# Technical report

We implemented our algorithm in C because it is a language with minimal runtime overhead, ensuring that the measured performance is closer to the actual execution time of the algorithm, an important factor when precise timing information is required for performance analysis. In languages like Python and Java, there are additional layers of abstraction, such as virtual machines and garbage collectors, which can introduce some level of runtime overhead. Being closer to the hardware, C often has lower overhead, making it more suitable for scenarios where every computational cycle counts. The additional layers in high-level languages can introduce variability in performance due to factors like just-in-time (JIT) compilation and automatic memory management. C, on the other hand, tends to offer more predictable performance, making it suitable for applications where precise control over timing is essential, such as the algorithmic complexity analysis that we perform on our algorithm.

## Single Buffer

```
input: event e and its timestamp ts

1 if lazySort or inOrder then
2 add(e)
3 else if eagerSort then
4 addInOrder(e, ts)
5
6 if tick then
7   if lazySort then heapSort()
8   windowInterval = scope(e, ts)
9   evictInterval = updateEvictInterval(windowInterval)
10  if report then
11    content = extractData(windowInterval)
12    reportContent(content)
13  if evict then
14    evictData = extractData(evictInterval)
15    evict(evictData)
```

**Algorithm 1**, Single Buffer *inOrder*, *lazySort* and *eagerSort* are the conditions that drive the addition of an element to the buffer; *report* and *evict* are the conditions that enable the respective procedures; *windowInterval* and *evictInterval* are triplets of start timestamp, end timestamp and size of the correspondent window or evict interval.

