent and fine-grained queries but increases baseline memory consumption. A smaller  $D_1$  improves space saving, but it may increase the number of dynamic-layer activations when the query workload touches many different

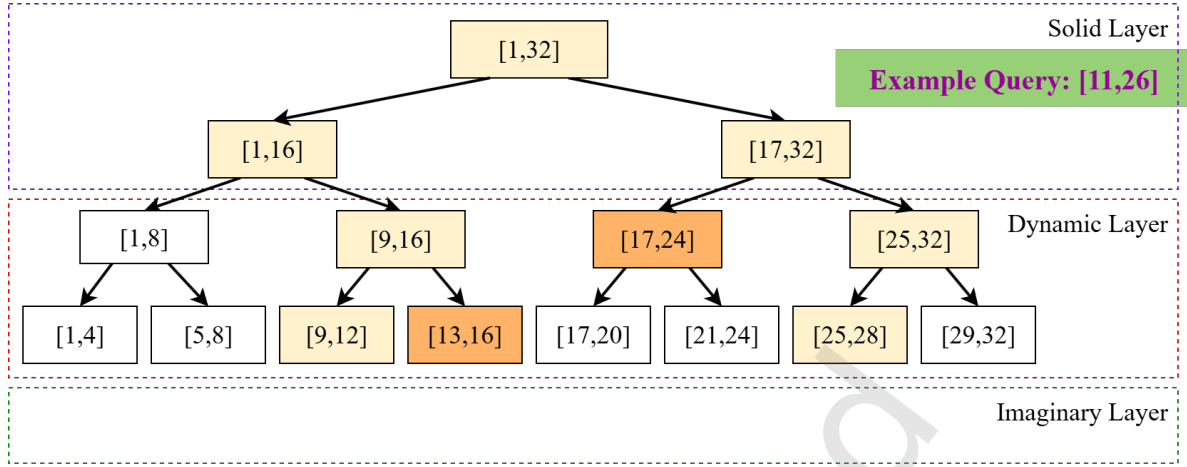


**Fig. 1** Schematic diagram of the subspace segment tree structure

intervals. The parameter  $D_2$  controls how much of the lower tree is handled through on-the-spot computation. Through practical usage and testing, we have found that maintaining the total depth of the imaginary layer at 2 to 3 has a relatively small impact on the overall performance of the algorithm, with acceptable time overhead. This means that the interval size associated with each node in the imaginary layer is roughly at the level of 2 or 4, with a maximum of no more than 8. In practice,  $D_1$  should be increased when the workload contains many repeated narrow-range queries, while it can be reduced when queries are coarse, stable, or strongly concentrated near upper-level intervals. The value of  $D_2$  should be selected so that the imaginary layer remains shallow enough to bound the cost of direct aggregation. Fig. 1 illustrates the resulting three-layer organization and shows how a query intersects nodes across the solid, dynamic, and imaginary layers.

### 3.3 Activation and Destruction of Nodes

The activation algorithm refers to the process of constructing nodes that have not been created or have been destroyed. Nodes in an inactive state do not occupy memory space, and when they are supposed to be put into use, we construct the node and allocate memory space for it.

#### Algorithm 1 Node Activation Algorithm (ActiveNode)

```

1: if memory not enough then
2:   collect()
3: end if
4: Node = malloc_new_node()
5: refresh(Node)
6: return Node

```

The process of Algorithm 1 is as follows: We first check

if the remaining memory pool is sufficient (line 1). If not, we execute Algorithm 2 to release space (line 2) to ensure enough room for allocating a new node. Then, we allocate new memory space for the current node (line 4), update the last usage time of the current node (line 5), and return the node (line 6).

Nodes that have been unused for a long time should be destroyed to release their space and make it available for other new and frequently used nodes. The destruction of a node should occur only when all its child nodes have been destroyed or are inactive. The node weight update mechanism mentioned below ensures that all child nodes are destroyed before their parent node.

When space needs to be released, we dequeue the least-used node from the root and release its memory space. We repeat this process until there is enough memory space to create a new node. The heap structure used here guarantees that the least recently used node is always dequeued, and the memory space management methods involved will be described in detail in subsequent chapters.

#### Algorithm 2 Space Release Algorithm (Collect)

```

1: while current rest memory not enough do
2:   Node = heap.top()
3:   Set the child pointer of Node.F to null
4:   delete(Node)
5: end while

```

The process of Algorithm 2 is as follows: If there is insufficient remaining space, the algorithm continues (line 1). First, the least recently used node is dequeued from the time heap (line 2). Then, the child pointer of its parent node,

represented by `Node.F`, is set to null (line 3). Following that, the memory space occupied by the current node is released (line 4).

### 3.4 Initial Tree Construction Method

The tree construction method for the subspace segment tree differs little from the typical segment tree construction approach (briefly introduced in Section 2.3). In essence, we construct the tree using the standard method and then discard several lower-level nodes.

