Continuous Query for Top-K Maximal Sum Intervals over Streaming Data

Zhongshuai Zhang
Beijing Institute of Technology
zhang_zhongshuai@126.com

Xiaochun Yang
Northeastern University
yangxc@mail.neu.edu.cn

Baihua Zheng
Singapore Management University
bhzheng@smu.edu.sg

Rui Zhu
Shenyang Aerospace University
zhurui@sau.edu.cn

Haomin Li
Northeastern University
lihm@mails.neu.edu.cn

Bin Wang
Northeastern University
binwang@mail.neu.edu.cn

ABSTRACT

The continuous identification of top-k maximal sum intervals using a sliding window over a data stream is a critical operation for applications in IoT and beyond. A maximal sum interval is a non-overlapping, contiguous subsequence with the maximal sum in a sequence of signed values. Existing algorithms are ill-suited for streaming contexts: they either exhaustively enumerate all intervals even for small k values, or depend on indexes that require frequent and costly restructuring. We propose a novel *partition*-based strategy. Our core insight is a partitioning scheme that guarantees that any maximal sum interval is fully contained within a single partition, enabling independent and parallel processing. This design provides two key advantages: it enables safe pruning of partitions that cannot contribute to top-k results, drastically narrowing the search space, and it enables efficient, incremental maintenance of the maximal sum intervals in each partition. We develop algorithms for partition construction, incremental partition updates, and partition-based top-k maximal sum interval search. Extensive experiments on real and synthetic datasets demonstrate that our approach significantly improves efficiency.

PVLDB Reference Format:
Zhongshuai Zhang, Xiaochun Yang, Baihua Zheng, Rui Zhu, Haomin Li, and Bin Wang. Continuous Query for Top-K Maximal Sum Intervals over Streaming Data. PVLDB, 19(9): 2289 - 2302, 2026.
doi:10.14778/3819518.3819551
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zhangzhongshuai/C-KMaxI-code.

1 INTRODUCTION

The Top-k Maximal Sum Intervals (KMaxI) problem [6, 26], a classic problem studied for over 25 years, aims to find the k non-overlapping, contiguous subsequences with the highest sums in a sequence of real numbers. These intervals are ranked such that the i-th interval has the maximum possible sum from the elements disjoint from all higher-ranked (1st to $(i-1)$-th) intervals. Identifying such high-impact contiguous segments is fundamental in time-series analysis, supporting applications such as burst detection, anomaly identification, and trend discovery in sequential data.

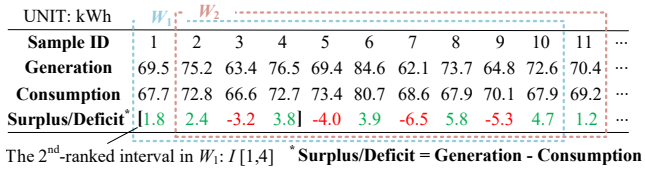

Figure 1: A running example on power generation, consumption, and storage of a microgrid.

This paper extends KMaxI to the challenging streaming context. In many modern applications, signals arrive continuously and must be analyzed in real time, making it necessary to support efficient interval discovery over sliding windows. To address this need, we introduce the Continuous KMaxI query (C-KMaxI), which provides real-time insights over a sliding window. Sliding windows can be time- or count-based [5, 9, 13–16, 20, 23, 25, 29]. In this work, we focus on the count-based model. Formally, a C-KMaxI query, denoted as $q(n, s, k)$, continuously monitors the n most recent signed numeric values in the query window W. Each time W advances by s positions, the query returns the top-k maximal sum intervals within the updated window. The techniques developed in this work can also be applied to time-based sliding windows.

Motivations and Challenges. The C-KMaxI extends interval-based analysis to streaming environments by continuously identifying high-impact contiguous periods within the sliding window. Such intervals capture sustained periods of strong cumulative signal values and are essential for detecting bursts, trends, or anomalous behaviors. In many real-world applications, identifying multiple disjoint high-impact periods is more informative than detecting a single peak interval, as systems often need to prioritize several candidate periods for downstream analysis or operational planning.

The C-KMaxI query is motivated by its importance in real-time analytics across diverse domains. For example, it can be used to identify significant genomic regions in high-throughput bio-informatics sequencing [18, 19, 26, 28], detect trending patterns in web scraping [24] and information retrieval systems [21], and extract informative temporal features for online AI models [17]. In these applications, the data naturally form signed sequences, where values may be positive or negative, representing gains and losses, increases and decreases, or surplus and deficit signals. While KMaxI handles real-valued sequences, many practical signals are signed. Supporting

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819551

signed sequences therefore enables systems to identify sustained positive trends despite temporary fluctuations.

A compelling application of C-KMaxI query arises in the Internet of Things (IoT), particularly in real-time monitoring and energy management for microgrids, which have received significant attention in recent years [11, 34]. Figure 1 illustrates a running example. Here, a C-KMaxI query continuously analyzes a *signed* microgrid signal to identify time periods with the largest *cumulative* energy surplus. Specifically, the input stream can be modeled as a surplus/deficit sequence, where each value represents the difference between power generation and consumption. Positive values indicate energy surplus, while negative values indicate energy deficit. The returned top-k non-overlapping maximal sum intervals therefore correspond to the k most significant, disjoint periods with the highest cumulative surplus. These intervals correspond to sustained periods of excess generation, enabling operators to identify when surplus energy should be stored, redistributed, or sold to the grid. While we illustrate the query using a microgrid monitoring scenario, the proposed C-KMaxI framework is broadly applicable to many streaming analytics tasks that require identifying sustained high-impact periods from continuously evolving signals.

As new measurements arrive, the query result evolves with the sliding window. Here, $I[i, j]$ denotes the contiguous interval from objects o_i to o_j and its associated sum is the total value over that interval. For example, the top intervals may change from $I[8, 8]$ with sum 5.8 and $I[1, 4]$ with sum 4.8 in window W_1, to $I[8, 11]$ with sum 6.4 and $I[6, 6]$ with sum 3.9 in window W_2. This dynamic update capability enables continuous monitoring of emerging surplus periods as the data stream evolves.

Despite its practical importance, efficiently computing C-KMaxI queries over data streams presents two formidable challenges. First, the combination of high-velocity data streams and the inherent computational complexity of the KMaxI problem makes it difficult to deliver low-latency results, especially for large window sizes. Second, the set of top-k intervals is highly volatile. As the window slides, intervals may split, merge, or be completely replaced. For instance, as shown in Figure 1, when the window advances to W_2, the result interval $I[8, 8]$ merges with adjacent intervals to form a new result interval $I[8, 11]$. This volatility renders static algorithms inefficient, as they cannot update results incrementally and must instead perform costly re-computations from scratch.

State-Of-The-Art Efforts. Existing research on the KMaxI problem is primarily designed for static data environments and falls into two categories. Scan-based approaches, such as Ruzzo-Tompa [26], identify all maximal intervals through a sequential scan but require a subsequent sorting step to select the top-k results, which becomes inefficient especially when k is much smaller than the total count. Index-based approaches, like the Tournament-Tree [6], construct auxiliary structures that allow direct retrieval of the top-k intervals without exhaustive comparisons. While effective in static scenarios, these approaches are not suitable for data streams. For example, adapting the Tournament-Tree would necessitate frequent and costly index reconstructions with every window slide, resulting in prohibitive computational overhead. Consequently, existing algorithms cannot *efficiently* support the C-KMaxI query.

Our Solution. To overcome these limitations, we propose a novel partition-based framework for efficient C-KMaxI processing. The key idea is to partition the query window into a set of disjoint partitions such that every maximal sum interval is fully contained within a single partition. This property enables two important advantages. First, since the top-k results must come from at most k partitions, many partitions can be safely pruned, drastically reducing the search space. Second, changes in maximal sum intervals are closely tied to the evolution of partitions as the window slides, which can be categorized into four distinct patterns, namely unaltered, shrinking, extended, and emergent. We develop efficient update strategies for each pattern, maximizing the reuse of previously identified maximal sum intervals.

Based on this insight, our solution proceeds in three stages. First, we construct the partitions and identify the top-1 maximal sum interval within each partition using a dynamic programming strategy. Next, we perform a partition-based k-maximal sum interval search algorithm that examines promising partitions (at most k partitions) to identify the top-k results. Finally, we design an incremental maintenance mechanism that updates partitions and their corresponding maximal intervals as the query window evolves. By carefully categorizing partition evolution patterns and reusing previously computed results whenever possible, our approach significantly reduces the computational overhead of C-KMaxI evaluation.

Table 1: Notations

Symbols	Description
o_i	the i-th real number object
$I[i, j]$, $S[i, j]$	the interval from o_i to o_j, the sum of objects in $I[i, j]$
$MaxI[i, j]$	maximal sum interval from o_i to o_j
$q(n, s, k)$	the query, modeled as a sliding window W monitoring most recent n objects, returning the top-k maximal intervals over a data stream $\mathcal{S}$, with s being object update number for each slide
$M[i]$, $M[\cdot]$	the largest sum of an interval starting at position i, $M[\cdot]$ is the collective term for M_i
l_j, r_j, LM_j	the leftmost index in partition P_j, the rightmost index in P_j, the index of the largest M in P_j
pTMS $M[u, v]$	a Potential Target M-Subsequence that has $M[u]/M[v]$ as its strict maximum/minimum M value.

2 PROBLEM DEFINITION

A *maximal sum interval* (or *maximal interval*, denoted as $MaxI[i, j]$), is a contiguous subsequence $(o_i, \cdots, o_j)$ in a sequence of real numbers $(o_1, o_2, \cdots, o_n)$ whose sum $S[i, j]$ is global maximal, i.e., $S[i, j] = \max_{1 \le i \le j \le n} S[i, j]$, with $S[i, j] > 0$. This paper considers the set of globally ranked, positive, and non-overlapping maximal intervals, following the iterative definition in [26]. Specifically, the interval with the largest sum in the entire sequence is ranked first; the i-th interval is the maximal interval computed from elements disjoint from all higher-ranked intervals. To avoid ambiguity under ties, we treat intervals containing non-empty zero-sum prefixes or suffixes as invalid. Otherwise, such intervals could create multiple overlapping intervals with identical sums when adjacent to positive-sum segments. Therefore, if a sequence has fewer than k valid maximal intervals, C-KMaxI may return fewer than k results.

Consider the sequence in query window W_1 in Figure 2. The interval $I[10, 10]$ (sum=12) is a global maximum sum interval since no other subsequence has a larger sum. Subsequences like $I[3, 3]$ and $I[11, 13]$ are invalid due to non-positive sums. While $I[1, 2]$ and $I[2, 2]$ both have sum 6, $I[1, 2]$ is invalid as it contains a zero-sum

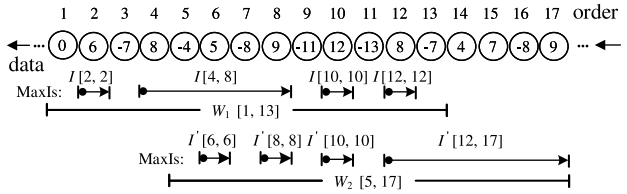

Figure 2: An example of $q(n = 13, s = 4, k = 2)$.

prefix. In addition, each lower rank interval must be disjoint from all higher-ranked ones, e.g., $I[4, 10]$ has a sum larger than $I[4, 8]$ but is inadmissible, as it overlaps with the 1st-ranked interval $I[10, 10]$.

