Educational Administration: Theory and Practice
2024, 30(7), 1254-1263
ISSN: 2148-2403
https://kuey.net/ **Research Article**

Comprehensive Analysis of Performance Metric on In-Order Data Stream Process for Sliding Window Aggregation on Business Intelligence

C. Kalyani[1*], Dr. S. Antelin Vijila[2], Dr.M.Safish Mary[3]

[1*]Research Scholar, Department of Computer Science, St.Xavier's College (Autonomous), Palayamkottai, (Affiliated to Manonmaniam Sundaranar University) Tirunelveli. India, Email: kals.sona@gmail.com
[2] Department of Computer Science and Engineering, Manonmaniam Sundaranar University, Tirunelveli, India,
Email: antelinvijila@gmail.com
[3] Department of Computer Science, St.Xavier's College (Autonomous), Palayamkottai, (Affiliated to Manonmaniam Sundaranar University) Tirunelveli, India, Email: marysafish@gmail.com

Citation: C. Kalyani et al. (2024) Comprehensive Analysis of Performance Metric on In-Order Data Stream Process for Sliding Window Aggregation on Business Intelligence, *Educational Administration: Theory and Practice*, 30(7) 1254-1263,
Doi: 10.53555/kuey.v30i7.9136

ARTICLE INFO

ABSTRACT

A data stream process is usually referred as a event stream process which uses data streaming platform to automatically integrate data from various sources, manage, organize and act upon the data on the fly as it is generated. As the world undergoes digital transformation, organizations leverage data streaming platforms to create new business opportunities. The platforms help to strengthen the competitive advantage and make the organizations' existing operation more efficient. In real-time applications, data is processed with an unbounded data stream which provides immediate insights for making any decision in analytical applications like marketing, finance, sales and more. Windowing is one of the most efficient processing methods to process data streams where unbounded stream of data or event is split into finite sets or windows based on specific criteria such as time, count of data elements or a condition. For making analytical decisions in real-time, it is a great challenge to handle data streams with efficient performance in an accurate manner. In this work, Sliding Window Aggregation (SWAG) algorithms are analyzed with synthetic datasets experimentally for a data stream process to measure the performance based on throughput and latency. Two-Stacks, Two- Stacks Lite, DABA, and DABA Lite algorithms are analyzed in this work on the synthetic dataset. Among these algorithms, DABA Lite performs well in terms of throughput and latency.

Keywords: Data stream process, Sliding window aggregation, Partial aggregate, Final aggregate, Dynamic memory allocation

1. INTRODUCTION

An introduction and need for data streaming and processing are presented in this section.

1.1 Data Streaming and Processing

Streaming means a delivery method of media content (either live or recorded) to a device using the internet in real time. Data streaming or streaming data is a continuous generation of data in various formats and sizes, by various sources like applications, sensors, server log files, website activities and more. It can also be referred as a technology that allows the users to access the data content immediately in real time rather than waiting for it to be downloaded. Thus streaming data can be used to analyze it in real or near real-time. Two common use cases of data streaming are streaming media (especially video) and real- time analytics. Real-time analytics refers to the processing and analyzing of undesirable events of fast moving data streams. It helps an organization to raise real-time actions or alerts automatically making proactive decisions instead of reactive decisions. It is analyzed on the basis of On-demand analytics or Continuous analytics. On-demand analytics provides data or computation results on the basis of demand made by the user/application and Continuous analytics process the streaming events in a continuous manner to end users, applications or a data store. Real-time analytics is an intellectual process usually done in business related fields. Business analytics and business intelligence are used interchangeably [1] whenever the terms differ in purpose and methodology. Business analytics provides predictive analysis (what will happen in the future) and Business intelligence provides

descriptive analysis (what happened and why it happened).
Stream Processing refers to the process of computing continuous data in motion (unbounded data stream) directly when it is received or produced. The unbounded stream has a beginning but no end and processing of it is referred as stream processing.
Now-a-days, as data scientists need to work with overwhelming amount of information for faster analytics, data stream access becomes an important case to fill in the space between IT and business. Because of its dynamic nature, data streams have to be scalably processed with limited time and memory. This is the motivation to work on this concept.

1.2 Need for Stream Processing

A data stream processing platform is needed to process vast amount of data with low latency and high throughput. A stream processor collects, analyzes and visualizes the continuously flowing data. In real-time, the majority of data is produced from a series of events as continuous streams like sensor events, financial trades and user activity in a website. On receiving an event from a stream, a stream processing application can react to that event, trigger an action, update an aggregate or a statistical value, and recall that event for further reference.
A stream processor is responsible for efficiently handling multiple jobs of an event, scaling and fault tolerant computation. Some of the use cases of a stream processor are fraud detection, anomalous event detection, and predictive analysis and so on. Generally, as data stream originates from heterogeneous sources a data stream processing platform must be capable to handle the following challenges:

- to deal with data loss and damaged data packets
- time critical nature of data stream demands fast enough, high performing fault tolerant system as mentioned by liang nei and others in [2]
- to handle the load dynamically based on the data stream transmission
- handle the out of order sequence of data efficiently [3]

In data stream processing, delay is an unavoidable criteria faced due to network congestions, slow processors or back pressure and would lead to loss of data, which in turn affects the throughput of the system. Thus the main objective of this work is to find an optimized technique providing a higher throughput and lower latency for a data stream process.
The contribution of this work concentrates in measuring the performance metric of a data stream process with various SWAGs in terms of throughput and latency which are the essential parameters that are responsible for an efficient data processing system.