The intermediate variables during this tree construction process do not require space for storage. The idea is similar to dynamic programming algorithms that utilize reusable space. We denote the aggregate values returned by the left and right subtrees by  $A_L$  and  $A_R$ , respectively. The specific algorithm is outlined below:

---

**Algorithm 3** The Metamethod for Constructing the Subspace Segment Tree (Initialization)

---

**Input:** Construction sequence  $S$ , current interval left endpoint  $L$ , current interval right endpoint  $R$

**Output:** Returns the node `Node` and the aggregated value  $A$

---

```

1: if  $L == R$  then
2:   return null,  $g(S[L])$ 
3: end if
4:  $M = (L + R)/2$ 
5:  $C_L, A_L = \text{Initialization}(S, L, M)$ 
6:  $C_R, A_R = \text{Initialization}(S, M + 1, R)$ 
7:  $A = f(A_L, A_R)$ 
8: if  $d \leq D_1$  then
9:   Node = malloc_new_node()
10:  Node. $\{L, R, C_L, C_R, A\} = \{L, R, C_L, C_R, A\}$ 
11:  return Node,  $A$ 
12: end if
13: return null,  $A$ 

```

---

Algorithm 3 is a meta-method for constructing segment trees, where it builds either an interval node for the range  $[L, R]$  in sequence  $S$  (when  $L < R$ ) or a leaf node (when  $L = R$ ). It returns the constructed result `Node` for the current node as well as the aggregated value  $A$  for that node. Note that during the initialization process of the subspace segment tree, we only actually store all information for nodes in the solid layer. For nodes in the dynamic and imaginary layers, we only compute their aggregated values without actually allocating space or creating tree nodes that occupy memory. When a node is generated in the dynamic or imaginary layer, the first return value of this method will be null.

The process of Algorithm 3 is as follows: First, it determines if the length of the current interval is 1 (i.e., not a proper interval,  $L = R$ , line 1). If true, it is considered a leaf node, and no construction is performed (returning null). It then generates the single-node aggregated value ( $g(x)$ ) for the current node as the aggregated result (line 2). Next, it finds the midpoint of the current interval (line 4) and recursively calls Algorithm 3 itself to construct the left and right subtrees (lines 5 and 6). Subsequently, it aggregates the aggregated results from the left and right subtrees to produce the aggregated result for the current node (line 7). If the current node is in the solid layer (line 8), memory space is allocated (line 9), the node values are initialized (line 10), and the current node is returned (line 11). Otherwise, no memory space is allocated, and only the aggregated result is returned (line 13).

Algorithm 3 is a method that recursively calls itself, with the recursive exit being the construction of leaf nodes (where no actual construction node is created, only the aggregation of initial values is returned). The entry point for calling this recursive method can be  $\text{Initialization}(S, 1, N)$ , which uses the entire range  $[1, N]$  to construct the root node of the segment tree by applying it to the entire sequence  $S$ .

### 3.5 Query Construction

Apart from initializing the segment tree by accessing all of its nodes, when subsequently accessing nodes in the imaginary layer to retrieve information or calling inactive or destructed nodes in the dynamic layer, we use the query construction method to aggregate the target range and generate the required nodes. Based on the position of the node, we decide whether to generate the current node and its child nodes.

When a query is performed on an inactive dynamic layer node, at the end of the query, that node needs to be generated and added to the tree structure. However, if the query requires the entire range of the interval, only the entire node for that range needs to be generated, without the need to generate its child nodes. Our algorithm controls the scope and number of nodes generated in the dynamic layer based on the query content, avoiding significant memory overhead from generating a large number of nodes at once.

For nodes that are not yet expanded in the tree, we use a simple method for function aggregation, which can be quite time-consuming. In practical production, most queries involve ranges that are frequently queried, and extreme cases requiring simple decomposition are rare. Each query will have at most one root node whose information can be derived from its two newly constructed subtrees, and the total time overhead consumed in the overall streaming query process

is not particularly significant. We use  $\text{Aggregate}([L, R])$  to denote the direct computation of  $f$  over the original sequence elements in interval  $[L, R]$ . We denote the aggregate result of the current query by  $Q_A$  and the results returned by its left and right recursive calls by  $Q_L$  and  $Q_R$ , respectively.

---

**Algorithm 4** Query Construction (PlainQueryGenerate)
 

---

**Input:** Node, Left Endpoint  $L$ , Right Endpoint  $R$

**Output:** Node and query aggregate value  $Q_A$

```

1: if  $d > D_2$  then
2:    $Q_A = \text{Aggregate}([L, R] \cap [\text{Node}.L, \text{Node}.R])$ 
3:   return null,  $Q_A$ 
4: end if
5: ActiveNode(Node)
6:  $\text{Node}.A = \text{Aggregate}([\text{Node}.L, \text{Node}.R])$ 
7: if  $[L, R] \cap [\text{Node}.L, \text{Node}.R] = \emptyset$  then
8:   return Node,  $z$ 
9: end if
10: if  $[L, R] \supseteq [\text{Node}.L, \text{Node}.R]$  then
11:   return Node,  $\text{Node}.A$ 
12: end if
13:  $M = (\text{Node}.L + \text{Node}.R) / 2$ 
14:  $(\text{Node}.C_L, Q_L) =$ 
    $\text{PlainQueryGenerate}(\text{Node}.C_L, L, M)$ 
15:  $(\text{Node}.C_R, Q_R) =$ 
    $\text{PlainQueryGenerate}(\text{Node}.C_R, M + 1, R)$ 
16:  $Q_A = f(Q_L, Q_R)$ 
17: return Node,  $Q_A$ 
  
```

---

Algorithm 4 performs a naive query on the interval  $[L, R]$  while simultaneously generating the aggregated result of the single query and the dynamic layer nodes. It consists of two specific parts: one is to complete the interval aggregation query for a specific node on the range  $[L, R]$ , noting that when calling this algorithm, there is no guarantee of the relationship between the queried interval and the current node. They may be subsets of each other, may intersect but not be subsets, or may not intersect at all. Specific return results need to be given based on different situations. The second part involves the potential activation of dynamic layer nodes and memory allocation, along with the recording of data changes in the tree structure. As the process recursively calls itself, a series of subtree nodes may be generated during this process. For notational simplicity,  $\text{Node}.L$ ,  $\text{Node}.R$ , and related fields denote the interval metadata of the conceptual node even before it is activated; an implementation passes this metadata as parameters until memory is allocated.