We now formally define **C-KMaxI**, the continuous query for top-k maximal sum intervals over streaming data.

Definition 1. ***C-KMaxI.*** *Given a stream $\mathcal{S}$ of signed values (hereafter referred to as objects), a C-KMaxI query, denoted as $q(n, s, k)$, maintains a sliding window W of the most recent n objects from $\mathcal{S}$. Each time the window slides by s positions (s objects expire and s new objects arrive), the query returns the top-k maximal intervals $\{MaxI_1, MaxI_2, \ldots, MaxI_k\}$ in the updated window W.*

Figure 2 illustrates an example C-KMaxI query $q(13, 4, 2)$. In window W_1, four maximal intervals exist: $MaxI[2, 2]$, $MaxI[4, 8]$, $MaxI[10, 10]$, and $MaxI[12, 12]$. The top-2 results are $MaxI_1[10, 10]$ (sum = 12) and $MaxI_2[4, 8]$ (sum=10). After the window slides to W_2, the intervals update to $MaxI'[6, 6]$, $MaxI'[8, 8]$, $MaxI'[10, 10]$, and $MaxI'[12, 17]$. The top-2 results become $MaxI'_1[12, 17]$ (sum=13) and $MaxI'_2[10, 10]$ (sum=12).

Streams with signed values naturally arise in many real-world applications. Examples include surplus signals in energy systems (supply−demand), financial price changes, sensor deltas, and residual signals obtained after subtracting a baseline. Residual signals, in particular, convert strictly positive or negative measurements into signed streams (e.g., by subtracting the mean or expected value). In all these scenarios, a C-KMaxI query identifies the top-k intervals with the largest cumulative deviations.

3 PARTITIONING

This section formalizes our partitioning strategy for optimizing C-KMaxI queries. The core idea of this approach is a guarantee that any maximal interval lies entirely within a single partition, which enables independent processing of partitions. This property yields two key advantages: safe pruning of partitions that cannot contain result intervals, and efficient incremental maintenance via partition evolution patterns. We begin by formally defining partitions and subsequently demonstrate how these characteristics are realized.

Definition 2. ***Partition.*** *A partition $\mathcal{P} = \{P_1, P_2, \ldots, P_p\}$ of a query window $W = (o_1, o_2, \ldots, o_n)$ is a division of W into contiguous, non-overlapping segments that collectively cover W. Each partition $P_i = \{o_{l_i}, o_{l_i+1}, \ldots, o_{r_i}\} = [l_i, r_i]$ with $1 \le l_i \le r_i \le n$ has a right boundary at position r_i, which is defined as the position that maximizes the cumulative sum from the start of the partition P_i; that is, $S[l_i, r_i] \ge S[l_i, t]$ for all $t \in [l_i, n]$.*

Figure 3 shows the partitioning of $W_1 = (o_1, \cdots, o_{19})$ into five partitions, $P_1, \ldots, P_5$. Consider partition $P_1 = (o_1, o_2, o_3, o_4) = [1, 4]$.

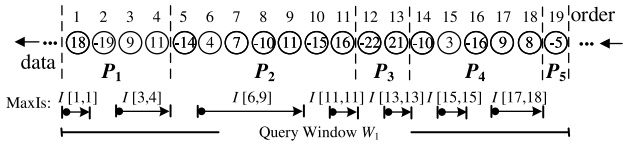

Figure 3: An example of partitions.

Its right boundary is at position 4 (i.e., $r_1 = 4$). This means that for a subsequence starting from position 1, the cumulative sum is maximized at position 4; any interval $[1, j]$ with $j \neq 4$ yields a smaller cumulative sum. This property extends to any subsequence starting from $s \in [1, 4]$, as formally established in Lemma 1. This follows directly from the functional definition of the partition's right boundary. We omit the formal proof in the interest of brevity.

Lemma 1. ***Right-bound Alignment.*** *Let partition $P_i = (o_{l_i}, o_{l_i+1}, \ldots, o_{r_i}) = P[l_i, r_i]$ be a partition of size greater than 1 of window $W = (o_1, o_2, \cdots, o_n)$, where r_i is its right boundary as defined in Definition 2. Then, for every start index $j \in [l_i, r_i]$, the cumulative sum starting from j also attains its maximum at the same right boundary r_i. Formally, $\forall j \in [l_i, r_i]$, $S[j, r_i] = \max_{t \in [j,n]} S[j, t]$.*

Beyond the right-bound alignment property, our partitioning scheme exhibits a more profound property: any maximal interval is entirely contained within a single partition. This guarantee is formally encapsulated by Theorem 1.

Theorem 1. *No maximal interval overlaps with any two adjacent partitions.*

Proof. We proceed by contradiction. Assume the theorem is false, i.e., meaning there exists a maximal interval $MaxI[a, b]$ that overlaps with two adjacent partitions $P_i = [l_i, r_i]$ and $P_{i+1} = [l_{i+1}, r_{i+1}]$ where $r_i = l_{i+1} - 1$, $l_i \le a \le r_i$, and $b \ge l_{i+1}$. By the definition of a maximal interval, we have $S[a, b] > S[a, r_i]$. On the other hand, the right-bound alignment property of a partition guarantees that $S[a, b] \le S[a, r_i]$. These two constraints introduce a contradiction, implying that our initial assumption is false, thereby proving the theorem. □

For W_1 in Figure 3, there are in total seven maximal intervals: $\{MaxI[1, 1]$, $MaxI[3, 4]$, $MaxI[6, 9]$, $MaxI[11, 11]$, $MaxI[13, 13]$, $MaxI[15, 15]$, $MaxI[17, 18]\}$. As guaranteed by Theorem 1, each maximal interval is contained entirely within a single partition (e.g. $MaxI[1, 1]$ and $MaxI[3, 4]$ lie in partition P_1).

A key benefit of our partitioning strategy is its support for efficient query processing by focusing on local maximal intervals. Since Theorem 1 guarantees that any maximal interval is entirely contained within a single partition, partitions can be processed independently. This containment enables a crucial pruning heuristic: we first identify the top-1 maximal interval within each partition. If this local champion for a given partition P_i does not qualify for the global top-k list, then no other intervals within P_i can be query results. Therefore, the partition P_i can be safely pruned.

Consider the query in Figure 3 that searches for top-3 maximal intervals within W_1. The top-1 maximal intervals with each partition are $MaxI[3, 4]$ (sum 20) in P_1, $MaxI[11, 11]$ (sum 16) in P_2, $MaxI[13, 13]$ (sum 21) in P_3, and $MaxI[17, 18]$ (sum 17) in P_4.

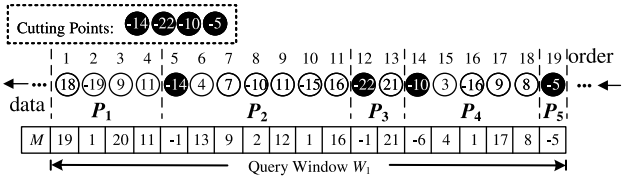

Figure 4: Constructing partitions.

P_5, only containing a non-positive object, can be skipped as it does not contain any maximal intervals. Among these candidates, $MaxI[11, 11]$ from P_2 is not among the top-3. Consequently, no other interval in P_2, such as $MaxI[6, 9]$, can be a query result, and the entire partition can be pruned.

4 THE INITIALIZATION CONSTRUCTION

In this section, we first introduce the partition construction algorithm and then propose a search algorithm to locate k-maximal intervals based on partitions.

4.1 The Partition Construction

We propose a dynamic programming-based partition construction algorithm. Moreover, while constructing the partitions, it provides direct access to the top-1 maximal interval within each partition, without requiring extra computation.

Given a query window $W = (o_1, \cdots, o_n)$, we define two arrays $M[1 \cdots n]$ and $E[1 \cdots n]$. Here, $M[i]$ stores the largest cumulative sum of any contiguous interval starting at position i, and $E[i]$ records the ending position of the corresponding interval, i.e., $S[i, E[i]] = M[i]$. Formally, $M[i] = S[i, E[i]] = \max_{1 \le i \le j \le n} \sum_{\mu=i}^{j} o_\mu$. Both arrays can be computed in reverse order using the following recurrence relation:

$$(M[i], E[i]) = \begin{cases} (o_i + M[i+1], E[i+1]), & \text{if } M[i+1] > 0, \\ (o_i, i), & \text{if } M[i+1] \le 0 \end{cases} \quad (1)$$

with the base case $M[n] = o_n$ and $E[n] = n$. Intuitively, if $M[i+1] \le 0$, the single-object interval $I[i, i]$ (i.e, just o_i) yields the maximal sum starting at i, as including any subsequent interval would only decrease the sum. Crucially, any position i where $M[i] \le 0$ is identified as a *cutting point*, as formalized in Definition 3. A non-positive $M[i]$ indicates that no interval $I[j, \cdot]$ starting at position $j < i$ can achieve a larger sum by extending to include any interval starting at position i. Consequently, these positions naturally form the boundaries of our partitions.

Definition 3. ***Cutting Points.*** *A cutting point is a position i in the query window W for which the value $M[i] \le 0$. The object o_i located at a cutting point is the physical marker for that boundary.*

A partition is determined by two consecutive cutting points, inclusive of its left cutting point and exclusive of the right one. The window boundaries (i.e., positions 1 and n of a query window of size n) serve as implicit partition boundaries if they are not themselves cutting points. As illustrated in Figure 4, the cutting points o_5, o_{12}, o_{14}, and o_{19} divide the entire sequence W_1 into five partitions.

Algorithm 1: Partition Construction

Input: The object array $o[1, \cdots, n]$
Output: M values; partitions $\mathcal{P}$; indices of the largest M in each partition $\mathcal{LM}$, and set of cutting points C.

1 Initialize array $M[1, \cdots, n]$, partition set $\mathcal{P}$, the largest M index set $\mathcal{LM}$, the cutting point index set C, the right cutting point index rc;
2 $M[n] \leftarrow o[n]$, $rc \leftarrow n+1$, $LM \leftarrow n$;
3 **for** $i \leftarrow n-1$ *to* 1 **do**
4 **if** $M[i+1] \le 0$ **then** // scan a cutting point
5 C.add $(i+1)$; $M[i] \leftarrow o[i]$;
6 **if** $P[i+1, rc-1]$ *is a non-degenerate partition* **then**
7 $\mathcal{P}$.add $(P[i+1, rc-1])$, $\mathcal{LM}$.add (LM);
8 $rc \leftarrow i+1$, $LM \leftarrow i$;
9 **else** // enter a non-degenerate partition
10 $M[i] \leftarrow o[i] + M[i+1]$;
11 **if** $M[i] > M[LM]$ **then**
12 $LM \leftarrow i$;
13 **if** $P[1, rc-1]$ *is a non-degenerate partition* **then**
14 $\mathcal{P}$.add $(P[1, rc-1])$, $\mathcal{LM}$.add (LM);
15 **return** M, $\mathcal{P}$, $\mathcal{LM}$, C;

The partition construction process is detailed in Algorithm 1. After initializing the key data structures (Lines 1-2), the algorithm performs a reverse scan of the window W to compute $M[i]$ values and identify cutting points (Lines 3-12). It maintains a pointer, rc, to track the position of previous cutting point. When a new cutting point is found at position $i+1$ (Lines 4-5), objects within the segment $[i+1, rc-1]$ are finalized as a new partition (Lines 6-7). In addition, rc is updated to $i+1$ (Line 8). Here, we want to highlight that not all partitions are retained in our algorithm; a degenerate partition containing only a single non-positive object is discarded, as it contain zero positive-sum maximal intervals. Only non-degenerate partitions are returned to serve C-KMaxI query.