2. Literature Review on SWAGs

Research community has contributed many innovative implementations on SWAGs. Before discussing the related works on SWAGs, a brief note on the popular optimization techniques for handling a data stream process named windowing and aggregation are given in this section.
In a data stream process, memory management acts as a basic metric factor to promote the performance. When a data stream process is handled with SWAG, the method of computation of aggregation is to be analyzed for a better performance.

2.1 Windowing for an Efficient Data Stream Process

A stream processor uses a vital concept called windowing to handle the unbounded incoming data. Windowing is an approach used to break the data streams into finite streams to apply some transformation in them. Different types of windowing are available and can be chosen based on the computation requirement. Different types of windowing strategies are tumbling, sliding, session and global windows.
In this work, a sliding window strategy is used for analysis and message count is taken as a windowing criteria. A Sliding window is defined with a window size and a slide size where computation of data is done during the slide size interval. It contains overlapping data that belong to more than one window.
Thus the different window types operate on some predefined mechanism to fire computation of data when a condition is met or a trigger is fired.

2.2 Need for Aggregation in Sliding Window Computation

Sliding window computation leads to a high degree of redundant computation due to a large number of common data elements as the window slides. Several optimization techniques have been developed by researchers for processing replicated data for the enhancement of effective storage and fast computation. Sliding window aggregation is one such optimization technique where data de-duplication is done by an aggregation function to produce aggregate values from a collection of data elements. This technique involves incremental computation as the window contents change over time.
Companies rely on software tools for data aggregation. The aggregated data is analyzed to create actionable business intelligence and guide for a decision-making process to improve the business as developed by Jun Wu

and Luo Zhong [4]. Hence data aggregation is an essential process in this fast world in almost all the fields like investment and finance, travel [5], banking, healthcare and education.
In the following section, SWAGs implemented by the research community are discussed.

2.3 Related works on SWAGs

Arasu and Jennifer designed a technique called B-Int (Base Interval) [6] with a shared window holding all the windows for computation of aggregation. It uses multi- level data structure containing dyadic (base) intervals of varying length capable of handling FIFO windows. Base intervals taken for computation, depends on the position of data item and not the position of the window. This algorithm uses simple array treated as a circular buffer and does not support dynamic memory management. Jini Li et al. proposed a technique named panes [7] for computing sliding window aggregate queries using disjoint panes where the pane aggregates are finally rolled up to get window aggregates within a single query. This technique does not execute multiple sliding window queries. Sailesh Krishnamurthy et al. designed an approach named pairs [8] for a streaming system that splits the window slides into two equal fragments where the length of the slide is computed by finding the least common multiple of the slides. It proves paired (shared) windows are superior to paned (unshared) windows. Thanaa et al. adopted two approaches for incremental evaluations namely input triggered and negative tuple approach [9]. In input triggered approach, expiration of tuples is based on the timestamp of the newly inserted tuple. In negative tuple approach, the unpredictable delay is handled by maintaining the tuple flow on any insertion or eviction. Anatoli at el designed a technique named Flatfit[10] that works with the reuse of intermediate window aggregates with heavy workloads with an index data structure. It is the first sliding window processing technique with time complexity of O (1). In a data stream process, on meeting a high degree of overlap across windows, Capri Balkesen and Nesime Tatbul designed a pane based partitioning strategy [11] with two alternative partitioning strategies based on batching and pane based processing. It promotes the scale -up behavior. Pramod Bhatotia et al. designed a sliding window computation framework called Slider [12] with self-adjusting contraction tree data structure that updates the computation during window slide and reuses result from prior computations. It uses balanced tree structure. Tangwongsan et al. proposed a technique called Reactive Aggregator [13] using balanced trees. Partial aggregated data of the sub trees are stored in the internal nodes and a flat array. It handles non invertible or commutative aggregation functions. The pointer less structure also works with Non-FIFO windows. Martin hirzel et al. presented a technique Subtract On Evict (SOE) [14] that uses the previous computed aggregated data to find the current one by updating the aggregated data on each insertion by finding the symmetric difference on each eviction. It does not support invertible operations in aggregation operations. Martin also suggested a SWAG algorithm named order statistics tree [14] for a median like aggregation operation where the data is stored in both queue and tree. The aggregation computation is made by implementing the median of the query using sub tree statistics in a balanced search tree. Carbone et al. designed a hybrid optimization technique [15] by combining pre aggregation technique cutty [Carbone et al., 2016] with window slicing [pairs] by a process of decomposing windows into non overlapping partial aggregates that are shared and combined to calculate the final aggregate. Anatoli U shein proposed an algorithm named Slick Deque[16] to process invertible and non-invertible aggregations uniformly supporting multi-query processing with optimized memory management. It does not support dynamic and multi-node environments. Jonas traub et al. also proposed a technique named Scotty [17] an efficient open-source operator to compute sliding window aggregation using stream processing systems like Apache Flink, Apache Beam, Apache kafka, Apache samza and spark for count based windows. Jonas traub et al. presented a general Stream slicing technique [18] for window aggregation to build partial aggregates and share among concurrent queries of overlapping windows maintaining a slice metadata holding the start and end timestamp of a slice. It scales with many number concurrent windows. Apart from the SWAGs discussed, optimized techniques based on the memory management, the following algorithms are analyzed in detail in the section 4 to measure the performance metric based on the throughput and latency which are the essential metrics for an efficient data stream process. Tangwongsan et al. used a FIFO window mechanism with a simple data structure named Two-Stacks [19] with two fields to hold data and its aggregation in two stacks namely front and back stack used for eviction and insertion of window elements respectively. In order to improve the space complexity, Tangwongsan also proposed an invariant of Two-Stacks technique named Two- Stacks Lite [20] comprising of a single double ended queue maintaining a single field either to hold data or aggregation. One additional aggregation field at the back stack is maintained to hold aggregation updation during insertion. Martin et al. implemented a chunked Array queue data structure interlinked with different reference pointers for aggregation operation called DABA (De Amortized Banker's Aggregator) [21]. The data structure maintains two virtual sub lists, the front and the back sub list for insertion and eviction of window elements. Two-Stacks and DABA works with a space complexity of 2n (for a window size n) to compute the aggregation. To improve the space complexity, Martin et al. also introduced an invariant of DABA named DABA Lite [22] with two additional aggregation fields for the front and the back sub lists for updating the aggregation. DABA Lite improves the space complexity by n+2 Thus it outperforms DABA in memory management.

