CBiX: Incremental Sliding-Window Aggregation For Real-Time Analytics Over Out-of-Order Data Streams

Savong BOU†, Hiroyuki KITAGAWA†, and Toshiyuki AMAGASA†

† Center for Computational Sciences, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577 Japan
E-mail: savong.bou@kde.cs.tsukuba.ac.jp, {kitagawa,amagasa}@cs.tsukuba.ac.jp

Abstract Stream processing has been extensively applied in wide varieties of fields. In the meantime, stream aggregating is a popular processing technique of big data. However, it suffers from serious setbacks when the orders of events (e.g., stream elements) occurring are different from the orders of events arriving to the systems. We call such data streams as "non-FIFO steams". This phenomenon usually happens in distributed environment due to many factors, such as network disruption, delay, etc. In many analyzing scenarios, efficient processing of such non-FIFO streams is strongly needed to meet various data processing requirements. This paper proposes an efficient scalable checkpoint-based bidirectional indexing approach, called $CBiX$, for real-time analysis over non-FIFO streams. CBiX maintains the partial aggregating results in an on-demand manner per checkpoint. We demonstrate that CBiX requires less time and memory complexity than the state-of-the-art approach. We plan to do extensive experiments to compare the performance of the proposed approach and the the state-of-the-art approach once we finish the implementations.

Key words Non-FIFO streams, Aggregation, Sliding Window, Incremental Computation

1. Introduction

The rapid growth of modern technological devices and SNSs increases the flow of the unbounded transmissions of record continuously in real time. These unbounded records are called data streams. Stream processing has been receiving a lot of attentions in both academic and industrial fields. To date, many data stream management systems (DSMS) have been developed to process data in various applications such as health care, disaster prevention, telecommunication, etc.

To deal with an infinite sequence of streaming records [1–5] using finite memory (or computational resources), a window is recognized as one of the most practical ways to extract partial data from an infinite stream. Specifically, given an operation over a sequence of records, a window limits the evaluation scope. Additionally, it is common to slide a window over the stream within a range to apply the operation many times. This practice is called a sliding window.

It is important to extract abstract or higher-levels of knowledge from the raw data. One common technique is aggregation. Often aggregation is combined with sliding windows to extract useful information from streams, which is called an aggregate continuous query (ACQ) [6].

It has been extensively studied, and a lot of algorithms have been proposed. Those algorithms shows promising performance when dealing with the data streams where the order of events occurring are the same as the orders of events arriving to the systems. We call such data streams as "FIFO streams". However, in real applications, the arrival orders of data streams are not always ideal. There are many cases where the occurring and arriving orders of streams are different, which is called "non-FIFO steams". This phenomenon usually happens in distributed environment due to many factors, such as network disruption, delay, etc.

Let us consider an example where a Youtuber wants to arrange an efficient online and offline advertising strategy. Advertisers/Youtubers want to know the viewing frequency/duration or contents/ads of their videos, the viewers' demographic groups, and the charged/paid advertising fees. These aggregating results are needed in real time so that they can adjust budgets, change campaign strategies, and plan future directions as soon as possible. Since the Internet has so greatly expanded within reach worldwide, non-FIFO steams are inevitable. In such scenario, the money is involved, so the aggregation results (e.g., billing fee) should be computed as soon as possible. Such unbounded, out-of-order, and global-scale data streams have become increasingly common in daily life, such as Web logs, mobile usage statistics, sensor networks, etc.

Flink [7] and Cloud Data Flow [8] take an initiative to address this challenging problem. They propose a general stream processing framework by introducing a flexible setting option to instruct the system how much delay it should expect for the late-arriving records from streams. For this purpose, they set the timeframe, called watermark, to instruct the system to recompute the results of the past windows within the watermark timeframe if there are late-arrival tuples on them.

An example of these approaches is shown in Fig. 1. Assume that the current time is at the 22^{nd} second. After next two seconds (at the 24^{th} second), the window slides. One late-arrival record 19 and two non-late-arrival records 23 and 24 arrive to the system. Since the late-arrival record 19 falls into two past windows as shown in the figure, the results of the affected past windows are recomputed. Both Flink [7] and Cloud Dataflow [8] are the general stream processing framework. The default implementation is very naive and inefficient. Their main problem is they compute the results of the affected past windows and the current window from scratch non-incrementally every time the window slides.

Recomputing the results of the affected past windows is trivial because the updated results can be obtained by aggregating the existing results with the late-arrival records. However, computing the result of the current window is challenging. Therefore, this paper only focuses on computing the aggregating result of the current window.