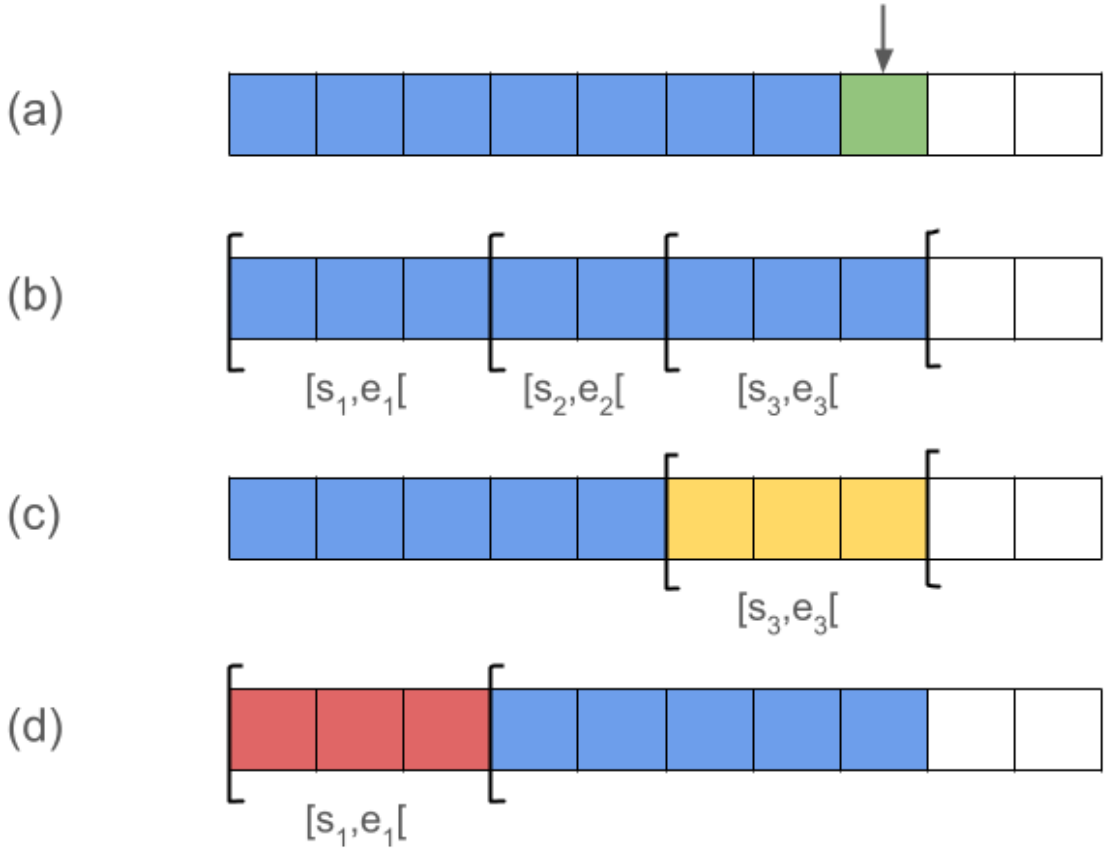


**Fig. 0** a) add (lines 1-4 and 7): insert a new processed element (green) into the buffer b) scope (line 8): compute the window of an element, return the time interval c) content (lines 10-12): extract and return the physical elements (yellow) of the stream corresponding to a given window time interval d) evict (lines 9 and 13-15): erase windows and their physical elements (red) from the buffer, given the set of windows

The single buffer variant of the algorithm enables the processing of the input stream by using a single data structure to store and manage all the windows related to the stream elements, Fig. 0 represents Algorithm 1 providing a high-level overview of the implemented functions. This solution is memory efficient because it stores only the stream elements in a C array and some selected relevant pointers such as the tail of the buffer or the pointer to the first element of the current frame, resulting in a single data structure that never exceeds the size of the input stream that can support the entire algorithm execution. This approach leads our algorithm to have all the main functions with time complexity coupled with the number of elements in the stream. To prevent running out of available cells in the stream elements array, we implemented a circular array that overwrites the cells of already evicted elements no longer valid (Fig. 1), giving the user the responsibility to select an eviction policy that does not lead to an overwrite of still valid elements, but guaranteeing the possibility of processing an infinite number of elements. The array data structure has been chosen because of its index-access property, enabling the implementation of procedures like binary search and heap sort.

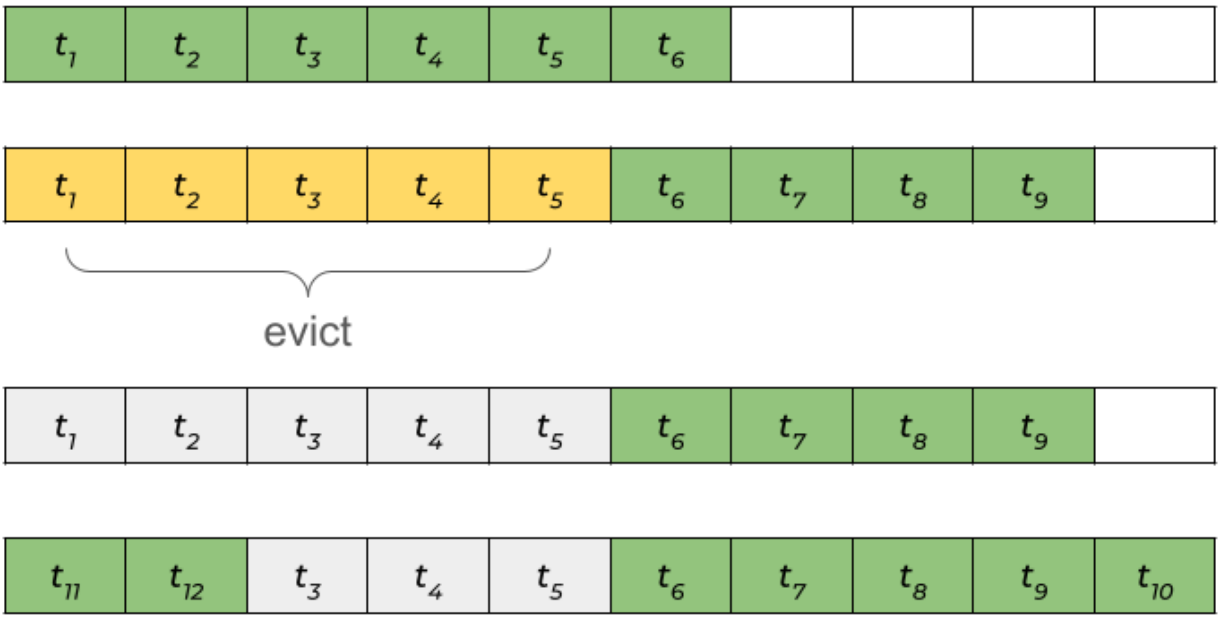


**Fig. 1** When the elements in the buffer are close to filling the total size of the array, a portion of the elements is evicted and marked as invalid. From this point on, the successive elements can overwrite the invalid ones, enabling the array to be circularly indexed.