The process of Algorithm 4 is as follows: If the current node

is an imaginary layer node (line 1), a naive query aggregation is performed (line 2), and the result is generated and returned (line 3). For a dynamic layer node, new memory space is allocated for this call (line 5), and the total aggregation information for the node itself is generated (line 6). When the queried interval does not intersect with the current node's interval (line 7), the zero element of the aggregation function is returned as the aggregated result of the current query (line 8). When the current node's interval is a subset of the queried interval (line 10), the aggregated result of the entire current interval is returned as the aggregated result of this query (line 11). Excluding these two cases, the queried interval must intersect with the current node's interval but does not fully encompass it. This requires splitting the current interval and performing further queries on the subtrees. Therefore, find the midpoint  $M$  of the current interval (line 13) and recursively call Algorithm 4 itself to perform aggregation queries and possible node construction on the left and right subtrees (lines 14 and 15). Then, based on the aggregation query results of the two subtrees, generate the aggregated query result on the current interval (line 16). Finally, return the generated interval node and the aggregated result (line 17).

Algorithm 4 is also a recursive method that calls itself. There are two recursive exit points: when performing a query on an imaginary layer node, the recursion terminates; and when querying a node on the dynamic layer, if the interval does not require further invocation of subtree nodes on the dynamic layer, the recursion will also terminate. This algorithm may be called during the process of performing tree-like queries on solid-layer nodes or activated dynamic-layer nodes to achieve aggregation queries on non-activated dynamic layer nodes and imaginary layer nodes, and activate the required nodes in the dynamic layer.

### 3.6 Query Method on Tree Structure

The query method on the subspace segment tree differs little from the conventional segment tree query method (briefly introduced in Section 2.3). The approach to dividing the interval and splitting the subproblems is quite similar. The only exception is when the query node is empty, in which case the current query needs to be handed over to the query construction method (i.e., Algorithm 4) for processing. For a given interval  $[L, R]$ , the specific algorithm for executing the query on node Node is as follows:

Algorithm 5 serves as the query meta-method for the subspace segment tree. It performs a query on node Node within the interval  $[L, R]$  and returns the aggregate value  $A$  obtained from the query on the current node. If Node in

**Algorithm 5** Query Meta-Method (Query)

**Input:** Node (node to be queried),  $L$  (left endpoint of the query interval),  $R$  (right endpoint of the query interval)

**Output:** Return the aggregate value  $A$

```

1: if Node = NULL then
2:   Node,  $Q_A$  = PlainQueryGenerate(Node,  $L$ ,  $R$ )
3:   return  $Q_A$ 
4: end if
5: if Node. $L$  =  $L$  and Node. $R$  =  $R$  then
6:   return Node. $A$ 
7: end if
8:  $M$  = (Node. $L$  + Node. $R$ )/2
9: if  $R \leq M$  then
10:  return Query(Node. $C_L$ ,  $L$ ,  $R$ )
11: end if
12: if  $L > M$  then
13:  return Query(Node. $C_R$ ,  $L$ ,  $R$ )
14: end if
15: return  $f$ (Query(Node. $C_L$ ,  $L$ ,  $M$ ),
    Query(Node. $C_R$ ,  $M + 1$ ,  $R$ ))

```

the dynamic layer is not constructed, it will complete the construction of the node and store the construction result. This process will affect the shape of the nodes in the dynamic layer, achieving the effect of reshaping the segment tree to adapt to the query.

The process of Algorithm 5 is as follows: Firstly, it checks if the current node is empty. If the current node is null (line 1), indicating that the node is either in the dynamic layer or the imaginary layer, it will invoke Algorithm 4 to generate the node, and for nodes in the dynamic layer, it will store the generation result (line 2). Subsequently, it produces the aggregate result and returns it (line 3).

If the interval stored in the current node matches the query interval exactly (line 5), it returns the aggregate result of the current node (line 6). This step is the core operation of the segment tree for optimizing the time performance of interval queries.

Otherwise, it finds the mid-point of the current interval (line 8). If the right endpoint of the current query is to the left of the mid-point (line 9), a recursive query is performed on the left subtree (line 10). If the left endpoint of the current query is to the right of the mid-point (line 12), a recursive query is performed on the right subtree (line 13). In both cases not being satisfied, indicating that the mid-point falls within the query interval, the query is split. Recursive queries are performed on both the left and right subtrees, and the results are aggregated to serve as the output result of the current

algorithm (line 15).

Algorithm 5 is also a recursive method that calls itself. The recursive exit conditions are when the node is null and another algorithm is called, or when the node interval matches the query interval. To initiate a query on the interval  $[L, R]$ , one can invoke the recursive method using Query(ROOT,  $L$ ,  $R$ ), where ROOT is the root node of the segment tree.