To the best of authors' knowledge, there is only one algorithm that can compute the results of the current window incrementally though its performance is very poor. FlatFAT [9] uses binary tree to keep all intermediate results to achieve incremental computation over both FIFO and non-FIFO streams. If n represents the number of partitions that can be derived from the given window W and slide S, and p represents the affected partitions by the late-arrival records, the computation complexity of FlatFAT is $p * \log(n)$. The proposed approach only needs $p1 * \log(\frac{n}{k}) + 3 * p2$ time complexity ($p1 + p2 = p$), where k is the number of checkpoints and $\frac{n}{k} \in [S, n]$. The performance improvement is significant for both time and space complexities.

This paper aims at addressing the above bottleneck by introducing an efficient approach that can handle the current window computation incrementally. The proposed approach use a novel checkpoint-based bidirectional index, called CBiX, to efficiently maintain all intermediate results. Since the implementation has not finished yet, we plan to do extensive experimental evaluations to show the effectiveness of CBiX over the state-of-the-art approach when dealing with non-FIFO streams.

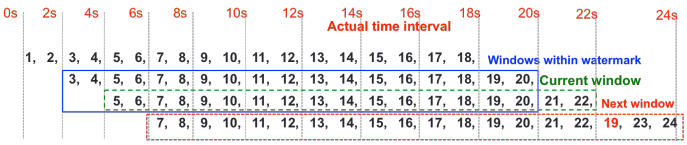


Figure 1: An example of non-FIFO stream aggregating given a query with the window size of 18 seconds, the slide size of 2 seconds, and Sum aggregation. For simplicity, each record represents both value and timestamp, and the watermark is set to 6 seconds.

2. Preliminaries

2.1. Aggregate Continuous Query (ACQ)

Aggregate Continuous Query (ACQ) consists of three main components: aggregation operation, window, and slide. Aggregation operation indicates the necessary kind of computation. Window denotes the bounded streams on which the aggregation operation is evaluated. Finally, slide instructs when to execute ACQ. The basic syntax of ACQ is shown below:

SELECT Aggregation
FROM Stream
WINDOW window **SLIDE** slide

Aggregation represents the aggregation operation, and *Stream* represents the data stream. *WINDOW* and *SLIDE* represent a window and a slide, respectively. In ACQ, the window and the slide can be expressed by either the number of records or the time interval. Non-FIFO streams do not happen when the window and slide are expressed as the number of records, so the window and slide are both expressed as time interval throughout this paper.

Note that ACQ also supports other properties such as *WHERE* and *ORDER BY* clauses. Since this work focuses on sliding-window aggregations, the other properties are ignored for simplicity.

2.2. Properties of Aggregation Operations

Aggregation operations are used to summarize the values of multiple records. Although there are a lot of aggregation operations [10], they can be categorized into three groups:

- *Distributive aggregation*: Suppose S is a sequence of values and x is a new value. Aggregation $f()$ is a distributive aggregation if $f(S.x) = f(f(S).x)$ where $S.x(f(S).x)$ is a sequence concatenating $S(f(S))$ and x. The distributive operation includes count, sum, min, max, product, sum of squares, etc.
- *Algebraic aggregation*: Algebraic aggregation is a combination of two or more different distributive aggregations. For example, the average of a sequence of values can be calculated by combining the sum and the count. Algebraic aggre-

gations include the average, standard deviation, geometric mean, range, etc.

- *Holistic aggregation*: All other types of aggregations are included into this category.

This paper focuses on distributive and algebraic aggregations. Holistic aggregation is beyond the scope of this study.

3. Related Work

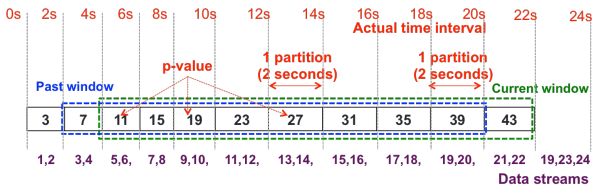


Figure 2: An example of Cutty for the ACQ over data streams in Fig. 1.

This section reviews related works of incremental sliding-window aggregations over data streams. Previous studies can be categorized into two groups: (1) partition aggregation and (2) final aggregation. For simplicity, we use Sum for example, but the discussion applies to other distributive and algebraic aggregations as well. In addition, each stream record in this paper represents the value and timestamp, and we only focus on computing the result of the current window.