## Windows computation

The first two operations performed by the algorithm are ***add*** and ***scope***, the first to add a new element into the buffer, and the latter to compute the time interval of the window to which the element belongs. At the arrival of an element, the processing engine behaves differently depending on the sorting policy adopted for the out-of-order management, needed for the correct processing of windows. If no out-of-order is expected from the element arrival, then the new element is just added at the tail of the buffer (Fig. 2a), achieving an element addition cost of O(1), otherwise, two exclusive out-of-order policies are distinguished based on their sorting approach. If the processing engine should sort the buffer at every element arrival, resulting in a continuously ordered buffer, it performs **eager sort**; if the processing engine sorts the element buffer only when the ***scope*** operation should be done, it performs **lazy sort**. The **eager sort** inserts the new element always in order into the buffer (Fig. 2b), eventually right-shifting all the successive elements, iterating O(n) elements for every *add* operation. With this strategy, we increase the cost of the addition from O(1) to O(n) w.r.t. the in-order case, but we maintain the ***scope*** operation cost equal to O(n) because the buffer is always ordered and ready to be traversed for the computation of the windows. When the ***lazy sort*** method is applied, a new element ingested by the engine is appended to the buffer even if it has a delayed event time, making it possible to perform the element addition still in O(1). Then, when the **scope** operation has to be performed, the program executes a sort of the entire buffer, a **heap sort** in O(nlogn) time, before the execution of the ***scope*** operation, needing a total of O(nlogn + n) time for completing the operation. The **scope** operation traverse and analyze all the elements while maintaining context variables that predicate when a window should be opened, updated, or closed, for each processed element. The

algorithm implements in the **scope** function the logical operator described in FRAMES[0], building the windows incrementally and returning the window time interval of the last element added into the buffer before the *scope*.

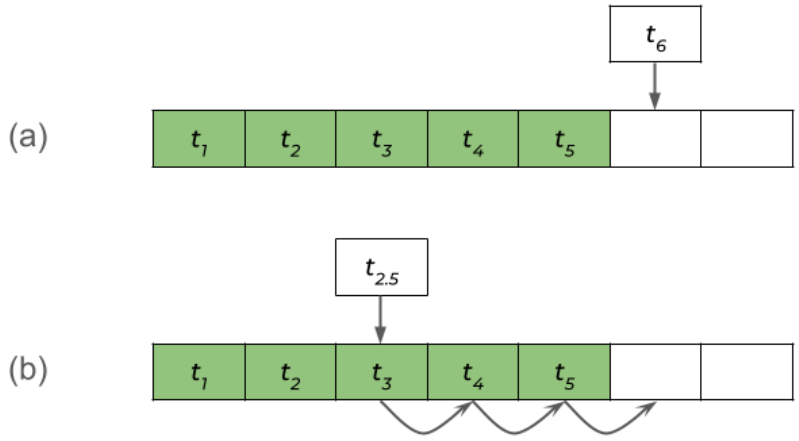


**Fig. 2a** A new element ***t6*** ingested from the stream processing engine is added to the tail of the buffer, event times of the elements are not considered. **Fig. 2b** A new element ***t2.5*** ingested from the stream processing engine is added in order into the buffer comparing its event time with the other elements' event time. Inserting the new element between two existent elements leads to a right-shifting of the successive ones until the end of the buffer.