**3.7 Modification Method on Tree Structure**

The single-point modification method on the subspace segment tree aims to update the value at a specific point in the sequence and affect all relevant interval nodes accordingly. The main difference from the conventional segment tree modification method lies in the way the aggregate values are updated. Since we cannot guarantee that both the left and right subtree nodes of every node are in an active state, we cannot directly access the aggregate values on the subtrees. Instead, we need to perform a query operation on either the left or right subtree to obtain the aggregate results. The specific algorithm is as follows:

**Algorithm 6** Single-Point Modification (Modify)

**Input:** Node (the node to be modified),  $i$  (the index of the modification point), NewValue (the new value for the modification)

**Output:** The current aggregate value  $A$  of the node

```

1: if Node is null then
2:    $S[i]$  = NewValue
3:   Node,  $Q_A$  = PlainQueryGenerate(Node,  $i$ ,  $i$ )
4:   return Node. $A$ 
5: end if
6:  $M$  = (Node. $L$  + Node. $R$ )/2
7: if  $i \leq M$  then
8:    $A_L$  = Modify(Node. $C_L$ ,  $i$ , NewValue)
9:    $A_R$  = Query(Node. $C_R$ ,  $M + 1$ , Node. $R$ )
10: end if
11: if  $i > M$  then
12:    $A_L$  = Query(Node. $C_L$ , Node. $L$ ,  $M$ )
13:    $A_R$  = Modify(Node. $C_R$ ,  $i$ , NewValue)
14: end if
15: Node. $A$  =  $f$ ( $A_L$ ,  $A_R$ )
16: return Node. $A$ 

```

Algorithm 6 implements the single-point modification method on the subspace segment tree. It modifies the value at position  $i$  of Node, sets the new value to NewValue, and then returns the updated aggregate value of the current node for use in generating the new aggregate value of the parent node.

The specific process is as follows:

First, it checks if the node is null. If the current node is null (line 1), it modifies the single-point value (line 2), invokes the query construction method to construct the current node (line 3), and then returns the aggregate result of the current node (line 4). Since the construction of the subspace segment tree includes an imaginary layer, and leaf nodes are always in the imaginary layer, non-null nodes must be in the dynamic or solid layer. Therefore, the recursive exit check can be skipped, and the re-aggregation calculation can be performed directly on the current node.

The algorithm finds the midpoint of the current node's interval (line 6) and determines the location of the node to be modified. If it falls within the left subtree (line 7), it recursively calls the single-point modification method on the left subtree (line 8) and performs a query method on the right subtree (line 9). If it falls within the right subtree (line 11), it performs a query method on the left subtree (line 12) and recursively calls the single-point modification method on the right subtree (line 13).

Finally, the algorithm aggregates the results from the left and right subtrees, stores and updates the aggregate result in the current node, and returns it to the parent node for aggregation.

Although Algorithm 6 is written as a recursively executed method, its nature is tail-recursive, meaning it could be rewritten in a loop form. However, writing it recursively, as presented, aids in understanding. The recursion terminates by invoking the query construction method when it reaches an imaginary layer node. To invoke this recursive method, one can use `Modify(ROOT, i, NewValue)`, where `ROOT` is the root node of the segment tree.

### 3.8 Discussion on Dynamic Query Workload Support

The proposed subspace segment tree is designed under the assumption of relatively stable query patterns, where frequently accessed nodes in the dynamic layer can be efficiently managed by the LRU-based memory pool. However, the algorithm demonstrates adaptability to dynamically changing workloads through its parameterization and memory management strategy.

When the query pattern changes significantly, the performance of the algorithm can be maintained by adjusting the hyperparameters  $D_1$  and  $D_2$ . For instance, if the new workload involves more fine-grained queries on smaller intervals, increasing the value of  $D_1$  (expanding the solid layer) can ensure that more frequently accessed nodes are permanently

resident in memory, reducing the activation overhead. Conversely, if the workload shifts towards coarser queries on larger intervals, a smaller  $D_1$  value can be used to maximize space savings, as fewer nodes in the dynamic layer will be activated. These adjustments do not change the worst-case asymptotic complexity, but they change the constant factors associated with activation, destruction, and direct aggregation in the imaginary layer.

Under unstable workloads, the activation and destruction mechanism may introduce additional overhead because nodes that have just been released can become useful again shortly afterward. This behavior is similar to cache thrashing in a small cache. The fixed memory pool bounds memory usage and prevents unbounded growth, but it does not eliminate the cost of rebuilding dynamic-layer nodes. For this reason, the proposed method is best suited for workloads with repeated range-access locality or with query patterns that change in phases rather than at every request. When the query stream is highly volatile, the system should use a larger solid layer, a larger dynamic memory pool, or a workload-monitoring policy that delays parameter changes until a stable pattern is observed.

The dynamic layer management strategy, which relies on a simple LRU policy for node destruction, is intentionally designed to be lightweight and efficient. While more sophisticated prediction mechanisms, such as those based on deep learning, could potentially preemptively activate nodes and further reduce time overhead, they are not suitable for the target embedded environment. The computational resources required for such complex models would likely outweigh the benefits gained from reduced activation costs, especially given the limited processing power and memory of embedded devices. Therefore, the current LRU-based strategy represents a practical balance between performance and resource consumption for the intended application scenario, while robustness under highly dynamic workloads remains an important direction for further empirical study.