The algorithm also maintains a list $\mathcal{LM}$ that stores, for each non-degenerate partition, the indices of the largest $M[i]$ value (Lines 11-12). Formally, for a non-degenerate partition $P_i = [l_i, r_i]$, $LM_i = \arg\max_{j \in [l_i, r_i]} M[j]$. Accordingly, the interval $I[LM_i, r_i]$ is the top-1 maximal interval in P_i and meanwhile also a valid maximal interval in W, as formally established in Lemma 2. The proof is provided in our technical report [36] due to space constraints. List $\mathcal{LM}$ provides direct access to the top-1 maximal interval in each partition, facilitating the processing of C-KMaxI as detailed later.

Lemma 2. *For a non-degenerate partition P_j that ends at position r_j, the interval $I[LM_j, r_j]$ is the maximal interval with the largest sum within P_j and also a maximal interval globally.*

In addition, the algorithm uses a list C to store the indices of all cutting points. These cutting points help determine how partitions evolve after the query window slides, as detailed in Section 5.1. Finally, we emphasize that the partitions generated by this process adhere strictly to the definition given in Definition 2. Specifically, as established by Lemma 3, the right boundary of each partition $P = [l_i, r_i]$ is correctly positioned at r_i, i.e., $\forall j \in [l_i, r_i]$, $E[j] = r_i$. The proof is omitted, as it can be derived directly from the definition of cutting points. This property is precisely why Algorithm 1 does not need to maintain $E[\cdot]$ array explicitly.

Lemma 3. *In a non-degenerate partition $P_i = [l_i, r_i]$ constructed by Algorithm 1, r_i is P_i's right boundary, i.e., $\forall j \in [l_i, r_i]$, $E[j] = r_i$.*

The example in Figure 4 demonstrates this process. The right boundary rc is initialized to 20, making position 19 the initial right boundary of the initial scanned partition. Since $M[19] < 0$, the segment $[19, 20)$ is identified as a degenerate partition (containing only o_{19}) and is discarded. The algorithm then processes a non-degenerate partition. Upon scanning the next cutting point at position 14, a non-degenerate partition $P_4 = [14, 19)$ is recorded. $LM_4 = 17$, as $M[17]$ is the maximum in this partition. Partitions P_3 and P_2 are identified similarly. Finally, the leftmost partition $P_1 = [1, 4]$ is verified as non-degenerate and recorded, with $LM_1 = 3$.

4.2 Partition-based k-Maximal Interval Search

In a streaming context, the query window evolves continuously. A naive method would compute all maximal intervals for each new window, which is computationally expensive. Our key observation is that the top-k results in any window can originate from at most k partitions. This follows from the fact that each maximal interval lies entirely within a single partition. Since the top-k maximal intervals must be mutually disjoint, these k intervals can belong to at most k partitions. This property enables effective pruning of the search space by restricting attention to a limited subset of promising partitions. We therefore rely on the top-1 maximal interval within each partition, identified during partition construction, to determine the k most promising partitions.

Algorithm 2 presents the "Partition-based k-Maximal Interval Search" algorithm. It maintains a size-k AVL tree that stores the sums of the current top-k maximal intervals discovered so far.

Initially, the sums of top-1 intervals of all partitions are inserted into the AVL tree (Lines 1-2). Those partitions P_j whose top-1 intervals (e.g., the interval $I[LM_j, r_j]$) remain within this tree are considered promising. The algorithm then processes these promising partitions (up to k) in descending order of their top-1 interval's sum (Lines 3-6). For each such partition P_j, an evaluation is invoked to locate the remaining valid maximal intervals within the partition that are disjoint from the top-1 interval $I[LM_j, r_j]$ (Line 4). The details of this evaluation procedure are described in the next section. The maximal intervals discovered in P_j are stored in a set CM_j. The sums of these newly discovered maximal intervals are inserted into the AVL tree (Lines 5-6), which continuously maintains the sums of current top-k maximal intervals found so far. The algorithm terminates once all partitions whose top-1 interval sums remain among the current top-k candidates have been evaluated. Finally, the union of all such CM_j forms the final candidate maximal interval set $\mathcal{CM}$, which will be used for subsequent window updates, and the maximal intervals whose sums remain in the final AVL tree form the final top-k results (Line 7).

The results returned by Algorithm 2 are guaranteed to be the top-k maximal intervals, as stated in Lemma 4. We omit the proof for brevity and provide it in Lemma 4 of our technical report [36].

Lemma 4. *Within a query window W_i, the k intervals returned by Algorithm 2 correspond to the top-k maximal intervals in the window.*

4.2.1 Identifying maximal intervals in a partition. Algorithm 2 invokes an evaluation procedure to identify all remaining maximal intervals within each promising partition P_j; specifically, those maximal intervals disjoint from the top-1 interval $I[LM_j, r_j]$. To

Algorithm 2: Partition-based k-Maximal Interval Search

Input: Partitions $\mathcal{P} = \{P_1, P_2 \cdots, P_p\}$ with their respective top-1 M values $(M[LM_1], M[LM_2], \cdots, M[LM_p])$, the candidate maximal interval set $\mathcal{CM} = \{CM_1, CM_2, \cdots, CM_p\}$, required k

Output: The top-k maximal intervals, the candidate maximal interval set.

1 Initialize an AVL tree *avl* with the size of k;
2 insert all $M[LM_j]$ for $j \in [1, p]$ into *avl*;
3 **while** *find* $M[LM_i]$ *in avl in descending order* **do**
4 incrementally update $\mathcal{CM}_i$; `// Algorithm 5`
5 **foreach** *maximal interval* $MaxI[u, v]$ *in* CM_i **do**
6 insert the sum of $MaxI[u, v]$ into *avl*;
7 **return** *maximal intervals whose sums are in avl,* $\mathcal{CM}$;

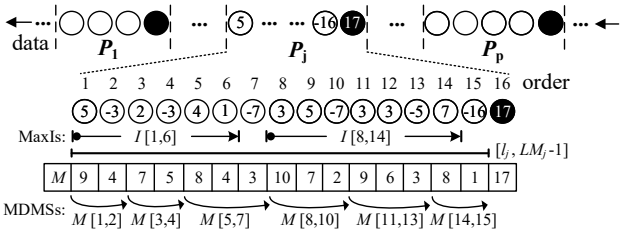

Figure 5: An example of maximal intervals in range $[l_j, LM_j - 1]$ in P_j where $l_j = 1$ and $LM_j = 16$.

support this procedure, we first derive a necessary but not sufficient condition that any maximal interval must satisfy, as formalized in Lemma 5. Again, we omit the proof due to space constraints; the full proof is provided in Lemma 5 of our technical report [36].

Lemma 5. *Let $MaxI[u, v-1]$ be a maximal interval in the sub-range $[l_j, LM_j - 1]$ of partition $P_j = [l_j, r_j]$. Then, $M[u] > M[i]$ for all $i \in (u, v]$, and $M[v] < M[i]$ for all $i \in [u, v)$. Thus, u is a strict local maximum and v is a strict local minimum of the $M[\cdot]$ values.*

Lemma 5 states that $I[u, v-1]$ can be a maximal interval only if the corresponding M-subsequence $M[u, v]$ has $M[u]$ as its strict maximum and $M[v]$ as its strict minimum. Any M-subsequence satisfying this condition is defined as a *Potential Target M-Subsequence* (pTMS for short). All other M-subsequences can therefore be safely filtered out. However, not every pTMS corresponds to a maximal interval. For instance, in Figure 5, $M[5, 7]$ is a pTMS, yet $I[5, 6]$ is not a maximal interval. Thus, while pTMS provides a filtered candidate set, additional refinement is needed.

We therefore seek a *Target M-Subsequence* (TMS for short), i.e., an $M[u, v]$ whose corresponding interval $I[u, v-1]$ is indeed a maximal interval. The key observation underlying this refinement is the mergeability property of pTMS. Two pTMS, $M[u_1, v_1]$ and $M[u_2, v_2]$ (with $v_1 < u_2$) are mergeable if their concatenation $M[u_1, v_2]$ also constitutes a pTMS. Based on this notion, we establish a key result (Lemma 6): a pTMS is a true TMS if and only if it is non-mergeable with any other pTMS in the sequence. Again, we omit the proof for brevity and provide it in Lemma 6 of our technical report [36].

Definition 4. ***Mergeability of two pTMS.*** *Let $M[u_1, v_1]$ and $M[u_2, v_2]$ be two pTMS in a partition $P_j = [l_j, r_j]$ such that $l_j \le u_1 < v_1 < u_2 < v_2 < LM_j$. They are mergeable iff the concatenated sequence $M[u_1, v_2]$ itself constitutes a valid pTMS (i.e., $M[u_1]$ is the*

Algorithm 3: Identify Maximal Intervals

Input: Array $M[l_a \dots LM_a - 1]$
Output: Maximal intervals

```
1  Initialize an empty double linked list L with a dummy head;
2  while traverse the array M from M[l_a] to M[LM_a − 1] do
3      scan a Maximal Monotonically Decreasing M SubSequence M[i, j];
4      b ← |L| − 1; L.append(M[i, j]);
5      while b ≥ 0 do // traverse L for mergeability checks
6          M[u, v] ← L[b];
7          if M[u] > M[i] ∧ M[v] > M[j] // mergeable then
8              L[b] ← M[u, j]; L[b].next ← NULL;
9              b ← b − 1; M[i, j] ← M[u, j];
10         else if M[u] ≤ M[i] ∧ M[v] > M[j] then
11             L.M[i, j].prev ← L[b].prev; b ← b − 1;
12         else // M[v] ≤ M[j]
13             L.M[i, j].prev ← L[b]; break; // stop traversing L
14 return elements of L;
```

strict maximum and $M[v_2]$ is the strict minimum over the extended range $[u_1, v_2]$). Otherwise, the two pTMS are non-mergeable.