## Elements retrieval

In the second part of Algorithm 1 (lines 9-14), the procedures that operate on the buffer's physical elements are performed after the buffer update and management. Firstly, if the report condition is met, the window interval propagated by the ***scope*** function is used to extract the physical elements from the stream that belongs to the window. This operation is called ***content***, and reports these elements to the query operator. To execute the operations that extract elements from the buffer efficiently, we decided to implement the latter as an indexed data structure, choosing the array, native of the C language. To find the elements, a binary search is implemented, achieving the full resolution of the search in O(logn) iterations. This operation looks for the first element of the window by using the timestamp as the key, and then the full window is extracted using its size, finding all the elements contiguously in the array. The timestamp of the first element of the window and the window size were returned by the *scope* function. The same technique is applied for the ***evict*** operation, finding the first of the elements that should be evicted with the binary search, then extracting all the others by using a size that can be cumulated after a certain number of ***scope*** operations that can be defined by the user. For instance, if the user decides that after X windows are created, Y should be evicted, with $X \leq Y$, then the timestamp of the first element of the first window that will be evicted is saved as the key for the future binary search, and the number of successive contiguous elements to extract is cumulated for each new processed elements, in this way we are sure to avoid cutting a window when evicting elements.

# Multi-Buffer

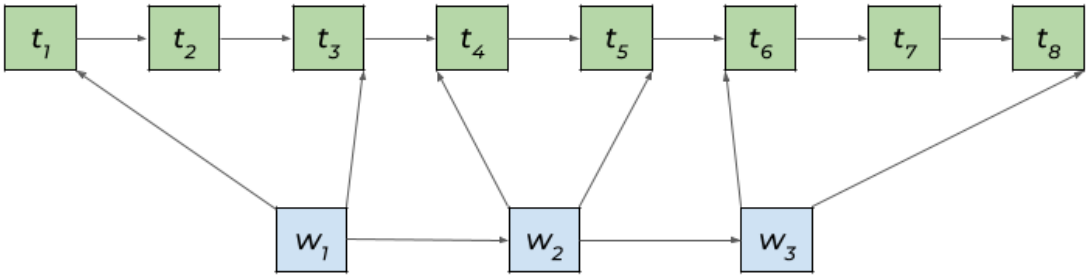


**Fig. 3** Data structure supporting the Multi Buffer implementation, an ensemble of two linked lists. The lower linked list with light blue nodes represents the windows, for each window, exists a pointer to the start and one to the end element of the window. The upper linked list with green nodes represents the elements of the stream.

The Multi Buffer version of the algorithm takes advantage of the implementation of multiple data structures to store and manage the windows state. The algorithm no longer uses a single array, but two coupled linked lists (Fig. 3), one used to store the stream elements and the other to represent the windows. Every node of the windows list represents a frame and has two main pointers to the elements list, one points to the start tuple and one to the end tuple of the window. The window node also stores the start and end timestamps of the corresponding time interval and the number of elements in the window. With this technique, we can dynamically change at runtime the value of the pointers of each window edge, keeping up-to-date the windows state and always having a reference to every window already computed. Having a data structure that stores the list of windows significantly decreases the algorithmic complexity of the implementation, making it possible to have the number of iterations in the main operation, that is the ***scope***, proportional to the number of windows instead of the number of elements in the stream. The flexibility and scalability of the linked list data structure, make it possible to insert and remove nodes easily at runtime in the inner parts of the structure, and also to grow in size without the need of circular data structures because of the not necessary declaration of needed memory in advance.

```
input: event e and its timestamp ts

1  windowInterval = scope(e, ts)
2  add(windowInterval, e, ts)
3
4  if tick then
5    recomputeBuffer(windowInterval)
6    if report then
7      content = extractData(windowInterval)
8      reportContent(content)
9    if evict then
10     evictWindows()
```

```
recomputeBuffer(windowInterval):

    newWindowInterval = recomputeFrame(windowInterval)
    if newWindowInterval.end < windowInterval.end then
        toMergeWindow = split(newWindowInterval)
        newWindowInterval = merge(toMergeWindow, windowInterval.next)
        recomputeBuffer(newWindowInterval)
    else if newWindowInterval.end = windowInterval.end then
        return
    else if newWindowInterval.end > windowInterval.end then
        newWindowInterval = merge(newWindowInterval, newWindowInterval.next)
        recomputeBuffer(newWindowInterval)
```

