Concurrent Data Structures for Efficient Streaming Aggregation

DANIEL CEDERMAN
VINCENZO GULISANO
YIANNIS NIKOLAKOPOULOS
MARINA PAPATRIANTAFILOU
PHILIPPAS TSIGAS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY / GÖTEBORG UNIVERSITY
Göteborg, Sweden, 2013
Concurrent Data Structures for Efficient Streaming Aggregation

Daniel Cederman  Vincenzo Gulisano  Yiannis Nikolakopoulos
Marina Papatriantafilou  Philippas Tsigas∗
Chalmers University of Technology
{cederman,vinmas,ioaniko,ptrianta,tsigas}@chalmers.se

ABSTRACT
In this paper, we study the problem of streaming multiway aggregation, where large data volumes are received from multiple input streams. Multiway aggregation is a fundamental computational component in data stream management systems, requiring low-latency and high throughput solutions. We focus on the problem of designing concurrent data structures for achieving low-latency and high throughput multiway aggregation; an issue that has been overlooked in the literature. Data structures act as articulation points between pipeline stages of streaming aggregation, supporting parallelism and balancing the work between the aggregate stages. By studying the use and limitations of existing aggregate designs and the data structures they use, we motivate the need for new ones. We present two new concurrent data structures and their lock-free linearizable implementations, supporting both order-sensitive and order-insensitive aggregate functions. We also present an extensive evaluation, based on a variety of aggregation queries on two large datasets, one with data extracted from SoundCloud, a music social network, and one with data from a smart grid metering network. In all the experiments, the proposed data structures and methods improved the aggregation performance significantly, up to one order of magnitude, in terms of both processing throughput and latency over the commonly-used techniques based on queues.

Keywords
data streaming, data structures, lock-free synchronization

∗The research leading to these results has been partially supported by the European Union Seventh Framework Programme (FP7/2007-2013) through the EXCESS Project (www.excess-project.eu) under grant agreement 611183, through the SysSec Project, under grant agreement 257007, through the FP7-SEC-285477-CRISALIS project, by the collaboration framework of Chalmers Energy Area of Advance and by the Chalmers Center for E-science.

1. INTRODUCTION
For data intensive computing that can support continuous complex analysis of large data, the data streaming processing paradigm is more appropriate than the “store-then-process” (dataset) one. The large continuously arriving volume of data makes it impractical to store all the information in memory before processing it. As emphasized in [7], the low-latency and high throughput requirements of the continuous real-time complex processing of such increasingly large data volumes makes parallelism a necessity.

Data streaming [19, 1, 6] emerged as an alternative to dataset computing. In data streaming, continuous queries (defined as directed acyclic graphs of interconnected operators) are executed by Stream Processing Engines (SPEs) that process incoming data in a real-time fashion, producing results on an on-going basis. A good portion of the research has so far focused on leveraging the processing capacity of clusters of nodes and originally centralized SPEs [1] evolved rapidly to distributed [3] and parallel [8, 11] ones.

A data streaming application can be seen as a pipeline where data is continuously produced, processed and consumed. In a parallel environment the underlying data structures should provide the means for organizing the data so that the communication and the work imbalance between the concurrent threads of the computation are minimized while the pipeline parallelism is maximised. Although providing the appropriate data structures that best fit the needs of the application in a concurrent environment is a key research issue [13], providing the data structures that best fit data streaming applications has been overlooked, to the best of our knowledge. Existing SPEs such as [3, 8] still rely on basic data structures such as queues; similar is the case with work focusing on the improvement of SPEs’ architectures [2] or continuous queries accessing the same stream of data [17].

By shedding light on the data structures, we identify new key challenges to improve data streaming aggregation, one of the most common and throughput-demanding monitoring application [17]. In particular, we focus on multiway aggregation, where huge volumes of data received from multiple input streams must be merged and sorted in order to be processed deterministically [8]. Sample application scenarios include monitoring applications in the context of social media, where information could be aggregated to study trends, or update real-time pricing in the context of smart grids.

Contributions. We study data structures as articulation points between pipeline stages of streaming aggregation. The shared access to the data by the collaborating threads defines new synchronization needs that can be integrated in the functionality provided by the shared data structures. By studying the use and limitations of existing aggregate designs and the data structures they use, we mo-
tivate the need for a new approach. We propose two concurrent and lock-free data structures (T-Gate and W-Hive) upon which we build three enhanced multiway aggregate operators that outperform existing implementations in both order-sensitive and order-insensitive functions. We provide an extensive study using two large datasets extracted from the SoundCloud social media and from a smart grid metering network. For both datasets the enhanced aggregation resulted in large improvements, up to one order of magnitude, both in terms of processing throughput and latency.