Lemma 6. *A pTMS within $[l_j, LM_j - 1]$ is a TMS iff it is non-mergeable with any other pTMS in $[l_j, LM_j - 1]$.*

Guided by Lemma 6, our task can be reduced to identifying non-mergeable pTMS. Directly determining non-mergeability of a pTMS $M[u_1, v_1]$ during a scan of a M sequence requires examining all future M values, leading to repeated scans. To avoid this inefficiency, we introduce *Maximal Monotonically Decreasing M-SubSequence* (MDMS for short), the longest contiguous subsequences where M values directly decrease (e.g., $M[1, 2]$ and $M[3, 4]$ in Figure 5). MDMS plays a crucial role in our evaluation procedure because it has three useful properties: i) each MDMS is itself a pTMS; ii) all MDMS can be identified in a single scan of the sequence, and iii) MDMS can be iteratively merged according to Definition 4 to eventually produce the desired TMS.

Based on these observations, we present the procedure to identify maximal intervals in a given M sequence, as outlined in Algorithm 3. The algorithm scans the M sequence in each partition P_a from l_a to $LM_a - 1$, identifying MDMS on the fly (Lines 2-13). During the scan, we maintain an ordered list L that stores current pTMS candidates. Whenever a new MDMS $M[i, j]$ is identified, it is appended to the tail of L (Lines 3-4). We perform mergeability checks to examine whether it can merge with existing pTMS in L, examining them sequentially from the tail (the element in L immediately preceding $M[i, j]$) toward the head (Lines 5-13). If a merge is possible, the two subsequences are merged to form a longer pTMS; the new and longer pTMS becomes the new tail so all pTMS contained within the merged range are removed from L. We then continue checking whether the newly formed pTMS can be further merged with earlier elements in L. Once this local mergeability checks complete, the tail pTMS (originating from the identified MDMS) is guaranteed to be non-mergeable with any other pTMS in L. The algorithm then proceeds to identify the next MDMS and repeats the mergeability checks. After all MDMS are processed, the final list L contains the desired TMSs, each corresponding to a maximal interval.

Figure 6 shows this process when MDMS $M[5, 7]$ is scanned (Step ②). The list L contains $M[1, 2]$ and $M[3, 4]$. The algorithm checks its mergeability with these candidates from tail to head sequentially.

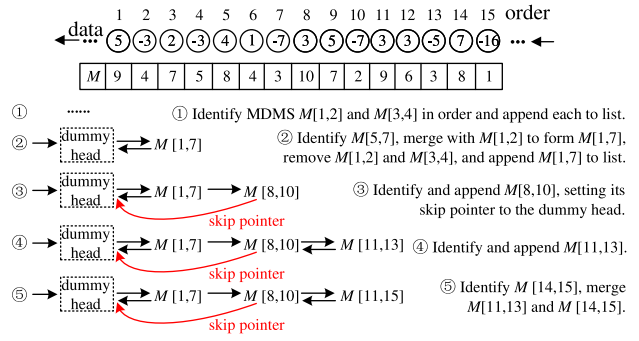

Figure 6: Build a skip-pointer when the algorithm scans $M[8, 10]$.

While $M[5, 7]$ cannot merge with $M[3, 4]$, it is mergeable with $M[1, 2]$, forming the new, longer pTMS $M[1, 7]$. This new pTMS now fully encompasses the range of $M[3, 4]$, making it obsolete. Consequently, the list is updated to $M[1, 7]$.

A naive implementation of mergeability checks that examines each new MDMS against every pTMS in L is inefficient, as it performs many unnecessary checks. We optimize it with a *skip-pointer* mechanism, based on two key insights: let $M[u, v]$ and $M[i, j]$ be two pTMSs in L with $v < i$, 1) if $M[v] \leq M[j]$, $M[i, j]$ *cannot* merge with $M[u, v]$ and any earlier pTMS in L, as the merged sequence would violate the requirement that its last element be the strict minimum M value; 2) if $M[u] \leq M[i]$, $M[u, v]$ *cannot* merge with $M[i, j]$ or any subsequent pTMSs in L, since the merged sequence would violate the requirement that the first element of a pTMS must have the strict maximum M value over the merged range.

Consequently, the mergeability of a pTMS (a new MDMS $M[i, j]$ or one formed by a previous merge) and an existing pTMS $M[u, v]$ in L is determined by evaluating the extended range $M[u, j]$. This leads to three cases based on whether $M[u]$ is the strict maximum and $M[j]$ is the strict minimum in the extended range $M[u, j]$.

- Case i): both conditions hold (Lines 7-9). This sequence is merged into a new pTMS $M[u, j]$. All pTMS contained within this new range are removed from L, and the algorithm continues checking the new pTMS against the remaining elements in L.
- Case ii): only $M[j]$ is minimal (Lines 10-11). $M[u, v]$ can not merge with $M[i, j]$ and future pTMS located after $M[i, j]$. However, $M[i, j]$ or future pTMSs $M[i', j']$ with $i' > j$ may be able to merge with pTMS $M[u', v']$ located before $M[u, v]$ in L if $M[u']$ has the strict maximum M value while $M[j]$ or $M[j']$ has the strict minimum M value in the range $[u', j]$ or $[u', j']$. Consequently, only $M[u, v]$ can be bypassed in this mergeability evaluation procedure, and `prev` pointer of $M[i, j]$ is set to that of $M[u, v]$, and the algorithm continues checking $M[i, j]$ against the preceding element in L.
- Case iii): $M[j]$ is not minimal (Lines 12-13). $M[i, j]$ cannot merge with $M[u, v]$ or any pTMS located before $M[u, v]$ in L. Consequently, we can safely terminate the mergeability checks of $M[i, j]$. In addition, `prev` pointer of $M[i, j]$ is set to $M[u, v]$.

In summary, we implement both insights in above mergeability check procedure, and we re-purpose the `prev` pointer in the ordered list L. For any pTMS $M[i, j]$ in L, its `prev` pointer does not simply

point to the immediate predecessor in L. Instead, it points to the nearest preceding pTMS $M[u, v]$ in L such that $M[v] \leq M[j]$, or to the dummy head element if no such preceding pTMS exists. This pointer effectively creates a "skip list" that allows the algorithm to bypass directly all intervening pTMS that are guaranteed to be unmergeable due to the fact that their initial M value is not the strict maximum and their last M value is not the strict minimum.

For example, as shown in Figure 6, when the scanning MDMS $M[8, 10]$ against a list containing $M[1, 7]$, its `prev` pointer links to the dummy head. This instantly tells that $M[1, 7]$ cannot merge with any subsequent pTMS, due to the existence of $M[8, 10]$. When MDMS $M[11, 13]$ is formed, we only need to check whether it is mergeable with $M[8, 10]$, bypassing the check against $M[1, 7]$.

Complexity analysis. Algorithm 1 scans the n objects in the query window to construct $|\mathcal{P}|$ partitions (where $|\mathcal{P}| < n$). This costs $O(n)$ time and $O(n)$ space. Algorithm 2 first spends $O(|\mathcal{P}| \cdot log\ k)$ time to find top-k promising partitions. It then traverses the AVL tree, which costs $O(k \cdot log\ k)$ time. During this process, Algorithm 3 is called to identify maximal intervals within each partition. Assuming partitions are distributed uniformly, each partition P_i contains $\frac{n}{|\mathcal{P}|}$ objects and $|CM_i|$ maximal intervals. Algorithm 3 is linear, so identifying them costs $O(\frac{n}{|\mathcal{P}|})$ time. They participate in comparison, requiring $O(|CM_i| \cdot log\ k)$ time. The total time complexity is $O\big(n + |p| \cdot log\ k + k \cdot log\ k + k \cdot (\frac{n}{|\mathcal{P}|} + |CM_i| \cdot log\ k)\big)$ $= O\big((1 + \frac{k}{|\mathcal{P}|}) \cdot n + (|P| + k + |CM|) \cdot log\ k\big)$.

For space complexity, Algorithm 1 maintains n M values, $|\mathcal{P}|$ partitions including their boundaries, $\mathcal{LM}$ set, and cutting points. Algorithm 2 maintains an AVL tree of size k and the candidate set $\mathcal{CM}$. Algorithm 3 processes a partition requiring space bounded by $O(\frac{n}{|\mathcal{P}|})$; when it completes a partition, the space can be released. Thus, the total space is bounded by $O(n + |\mathcal{P}| + k + |\mathcal{CM}| + \frac{n}{|\mathcal{P}|})$.

5 THE INCREMENTAL MAINTENANCE

Our partitioning strategy is designed not only to support C-KMaxI search within a single window but also to incrementally update the $\mathcal{LM}$ and $\mathcal{CM}$ sets[1], which are essential for identifying promising partitions and searching for results in the new window. In the following, we first explain how to update the partitions and then detail the maintenance of the $\mathcal{LM}$ and $\mathcal{CM}$ sets corresponding to the updated window.

5.1 Update Partitions

Sliding the window can significantly alter partitions. As shown in Figure 7, when W_1 slides to W_2, partitions evolve: P_2 shrinks, P_4 extends, and P_4' emerges. These changes result from updated cutting points, which shift from $\{o_5, o_{12}, o_{14}, o_{19}\}$ to $\{o_{12}, o_{14}, o_{22}\}$. A naive approach would track these changes by recomputing all affected M values via a full window scan. However, this is highly inefficient, as many objects' M values remain unchanged and many updated partitions do not contribute to query results. To eliminate these unnecessary computations, we propose a maintenance strategy that updates only a subset of the M values.

[1] $LM_j \in \mathcal{LM}$ and $\mathcal{CM}_i \in \mathcal{CM}$ record the position of the top-1 interval and the remaining maximal intervals, if identified, in partition P_j, respectively.

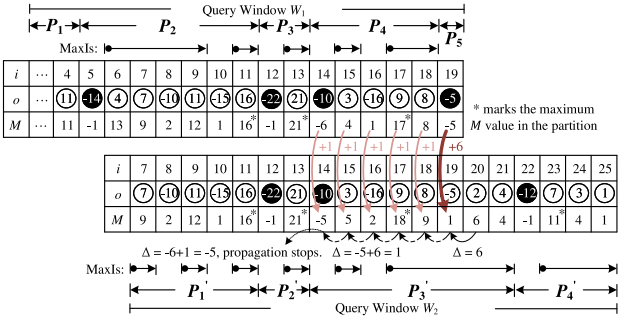

Figure 7: An example of window sliding from W_1 to W_2.

Each time the window W advances, the first s objects expire and s new objects arrive. We first calculate the M values for the new objects, in reverse order, from $M[n + s]$ to $M[n + 1]$, using Equation (1). If $\Delta = M[n + 1] \leq 0$, the update is complete, as this non-positive value blocks further propagation to preceding M values. Otherwise, we propagate the update backward through the existing partitions. This backward propagation process traverses the partitions in reverse order, starting from the last partition P_p in W_1. For each partition $P_i = [l_i, r_i]$, if $\Delta_i = M[r_i + 1] > 0$, we add Δ_i to all M values within P_i. We then proceed to partition P_{i-1}. The propagation terminates at the first partition P_j where $\Delta_j = M[r_j + 1] \leq 0$, as this cutting point prevents the cumulative impact from propagating further. Note that all partitions, including degenerate partitions (which can be easily recovered from cutting points), participate in this process.