## Split & Merge

The high-level overview of the Multi Buffer procedure is represented by Algorithm 2. Thanks to the data structure that keeps track of the windows, it is possible to update the overall state by first executing a ***scope*** operation on the windows list to find the window to which the new processed element belongs, then executing the **add** operation on the corresponding found window. Differently from the Single Buffer version, it is not longer needed to recompute all the windows, by traversing all the stored stream elements, each time a new element is processed, instead, we can perform the ***scope*** operation in O(w) iterations, with w equal to the number of windows in the stream. The ***add*** operation inserts the new element in its corresponding window right after the reference is found, making it possible to do it in O(1). When all the events arrive by their event time order, the Multi Buffer data structure remains consistent at each addition, because the algorithm computes the windows incrementally. When the events arrive out of order w.r.t. their timestamp, the function **recomputeBuffer** repairs the data structure to ensure the correctness of the windows set. The algorithm needs this procedure because the addition of an element to a window is made by checking the timestamp in this case, but not its attributes, which is instead required in data-driven windows. To restore the correctness of the data-driven windows, the algorithm recomputes

the window that was updated with the new element, then starts to perform the Split & Merge procedure shown in Fig. 4 and described by the pseudo-code function *recomputeBuffer* in Algorithm 2. The worst case occurs when the event time of an element is much later than the processing time and belongs to a window that has been closed for a long time, i.e. the window is represented by a node that is at the beginning of the concatenated list of windows. In this case, the reconstruction via split and merge may have to reconstruct the entire multi-buffer, possibly splitting each existing window and then merging it with the next one, for each event in the buffer, finding itself performing O(nw) operations. This operation is implemented by allocating a new node in the linked list when doing a split and by removing a node when doing a merge, coherently with the update of the pointers to the edge elements of the window. The linked list was necessary to implement this functionality because makes it possible to insert and remove nodes from the structure easily and by allocating or freeing only the necessary memory. A key optimization in this procedure is to stop the split and merge mechanism at the moment when the recomputation of the window coincides with the original window, realigning with the previously computed buffer. This exit condition from the procedure is obtained by the algorithm by comparing the end timestamp of the windows.

### Elements access optimization

Differently from the single-buffer implementation, the multi-buffer version does not use anymore an indexed data structure as it was the native C array, instead, with the multi-buffer we need to traverse the windows list to find the one we want to report to the query operator. The function that extracts the elements iterates O(w) elements to have all the references to the elements available, it is not possible to achieve logarithmic complexity because the Binary Search algorithm is designed for an array-like data structure. For this reason, even if the buffer is sorted, we cannot achieve an exact O(logn) time complexity, but still a linear time complexity, proportional to the number of windows. The same case applies to the search of the windows to evict. Depending on the evict policy, that can be specified by the user, the evict functionality traverses the windows' buffer to find all the windows to erase, which can be easily deleted from memory by freeing the nodes representing the windows and the nodes representing the tuples. Should be noted that every time we perform a scope operation, it is possible to update context variables and pointers to optimize the successive element access operation such as content and evict. By saving the pointers to the elements that should be reported or evicted when they are found during scope, the respective functions can combine this information with context variables for being executed in O(1), i.e. the cost of accessing a pointer. With these optimizations, we can drop the cost of the element access operations done after the management of the windows.

## References

[0] Grossniklaus, Michael & Maier, David & Miller, James & Moorthy, Sharmadha & Tufte, Kristin. (2016). Frames: Data-driven Windows. 13-24. 10.1145/2933267.2933304.