The paper is organized as follows. Section 2 introduces the data streaming processing paradigm, the multiway aggregate operator and basic concurrency concepts. Section 3 proposes, by rethinking parallelism in this context, how the efficiency of the multiway aggregate operator can be enhanced by means of concurrent data structures. Section 4 presents a detailed overview of the algorithmic design of the enhanced operators and data structures that we propose. In Section 5 we show the liveness and safety properties, namely lock-freedom and linearizability, of the proposed operators and data structures implementations. Section 6 presents the experimental evaluation. We discuss related work in Section 7 and conclude in Section 8.

2. PRELIMINARIES AND PROBLEM DESCRIPTION

Data streaming and multiway aggregation. A stream is defined as an unbounded sequence of tuples \( t_0, t_1, \ldots \) sharing the same schema composed by attributes \( \{ts, A_1, \ldots, A_n\} \). Given a tuple \( t \), attribute \( t.ts \) represents its creation timestamp at the data source while \( A_1, \ldots, A_n \) are application related attributes. Following the data streaming literature (e.g., [3, 8]), we assume that each stream contains timestamp-sorted tuples. In the presence of multiple streams, tuples from different streams may arrive out of timestamp-order, posing a need to merge and synchronize them before processing; in this section we describe this issue in detail.

Data streaming continuous queries (or simply queries) are defined as directed acyclic graphs. Nodes represent operators that consume and produce tuples, while edges specify how tuples flow among operators. Operators are divided into stateless or stateful, depending on whether they keep any state while processing tuples. Due to the unbounded nature of streams, stateful operations are computed over a sliding window, defined by parameters size and advance. Sliding windows can be time-based (e.g., to group tuples received during periods of 5 minutes every 2 minutes) or tuple-based (e.g., to group the last 10 received tuples every 3 incoming tuples). We focus in this paper on time-based sliding windows (or simply windows); we use POSIX notation to specify the periods covered by a window and assume all windows start at time 0. That is, a window with size and advance of 10 and 2 time units, respectively, will cover periods \([0,10), [2,12), [4,14]\), and so on.

The multiway aggregate operator is defined by its window’s size and advance parameters, by a function \( F \) applied to the tuples and by an optional group-by parameter \( K \) (a subset of the input tuple’s attributes), which specifies if \( F \) is applied independently on tuples with different \( K \) value. We focus on deterministic functions, which can be order-sensitive (e.g., forward only the first received tuple) or order-insensitive (e.g., count the number of tuples) with respect to the processing order of the tuples that contribute to the same window. For ease of explanation we focus on individual functions. The proposed methods also apply for sets of functions. If the group-by parameter \( K \) is defined, the operator needs to keep separate windows not only for different time intervals, but also for different values of \( K \). We define a winset as the set of windows covering the same time interval for different values of \( K \).

In scenarios such as parallel-distributed SPEs [8, 4] and replica-based fault tolerant SPEs [3], it is desirable to provide deterministic processing of input tuples (i.e., to produce the same sequence of output tuples given the same sequences of input tuples). When dealing with multiple input streams, processing is not deterministic if tuples are simply processed in the order they are received (i.e., if the processing order depends on the input streams’ inter-arrival times). To ensure deterministic processing, tuples from multiple input streams need to be merged into one sequence and sorted in timestamp order [8], an operation we refer to as S-Merge. A tuple is ready to be processed if at least one tuple with an equal or higher timestamp has been received at each input stream.

Definition 1. Let \( t^i \) be the \( i \)-th tuple received from input \( i \). \( t^i \) is ready to be processed if \( t^i.ts \leq \text{merge}_{i+} \), where \( \text{merge}_{i+} = \min_j \{\max_i(t^i.ts), ts\} \) is the minimum among the latest timestamps received from each input \( j \).

The aggregate operator is composed by four main stages:

1. Add: fetching incoming tuples from each input stream.
2. S-Merge: merging and sorting of input streams’ tuples.
3. Update: updating of the windows a tuple contributes to.

Figure 1 presents an example multiway aggregation query used in a smart grid application to count the number of power outages reported by smart meters and sample execution of the aggregate operator.

Concurrency. We consider systems of concurrent threads. Communication and information exchange relies on shared data and
concurrent shared data structures provide common means for that. Concurrent shared data structures can be implemented in a lock-free way, i.e. guaranteeing that at least one of the threads operating on it is guaranteed to finish its operation in a bounded number of its own steps. Lock-free data structures have been shown to increase throughput and are part of the Java and C# standard libraries. The correctness of such implementations is commonly shown through linearity [10], which guarantees that, given a history of concurrent operations, there exists a sequential ordering of them, consistent with their real-time ordering and with the sequential semantics of the data structure.