Figure 7 illustrates this process. When W_1 advances to W_2 by $s = 6$ objects, we first calculate the M values for the new objects, i.e., $M[20, 25]$. Since $\Delta = M[20] = 6 > 0$, we propagate the update backward through the partitions: 1) Partition P_5: the M value is increased by $\Delta = 6$. 2) Partition P_4: As $\Delta_4 = M[19] = 1 > 0$, propagation continues and M values are increased by Δ_4. 3) Partition P_3: Since $\Delta_3 = M[14] = -5$, the propagation terminates, leaving M values for objects in partitions P_3, P_2, and P_1 unchanged.

We want to highlight that when propagating the update backward through the existing partitions, Δ value reduces monotonically, as stated in Lemma 7. The proof is skipped, as it can be derived by the fact that $\Delta_i = \Delta_{i+1} + M[r_i + 1]$ and $M[r_i + 1] = M[l_{i+1}] < 0$ (position l_{i+1} is a cutting point). In addition, a non-cutting point in the original window will not become a cutting point in the updated window, as the M values of objects in a partition P_i change only when the corresponding $\Delta_i > 0$. A non-cutting point has a positive M value, and this value can only increase (or remain unchanged) during backward propagation. Consequently, it cannot become a cutting point, as stated in Lemma 8.

Lemma 7. *Let P_i and P_{i+1} be consecutive partitions whose M values are updated by backward propagation with changes Δ_i and Δ_{i+1} respectively. Then, $\Delta_i < \Delta_{i+1}$.*

Lemma 8. *When the window advances from W to W', let C and C' indicate the sets of cutting points in W and W' respectively, and let $\mathcal{E}$ and $\mathcal{A}$ denote the sets of expired and newly arrived objects. A non-cutting point in W will certainly not become a cutting point in C'. Moreover, C' is a subset of $(C \setminus \mathcal{E}) \cup \mathcal{A}$.*

Algorithm 4: Partition Update

Input: Cutting points C, expired objects $\mathcal{E}$, newly arrived objects $\mathcal{A}$, M array, $\mathcal{LM}$ list
Output: Updated cutting points C', and updated $\mathcal{LM}'$ list

1 $C' \leftarrow C \setminus \mathcal{E}$;
2 $M_{new}, C_{new} \leftarrow$ DP-BuildNewCPs($\mathcal{A}$) ; // call Algorithm 1
3 $C' \leftarrow C' \cup C_{new} \setminus$ BackwardPropagation(C', M, M_{new});
4 $\mathcal{LM}' \leftarrow$ UpdateLM($C', C, \mathcal{LM}, M, M_{new}$);
5 **return** $C', \mathcal{LM}'$;

After the M values are updated, cutting points can be identified to form partitions for the new window. In our example shown in Figure 7, given cutting points o_{12}, o_{14}, and o_{22}, four partitions are formed: $P'_1 = [7, 11]$, $P'_2 = [12, 13]$, $P'_3 = [14, 21]$, and $P'_4 = [22, 25]$. We observe that partition evolution can be described using four fundamental patterns, illustrated in Figure 7: (*i*) Shrinking, where a partition loses objects from its head without incorporate new ones (e.g., P'_1); (*ii*) Unaltered, where a partition remains unaltered (e.g., P'_2); (*iii*) Extended, where a partition extends by shifting its right boundary forward (e.g., P'_3); or (*iv*) Emergent, where a new partition forms entirely from newly arrived objects (e.g., P'_4). If a shrinking partition exists, it must be the first partition in the new window. For presentation clarity, we assume that all partition changes conform to one of these four patterns. This categorization is well supported by the property that a non-cutting point cannot become a cutting point after a window slides (Lemma 8). However, in an extreme case, the first partition in the new window may both lose initial objects (due to object expiry) and extend its range, although this occurs only if backward propagation reaches the first cutting point, which is not common in practice. Importantly, our approach fully supports this corner case. It can be interpreted as a combination of a Shrinking followed by an Extension, thus does not affect the correctness of the algorithm or the returned top-k intervals.

Algorithm 4 updates partitions as follows. It removes expired cutting points (Line 1), processes new objects via Algorithm 1 to find new cutting points (Line 2), and runs `BackwardPropagation` function propagates their cumulative impact backward to remove cutting points whose updated M values become positive (Line 3). Finally, function `UpdateLM` updates LM', the index of top-1 M value in each partition P' in the new window (Line 4). Let $P' = [l', r']$ be an updated partition in the new window. We detail its LM' update operation below, dependent on the evolution pattern of P'. i) Shrinking (from partition P_i): If $l' \leq LM_i$, $LM' = LM_i$; otherwise, we scan M values of objects in P' to locate LM'. ii) Unaltered (same as partition P_i in previous window): $LM' = LM_i$. iii) Extended: assume P' covers $P_i, \cdots, P_j$ ($j \geq i$) in the original window, and possibly including new objects $\mathcal{A}' \subseteq \mathcal{A}$. Due to the nature of backward propagation, the position LM_i within any partition P_i in W continues to have the largest M value among all objects in that partition, even after their M values are updated. Consequently, LM' must lie among the positions $\{LM_i, \cdots, LM_j\}$, and, when $\mathcal{A}' \neq \emptyset$, among the newly arrived objects $\{n+1, \cdots, n+|\mathcal{A}'|\}$, where n indicates the right boundary of the previous window W. We identify LM' by scanning the M values at these positions. iv) Emergent: we scan M values of objects in P' to locate LM'.

Implementation Optimization. In the above description, we assume that the backward propagation stops only after encountering a cutting point say c_i with non-positive M value. This is mainly for ease of explanation. In practice, however, it is not necessary to update the M values for all objects located after c_i. In our implementation, we adopt a lazy update strategy. We update the M values only for new objects and for existing cutting points associated with previous window. The M values of objects within a partition are updated only when necessary (e.g., when determining LM' for a shrinking partition). A binary flag associated with each partition records whether the M values of its objects are updated. When a window slides, the flags of all the partitions are initialized to 0.

Complexity analysis. Removing expired cutting points requires $O(|\mathcal{E}|)$ time. Constructing new cutting points from the newly arrived objects costs $O(|\mathcal{A}|)$ time. Backward impact propagation only updates cutting points' M values rather than all, which requires $O(|C|)$. The total time complexity is $O(|\mathcal{E}| + |\mathcal{A}| + |C|)$. We need auxiliary space to record the cutting points and the M values of new objects, so the total space complexity is $O(|C| + |C_{new}| + |\mathcal{A}|)$.

Algorithm 5: Incremental Maximal Interval Maintenance

Input: Partition $P' = [l', r']$ in new window with LM' indicating the top-1 M value, original partitions $\mathcal{P} = \{P_1, \cdots, P_p\}$, original $\mathcal{LM}$ list, reusable maximal intervals $\mathcal{CM} = \{CM_1, \cdots, CM_p\}$
Output: Maximal intervals CM' in partition P'

1 **if** $\exists P_j \in \mathcal{P}$, Unaltered($P', P_j$) $\wedge$ MaxIReusable(P_j) **then**
2 $CM' \leftarrow CM_j$;
3 **else if** $\exists P_j \in \mathcal{P}$, Shrinking($P', P_j$) $\wedge$ MaxIReusable(P_j) **then**
4 $CM' \leftarrow CM_j \setminus \{$expired $MaxI[a, b]$ with $b < l'\}$;
5 **if** $\exists MaxI[a, b] \in CM'$ *such that* $l' \in (a, b]$ **then**
6 CM'.remove($MaxI[a, b]$);
7 CM'.add(reidentify $MaxI$ within $[l', b]$ via Algorithm 3);
8 **else if** $\exists P_i, \cdots, P_j \in \mathcal{P}$, Extended($P', P_i, \cdots, P_j$) **then**
9 **if** $\exists a \in [i, j]$, $LM_a = LM' \wedge$ MaxIReusable($P_i, \cdots, P_a$) **then**
// Case 1
10 $CM' \leftarrow \cup_{t=i}^{a} CM_t \cup MaxI[LM', r'] \setminus \{MaxI$ in $[LM', r']\}$;
11 **else if** $LM' > n \wedge$ MaxIReusable($P_i, \cdots, P_j$) **then** // Case 2
12 $CM' \leftarrow MaxI[LM', r'] \cup$ reidentify $MaxI$ within $[l', LM' - 1]$ via Algorithm 3;
13 **else** // emergent partition or partitions without reusable MaxI
14 $CM' \leftarrow$ Identify Maximal Interval via Algorithm 3;
15 **return** CM';

5.2 Maintain Maximal Intervals Incrementally

After updating the partitions and $\mathcal{LM}$, we answer C-KMaxI using the partition-based k-maximal interval search (Algorithm 2). Recall that $\mathcal{LM}$ only provides access to the top-1 maximal interval in each partition. During AVL-tree traversal, however, we must still retrieve other maximal intervals (beyond the top-1) from each promising partition whose top-1 enters the current top-k. A naive strategy computes all maximal intervals in these partitions using Algorithm 3, but this is inefficient because it ignores maximal intervals already obtained in the previous window (maintained in set $\mathcal{CM}$). Depending on how partitions evolve, many intervals in $\mathcal{CM}$ remain valid and can be reused. To avoid redundant computation, we incrementally maintain maximal intervals by exploiting partition evolution patterns and reusing prior results. Algorithm 5 summarizes this procedure: it first determines the current partition's evolution pattern and the availability of reusable maximal intervals, then applies the corresponding maintenance strategy.

Specifically, for new partitions or partitions without reusable maximal intervals (Lines 13-14), maximal intervals must be computed from scratch using Algorithm 3. When reusable maximal intervals exist (determined by function `MaxIReusable`), unaltered partitions can directly reuse all previously computed maximal intervals (Lines 1-2). For shrinking and extended partitions, the corresponding maintenance strategies are detailed below.

Shrinking Partitions. Let partition $P' = [l', r]$ be a shrinking partition derived from partition $P = [l, r]$, where $l' > l$. The maximal intervals containing only expired objects are removed. If position $l' - 1$ falls within an existing maximal interval $MaxI[a, b]$, we need to perform the following updates. First, $MaxI[a, b]$ shrinks to $I[l', b]$. It is guaranteed that pTMS $M[l', b + 1]$ corresponding to interval $I[l', b]$ will not merge with any other pTMS in this partition. This is because in the maximal interval $MaxI[a, b]$, $M[a]$ has the strict maximal M value, and therefore $M[a] > M[l']$. The fact that $MaxI[a, b]$ is a maximal interval in $P = [l, r]$ and $M[a] > M[l']$ guarantees that $M[l', b + 1]$ cannot merge with any other pTMS in this partition. Next, we invoke Algorithm 3 to identify maximal intervals, if any, in the range of $[l', b]$.

In summary, in a shrinking partition, maximal intervals containing only expired objects are removed, the maximal interval containing both expired and valid objects, if it exists, requires further evaluation, and all other maximal intervals remain unchanged. For example, in Figure 7, partition P_2 in W_1 shrinks to P_1' in W_2. Objects o_5 and o_6 in P_2 expire, affecting only maximal interval $MaxI[6, 9]$, which shrinks to interval $I[7, 9]$. We then locate two maximal intervals $I[7, 7]$ and $I[9, 9]$ in this reduced range via Algorithm 3. The other maximal interval, $MaxI[11, 11]$, remains unchanged.