3.1. Partition Aggregation

A partition aggregation divides a window into partitions based on the given window size and slide size to calculate the aggregation for the entire window by sub-aggregations for partitions. Such partitioning is important for two reasons. First, the effects of all expired records can be eliminated simultaneously without checking each individual record when the window slides. Second, the sub-aggregation for each partition, called $p - value$, can be reused to calculate the query results of ACQ. Consequently, both CPU and memory cost can be conserved. The following techniques are typically used to partition all records in a window: Panes [11], Pairs [12], and Cutty [13].

Cutty [13] is the most efficient approach for this purpose. This approach creates a new partition only when the window slides. Fig. 2 illustrates how Cutty works. Since the window size is 18 seconds and slide size is 2 seconds, Cutty partitions the window into nine partitions. Each partition keeps a sub-aggregation ($p - value$) for all records within two seconds. When the window slides, the query result is computed by accumulating $p - value$s of all partitions.

3.2. Final Aggregation

To improve the performance, final aggregation approaches were proposed on top of the partition aggregation approaches [6–9, 14]. These algorithms can be categorized into two groups: (1) FIFO algorithms [6, 9, 14], and (2) Non-FIFO algorithms [7–9].

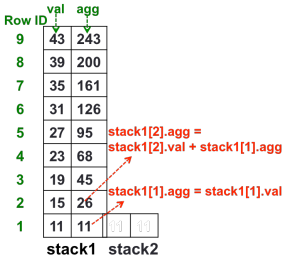
3.2.1. FIFO algorithms

There are several algorithms that can achieve $O(1)$ time complexity. They can be categorized into: (1) Worst-$O(1)$ algorithms, and (2) Average-$O(1)$ algorithms. Worst-$O(1)$ algorithm is referred to all algorithm that required at most $O(1)$ time complexity to compute the aggregation when the window slides. Average-$O(1)$ algorithm requires in average $O(1)$ time complexity, but at some slides, it requires at worst $O(n)$ time complexity where n is the total number of slides in the window.

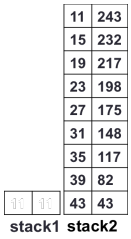
DABA [14] and FOA&IOA [14] are the worst-$O(1)$ algorithms. FlatFIT [6] and Two-Stacks [14] are the average-$O(1)$ algorithms. Two-Stacks [14] is the state-of-the-art approach. It uses two stacks to maintain the aggregating values from the previous computations to achieve $O(1)$ time complexity on average. Each stack's element consists of two value: (1) *val*: is the left aggregating value for one partition, and (2) *agg*: is the right aggregating value for one or more partitions. Maximum size of each stack is equal to the number of partitions in the window. When the window slides, one element is pushed into stack1, and top element of stack2 is popped off. The *val* of the pushed element is the aggregating value of the coming records, and the *agg* is computed by aggregating the *agg* of the top element with the *val* of the coming records. The query result is computed by aggregating the *agg* values from the top elements of both stacks. When the stack2 is empty, all elements from stack1 are popped off and pushed into stack2 one after another.

Fig. 3 is an example for this approach. Since the window size is 18 seconds, and the slide size is 2 seconds, the number of partitions is nine. Fig. 3a shows its status when maintaining all records in the current window. In stack1, *agg* values are computed by aggregating the *agg* values from the bottom to the top. Since the size of stack1 is same as the number of partitions in the window, which is nine and stack2 is empty, all elements from stack1 are popped off and pushed into stack2 by aggregating the *agg* value from the bottom to the top as shown in Fig. 3b. When the window slides, records 23 and 24 arrive (Fig. 3c). The aggregating value of the arriving records is pushed into stack1, and one element is popped off stack2. The query result is computed as shown in Fig. 3c. Same process occurs when other records arrive.

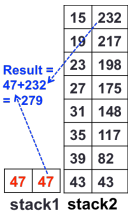
Though, these algorithms show good compromise for both speed and memory usage when dealing with FIFO streams, they cannot handle the out-of-order streams or non-FIFO streams as stated in the respected papers.



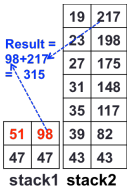
(a) Records (5, 6, ..., 21, 22) are in the window.



(b) Popped elements from Stack1 and pushed into Stack2.



(c) Records 23 and 24 arrive.



(d) Records 25 and 26 arrive.

Figure 3: An example of Two-Stacks for the ACQ over data streams in Fig. 1. In this example, we assume that there is no late-arrival record 19 coming when the window slides at the 24^{th} second.