3. Design of SWAG model

Data stream processing architecture depicting a sliding window aggregator approach is presented in this

section. Before analyzing the SWAGs for its performance, it would be a great advantage to view a design model of SWAG for a better understanding.
SWAG is a technique generally based on the term sliding window processing approach where the content of the window and how often the aggregation is to be computed is determined dynamically based on various windowing semantics like count based (for eg. past 100 elements), time-based (for eg. past 10 mins) as in [23] or on the basis of both. Based on the data stream processing architecture, a SWAG model can be designed with input, process and output modules to handle the data stream process efficiently as shown in the figure 3.1.

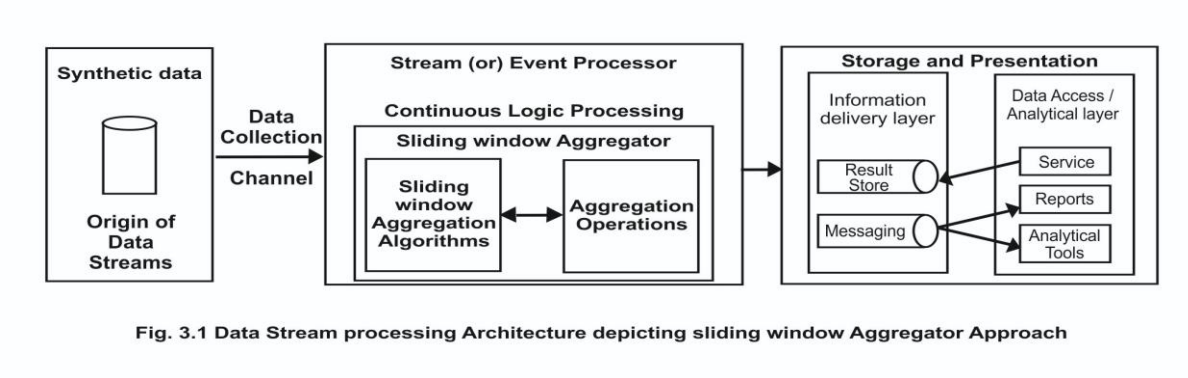

Fig. 3.1 Data Stream processing Architecture depicting sliding window Aggregator Approach

The SWAG model comprises of the following modules:

- **Origin of Data streams (as input module)**
- **Stream or Event Processor (as process module)**
- **Storage and Presentation (as output module)**