Extended Partitions. Let partition $P' = [l_i, r']$ be an extended partition originating from $P_i = [l_i, r_i]$ $(r' > r_i)$, covering the sequence of partitions $P_i, \cdots, P_j$ for some $j \geq i$, and possibly including new objects $\mathcal{A}' \subseteq \mathcal{A}$. Let LM' denote the new position in P' with the largest M value. The maintenance operations depend on whether $LM' > n$, where n denotes the ending position of the previous window W, as described below.

Case 1): $LM' \leq n$. This means that the position with the top-1 M value lies in an existing partition, say P_a (i.e., $LM' = LM_a$). Accordingly, $MaxI[LM', r']$ is the top-1 maximal interval in the extended partition P', and all maximal intervals in this range become invalid and are removed. However, maximal intervals that end before position LM' remain unchanged. In other words, if CMs corresponding to $P_i, \cdots, P_a$ are available, we can immediately identify CM' by re-using all maximal intervals ending before LM_a. Otherwise, we only need to compute CMs of those partitions among $P_i, \cdots, P_a$ whose CMs are unavailable. We did not explicitly highlight this in our pseudo-code due to space constraints and for code simplicity. Consider the extended partition $P_3' = [14, 21]$ in Figure 7 as an example. P_3' (with $LM' = 17$) originates from P_4, covers partitions P_4 and P_5, and includes new objects $\mathcal{A}' = \{o_{20}, o_{21}\}$. Accordingly, $MaxI[17, 21]$ becomes the new top-1 maximal interval, and the previous maximal interval $MaxI[17, 18]$ is removed. All maximal intervals ending before position 17 (i.e., $MaxI[15, 15]$) remain valid.

Case 2): $LM' > n$. This means the position with the top-1 M value is located among new objects. $I[LM', r']$ becomes the top-1 maximal interval in the extended partition P', we need to relocate the maximal intervals in the range of $[l_i, LM' - 1]$ via Algorithm 3.

Table 2: Parameter Setting

Parameters	Value Ranges	Default
n	0.1%, 0.5%, 1%, 5%, 10% ($\times\|D\|$)	1% $\times$ $\|D\|$
s	0.01%, 0.1%, 0.5%, 1%, 10% ($\times n$)	0.1% $\times$ n
k	5, 10, 50, 100, 1000	50

Complexity analysis. Let $|\mathcal{P}|$ partitions with a uniform distribution exist in the window with n objects. In Algorithm 5, only when the worst case (case 2 in extended partitions) occurs, maximal intervals in the partition need to be re-identified. In this case, the algorithm degenerates to the scenario where Algorithm 3 processes one partition, which requires $O(\frac{n}{|\mathcal{P}|})$ time and $O(\frac{n}{|\mathcal{P}|})$ space.

6 EXPERIMENTS

6.1 Experiment Setting

Datasets. Three real datasets, Stock, Power, and Genome, and two synthetic datasets, Uniform and Normal, are used. 1) **Stock**. Bitcoin records from Alpha Vantage [1] (Nov. 4, 2013–Mar. 31, 2025), converted into a 5-minute price-delta stream (close price − open price) with 9M records; top-k maximal intervals indicate the largest cumulative price increases. 2) **Power**. UK's half-hourly power data (Nov. 2008–Dec. 2022), compiled by Kaggle from the Elexon Portal and National Grid [2]. Generation minus consumption forms a surplus stream with 0.24M records; top-k maximal intervals indicate the largest cumulative surplus. 3) **Genome**. The Haemophilus influenzae genome from NCBI [3]; following prior work [26], CpG dinucleotides score 20 and all others score −1, yielding 1.8M records. 4) **Uniform** and 5) **Normal**. Synthetic streams of 10M real numbers each, drawn from uniform and normal distributions.

Parameter Setting. We vary three key parameters: window size n, slide size s, and result size k. For each dataset, we vary one parameter at a time using the values in Table 2, while fixing the others at their defaults. Since suitable values depend on application needs, no single configuration is universally optimal. Following streaming and sliding-window evaluations [8, 23, 27, 30–33, 35, 37, 38], we choose parameter ranges that (i) cover multiple orders of magnitude, (ii) are normalized across datasets for comparability, and (iii) stress scalability under different workloads. Using percentages prevents parameter choices from being tied to a particular dataset scale and enables consistent comparison across datasets with different cardinalities. The default (n, s, k) configuration represents a representative mid-range streaming workload with substantial window overlap and a moderate number of requested disjoint intervals.

Performance Metrics. Following prior evaluations [8, 23, 27, 30–33, 35, 37, 38], we use five metrics: 1) *Overall running time*: total query and update time. 2) *Data throughput*: updates processed per second. 3) *Candidate set size*: average number of candidate maximal intervals. 4) *Memory usage*: peak memory across algorithms and datasets. 5) *Impact of number of partitions*: average running time per window under different partition counts.

Competitors. We compare our partition-based algorithm with RT [26], Tournament [6], and their incremental streaming variants, Streaming RT and Streaming Tournament; implementation details appear in Appendixes A and B of our technical report [36]. All algorithms are implemented in C++ and evaluated on a Windows 11 PC with an Intel Core i5-13500H CPU and 32GB memory.

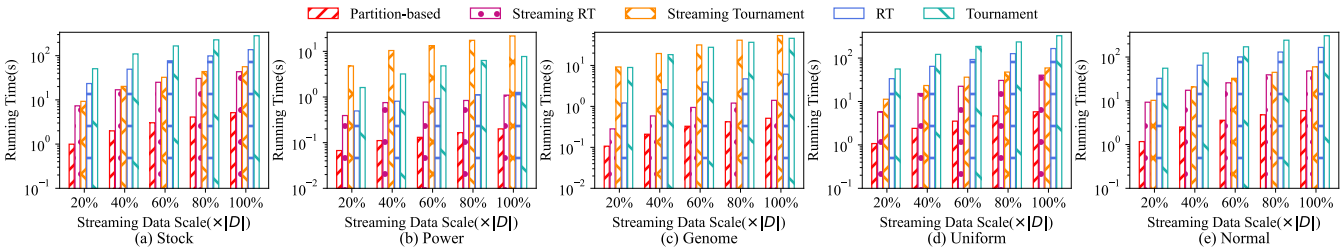

Figure 8: Running time comparison under different data scales.

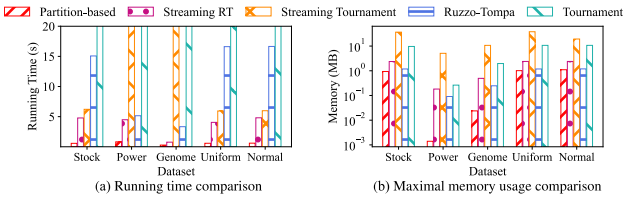

Figure 9: Performance comparison for datasets of 1M records.

6.2 Algorithm Comparisons

Running time comparison. We assess the running time of all algorithms under varying data scales, ranging from 20% to 100% of each dataset. For example, using 20% of Stock means that 20% of the original records are selected to form a new dataset containing approximately 9M ×0.2 = 1.8M records. The running time required to process C-KMaxI while simultaneously updating partitions for the entire dataset is reported in Figure 8. The results demonstrate that the partition-based approach consistently outperforms all competing algorithms across all datasets, achieving substantial performance gains. For example, on the Stock dataset, it achieves speedups of 8× and 10× over Streaming RT and Streaming Tournament, respectively. Additionally, it outperforms the original RT and Tournament by more than an order of magnitude. Similar advantages are observed across the remaining datasets, where the partition-based algorithm exhibits consistent and often significant superiority.

The superior performance of our method can be primarily attributed to the proposed partitioning strategy. Compared with competitors, our method has three advantages. First, unlike Streaming RT, it identifies partitions likely to contain query results and confines maximal interval identification and comparison to them, substantially reducing the solution space. Second, unlike Streaming Tournament, our method avoids index structures and their restructuring overhead. Third, its partition-based incremental maintenance accurately identifies affected ranges and maximizes reuse of previously computed maximal intervals. In contrast, although both Streaming RT and Streaming Tournament support incremental maintenance, Streaming RT incurs a larger affected range due to the absence of partitions, while Streaming Tournament frequently requires expensive structural adjustments. Furthermore, without streaming incremental maintenance, original RT and Tournament must re-identify all maximal intervals in each window, yielding lower efficiency.

We also observe that the performance gap varies considerably across datasets. For example, on the Stock dataset, the partition-based approach outperforms Streaming Tournament by approximately an order of magnitude on average, whereas its advantage increases to nearly two orders of magnitude on the Power and Genome datasets.

To further examine the sensitivity of the algorithms to data distribution, we analyze the running time required by each algorithm to process one million objects across multiple datasets. The results are presented in Figure 9(a). We observe that, for each algorithm, the time needed to process one million updates varies across datasets. This variation arises because maximal intervals, the targets of interest, are inherently sensitive to the underlying data distribution. As subsequences of signed values, both their length and their counts are influenced by distributional characteristics.

However, it is evident that our approach exhibits the least sensitivity to data distribution. This robustness stems from the partitioning strategy: although the length and number of maximal intervals may fluctuate significantly across datasets, our method processes only a small subset of the data, specifically up to k partitions most likely to contain query results. This effectively mitigates dataset-specific variations and reduces the impact of distributional differences. Consequently, we conclude that our approach demonstrates notably greater stability compared to all competing algorithms.

Data throughput comparison. Figure 10 presents data throughput of all algorithms under varying parameter settings across different datasets, showing that our approach consistently outperforms all competitors by a significant margin. As shown in Figures 10(a)-(e), as n increases, the throughput of both our approach and Streaming Tournament improves, while the throughput of the other algorithms remains relatively stable. Note, due to the small size of the Power dataset, its n value starts at 0.5% rather than 0.1%. A larger n indicates that the query window contains more objects and thus more maximal intervals. However, our approach continues to process only the partitions likely to contain query results, while Streaming Tournament directly searches for the top-k results based on the index. Consequently, the number of updates processed per unit time increases for both methods.