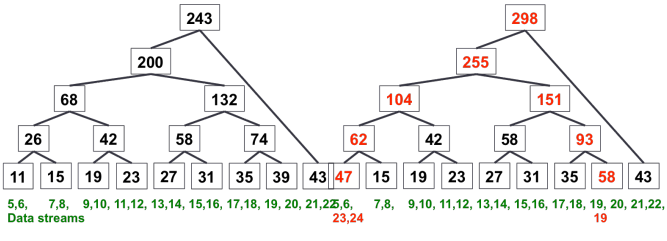
3.2. 2. Non-FIFO algorithms

Few algorithms [7–9] focus on dealing with non-FIFO streams. To deal with non-FIFO streams, Flink [7] and Cloud Dataflow [8] introduce a "watermark" concept to instruct the systems how much time the systems should expect the arrival of late-arriving records from streams. They set the timeframe, called watermark, to instruct the system that the past windows within a watermark duration need to be re-executed if late stream records fall into them.

An example of these approaches is shown in Fig. 1 as previously explained in the Section 1. Both Flink [7] and Cloud Dataflow [8] compute the results of the affected past windows and the current window from scratch non-incrementally every time the window slides.

Recomputing the results of the affected past windows is trivial. However, computing the result of the current window is challenging. Therefore, this paper only focuses on computing the aggregating result of the current window.

To the best of authors' knowledge, there is only one algorithm that can compute the results of the current window incrementally though its performance is very poor. FlatFAT [9] uses binary tree to keep all intermediate results to achieve incremental computation. An example of FlatFAT when dealing with non-FIFO streams in Fig. 1 is shown in Fig. 4. The current status of FlatFAT is shown in Fig. 4a. When the window slides (Fig. 4b), the non-late-arrival records (23, 24) replace the expired records in the first leaf node (left-most leaf node). Then, the binary is updated upward to the root node. Similarly, the late-arrival record (19) falls into the 8^{th} leaf node (counting from left), then the binary tree is also updated upward to the root node. Then, the query result is the value of the root node.



(a) FlatFAT for the current window (at the $22^{nd} second$).

(b) FlatFAT after the window slides (at the $24^{th} second$).

Figure 4: An example of FlatFAT for the ACQ over data streams in Fig. 1.

3.3. Discussion

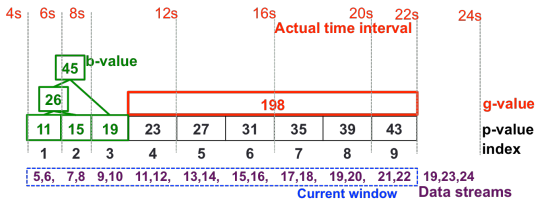
As stated in the respected papers of FIFO algorithms [6,9, 14], FIFO algorithms cannot handle non-FIFO streams because they provide inaccurate aggregating results. DABA [14], FOA&IOA [14], and FlatFIT [6] rely on man complicated pointers to maintain their many inter-related intermediate results. When the window slides, not only the intermediate results but also their many pointers need to be updated. Extending these algorithms to support non-FIFO streams is unrealistic because it is very complicated, and would result in very poor performance. Therefore, these algorithms are not included as the comparative approaches to the proposed algorithm.

The work in [14] has proved that Two-Stacks approach is not only simple to implement but also more efficient than other existing algorithms for FIFO streams. Two-Stacks can be easily extended to deal with non-FIFO streams. The extending is to recompute all affected intermediate results if there are late-arrival records. However, it eagerly maintains many inter-related intermediate results, which is very prone to be heavily affected by the late-arrival records.

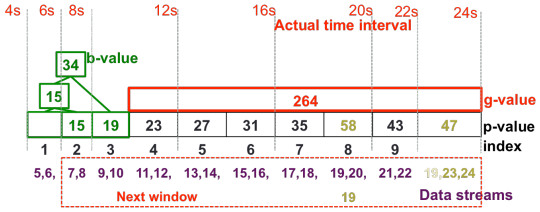
For example, in Fig. 3, when the window slides at the 24^{th} second (Fig. 3c), if one late-arrival tuple 19 together with non-late-arrival tuples 23, 24 come, all intermediate results in stack2 are affected. Recompute these affected intermediate results is very costly but necessary to guarantee the correct computation of the query result.

We assume that n represents the number of partitions that can be created from the ACQ with window size W and slide size S, and p partitions are affected by the late-arrival records per slide.

For the Two-Stacks, the numbers of elements in both



(a) Current status of CBiX.



(b) CBiX efficiently updates the affected intermediate results over non-FIFO streams when the window slides.