Generally, in data stream applications, origin of data streams takes place from different data sources like IoT devices, sensors or applications. This module acts as an input layer for a data stream process. In this work, the synthetic dataset is utilized and is originated from the DEBS 2012 Grand Challenge, a public dataset. Synthetic dataset is an artificial production of data that resembles the real world events. It is algorithmically created that can be used as a test dataset of production (as used in this work) or as an operational data to manipulate mathematical models or to train machine learning models. These data are ingested into the stream processing platform through a data collection channel.
Stream or Event processor acts as a central platform for continuous logic processing. Generally, Data Stream Processing System (DSPS) processes the live, raw data immediately as it arrives. The requirements of a DSPS are provided by Stone braker et al. [24] in a detailed manner. It also handles the challenges of incremental processing, scalability and fault tolerance in a data stream process. Hence a DSPS is necessary to overcome gaps in processing huge data volumes as data streams have to be processed on the fly. In this work, computation of the aggregation is the data stream process and is performed by SWAG using a user defined aggregation operation like sum, count, max, maxcount and so on. In this work, input data is originated from synthetic dataset and the aggregation operation to be performed (in this work max and maxcount) is defined by the user. The aggregation operation max returns the maximum of window elements and maxcount returns count of occurrence of the maximum in the window elements. SWAG uses a popular optimization technique called windowing to break data into finite streams with an implementation of data structures like stack and queue. A stream processor opens a window when it receives the first data element and closes when it meets some criteria. The criteria can be based on time interval, count of data or a condition. This work deals with count based data which is also given as an input. The window size varies from 2,4,8,16,32....512. For a specific window size n, computation of aggregation is performed on each window slide by various SWAG algorithms. The window slide is taken as one. The computed information is measured for throughput and latency performance. The combined technique of sliding window with aggregation operation is called Aggregate Continuous Query (ACQ). Basically, SWAG algorithms are implemented to handle in-order and out-of-order data stream [25]. This work computes the aggregates for the synthetic datasets to measure the performance metric by analyzing various in-order SWAG algorithms.
Storage and presentation are the two main supporting systems in data stream processing. This module acts as an output layer in the design model. The storage system acts as an information delivery layer which maintains a summary of the input data stream and the results of the computation that can be used for future references. It also acts as a service on demand by an end user. The presentation system can be used by the consumers to visualize the data. It acts as a messaging system to generate reports /an analytical system to alert the end users. In case of real-time applications for example in case of online retail store, the order placement (data) would be more frequent during working (peak) hours and less frequent during non-working hours. So it is necessary to scale the window sizes based on the requirement. In this work, four In-order SWAG algorithms are analyzed experimentally on a synthetic dataset of 200 million items for 200 iterations and the performance is measured in terms of throughput and latency for varying window sizes.

4. In-order SWAG algorithms

In data stream processing, selection of suitable windowing and aggregation mechanism helps to minimize the latency to a greater extent. Memory footprint plays a major role in improving the performance metric of a data stream process. Memory allocation and deallocation of a data stream process is efficiently handled dynamically using pointers. In this work, data from synthetic dataset ingested into the window elements are taken for continuous logic processing for a user defined aggregation operations max and maxcount. The continuous logic processing in SWAGs involve the three following basic operations namely insert, evict and query.
Insert () - Inserts window elements in to the data structure maintained. In SWAG, during initial insertion of window elements in to the data structure, the aggregation is computed with the identity element.
Evict () - Removes the oldest window element from the data structure during a window slide.
Query () - Returns the computed aggregated data for a given data stream process. In case if the data structure is found empty, the aggregation is computed as one.
Following section details the four in-order SWAG algorithms taken for this comparative study.

4.1 Two-Stacks algorithm

Two-Stacks is a simple in-order SWAG algorithm with a data structure comprising of two stacks namely, the front stack (being rotated 90° left) and the back stack (rotated 90° right). These stacks maintain three pointers F, B and E. The pointer F denotes the top of the front stack that facilitates insertion of window element into it and the E denotes the top of the back stack that facilitates eviction of window elements from it. The pointer B denotes the bottom of the two stacks that is visualized at the center to avoid clutter. The figure 4.1 depicts the general form of Two Stacks algorithm.

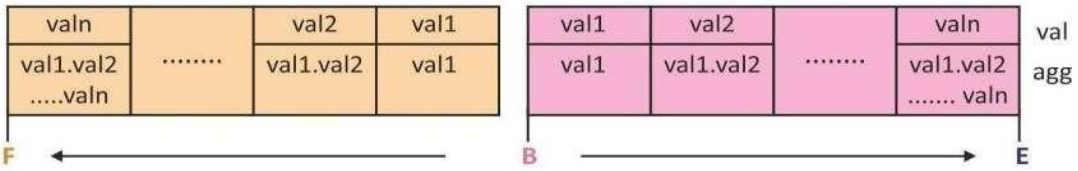

Fig 4.1 General form of Twostacks data structure

In the Two-Stacks algorithm, each element in the stack contains two fields val, denote the window elements as val1, val2...and agg , which denote the computed aggregated data as val1.val2.val3 where '.' denotes the aggregation operation performed chosen by the user.

4.1.1 *Pseudo Code for Insert, Evict and Query operation in Two-Stacks*

Insert ( )

1. Pushes an element (say val1) onto the top of the back stack
2. Computes partial aggregation of pushed element and the previously inserted elements i.e., val1.val2. valn.

Evict ( )

1. Pops element from the front stack if the front stack is non-empty.
2. If front stack found empty, a flip operation is activated by pushing all elements from the back stack to front stack along with the partial aggregation computation done in a reverse direction.

Query ( )

1. Pops the computed aggregated value from front and the back stack
2. If the front or the back stack is empty, the aggregation is set to the identity element 1 for computation of aggregation.

In this algorithm, as the window size increases, the implementation of flip operation leads to the degradation of performance thus affecting the throughput. For a window size n, it consumes a space complexity of 2n stored partial aggregates. In order to improve the performance, Two-Stacks Lite algorithm has been developed.