Conversely, Figures 10(k)-(o) show that as k increases, the data throughput of the partition-based algorithm experiences a slight decrease, while that of Streaming Tournament decreases sharply. Meanwhile, the throughput of other algorithms remains relatively stable. This behavior arises because the partition-based algorithm

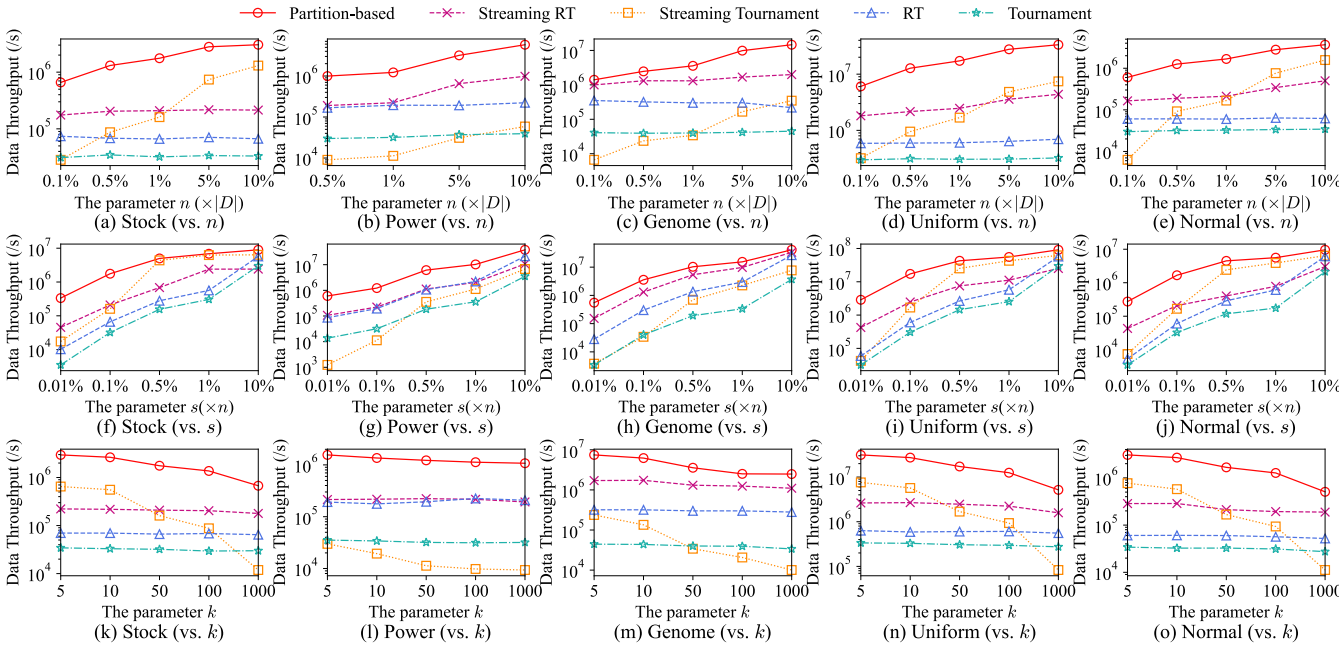

Figure 10: Data throughput comparison under different datasets.

Table 3: The average size of the candidate set (UNIT:*count*)

Solutions	Stock	Power	Genome	Uniform	Normal
Partition-based	930	16	194	1270	1252
Streaming RT	1927	17	207	2961	3032
RT	1927	17	207	2961	3032

must process more partitions as k grows, whereas Streaming Tournament performs more queries. Both RT and Streaming RT retain all maximal intervals, and Tournament incurs substantial overhead due to frequent index restructuring. As a result, their performance is less sensitive to k. Importantly, even when k is large, the partition-based algorithm still maintains a clear performance advantage.

Figures 10(f)-(j) illustrate that all algorithms exhibit increased throughput as s becomes larger. When s is small, there is considerable overlap between windows, causing many objects to be recomputed repeatedly. As s increases, the overhead from redundant computations is reduced, leading to an overall improvement.

Candidate set size. Table 3 reports the average number of maximal intervals maintained in the candidate set (i.e., $\mathcal{CM}$ in our algorithm). The two Tournament-Tree-based-approaches do not rely on a candidate set and are therefore excluded from this statistic. Both Streaming RT and RT maintain all maximal intervals, so their values in the table reflect the average number of maximal intervals per window. In contrast, our approach significantly reduces the number of intervals in the candidate set. This improvement arises because we incrementally maintain maximal intervals only within the partitions likely to contain query results. The Power dataset is the only exception: our method reduces candidate set only from 17 to 16, because this dataset inherently produces very few maximal intervals, leaving limited room for enhancement.

Table 4 presents the average per-window statistics for candidate count and usage. Count denotes the number of maximal intervals identified, while usage counts how many times these intervals are used, including comparisons for top-k selection and candidate-set maintenance. The results show that our approach achieves substantial improvements in both metrics. This advantage stems from our partitioning strategy: during maximal interval identification, top-k comparison, and incremental maintenance, the strategy avoids exhaustively processing all maximal intervals and instead focuses only on partitions likely to contain query results.

Memory usage comparison. Figure 9(b) illustrates the maximum memory usage of all the algorithms during their execution across various datasets. Among these, the partition-based approach consumes the least memory, as the partitioning strategy minimizes the number of partitions processed. Both Streaming RT and RT require comparing all maximal intervals to select the top-k, necessitating the use of more memory to identify and maintain all maximal intervals. While Streaming Tournament and Tournament do not maintain a candidate set, they still incur higher memory costs compared to the other algorithms due to the construction of the Tournament tree. Furthermore, the memory usage of the partition-based approach is notably lower than that of the other algorithms, particularly on the Power dataset. A detailed examination of this dataset reveals that most maximal intervals exist in the form of the maximal interval with the largest sum within its respective partition. These intervals can be directly obtained during partition construction, eliminating the need to invoke the maximal interval identification algorithm (Algorithm 3), which results in a substantial reduction in memory usage.

Impact of number of partitions. Our partitioning scheme is data-driven: for a fixed window, the cutting points and thus the

Table 4: Average count/usage statistics of candidates per window (UNIT:*count* for count, and *times* for usage)

Dataset	Partition-based	Streaming RT	RT
Stock	49.29 / 370.82	1244.15 / 1927.74	39401.89 / 1927.74
Power	0.51 / 0.97	92.15 /17.13	431.62 / 17.13
Genome	1.67 / 6.30	6.90 / 207.73	724.41 / 207.73
Uniform	50.74 / 569.54	1228.59 / 2961.59	49977.34 / 2961.59
Normal	51.83 / 620.57	1137.16 / 3032.76	49972.99 / 3032.76

Table 5: Average running time per sliding window under different numbers of partitions on Stock dataset.

Groups by $\|\mathcal{P}\|$	1-100	101-200	201-300	301-400	401-500	501-600
Groups' Frequency	42.95%	30.74%	17.92%	6.41%	1.68%	0.30%
Time/$\|W\|$ ($\times 10^{-5}$s)	5.45	4.83	4.34	3.89	3.35	2.91
Baselines	Streaming RT		Streaming TT		RT	TT
Time/$\|W\|$ ($\times 10^{-5}$s)	43.56		56.64		137.36	283.38

number of partitions $|\mathcal{P}|$ are uniquely determined by the distribution of signed values within the window. Consequently, $|\mathcal{P}|$ is not controlled by any algorithmic parameter, but varies naturally with the data. To evaluate how the partition count affects performance in practice, we conduct a window stratification study under the default parameters. Specifically, we process the entire stream and record for every sliding window, then group windows into ranges according to partition counts $|\mathcal{P}|$ (e.g., 1-100, 101-200, etc). For each group, we report both its frequency and the average running time per window. Table 5 reports the results on the Stock stream, together with the average running time per window required by other baselines. Note, Streaming TT and TT represent Streaming Tournament Tree and Tournament Tree, respectively. Please refer to our technical report [36] for results on other datasets.

The results reveal a clear monotonic trend: windows with larger $|\mathcal{P}|$ have lower average running time. This behavior reflects the key advantage of the partition-based framework. A larger number of partitions provides finer-grained localization and more effective pruning, allowing the top-k search to focus on fewer relevant regions and thereby reducing overall computation. As a result, the reduction in search space outweighs the relatively small overhead of maintaining additional partitions. Across all partition ranges, our method consistently outperforms these baselines by a substantial margin. Importantly, even for windows with small $|\mathcal{P}|$, the running time remains stable and highly competitive, indicating that our method does not degrade under coarse partitioning.

7 RELATED WORK

To the best of our knowledge, no existing algorithm efficiently addresses the C-KMaxI problem over streaming data. Prior studies mainly address maximal interval discovery in static sequences or consider different interval optimization objectives.

Maximal interval discovery in static sequences. Finding maximal sum intervals is a classic problem derived from the maximum subarray problem [4]. Existing solutions mainly fall into two categories. Scan-based approaches, such as the Ruzzo-Tompa (RT) algorithm [26], identify all maximal intervals in linear time. However, RT always enumerates the full set of maximal intervals, even when only the top-k results are needed, and the intervals are not ranked during enumeration, requiring an additional selection step. Index-based approaches, such as the Tournament-Tree method [6], build a tree-structured index that enables direct retrieval of the highest-ranked interval. After each interval is retrieved, its elements are marked as holes to prevent reuse in subsequent results. While effective for static data, these approaches assume a fixed input sequence and are not designed for continuous updates.

Streaming Interval Extraction. Several studies investigate extracting interval-shaped patterns from data streams under different formulations. For example, [22] proposes an online framework (oPIEC) for detecting probabilistic maximal intervals in noisy streams. This approach operates on probability and locates the threshold-based intervals that may overlap. In contrast, C-KMaxI operates on signed streams and returns globally ranked disjoint maximal intervals, making the problem fundamentally different.

Other Variants. Some work [7, 10, 12] also studies selecting k disjoint subsequences that maximize the total sum. This objective differs fundamentally from C-KMaxI. For instance, given the sequence $(10, -9, 10, -11, 1)$ and $k = 2$, these variants would select intervals $I[1, 1]$ and $I[3, 3]$ (total sum = 20), whereas C-KMaxI selects $MaxI[1, 3]$ and $MaxI[5, 5]$).

Limitations of existing approaches. Although static KMaxI algorithms could in principle be adapted to streaming settings through incremental maintenance, such adaptations remain costly. Scan-based methods must maintain and repeatedly compare all maximal intervals within the sliding window to derive the top-k results, leading to large candidate sets and high update overhead. For index-based methods require continuous structural updates to maintain correctness under window slides, including node insertions, deletions, and restoration of previously excluded elements. These operations introduce significant maintenance costs in high-velocity streams. In contrast, our method avoids both full enumeration and global indexing. By exploiting the partition containment guarantee, we prune partitions that cannot contribute to the top-k results and restrict interval identification and maintenance to a small number of promising partitions, enabling efficient and robust incremental processing over data streams.

8 CONCLUSION

This paper introduces an innovative partitioning strategy designed to support C-KMaxI over streaming data. The proposed strategy demonstrates significant efficiency in candidate identification, candidate set maintenance, and query result retrieval. Its adaptability enables effective support for C-KMaxI using a compact candidate set. Our comprehensive experiments across diverse datasets validate the superior performance and relative stability of the proposed algorithms. In the near future, we would like to study parallel algorithms for C-KMaxI over streaming data, leveraging the inherent property that partitions can be processed independently.

ACKNOWLEDGMENTS

The work was partially supported by the National Key Research and Development Program of China (No. 2024YFF0617702); the National Natural Science Foundation of China (Nos. U22A2025, 62232007, and U23A20309); the 111 Project (No. B16009); the Ant Group Research Program (No. 2025021900003).

REFERENCES

[1] 2025. *Alpha Vantage.* https://www.alphavantage.co/ Accessed: October 15, 2025.

[2] 2025. *Electrical Grid Half Hourly (UK).* https://www.kaggle.com/datasets/thedevastator/gb-electrical-grid-half-hourly-data-2008-present Accessed: October 15, 2025.