Figure 5: The main idea of CBiX when dealing with the ACQ over data streams in Fig. 1.

stacks is n. In the stack1, the only intermediate results that need to be recomputed are the affected *vals* and the top-most *agg*. However, in the stack2, the affected *vals* and all *aggs* intermediate results upwards need to be recomputed. Therefore, finding size of stack2 is important. Size of stack2 shrinks from n to 0, then restarts from n again. Therefore, getting the average size after one full processing of stack2 is a fair comparison. After one full processing of stack2 or after n slides, the average size of stack2 is $\frac{n+(n-1)+...+2+1}{n} = \frac{\frac{n*(n+1)}{2}}{n} = \frac{n+1}{2}$. The late-arrival records can fall on the top or bottom elements, which results in small or large number of updates respectively. It is fair to assume that the late-arrival records fall into the middle element, which requires to update $\frac{\frac{n+1}{2}}{2} = \frac{n+1}{4}$ elements. Therefore, the average number of updates to both stacks when there are p partitions affected by the late-arrival records per slide is $p * \frac{n+1}{4}$, which is quite expensive.

Similarly, the binary tree in FlatFAT has n leaf nodes. When the window slides, if there are p leaf nodes affected by the late arrival records, the cost to maintain the binary tree per slide is $(p+1)*\log(n)$, which is also very inefficient. The proposed approach needs only $(p1+1)*\log(\frac{n}{k})+3*p2$ updates to CBiX where $p1+p2=p$, which is explained in Section 4. Table 1 summarizes these computational complexities.

4. Proposed Approach

4.1. Motivation

Different from the existing approaches that eagerly aggregate the intermediate results, the proposed approach maintains the intermediate results in an on-demand manner. Fig. 5 is used as an example. The main idea is to group all records in the window into two parts:

- A small number of oldest records (Blue part in Fig. 5a): Some records in this part will expire soon, so it is necessary to eagerly aggregate all intermediate results. The number of all intermediate results is small, so it is cheap to update them when some records expire or late-arrival records fall into this part. The propose approach uses binary tree to maintain all intermediate results on this oldest part, where leaf nodes are all $p-values$, and other nodes maintain intermediate results. The root node maintains one intermediate result, called $b-value$, covering all $p-values$ for this part.
- A large number of records with longer lifespan (Red part in Fig. 5a): Only one global intermediate result, called $g-value$, covering all records in this part, is maintained. Even if there are many late-arrival records falling into this part when processing future incoming streams, only one intermediate result ($g-value$) is updated a part from updating the affected $p-values$.

Fig. 5 illustrates the above idea. Current status of CBiX at the 22^{nd} second is shown in Fig. 5a. The binary tree maintains the intermediate results from the last oldest six seconds of records. Other records with longer lifespan are aggregated into one $g-value$. At the 24^{th} second, the window slides, records 19, 23, and 24 arrive, and all records in the 1^{st} $p-value$ are expired. Then, two main updates on CBiX are executed: (1) The binary tree is updated by purging the first leaf node, and (2) Records 19, 23, and 24 are aggregated into $g-value$ (Fig. 5b). The result is computed by aggregating $b-value$ and $g-value$ ($A_{result} = b-value+g-value = 34+264 = 298$).

As can be seen, though there is a late-arrival tuple (e.g., record 19), $g-value$ is the only intermediate result to be updated, which results in very little performance impact. Note that, for efficient purging of expired records, the late-arrival record (e.g., 19) needs to be aggregated into the right $p-value$ (e.g., the 8^{th} $p-value$).

4.2. Structure of *CBiX*

Given the window and slide defined in the ACQ query, the proposed algorithm divides the window into n partitions. After partitioning, checkpoint-based circular bidirectional-index (CBiX) is created. CBiX is formally defined using Definition 41

Definition 41. *CBiX is a checkpoint-based bidirectional indexed array, which consists of:*

- *n elements: Represent all partitions. The aggregating value of each partition is called $p-value$.*
- *k checkpoints: The size of each checkpoint is $|\frac{n}{k}|$ partitions. Each checkpoint maintains one $c-value$ by aggre-*

Table 1: Complexity Analysis

Algorithm	Avg. # of update operations	Space
Two-Stacks	$p * \frac{n+1}{4}$	$2n$
FlatFAT	$(p+1) * \log(n)$	$2^{\lceil log(n) \rceil + 1}$
CBiX	$(p1+1) * \log(\lvert\frac{n}{k}\rvert) + 3 * p2$	$(n - \lvert\frac{n}{k}\rvert) + k + 1 + 2^{\left\lceil \log(\lvert\frac{n}{k}\rvert) \right\rceil + 1}$

gating all $p-values$ of all $|\frac{n}{k}|$ partitions.