4.2 Two-Stacks Lite algorithm

Two-Stacks Lite algorithm data structure comprises of a single double-ended queue with three pointers F, B and E. The pointer F points to the start of the queue, B points to the location between the start and end of the queue and E denotes the end of the queue. The pointer B separates the queue into two virtual sub lists I_F and I_B. For a window size n, in order to reduce the space complexity to store the computed partial aggregated data, Two-Stacks Lite maintains data elements with one additional aggregate field aggB in the back sub list I_B to store the aggregated data instead of n aggregate fields as in Two-Stacks algorithm. The general form of Two-Stacks Lite algorithm is depicted in the following figure 4.2.

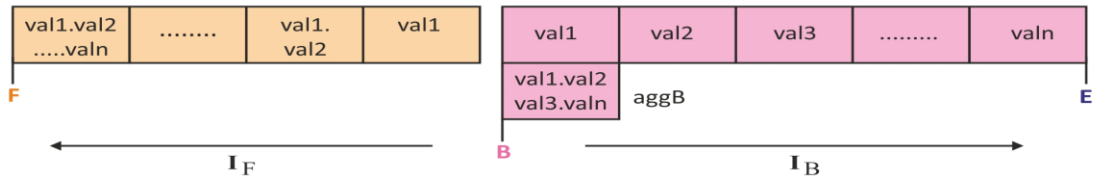

Fig 4.2 General form of TwostacksLite algorithm

Fig 4.2 General form of Two-Stacks Lite data structure

Similar to Two-Stacks algorithm, insertion of window elements is made in the back sub list I_B and eviction of window elements is made in the front sub list I_F. In Two-Stacks Lite algorithm, the partial aggregation is carried out towards right in the sub list I_B and towards left in the sub list I_F. Unlike Two-Stacks algorithm maintaining n partial aggregates, the Two-Stacks Lite algorithm maintains the partial aggregation of the entire front sub list I_F at the front end of the queue F and the partial aggregate of the entire sub list I_B in the additional partial aggregate field aggB. Thus it consumes a space complexity of n+1 partial aggregates.

4.2.1 Pseudo code for Insert, Evict and Query operation in Two-Stacks Lite

Insert ()

1. Enqueue the window element in to the back sub list.
2. Computes the partial aggregation of inserted elements and is maintained in the aggB field

Evict ()

1. Dequeue the element from the front sub list if the front sub list is non-empty
2. If front sub list found empty, all the elements from the back sub list are enqueued in to the front sub list. Computes the partial aggregates of the enqueued elements in the reverse direction in the front sub list.

Query ()

1. Dequeue the computed aggregation by aggregating the front sub list and the back sub list (maintained in the aggB field)
2. The aggregation is computed as an identity element 1 in case of empty sub lists.

Even though the space complexity is improved than Two-Stacks algorithm, the flip operation again makes the implementation expensive and time-consuming as the window size increases. Thus to handle the flip operation efficiently, DABA algorithm has been developed.

4.3 DABA (De Amortized Banker's Aggregator) Algorithm

DABA is an in-order SWAG algorithm that stands for De Amortized Banker's algorithm. It differs from Two-Stacks and Two-Stacks Lite algorithms by an early flip operation to improve the performance.
DABA data structure comprises of a queue of data elements which is a structure containing two fields val to store the window elements and agg to store the partial aggregates as shown in the following figure.

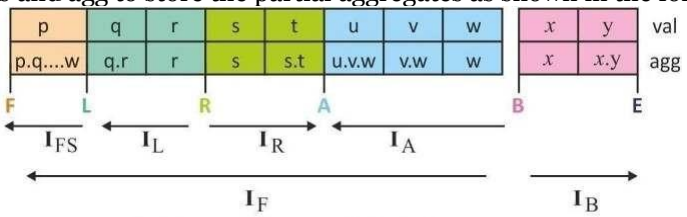

Fig 4.3 General form of DABA datastructure

The data structure comprises of chunks of memory separated with six pointers F, L, R, A, B and E in the queue. The pointers F and B forms a boundary for two virtual sub list I_F and I_B. The sub list I_F facilitates eviction of data elements whereas sub list I_B facilitates insertion of window elements. From the Fig 4.3, it is observed that the front sub list I_F is further divided into four sub lists namely I_{FS}, I_L, I_R and I_A.
The direction of computation of aggregation and the pointers maintaining the partial aggregation of the sub lists can be tabulated.

Table 4.1 Pointers maintaining the aggregation value with the aggregation direction in the sub list

Sub list	I_{FS} , I_L, I_A	I_R, I_B
Pointer holding the aggregation	F, L, A	R, E
Aggregation direction	Left	Right

Insertion and Eviction Operations

During insertion and eviction of data elements of the window into the data structure, this algorithm performs

some pointer restoration operations to restore the original size invariants of the pointer. It is shown in table 4.2.