## 4 Experiment

### 4.1 Experiment Design

This paper designs a straightforward comparative experiment to demonstrate the spatial optimization effect and additional time overhead of the subspace segment tree under different parameter choices. The plain segment tree is used as the primary experimental baseline because it has the same exact range-aggregation semantics and the same update model as the proposed method, making the measured space and time differences directly attributable to the proposed memory-

management strategy. Other candidates, including Fenwick trees, hash-based aggregation, approximate query processing, and specialized compressed indexes, have different query semantics or accuracy assumptions, so we discuss them in the motivation rather than treating them as direct one-to-one experimental baselines.

In this experiment, we did not utilize GPU computation. The CPU configuration used for the experiment is a 12th Gen Intel(R) Core(TM) i9-12900H with a base clock speed of 2.50 GHz. The memory configuration is 32.0 GB DDR5. The operating system is Windows 11 Pro 23H2, and the Python version is 3.10.4.

The experiment compares two main parameters: the space consumed by the segment tree and the time required for tree construction and queries. It is important to note that since the storage medium for the original data may differ from the memory used by the data structure, the measurement of space consumption does not include the occupancy of the original data in memory. Under the assumption of a sufficiently large memory pool, we evaluate the spatial consumption based on the number of tree nodes constructed. This measurement reflects the structural memory footprint of the index. It does not claim to replace device-level measurements such as allocator fragmentation, cache behavior, or long-term memory stability, which require deployment on embedded hardware.

## 4.2 Data Processing

For the segment tree experiments, we utilize randomly generated data. While biased and dense queries in real-world data would likely favor our dynamic generation algorithm, even purely random data still exhibits sufficient advantages to demonstrate the effectiveness of our algorithm. The generation of multiple sets of random data is consistent with no significant differences, therefore we select a representative set of data to illustrate the experimental results. On a data sequence with a length of 1,500,000, we perform 10,000 query operations. To simplify the explanation, we assume that modification operations are relatively sparse in the problem context, and these operations are not included in the overall measurements due to their small number. Complex operations may inherently lead to additional spatial overhead. We consistently set the depth threshold  $D_2$  to 19 and vary  $D_1$  to explore the impact of changes in the data structure's shape on efficiency. This setting corresponds to a shallow imaginary layer and allows the experiment to focus on how much of the upper and middle tree should be kept materialized. A smaller  $D_1$  represents an aggressive space-saving configuration, while a

larger  $D_1$  represents a more conservative configuration that spends more memory to reduce dynamic activation overhead.

To evaluate robustness under dynamic query workloads, we further construct two workload patterns on the same data scale. The first pattern is a phase-changing workload, where 9,000 queries are divided into three consecutive phases and each phase focuses on a different hot range. The second pattern is a volatile-alternating workload, where queries frequently switch among four hot ranges and periodically include random whole-domain intervals. This second pattern intentionally stresses the activation and destruction mechanism. For this robustness experiment, the memory pool is limited to 30,000 nodes, and each query result is verified against prefix-sum ground truth.

## 4.3 Experimental Results

By measuring the changes in time and space as  $D_1$  varies, we obtain Table 2.

Among them, the last row presents the time and space consumption of the plain segment tree method, serving as a benchmark for comparing the performance of our approach under varying parameter settings. To obtain the space optimization rate, we divide the number of nodes consumed by our method by the number of nodes in the plain segment tree, and then subtract the result from 1. Similarly, the time optimization rate is calculated using the same methodology. The resulting trends are plotted in the line chart as follows:

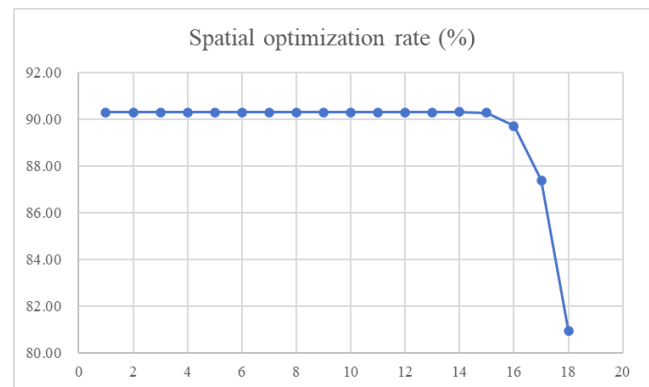
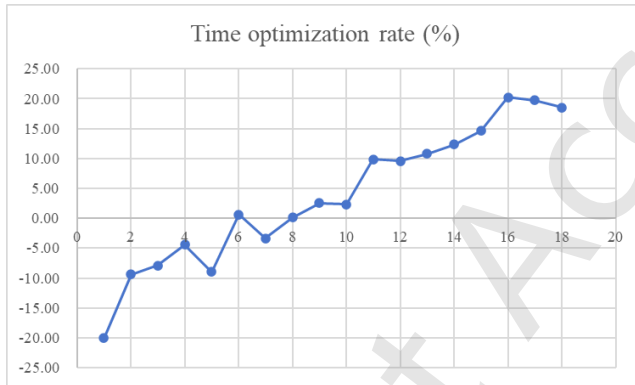


Fig. 2 Spatial optimization rate changes with  $D_1$

As shown in Figure 2, the experimental results for space optimization rate indicate that this algorithm can optimize space usage by over 80%. Even when the  $D_1$  depth is set to a relatively large value, it still achieves significant optimization effects. However, an overly small  $D_1$  value does not result in substantial space optimization, as many nodes in the dynamic layer may remain active in the tree structure for an extended period.