- *After the current checkpoint is entirely processed or after every $|\frac{n}{k}|$ slides, the followings are maintained:*
  - *One $g-value$ that is recomputed by aggregating all k $c-values$.*
  - *A binary tree is built on the oldest checkpoint, where all $p-values$ in the checkpoint are the leaf nodes. The aggregating value of the root node is called $b-value$.*

An example of CBiX is shown in Fig. 6. Setting the optimal number of checkpoints to achieves the best performance of CBiX is explained in Section 4.4.

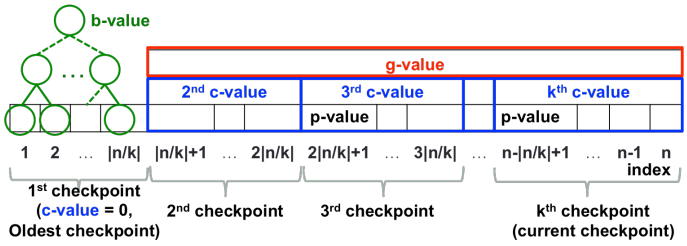


Figure 6: A CBiX that is created from a query with window W time unit and slide S time unit. Based on the slide S, window W can be divided into n partitions. Suppose that, k checkpoints with size of $|\frac{n}{k}|$ partitions can be created.

4.3. Main Algorithms

CBiX is visited and filled in by the aggregating values in a circular fashion from the 1^{st} $p-value$ to the last $p-value$ (n^{th} $p-value$) as new records arrive. When the n^{th} $p-value$ is visited, it restarts the process from the 1^{st} $p-value$.

Given a CBiX with n elements, and the aggregation operation $\oplus$. We assume that every slide affects $p1$ partitions of the oldest checkpoint and $p2$ partitions of other checkpoints, the aggregating value of the coming records of each affected partition is $pValue$, and the oldest $p-value$ to be expired is $expirepValue$. The detail is explained in Algorithm 1. When the window slides, the following processes are executed:

- Purge the expired records by removing the oldest $p-value$ ($expirepValue$) from the binary tree (Line 9).

$$b-value = updateTree(expirepValue). \quad (1)$$

- For each aggregating value (pValue) of $p1$ (Line 10), update the Tree by using Equation 2 (Lines 10–12).

$$b-value = updateTree(pValue). \quad (2)$$

- For each partition index (pIndex), checkpoint index (cIndex), and aggregating value (pValue) of $p2$ (Line 13), the following processes are executed:
  - Update each affected partition by Equation 3 (Line 14):

$$p-value[pIndex] \oplus = pValue. \quad (3)$$

  - Update each affected checkpoint by Equation 4 (Line 15):

$$c-value[cIndex] \oplus = pValue. \quad (4)$$

  - Update the $g-value$ by Equations 5 (Line 16):

$$g-value \oplus = pValue. \quad (5)$$

- If the current checkpoint is fully processed or after every $|\frac{n}{k}|$ slides (Line 18), the two following processes are executed:
  - Create a binary tree from all partitions ($p-values$) in the oldest checkpoint by Equation 6 (Lines 19–21).

$$b-value = createTree(pValues). \quad (6)$$

  - Recompute the $g-value$ by aggregating all k $c-values$ following Equation 7 (Lines 22–24).

$$g-value = c-value[1] \oplus ... \oplus c-value[k] \quad (7)$$

- The result (A_{result}) of the ACQ is computed by Equation 8 (Line 26).

$$A_{result} = b-value \oplus g-value \quad (8)$$

For simplicity, the functions $updateTree$ and $createTree$ respectively represent update and create the binary tree by returning the updated binary tree and $b-value$.

Example 41. *We explain how CBiX works for the example in Fig. 1.*

Nine partitions and three checkpoints can be derived from the given window and slide. Each checkpoint consists of three partitions. Hence, CBiX has nine $p-values$, three $c-values$, and one $g-value$.

The current status of CBiX is shown in Fig. 7a. The green part represents the binary tree, created on the latest checkpoint, the blue parts represent all checkpoints ($c-values$),

Algorithm 1 *CBiX*

Input: An ACQ with window W and slide S, aggregation operation $\oplus$, and data stream Str