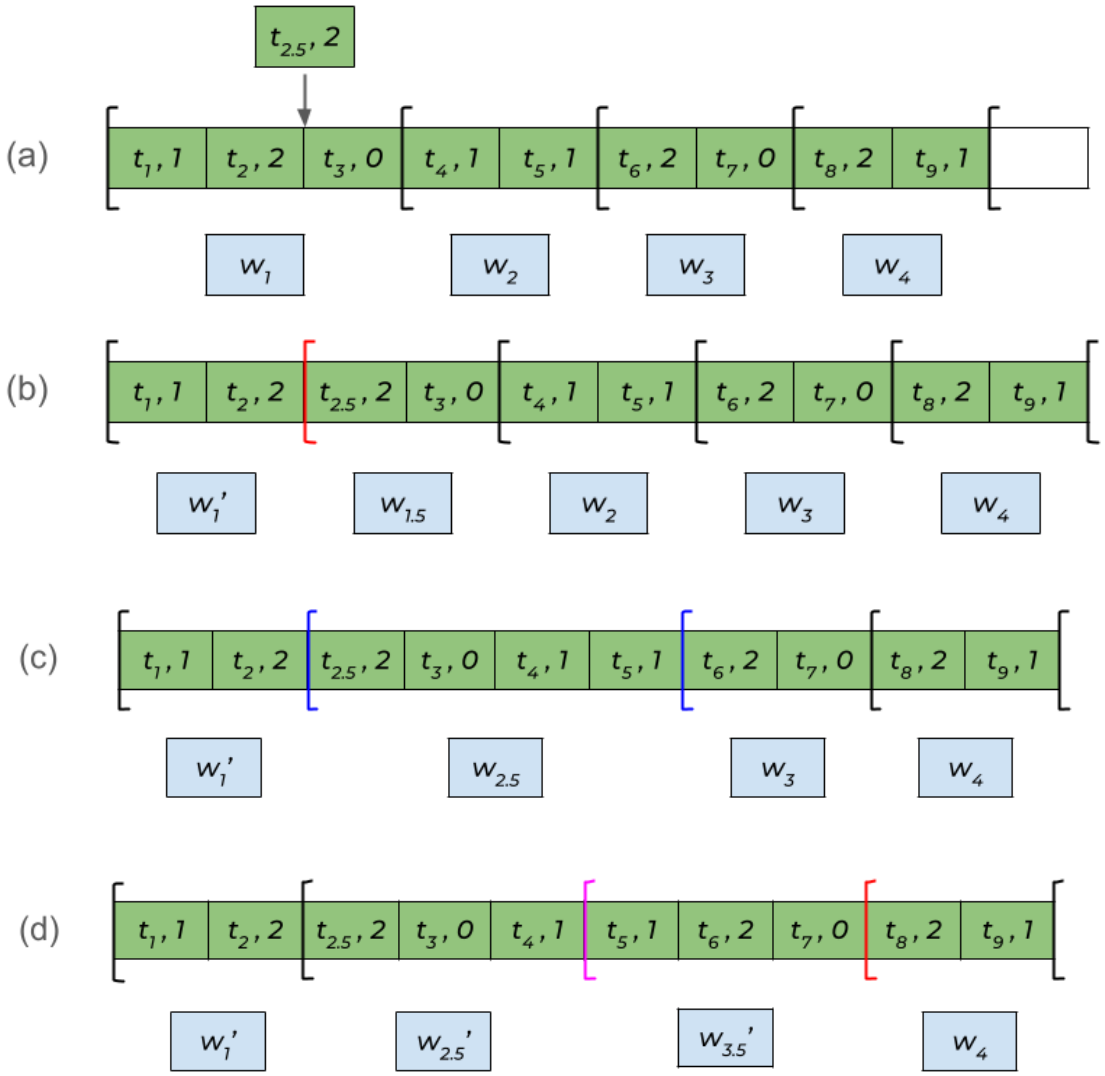


**Fig. 4** The green elements represent the nodes of the linked list storing the stream events, each one composed of the event timestamp t**i** and the attribute used for data-driven segmentation, the blue elements represent the nodes of the linked list storing the windows, and the separators between the events show the window edges implemented by pointers stored in the respective window node. The index *j* of the windows w*j* is a unique identifier. The aggregation framing scheme is adopted in this example, with the sum as the aggregation function, and a threshold of 3. (a) A delayed element is added in order into the window that encompasses the element event timestamp. This operation is done by the **scope** and **add** functions in lines 1-2. (b) The w1 window is recomputed after the new addition (line 5), resulting in a split of the window into two windows, namely w1' and w1.5. In the pseudo-code, in the function ***recomputeBuffer***, the result of the split function ***toMergeWindow*** is window w1.5. (c) The recomputation continues by merging window w1.5 with window w2, resulting in a new window, namely w2.5. (d) Another round of split and merge is executed until the recomputed window interval of window w3.5' matches is actual correct interval, resulting in the realigning with the already computed buffer, stopping the split and merge procedure.