3. RETHINKING AGGREGATION’S PARALLELISM

State of the art. Widely used SPEs such as Borealis [3] or StreamCloud [8] perform multiway aggregation by relying on per-input queues to store incoming tuples. Distinct threads insert and remove tuples from such queues and concurrent accesses are synchronized with the help of locks. Figure 2a presents this design, which we refer to as Multi-Queue (MQ). The output thread O_t peeks the first tuple in each queue to determine which one is ready to be processed (input threads and output threads are denoted by I_t and O_t respectively). The same thread is also responsible for the Update and Output operations. Since O_t is the only thread in charge of updating windows, no locking mechanism is required to access the winsets, usually implemented as hash tables to easily support arbitrary numbers of windows and to locate them quickly given the tuple’s group-by parameter K. By changing the queue implementation to a lock-free one we get the lock-free version of MQ (MQ_LF). It is useful to emphasize that we consider the MQ_LF design for the sole purpose of showing that the multiple queue MQ’s (i.e. the common option in state of the art SPEs) throughput and scalability is penalized mainly by the type of data structure used and not from locking. In the following paragraphs we are rethinking the parallelism in data streaming aggregation and identifying challenges that go beyond a simple adoption of lock-free queues.

Parallelization challenges. In existing implementations, S-Merge usually relies on simple sorting techniques, whose cost is linear to the number of inputs. Examples include the Input Merger operator [8] or the SUnion operator [3]. For this reason, (1) the first challenge relies on the parallelization of the S-Merge operation. To this end, extra inter-thread synchronization is needed to ensure that Update is only invoked on tuples that are ready to be processed. (2) The second challenge relies on the parallelization of the Update stage. To guarantee deterministic processing, the result of a window should be outputted only after all its contributing tuples have been processed. For order-sensitive functions, Update cannot be invoked in parallel on tuples sharing the same K value or when no group-by parameter is defined. This restriction can be relaxed for order-insensitive functions, since the result of a window would not be affected by the order in which concurrent threads update it. In both cases, parallelism can be enhanced by a concurrent data structure that synchronizes the access to the operator’s windows.

Utilizing concurrent data structures. We are looking for concurrent data structures that are capable of sorting input tuples at insertion time. In principle, tree-like data structures could provide concurrent logarithmic-time insertion operations. The need for extracting such tuples in timestamp order though, made us believe that a lock-free concurrent skip list [20] would be the right candidate due to its nodes’ structure. Nevertheless, a skip list would not differentiate between tuples that are ready and tuples that are not. Because of that, checking whether a tuple is ready or not would still be penalized by a cost that is linear to the number of inputs as for the multi-queue implementations. Furthermore, it would provide unnecessary functionality (i.e., a more complex implementation) such as deletion of elements at arbitrary positions (only head elements need to be removed in our scenario). Similar considerations hold for a lock-free concurrent skip list used to maintain the operator’s winsets. It would not differentiate between the winsets that have been updated by all their contributing tuples and the ones that have not, and would provide unnecessary functionality. The multi-level shortcuts that the skip list design provides, allow for a logarithmic find of the insert position in the list and this is a feature that the concurrent data structures that are presented in the following section inherited from them. Because of space constraints we refer the reader to Appendix A for a brief performance evaluation complementing the qualitative estimation presented in this paragraph.

3.1 New aggregate designs

This section overviews our enhanced aggregate operators. Figure 2 presents the different designs and describes how operations are assigned to threads. While presenting the different designs, we discuss the data structures needed to maintain tuples and winsets. We also introduce our new concurrent data structures and their APIs (table 1), with the functionality of each method.

**Tuple Merged List - Single Consumer (TuML_C)**. This design (Fig. 2b) addresses the first parallelization challenge by easily performing both Add and S-Merge in parallel. TuML_C relies on the Tuple-Gate (T-Gate), a new concurrent data structure whose API provides two methods, their definitions are given in table 1. Method `insertTuple(tup, input)` allows for tuples to be inserted while being merged and sorted by multiple input streams in parallel. Method `getNextReadyTuple()` guarantees that no tuple is returned for processing before it is ready. The T-Gate stores an ordered list of tuples. Similarly to a skip list, it uses the idea of multi-level shortcuts to locate quickly where to insert a new tuple. By keeping track of the latest added tuple from each input stream, it can quickly check if the first tuple in the list is ready. The output thread reads sorted and ready tuples from the T-Gate and performs the Update and Output stages.