Output: Query Result A_{result}

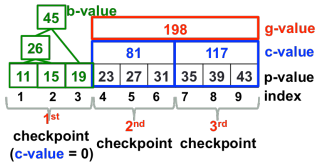
```
1: A_result = 0
2: g − value = 0
3: b − value = 0
4: n: number of partitions from W and S.
5: k: number of checkpoints from n.
6: Array < double > c − value[k]
7: Array < double > p − value[n]
8: for Window slides as new streams Str arrive do
9:    b-value = updateTree(expirepValue)
10:   for pValue in p1 do
11:      b-value = updateTree(pValue)
12:   end for
13:   for ([pIndex, cIndex] → pValue) in p2 do
14:      p − value[pIndex] ⊕= pValue
15:      c − value[cIndex] ⊕= pValue
16:      g − value ⊕= pValue
17:   end for
18:   if Current checkpoint is fully processed then
19:      for pValue in oldest checkpoint do
20:         b-value = createTree(pValue)
21:      end for
22:      for i ∈ {1, . . . , k} do
23:         g − value ⊕= c-value[i]
24:      end for
25:   end if
26:   A_result = g − value ⊕ b − value
27: end for
```

and the red part represents the of $g-value$. Note that, the 1^{st} $c-value$ is currently empty.

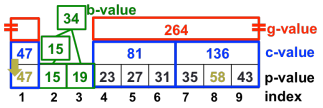
At the 24^{th} second, one late-arrival record 19, and non-late-arrival records 23 and 24 arrive. Based on the timestamp, record 19 is aggregated to the 8^{th} $p-value$, and records 23 and 24 are aggregated and replace the value of the 1^{st} $p-value$ (Fig. 7b) because the current value of the 1^{st} $p-value$ are expired. Consequently, the binary tree is updated by purging the expired $p-value$. Then, the record 19 is aggregated into the 3^{rd} $c-value$ (The 8^{th} $p-value$ is in the 3^{rd} $c-value$), and the $g-value$. Similarly, the records 23 and 24 are aggregated into the 1^{st} $c-value$ (The 1^{st} $p-value$ is in the 1^{st} $c-value$), and the $g-value$. The query result is computed by aggregating $b-value$ (34) and $g-value$ (264), which is 298.

Same procedures are done when the window slides at the 26^{th} second (Fig. 7c), and 28^{th} second (Fig. 7d).

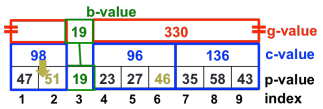
After the window slides at the 28^{th} second (Fig. 7d), the current checkpoint (1^{st} checkpoint) is entirely processed. Therefore, the new binary tree is created on the oldest checkpoint (Fig. 7e). At the same time, new $g-value$ is recomputed by aggregating all $c-values$ because the current $g-value$ will become invalid due to the expiration of the records in the 5^{th} $p-value$. Same procedures are done when other records subsequently arrive.



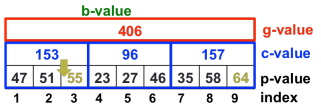
(a) Current records in the window.



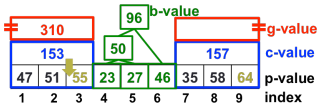
(b) At the 24^{th} second (current time), one late-arrival record 19, and non-late-arrival records 23 and 24 arrive. $A_{result} = b-value + g-value = 34 + 264 = 298$.



(c) At the 26^{th} second. Records 15, 25 and 26 arrive. $A_{result} = b-value + g-value = 19 + 330 = 349$.



(d) At the 28^{th} second. Record 21, 27 and 28 arrive. $A_{result} = b-value + g-value = 0 + 406 = 406$.



(e) The current checkpoint is entirely processed, so new binary tree on the oldest checkpoint is created, and new $g-value$ is recomputed.

Figure 7: Running example of CBiX for the ACQ over data streams in Fig. 1.

4.4. Optimal Number of Checkpoints

The performance improvement of CBiX relies on setting the right number of checkpoints. The optimal number of checkpoint is defined in Theorem 41.

Theorem 41. *The optimal number of checkpoints in CBiX is* $\sqrt{\frac{n*(p1+2)}{2*\ln(10)}}$.

Proof. The optimal number of checkpoints is obtained if the number of updates to CBiX is minimum. The number of updates to CBiX can be varied, so it is important to find the average one per checkpoint.

- After every $|\frac{n}{k}|$ slides, the following updates are executed:
  - Construct a binary tree on the oldest checkpoint,

which requires $|\frac{n}{k}| * \log(|\frac{n}{k}|)$.

– Recompute the $g-value$ by aggregating all $c-value$, which requires k.

- Purge the expired records at the oldest $c-value$ from the binary tree, which requires $\log(|\frac{n}{k}|)$.
- Every slide affects $p1$ partitions of the oldest checkpoints, so it requires $p1 * \log(|\frac{n}{k}|)$ updates to the binary tree.
- Every slides affects $p2$ partitions of other checkpoints, so it is required to update $p-values$, $c-values$, and $g-value$ $p2$ times, which requires $3 * p2$.