Table 4.2 Pointer restoration operation to restore the size invariants of the pointers

Pointer Restoration operation	**Time of Activation**	**Description**	**Pointer Restoration Code**
Flip	Insertion, Eviction	Flip operation is activated during initial insertion and when | I_B | >= | I_F |	Flip operation makes | I_B | = | I_L | = | I_R | = | I_A | =0
Shrink	Insertion, Eviction	Shrink is activated when |I_{FS} | and | I_L | are non- empty, and always after aflip operation. Shrink executes with or without flip	During insertion | I_B | = 0, | I_A | = | I_A | + 1,| I_{FS} | =1, | I_L | = | I_R | During insertion, shrink without flip makes | I_B | = | I_B | + 1,| I_A | = | I_A | + 1, | I_L | = | I_R| During eviction, Shrink without flip makes | I_A | = | I_A | + 1, | I_L | = | I_R| During initial insertion, shrink with flip is activated and is referred to as the singleton case
Shift	Insertion	Shift operation is performed during the insertion of window elements into the datastructure when | I_{FS} |> 0 and | I_L | and | I_R| are empty	Shift makes | I_B | = | I_B | + 1, | I_A | = | I_A | - 1

Query operation

Query operation is activated to retrieve the computed final aggregated data at an instant of time from the computed aggregation of I_F and I_B.

Even though the performance is improved by an early flip operation, two separate fields for each data for maintaining window elements and its aggregation consumes memory and needs again a space complexity of 2n to store the partial aggregates. Thus to improve the performance, DABA Lite has been developed.

4.4 DABA Lite Algorithm

The working of DABA Lite algorithm resembles the DABA algorithm but it differs in data structure in maintaining the data and its aggregation fields. The data structure of DABA Lite is depicted in the following figure 4.4.

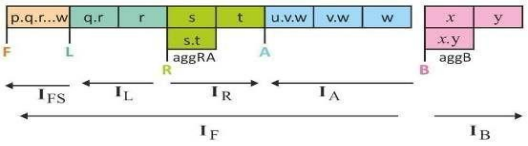

Fig 4.4 General form of DABALite data structure

Insertion and Eviction Operations

DABA Lite functions same as the DABA algorithm. During insertion and eviction, it follows the same set of pointer restoration operations shown in table 4.2.

Query operation

During query operation unlike DABA, DABA Lite retrieves the final aggregation at any instant from the aggregation of I_{FS} and aggB field.

Thus in DABA Lite, the performance is improved by efficiently managing the memory with only two additional fields for the entire sub lists. Thus it consumes only a space complexity of n+2 to store the partial aggregtes.

From the working nature of the above discussed algorithms, taking the window size to be n, the space complexity for maintaining partial aggregations of SWAG can be summarized in table 4.3

Table 4.3 Space Complexity of In-order SWAG to compute aggregates

Algorithm	**Space Complexity**
Two-Stacks	2n
Two-Stacks Lite	n+1
DABA	2n
DABA Lite	n+2

On analyzing the working nature of the discussed SWAG, Two-Stacks Lite and DABA Lite algorithms handles the memory resources efficiently. But DABA Lite outperforms Two- Stacks Lite by an early flip operation during the insertion and eviction of window elements. For further analysis, the four SWAGs are measured for throughput and latency for varying window sizes.

5. Result Analysis

Experimental analysis is made in this section with synthetic datasets with implementation in c++ and python. The SWAG algorithms Two-Stacks, Two-Stacks Lite, DABA and DABA Lite are analyzed to determine the performance metric in terms of throughput and latency.

5.1 Throughput Analysis

In a data stream process, the purpose of SWAG algorithm plays a vital role in analytical applications. In this work, throughput is measured as the number of computation of aggregates per second. By varying the window sizes from 1 to 512(where each window size is a power of 2, say 1, 2, 4, 8....512) on the synthetic dataset of 200 million data items, throughput analysis is made. The throughput is expressed as how many million items (aggregates computed) are processed per second. The aggregation is calculated during the window slide for the entire window elements. The throughput performance of the four algorithms Two- Stacks, Two-Stacks Lite, DABA and DABA Lite is calculated experimentally for varying window sizes and is depicted in the table 5.1

Table 5.1 Throughput Analysis for varying window sizes

	Window size							
Algorithm	1	2	4	8	16	32	256	512
DABA	23.48	21.27	21.24	21.33	20.24	21.38	22.95	23.75
DABA Lite	26.13	25.31	26.57	27.10	27.56	26.93	30.69	30.64
Two-Stacks	23.97	23.89	23.97	23.97	23.90	23.82	23.18	22.39
Two-Stacks Lite	22.79	23.33	23.38	23.66	23.77	23.49	23.30	23.21

From the table 5.1, following observations can be noted. When the window size is gradually increased, the algorithms other than DABA Lite do not exhibit much improvement in performance. But DABA Lite shows an improvement in performance for increasing window sizes. Unlike DABA, DABA Lite maintains window elements and its partial aggregates in separate sub lists for faster computation. So DABA Lite outperforms in throughput performance than DABA with 26% increase. An early flip operation and the chunked memory implementation in DABA Lite outperforms Two-Stacks with 17% increase and Two-Stacks Lite with 18% increase. This brings the result that DABA Lite performs well with a stable increase in throughput than other algorithms for varying window sizes.

The following figure 5.1 shows the graphical representation of throughput analysis for clear visualization.