[3] 2025. *Haemophilus influenzae whole genome.* https://www.ncbi.nlm.nih.gov/Traces/wgs/JAMLFG01 Accessed: October 15, 2025.

[4] 2025. *Maximum subarray problem.* https://en.wikipedia.org/wiki/Maximum_subarray_problem Accessed: October 6, 2025.

[5] Miklós Ajtai, T. S. Jayram, Ravi Kumar, and D. Sivakumar. 2002. Approximate counting of inversions in a data stream. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, John H. Reif (Ed.). ACM, 370–379. https://doi.org/10.1145/509907.509964

[6] Sung Eun Bae and Tadao Takaoka. 2007. Algorithms for k-Disjoint Maximum Subarrays. *Int. J. Found. Comput. Sci.* 18, 2 (2007), 319–339. https://doi.org/10.1142/S012905410700470X

[7] Fredrik Bengtsson and Jingsen Chen. 2006. Computing Maximum-Scoring Segments in Almost Linear Time. In *Computing and Combinatorics, 12th Annual International Conference, COCOON 2006, Taipei, Taiwan, August 15-18, 2006, Proceedings (Lecture Notes in Computer Science)*, Danny Z. Chen and D. T. Lee (Eds.), Vol. 4112. Springer, 255–264. https://doi.org/10.1007/11809678_28

[8] Christian Böhm, Beng Chin Ooi, Claudia Plant, and Ying Yan. 2007. Efficiently Processing Continuous k-NN Queries on Data Streams. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis (Eds.). IEEE Computer Society, 156–165. https://doi.org/10.1109/ICDE.2007.367861

[9] Ben Chen, Zhijin Lv, Xiaohui Yu, and Yang Liu. 2017. Sliding Window Top-K Monitoring over Distributed Data Streams. *Data Sci. Eng.* 2, 4 (2017), 289–300. https://doi.org/10.1007/S41019-017-0053-1

[10] Miklós Csürös. 2004. Maximum-Scoring Segment Sets. *IEEE ACM Trans. Comput. Biol. Bioinform.* 1, 4 (2004), 139–150. https://doi.org/10.1109/TCBB.2004.43

[11] Eros D. Escobar, Daniel Betancur, and Idi A. Isaac. 2024. Optimal Power and Battery Storage Dispatch Architecture for Microgrids: Implementation in a Campus Microgrid. *Smart Grids and Sustainable Energy* 9, 2 (2024), 27. https://doi.org/10.1007/s40866-024-00210-8

[12] Pawel Gawrychowski and Patrick K. Nicholson. 2015. Encodings of Range Maximum-Sum Segment Queries and Applications. In *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings (Lecture Notes in Computer Science)*, Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro (Eds.), Vol. 9133. Springer, 196–206. https://doi.org/10.1007/978-3-319-19929-0_17

[13] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. 2007. A Load Shedding Framework and Optimizations for M-way Windowed Stream Joins. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis (Eds.). IEEE Computer Society, 536–545. https://doi.org/10.1109/ICDE.2007.367899

[14] Xiaohui Gu, Philip S. Yu, and Haixun Wang. 2007. Adaptive Load Diffusion for Multiway Windowed Stream Joins. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis (Eds.). IEEE Computer Society, 146–155. https://doi.org/10.1109/ICDE.2007.367860

[15] Parisa Haghani, Sebastian Michel, and Karl Aberer. 2009. Evaluating top-k queries over incomplete data streams. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, David Wai-Lok Cheung, Il-Yeol Song, Wesley W. Chu, Xiaohua Hu, and Jimmy Lin (Eds.). ACM, 877–886. https://doi.org/10.1145/1645953.1646064

[16] Cheqing Jin, Ke Yi, Lei Chen, Jeffrey Xu Yu, and Xuemin Lin. 2010. Sliding-window top-k queries on uncertain streams. *VLDB J.* 19, 3 (2010), 411–435. https://doi.org/10.1007/S00778-009-0171-0

[17] Yuan Jin, Yunliang Lai, Azadeh Noori Hoshyar, Nisreen Innab, Meshal Shutaywi, Wejdan Deebani, and A Swathi. 2025. An ideally designed deep trust network model for heart disease prediction based on seagull optimization and Ruzzo Tompa algorithm. *Scientific Reports* 15, 1 (2025), 6035.

[18] S Karlin and S F Altschul. 1993. Applications and statistics for multiple high-scoring segments in molecular sequences. *Proceedings of the National Academy of Sciences* 90, 12 (1993), 5873–5877. https://doi.org/10.1073/pnas.90.12.5873 arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.90.12.5873

[19] Samuel Karlin and Volker Brendel. 1992. Chance and Statistical Significance in Protein and DNA Sequence Analysis. *Science* 257, 5066 (1992), 39–49. https://doi.org/10.1126/science.1621093 arXiv:https://www.science.org/doi/pdf/10.1126/science.1621093

[20] Tianyi Li, Yu Gu, Xiangmin Zhou, Qian Ma, and Ge Yu. 2017. An Effective and Efficient Truth Discovery Framework over Data Streams. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß (Eds.). OpenProceedings.org, 180–191. https://doi.org/10.5441/002/EDBT.2017.17

[21] Shangsong Liang, Zhaochun Ren, Wouter Weerkamp, Edgar Meij, and Maarten de Rijke. 2014. Time-Aware Rank Aggregation for Microblog Search. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management* (Shanghai, China) *(CIKM '14)*. Association for Computing Machinery, New York, NY, USA, 989–998. https://doi.org/10.1145/2661829.2661905

[22] Periklis Mantenoglou, Alexander Artikis, and Georgios Paliouras. 2023. Online event recognition over noisy data streams. *Int. J. Approx. Reason.* 161 (2023), 108993. https://doi.org/10.1016/J.IJAR.2023.108993

[23] Kyriakos Mouratidis, Spiridon Bakiras, and Dimitris Papadias. 2006. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 635–646. https://doi.org/10.1145/1142473.1142544

[24] Jeff Pasternack and Dan Roth. 2009. Extracting article text from the web with maximum subsequence segmentation. In *Proceedings of the 18th International Conference on World Wide Web* (Madrid, Spain) *(WWW '09)*. Association for Computing Machinery, New York, NY, USA, 971–980.

[25] Víctor Rampérez, Shima Zahmatkesh, and Emanuele Della Valle. 2021. Scaling the monitoring of approximate top-k queries in streaming windows. In *2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021*, Yixin Chen, Heiko Ludwig, Yicheng Tu, Usama M. Fayyad, Xingquan Zhu, Xiaohua Hu, Suren Byna, Xiong Liu, Jianping Zhang, Shirui Pan, Vagelis Papalexakis, Jianwu Wang, Alfredo Cuzzocrea, and Carlos Ordonez (Eds.). IEEE, 181–189. https://doi.org/10.1109/BIGDATA52589.2021.9672041

[26] Walter L. Ruzzo and Martin Tompa. 1999. A Linear Time Algorithm for Finding All Maximal Scoring Subsequences. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology, August 6-10, 1999, Heidelberg, Germany*, Thomas Lengauer, Reinhard Schneider, Peer Bork, Douglas L. Brutlag, Janice I. Glasgow, Hans-Werner Mewes, and Ralf Zimmer (Eds.). AAAI, 234–241. http://www.aaai.org/Library/ISMB/1999/ismb99-027.php

[27] Avani Shastri, Di Yang, Elke A. Rundensteiner, and Matthew O. Ward. 2011. MTopS: scalable processing of continuous top-k multi-query workloads. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, Craig Macdonald, Iadh Ounis, and Ian Ruthven (Eds.). ACM, 1107–1116. https://doi.org/10.1145/2063576.2063736

[28] John L Spouge, Leonardo Mariño-Ramírez, and Sergey L Sheetlin. 2012. The ruzzo-tompa algorithm can find the maximal paths in weighted, directed graphs on a one-dimensional lattice. In *2012 IEEE 2nd International Conference on Computational Advances in Bio and medical Sciences (ICCABS)*. IEEE, 1–6.

[29] Julien Subercaze, Christophe Gravier, Syed Gillani, Abderrahmen Kammoun, and Frédérique Laforest. 2017. Upsortable: Programming TopK Queries Over Data Streams. *Proc. VLDB Endow.* 10, 12 (2017), 1873–1876. https://doi.org/10.14778/3137765.3137797

[30] Yufei Tao and Dimitris Papadias. 2006. Maintaining Sliding Window Skylines on Data Streams. *IEEE Trans. Knowl. Data Eng.* 18, 2 (2006), 377–391. https://doi.org/10.1109/TKDE.2006.48

[31] Luan V. Tran, Liyue Fan, and Cyrus Shahabi. 2016. Distance-based Outlier Detection in Data Streams. *Proc. VLDB Endow.* 9, 12 (2016), 1089–1100. https://doi.org/10.14778/2994509.2994526

[32] Luan V. Tran, Minyoung Mun, and Cyrus Shahabi. 2020. Real-Time Distance-Based Outlier Detection in Data Streams. *Proc. VLDB Endow.* 14, 2 (2020), 141–153. https://doi.org/10.14778/3425879.3425885

[33] Di Yang, Avani Shastri, Elke A. Rundensteiner, and Matthew O. Ward. 2011. An optimal strategy for monitoring top-k queries in streaming windows. In *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, Anastasia Ailamaki, Sihem Amer-Yahia, Jignesh M. Patel, Tore Risch, Pierre Senellart, and Julia Stoyanovich (Eds.). ACM, 57–68. https://doi.org/10.1145/1951365.1951375

[34] Meng Yang, Rui Xie, Yongjun Zhang, and Yue Chen. 2025. Robust Microgrid Dispatch With Real-Time Energy Sharing and Endogenous Uncertainty. *IEEE Transactions on Smart Grid* 16, 4 (2025), 3085–3098. https://doi.org/10.1109/TSG.2025.3569095

[35] Susik Yoon, Jae-Gil Lee, and Byung Suk Lee. 2019. NETS: Extremely Fast Outlier Detection from a Data Stream via Set-Based Processing. *Proc. VLDB Endow.* 12, 11 (2019), 1303–1315. https://doi.org/10.14778/3342263.3342269

[36] Zhongshuai Zhang, Xiaochun Yang, Baihua Zheng, Rui Zhu, Haomin Li, and Bin Wang. 2026. Continuous Query for Top-K Maximal Sum Intervals over Streaming Data (Long Version). https://github.com/zhangzhongshuai/C-KMaxI/blob/main/Continuous_Query_for_Top-K_Maximal_Sum_Intervals_over_Streaming_Data_Long_Version.pdf

[37] Rui Zhu, Bin Wang, Xiaochun Yang, and Baihua Zheng. 2023. Closest Pairs Search Over Data Stream. *Proc. ACM Manag. Data* 1, 3 (2023), 205:1–205:26. https://doi.org/10.1145/3617326

[38] Rui Zhu, Bin Wang, Xiaochun Yang, Baihua Zheng, and Guoren Wang. 2017. SAP: Improving Continuous Top-K Queries Over Streaming Data. *IEEE Trans. Knowl. Data Eng.* 29, 6 (2017), 1310–1328. https://doi.org/10.1109/TKDE.2017.2662236