Therefore, the average number of updates to CBiX per slide is:

$$f(k) = \frac{|\frac{n}{k}| * \log(|\frac{n}{k}|) + k}{|\frac{n}{k}|} + \log(|\frac{n}{k}|) + p1 * \log(|\frac{n}{k}|) + 3 * p2$$

$$= (p1+2) * \log(n) - (p1+2) * \log(k) + \frac{k^2}{n} + 3p2.$$

Mathematically, the value of $f(k)$ is minimum at $k = k_0$ if

- k_0 is the solution of the derivative of $f(k)$ ($f'(k_0) = 0$), and
- The double derivative of $f(k)$ at $k = k_0$ is positive ($f''(k_0) > 0$)

We have

$$f'(k) = -\frac{p1+2}{k * \ln(10)} + \frac{2*k}{n}$$

$$f'(k) = 0, \text{ then } k_0 = \sqrt{\frac{n*(p1+2)}{2*\ln(10)}}$$

$$f''(k) = (-)*(-)*\frac{p1+2}{\log(10)} * \frac{1}{k^2} + \frac{2}{n}$$

$$= \frac{p1+2}{k^2 * \log(10)} + \frac{2}{n}$$

Replace the value of k_0 to $f''(k)$. Then, we get $f''(k_0) > 0$. The theorem is proved. □

5. Conclusion

We have proposed a novel checkpoint-based bidirectional index (CBiX) for incremental sliding-window aggregation over out-of-order data streams. The proposed algorithm efficiently compute all intermediate results in an on-demand manner. All intermediate results are categorized into two main groups: (1) $b-value$ and (2) $g-value$ with less impact from the late-arrival records.

For future work, we are going to finish the implementations. Then, the experiments will be performed.

Acknowledgment

This work has been partly supported by the **NICT BigClouT** project.

References

[1] S. Bou, T. Amagasa, and H. Kitagawa, "Scalable keyword search over relational data streams by aggressive candidate network consolidation," in *Information Systems - ELSEVIER*, ser. C, vol. 81, 2019, pp. 117–135.

[2] ——, "Filtering xml streams by xpath and keywords," in *Proceedings of the 16th International Conference on Information Integration and Web-based Applications & Services (iiWAS 2014)*, 2014, pp. 410–419.

[3] ——, "Path-based keyword search over xml streams," in *INTERNATIONAL JOURNAL OF WEB INFORMATION SYSTEMS (IJWIS)*, ser. 3, vol. 11, 2015, pp. 347–369.

[4] ——, "An improved method of keyword search over relational data streams by aggressive candidate network consolidation," in *In: Hartmann S., Ma H. (eds) Database and Expert Systems Applications. DEXA 2016. Lecture Notes in Computer Science*, vol. 9827, 2016, pp. 336–351.

[5] ——, "Scalable keyword search over relational data streams by aggressive candidate network consolidation," in *Information Systems (2018)*, 2018, https://doi.org/10.1016/j.is.2018.12.004.

[6] A. U. Shein, P. K. Chrysanthis, and A. Labrinidis, "Flatfit: Accelerated incremental sliding-window aggregation for real-time analytics," in *In Proc. International Conference on Scientific and Statistical Database Management*, ser. Article No. 5, 2017, pp. 1–12.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," in *IEEE Data Eng. Bull*, ser. 4, vol. 38, 2015, pp. 28–38, https://flink.apache.org/.

[8] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," in *Proceedings of the VLDB Endowment - Proceedings of the 41st International Conference on Very Large Data Bases*, ser. 12, vol. 8, Kohala Coast, Hawaii, pp. 1792–1803.

[9] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," in *In Proc. the VLDB Endowment*, ser. 7, vol. 8, 2015, pp. 702–713.

[10] 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," in *Data Mining and Knowledge Discovery*, ser. 1, vol. 1, 1997, pp. 29–53.

[11] D. M. Jin Li, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," in *In ACM SIGMOD Record*, ser. 1, vol. 34, 2005, pp. 39–44.

[12] S. Krishnamurthy, C. Wu, and M. J. Franklin, "On-the-fly sharing for streamed aggregation," in *In Proc. the ACM SIGMOD international conference on Management of data*, 2006, pp. 623–634.

[13] P. Carbone, J. Traub, A. Katsifodimos, and V. M. Seif Haridi, "Cutty: Aggregate sharing for user-defined windows," in *In Proc. the ACM International on Conference on Information and Knowledge Management*, 2016, pp. 1201–1210.

[14] B. Moon, I. F. V. Lopez, and V. Immanuel, "Low-latency sliding-window aggregation in worst-case constant time," in *In Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, 20017, pp. 66–77.