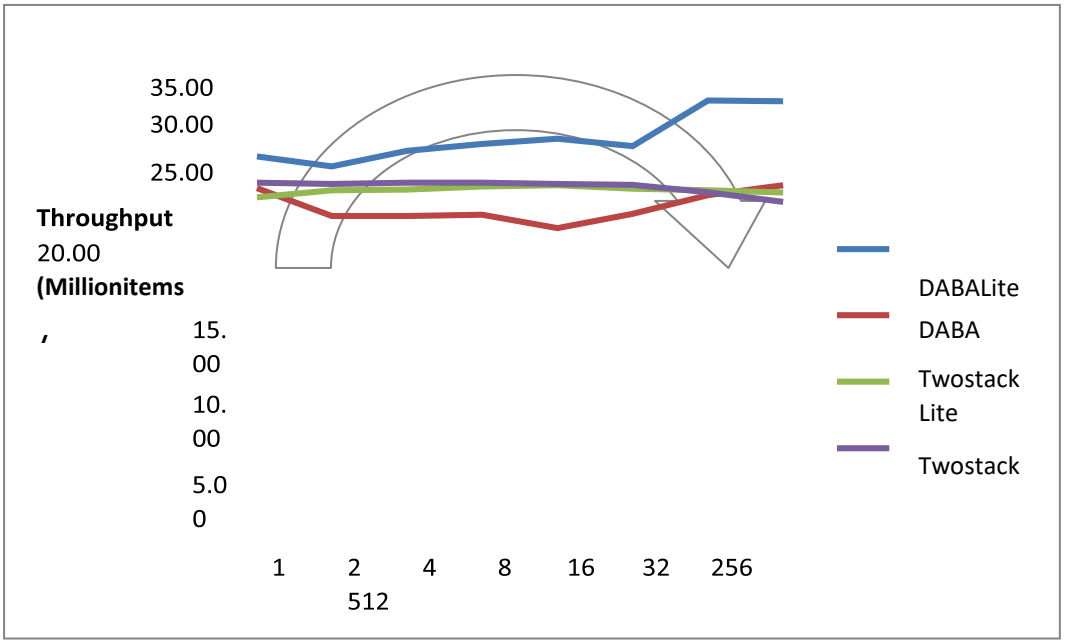

Fig 5.1 Throughput Analysis of SWAG Algorithms

5.2 Latency Analysis

Latency is measured as the difference between the time taken by the data entering the stream processor (taken as initial time) and performing the basic aggregation operations insert, evict and query (taken as processing time) for each iteration. It is measured in terms of processor cycle which denote the delay between initial and processing time. Executing each algorithm for 200 iterations, the least and the largest latency value in terms of processor cycles are tabulated in the following table. The performance of SWAG algorithm will be high when the latency value is low. The latency performance of the four algorithms Two- Stacks, Two-Stacks Lite, DABA and DABA Lite is tested experimentally and is depicted in the table 5.2.

Table 5.2 Latency analysis of in-order SWAG algorithms for varying window sizes on a synthetic dataset of 200 million items for 200 iterations

	window size													
Algorithm	2		4		8		16		32		256		512	
	Least	Large	Least	Large	Least	Large	Least	Large	Least	Large	Least	Large	Least	Large
DABA	155	70784	141	72475	131	70209	131	93318	131	50041	127	49389	127	56698
DABA Lite	133	40794	115	51340	111	55701	111	53139	109	44560	109	85138	107	51846
Two-Stacks	121	43424	121	51859	119	49442	121	79988	121	32770	121	48854	119	71681
Two-Stacks Lite	23	51371	101	43071	101	33260	101	48059	101	48612	101	46619	101	88392

From the table 5.2, the following observations are noted. Latency values for the first processed data is considered the least value and the last processed data is considered as the large value for 200 iterations. In all the analyzed algorithms, the least latency value is more or less same. But the largest latency value varies for varying window sizes. It can be observed that for a drastic change in window sizes, DABA Lite performs well when compared with other analyzed algorithms. And in case of small change in window sizes, Two-Stacks Lite performs well. Though both maintain a same structure with window elements and its aggregates in separate sub lists, the flip operation in Two-Stacks Lite degrades its performance for larger window sizes than DABA Lite. Even though the latency performance of Two-Stacks Lite and DABA Lite algorithms is more or less similar, DABA Lite outperforms Two-Stacks Lite in throughput performance. So it is observed that DABA Lite can be used for varying window environments.

6. Conclusion and Future Enhancement

6.1 Conclusion

In this work, in-order data streams are analyzed experimentally on synthetic dataset for the SWAG algorithms namely Two-Stacks, Two-Stacks Lite, DABA and DABA Lite. Result analysis proves that DABA Lite performs well in terms of both **t**hroughput and latency. The performance is consistent for adaptive window environments. It is observed that the better performance of DABA Lite is achieved using chunked array concept.

6.2 Future Enhancement

From the analysis, it is observed that chunked array concept provides a better performance. So it can be implemented to handle out of order data streams. As windowing strategies perform with re-evaluation while processing the data streams, the latency can still be improved using SWAG algorithms being implemented with different data structures to achieve optimized solutions.