**Table 2** Results of spatial and temporal consumption changes with  $D_1$ 

| $D_1$              | Node Num | Spatial optimization rate (%) | Query cost time (s) | Time optimization rate (%) |
|--------------------|----------|-------------------------------|---------------------|----------------------------|
| 1                  | 146011   | 90.31                         | 3.68                | -20.05                     |
| 2                  | 146011   | 90.31                         | 3.35                | -9.36                      |
| 3                  | 146011   | 90.31                         | 3.30                | -7.89                      |
| 4                  | 146011   | 90.31                         | 3.20                | -4.45                      |
| 5                  | 146011   | 90.31                         | 3.34                | -9.00                      |
| 6                  | 146011   | 90.31                         | 3.04                | 0.61                       |
| 7                  | 146011   | 90.31                         | 3.17                | -3.38                      |
| 8                  | 146011   | 90.31                         | 3.06                | 0.14                       |
| 9                  | 146012   | 90.31                         | 2.98                | 2.56                       |
| 10                 | 146012   | 90.31                         | 2.99                | 2.30                       |
| 11                 | 146016   | 90.31                         | 2.76                | 9.83                       |
| 12                 | 146023   | 90.31                         | 2.77                | 9.61                       |
| 13                 | 146044   | 90.31                         | 2.73                | 10.76                      |
| 14                 | 145827   | 90.32                         | 2.68                | 12.33                      |
| 15                 | 146382   | 90.29                         | 2.61                | 14.62                      |
| 16                 | 154895   | 89.72                         | 2.44                | 20.22                      |
| 17                 | 190096   | 87.39                         | 2.46                | 19.71                      |
| 18                 | 287257   | 80.94                         | 2.49                | 18.53                      |
| Plain Segment Tree | 1506974  | \                             | 3.06                | \                          |

**Fig. 3** Plot of time optimization rate change with  $D_1$ 

As depicted in Figure 3, the experimental results for time optimization rate reveal that an excessively small  $D_1$  value only leads to significant additional time overhead. Therefore, it is crucial to set the hyperparameter  $D_1$  appropriately for this method. When the  $D_1$  value is larger, it not only performs better in terms of space usage but also achieves optimization in efficiency in terms of time. This is because many nodes in the original tree avoid the node construction process, saving considerable time overhead in tree construction operations. However, it is foreseeable that when the number of queries increases to the point where the cost of tree construction becomes negligible, our method may exhibit some disadvantages in terms of time overhead.

Table 3 shows that the proposed mechanism remains correct under dynamic workloads, but the cost depends strongly on workload stability. Under the phase-changing workload, no

node destruction is triggered because the active working set remains within the bounded memory pool. The average query time decreases from 0.0222 ms to 0.0162 ms as  $D_1$  increases from 8 to 14, indicating that a larger solid layer reduces dynamic activation overhead for phase-local workloads. Under the volatile-alternating workload, the memory pool reaches its effective capacity, and the LRU-based destruction mechanism is activated. The number of destruction operations ranges from 24,892 to 33,169, and the average query time is higher than in the phase-changing workload. This confirms that frequent query pattern changes introduce measurable overhead, as expected. However, increasing  $D_1$  still reduces the average query time from 0.0875 ms to 0.0523 ms, suggesting that a larger permanently resident upper layer can partially stabilize performance under unstable workloads.

#### 4.4 Factors in Real Embedded Systems

In real-world embedded systems, factors such as dynamic query patterns, high-frequency data updates, allocator behavior, cache capacity, and long-running memory stability may affect performance. Our parameterized design allows runtime adjustment of  $D_1$  and  $D_2$  to adapt to shifting workloads. For instance, increasing  $D_1$  can stabilize performance under fine-grained query bursts. Additionally, the fixed-size memory pool mitigates fragmentation risks because dynamic nodes are allocated and reclaimed from a bounded region rather than through unrestricted heap growth. This design is intended to improve long-term stability in continuous operation scenarios, although full validation still requires device-level stress

**Table 3** Robustness results under dynamic query workloads

| Workload             | $D_1$ | Avg. query time (ms) | Dynamic activations | Destructions | Max active nodes |
|----------------------|-------|----------------------|---------------------|--------------|------------------|
| Phase-changing       | 8     | 0.0222               | 4810                | 0            | 5065             |
| Phase-changing       | 12    | 0.0185               | 4755                | 0            | 8850             |
| Phase-changing       | 14    | 0.0162               | 4637                | 0            | 21020            |
| Volatile-alternating | 8     | 0.0875               | 52671               | 25928        | 27036            |
| Volatile-alternating | 12    | 0.0563               | 47795               | 24892        | 27026            |
| Volatile-alternating | 14    | 0.0523               | 43784               | 33169        | 27018            |

testing.

The proposed subspace segment tree is designed for resource-constrained embedded systems, typically featuring limited RAM (e.g., 128 MB to 1 GB), low-power processors (e.g., ARM Cortex-A series), and constrained storage (e.g., eMMC or SPI Flash). Although our experiments were conducted on a high-end desktop, the algorithm's memory footprint can be precisely controlled via hyperparameters  $D_1$  and  $D_2$ . For instance, setting a smaller  $D_1$  reduces the solid layer size, minimizing baseline memory usage, while setting a larger  $D_1$  is preferable when a deployment can spend more RAM to reduce activation overhead. Furthermore, the lightweight LRU-based memory management in the dynamic layer ensures minimal CPU overhead, making it suitable for low-power embedded processors. By significantly reducing in-memory index size, our approach also alleviates pressure on storage systems with limited write endurance.