References

1. Ajah, Ifeyinwa Angela, and Henry Friday Nweke. “Big Data and Business Analytics: Trends, Platforms, Success Factors and Applications.” *Big Data and Cognitive Computing*, vol. 3, no. 2, 2019, p. 32., doi:10.3390/bdcc3020032.
2. Zhang, Liangwei, et al. “Sliding Window-Based Fault Detection from High- Dimensional Data Streams.” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2016, pp. 1–15., doi:10.1109/tsmc.2016.2585566.
3. Tangwongsan, Kanat, et al. “Optimal and General out-of-Order Sliding-Window Aggregation.” *Proceedings of the VLDB Endowment*, vol. 12, no. 10, 2019, pp. 1167– 1180., doi:10.14778/3339490.3339499.
4. Wu, Jun, and Luo Zhong. “A New Data Aggregation Model for Intelligent Transportation System.” *Advanced Materials*

Research, vol. 671-674, 2013, pp. 2855–2859., doi:10.4028/www.scientific.net/amr.671-674.2855.

5. Hochreiner, Christoph, et al. "Elastic Stream Processing for the Internet of Things." *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, 2016, doi:10.1109/ cloud.2016.0023.
6. Arasu, Arvind, and Jennifer Widom. "Resource Sharing in Continuous Sliding- Window Aggregates." *Proceedings 2004 VLDB Conference*, 2004, pp. 336–347., doi:10.1016/b978-012088469-8.50032-2.
7. Li, Jin, et al. "No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams" *ACM SIGMOD Record*, vol. 34, no. 1, 2005, pp. 39–44., doi:10.1145/1058150.1058158.
8. Krishnamurthy, Sailesh, et al. "On-the-Fly Sharing for Streamed Aggregation." *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, 2006, doi:10.1145/1142473.1142543.
9. Ghanem, Thanaa M., et al. "Incremental Evaluation of Sliding-Window Queries over Data Streams." *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 1, 2007, pp. 57–72., doi:10.1109/tkde.2007.250585.
10. Shein, Anatoli U., et al. "FlatFIT : Accelerated Incremental Sliding-Window Aggregation For Real-Time Analytics", *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, 2017, doi:10.1145/3085504.3085509.
11. Cagri Balkesen and Nesime Tatbul, '*"Scalable Data Partitioning Techniques for Parallel Sliding Window"* ,8th International Workshop on Data Management for Sensor Networks (DMSN 2011) Copyright 2011.
12. Bhatotia, Pramod, et al. "Slider: Incremental Sliding Window Analytics" *Proceedings of the 15th International Middleware Conference on - Middleware '14*, 2014, doi:10.1145/2663165.2663334.
13. Tangwongsan, Kanat, et al. "General Incremental Sliding-Window Aggregation." *Proceedings of the VLDB Endowment*, vol. 8, no. 7, 2015, pp. 702–713., doi:10.14778/2752939.2752940.
14. *Tutorial: Sliding-Window Aggregation Algorithms*,DEBS '17, Barcelona, Spain , 978- 1-4503-5065-5/17/06, 2017
15. Carbone, Paris, et al. "Stream Window Aggregation Semantics and Optimization." *Encyclopedia of Big Data Technologies*, 2018, pp. 1–9., doi:10.1007/978-3-319- 63962-8_154-1.
16. *Slickdeque: High Throughput and Low Latency Incremental Sliding-Window ...* openproceedings.org/2018/conf/edbt/paper-197.pdf.
17. Traub, Jonas, et al. "Scotty." *ACM Transactions on Database Systems*, vol. 46, no. 1, 2021, pp. 1–46., doi:10.1145/3433675.
18. *Efficient Window Aggregation with General Stream Slicing*. www.nebula.stream /paper/ traub_edbt2019.pdf.
19. *Hammer Slide: Work- and CPU-Efficient Streaming Window Aggregation*. spiral.imperial.ac.uk/bitstream/10044/1/62249/2/SIMDWindowPaper_ADMS.pdf.
20. Tangwongsan, Kanat, et al. "In-Order Sliding-Window Aggregation in Worst-Case Constant Time." *ArXiv.org*, 29 Sept. 2020, arxiv.org/abs/2009.13768.
21. Kanat Tangwongsan , Martin Hirzel, Scott Schneider 'Constant-Time SlidingWindow Aggregation', IBM Research Report, RC25574 (WAT1511-030)November 11, 2015
22. Tangwongsan, Kanat, et al. "In-Order Sliding-Window Aggregation in Worst-Case Constant Time." *The VLDB Journal*, vol. 30, no. 6, 2021, pp. 933–957., doi:10.1007/s00778-021-00668-3.
23. *Data Deduplication Techniques for Big Data Storage Systems*. www.ijitee.org/wp-content/uploads/papers/v8i10/J91290881019.pdf.
24. Stonebraker, Michael, et al. "The 8 Requirements of Real-Time Stream Processing." *ACM SIGMOD Record*, vol. 34, no. 4, 2005, pp. 42–47., doi:10.1145/1107499.1107504.
25. Bou, Savong, et al. "CPIX: Real-Time Analytics over out-of-Order Data Streams by Incremental Sliding-Window Aggregation." *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 11, 2022, pp. 5239–5250., doi:10.1109/tkde.2021.3054898.