The current experiment should therefore be interpreted as an algorithmic evaluation of the memory-management design rather than a complete embedded deployment benchmark. The dynamic workload experiment provides direct evidence of activation and destruction behavior under a bounded memory pool, while broader validation on physical embedded boards would further quantify robustness under RAM pressure, fragmentation risk, and long-term operation. These hardware-specific measurements are left as future deployment-oriented work because they require device-level instrumentation beyond the scope of the present algorithmic comparison.

## 5 Conclusion

This paper introduces an aggregated query data structure method based on hierarchical dynamic space optimization, known as the Subspace Segment Tree. This method significantly reduces the memory space occupied by the tree structure and optimizes the memory utilization rate on the basis of the traditional segment tree structure. Experiments show that this structure can substantially optimize the space occupancy of the segment tree without incurring significant disadvantages in terms of time, indicating promising

application prospects.

There is still room for improvement in the current work. When the data structure needs to be deployed on devices sensitive to I/O operations, it may be further improved to resemble a B+ tree structure. During the segmentation of the segment tree, intervals with a length of at least 2 can be split into  $K$  approximately equal subintervals, where  $K \geq 2$  is the branching factor, generating several subtree nodes to define new interval nodes. Furthermore, while our work focuses on algorithmic optimization, it is worth noting that our method is particularly well-suited for extension to distributed environments. The hierarchical nature of the subspace segment tree allows different tree segments to be naturally managed by separate nodes in a cluster, making it highly compatible with distributed deployment and enabling scalable processing of large-scale streaming data. This structural characteristic provides a significant advantage for scaling the system, as it can seamlessly accommodate distributed architectures when needed. A promising future direction is to explore smarter prediction mechanisms, such as query pattern prediction using lightweight machine learning models, to preemptively activate nodes in the dynamic layer and further reduce time overhead from frequent activation and destruction operations. Another important direction is to evaluate the method under longer-running, more diverse workloads and on real embedded hardware, so that the effects of RAM pressure, allocator fragmentation, and long-term stability can be measured directly.

## Acknowledgment

This paper was supported by the State Grid Corporation of China's technology project (Project No.:5100-202455379A-3-2-ZX).

## Reference

- [1] A. Nori, Mobile and embedded databases, in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '07. New York, NY, USA:

- Association for Computing Machinery, 2007, p. 1175–1177. [Online]. Available: <https://doi.org/10.1145/1247480.1247648>
- [2] X. Ding, H. Wang, G. Li, H. Li, Y. Li, and Y. Liu, IoT data cleaning techniques: A survey, *Intell. Converged Netw.*, vol. 3, no. 4, pp. 325–339, 2022. [Online]. Available: <https://www.sciopen.com/article/10.23919/ICN.2022.0026>
  - [3] O.-C. Marcu and P. Bouvry, Big data stream processing, University of Luxembourg, Tech. Rep., Sep. 2024. [Online]. Available: <https://hal.science/hal-04687320>
  - [4] H. Xu, A. Li, B. Wheatman, M. Marneni, and P. Pandey, BP-tree: Overcoming the point-range operation tradeoff for in-memory B-trees, *Proc. VLDB Endow.*, vol. 16, no. 11, p. 2976–2989, Jul. 2023. [Online]. Available: <https://doi.org/10.14778/3611479.3611502>
  - [5] H. Zhao, M. Dong, F. Wu, and H. Chen, Optimizing tree-structure indexes for CXL-based heterogeneous memory with SINLK, 2025. [Online]. Available: <https://arxiv.org/abs/2507.18559>
  - [6] W. Kang, S. H. Son, and J. A. Stankovic, QeDB: A quality-aware embedded real-time database, in *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009, pp. 108–117.
  - [7] —, Design, implementation, and evaluation of a QoS-aware real-time embedded database, *IEEE Trans. Comput.*, vol. 61, no. 1, pp. 45–59, 2012.
  - [8] J. Lv, S. Xu, and Y. Li, Application research of embedded database SQLite, in *2009 International Forum on Information Technology and Applications*, vol. 2, 2009, pp. 539–543.
  - [9] M. Olson, Selecting and implementing an embedded database system, *Computer*, vol. 33, no. 9, pp. 27–34, 2000.
  - [10] S. Wen, H. Zhang, and Y. Zhang, A construction and implementation of high performance embedded database, in *2011 International Conference on Electrical and Control Engineering*, 2011, pp. 4238–4241.
  - [11] S. Bou, T. Amagasa, and H. Kitagawa, Filtering XML streams by XPath and keywords, in *Proceedings of the 16th International Conference on Information Integration and Web-Based Applications & Services*, ser. iiWAS '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 410–419. [Online]. Available: <https://doi.org/10.1145/2684200.2684309>
  - [12] P. M. Grulich, J. Traub, S. Breß, A. Katsifodimos, V. Markl, and T. Rabl, Generating reproducible out-of-order data streams, in *Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 256–257. [Online]. Available: <https://doi.org/10.1145/3328905.3332511>
  - [13] A. Arasu, S. Babu, and J. Widom, The CQL continuous query language: semantic foundations and query execution, *VLDB J.*, vol. 15, no. 2, p. 121–142, Jun. 2006. [Online]. Available: <https://doi.org/10.1007/s00778-004-0147-z>
  - [14] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational geometry: algorithms and applications*, 2nd ed. Berlin, Heidelberg: Springer-Verlag, 2000.
  - [15] X. Mei, X. Sun, W. Dong, H. Wang, and X. Zhang, Segment-tree based cost aggregation for stereo matching, in *2013 IEEE Conference on Computer Vision and Pattern Recognition*, 2013, pp. 313–320.
  - [16] C. Zheng, G. Shen, S. Li, and S. Shenker, Distributed segment tree: Support of range query and cover query over DHT, in *International Workshop on Peer-to-Peer Systems*, 2006. [Online]. Available: <https://api.semanticscholar.org/CorpusID:9015133>
  - [17] Y.-K. Chang and Y.-C. Lin, Dynamic segment trees for ranges and prefixes, *IEEE Trans. Comput.*, vol. 56, no. 6, pp. 769–784, 2007.
  - [18] L. Wang and X. Wang, A simple and space efficient segment tree implementation, *MethodsX*, vol. 6, pp. 500–512, 2019. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S2215016119300391>
  - [19] S. Yadalam, K. Kanellis, M. Swift, and S. Venkataraman, ARMS: Adaptive and robust memory tiering system, 2025. [Online]. Available: <https://arxiv.org/abs/2508.04417>
  - [20] T. Wang, L. Liang, G. Yang, T. Heinis, and E. Yoneki, A new paradigm in tuning learned indexes: A reinforcement learning enhanced approach, *Proc. ACM Manag. Data*, vol. 3, no. 3, Jun. 2025. [Online]. Available: <https://doi.org/10.1145/3725257>
  - [21] Z. Li, X. Ding, and H. Wang, An effective constraint-based anomaly detection approach on multivariate time series, in *Web and Big Data*, X. Wang, R. Zhang, Y.-K. Lee, L. Sun, and Y.-S. Moon, Eds. Cham: Springer International Publishing, 2020, pp. 61–69.
  - [22] X. Ding, Y. Li, H. Wang, C. Wang, Y. Liu, and J. Wang, TSDDISCOVER: Discovering data dependency for time series data, in *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, 2024, pp. 3668–3681.
  - [23] X. Ding, Y. Song, H. Wang, D. Yang, C. Wang, and J. Wang, Clean4TSDB: A data cleaning tool for time series databases, *Proc. VLDB Endow.*, vol. 17, no. 12, pp. 4377–4380, 2024. [Online]. Available: <https://doi.org/10.14778/3685800.3685879>
  - [24] P. M. Fenwick, A new data structure for cumulative frequency tables, *Softw. Pract. Exper.*, vol. 24, no. 3, pp. 327–336, 1994. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380240306>
  - [25] Q. Liu, S. Han, Y. Qi, J. Peng, J. Li, L. Lin, and L. Chen, Why are learned indexes so effective but sometimes ineffective? *Proc. VLDB Endow.*, vol. 18, no. 9, p. 2886–2898, May 2025. [Online]. Available: <https://doi.org/10.14778/3746405.3746415>
  - [26] R. Qiu, J.-G. Jang, X. Lin, L. Liu, and H. Tong, TUCKET: A tensor time series data structure for efficient and accurate factor analysis over time ranges, *Proc. VLDB Endow.*, vol. 17,

no. 13, pp. 4746–4759, Sep. 2024. [Online]. Available: <http://dx.doi.org/10.14778/3704965.3704980>

- [27] S. Bou, H. Kitagawa, and T. Amagasa, CPiX: Real-time analytics over out-of-order data streams by incremental sliding-window aggregation, *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 11, pp. 5239–5250, 2022.
- [28] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh, Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals, *Data Min. Knowl. Discov.*, vol. 1, no. 1, pp. 29–53, 1997.

### Author biography



**Jiaxuan Su** received the master's degree in Computer Science and Technology from Harbin Institute of Technology, China, in 2019. He is currently pursuing the doctoral degree in Computer Science and Technology at Harbin Institute of Technology, China. His research interests include large-scale streaming data processing and analysis.



**Xiaou Ding** is currently an assistant professor in the School of Computer Science and Technology, Harbin Institute of Technology, China. She received her PhD from Harbin Institute of Technology in 2021. Her research covers data cleaning and integration and temporal data quality management. She has published more than 15 academic papers

in database conferences and journals, including TKDE, VLDB, ICDE, CIKM, and DASFAA, and has published three papers in leading Chinese journals. She received the Excellent Academic Paper Award from the China Association for Science and Technology in 2020.



**Haibin Qin** received the bachelor's degree from Harbin Institute of Technology, China, in 2018. He is currently pursuing the master's degree at the Massive Data Computing Lab of Harbin Institute of Technology, China. His research interests include data cleaning, data analysis, and data value assessment.



**Hongzhi Wang** received the PhD degree from Harbin Institute of Technology, China, where he studied from 2003 to 2008. He is a professor and PhD supervisor at Harbin Institute of Technology, the secretary general of ACM SIGMOD China, and a CCF distinguished member. His research interests include big data management and analysis

and data quality. He has led more than ten national or international projects, including key and general projects of the National Natural Science Foundation of China and a National Science and Technology Support Program project. He has received a first prize for technological advancement from the Ministry of Education, a Microsoft Fellowship, an IBM PhD Fellowship, and the Chinese Excellent Database Engineer Award. He has published more than 200 papers, including papers in VLDB and SIGMOD, and six books.