**Tuple Merged List - Multiple Consumer (TuML_MC)**. TuML_MC (Fig. 2c) extends the TuML_C, addressing the second parallelization challenge by performing also the Update operation in parallel. Multiple update threads, U_t, get ready tuples from T-Gate concurrently by invoking `getNextReadyTuple()` and update the windows to which each tuple contributes to. Thus, winsets are now accessed and updated concurrently by the U_t threads. For managing the winsets and synchronizing such access we introduce a new data structure, that we refer to as Window-Hive (W-Hive).

As the T-Gate encapsulates the logic to differentiate between tuples that are ready or not, the W-Hive is able to differentiate between the winset to which incoming tuples are still contributing and the ones whose results can be outputted. It provides two methods: `updateWindows(tup)` allows for multiple threads to synchronize and safely create and update active winsets while `getNextWinSet()` returns the earliest winset no longer being updated by any thread. This method is invoked by the output thread O_t, in charge of forwarding the operator’s output tuples. W-Hive uses similar techniques as the T-Gate to quickly find the right location of where to insert a new winset. To preserve the correctness of
Window Merged List (WiML). This design (Fig. 2d) further enhances the parallelization of the aggregate’s stages for order-insensitive functions. Operations Add, S-Merge and Update are performed in parallel by the \( I_n \) threads. Input tuples do not need to be sorted, they are immediately processed to update the windows they contribute to. The required synchronization is managed by the getNextWinSet() method provided by the W-Hive, which ensures that output tuples for a given winset are outputted only after all its contributing tuples have been processed.

### 4. AGGREGATE AND DATA STRUCTURE IMPLEMENTATIONS

In this section we discuss in detail the aggregate implementations and their supporting data structures. The pseudocode for the enhanced aggregate implementations can be found in Algorithms 2, 3 and 6. Methods’ names have been chosen according to which of the four main stages specified in Section 2 they implement.

#### 4.1 Common components

The base component of the aggregate operator is the Window. It represents a time interval and provides functionality to aggregate the tuples that contribute to it. Algorithm 1 shows the window interface and the implementation of a sample sum aggregation.

As discussed in Section 3, the winset can be implemented as a hash table to easily support an arbitrary number of windows and to locate them quickly given the tuple’s group-by parameter \( K \) (hereinafter referred to as the tuple’s key). In the WiML and TuMLSC implementations, the winset needs to be accessed by multiple threads. In these cases we use the lock-free concurrent hash table implementation provided by the W-Hive [12]. For the MQ, MQCL and TuMLSC implementations, since the access to the winset is sequential, a sequential implementation of a hash table is sufficient.

#### 4.2 Baseline implementations - MQ and MQCL

These are baseline implementations based on the ones used in SPES systems such as Borealis [3] or StreamCloud [8]. The multi-queue design consists of two main methods (see Algorithm 2). The Add method is used to deliver tuples to the aggregate and placing them in their respective input queue (L12). The queues are protected by a lock to allow concurrent access. The MQCL variation lowers the synchronization overhead by instead relying on the lock-free queue method by Michael and Scott [14].

The main work is performed by the second method, S Merge Output. It checks all the queues to make sure a tuple has been received from each input (L15). It then reads the tuple with the lowest timestamp among the inputs (L16–17). This guarantees that all tuples will be read in timestamp order. The currently active winsets are stored in a linked list. The method getTargetWindowTSs creates a list of windowTSs, of the starting timestamps of the windows that the tuple contributes to. If the starting timestamp of a winset in the window list is lower than the earliest timestamp in windowTSs, the aggregated results

---

**Figure 2:** Overview of aggregate designs.

**Table 1:** Methods supported by the data structures.

<table>
<thead>
<tr>
<th>T-Gate</th>
<th>W-Hive</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>insertTuple(tuple, input)</code></td>
<td>Inserts a tuple from the input stream in sorted order.</td>
</tr>
<tr>
<td><code>getNextReadyTuple()</code></td>
<td>Returns the earliest ready tuple (cf. definition 1). Each tuple is returned only once.</td>
</tr>
<tr>
<td><code>updateKWindows(tupl, Key)</code></td>
<td>Updates (and creates, if needed) the windows that the tuple contributes to.</td>
</tr>
<tr>
<td><code>getNextWinSet()</code></td>
<td>Returns the earliest winset to which no new tuples can contribute anymore. Each winset is returned only once.</td>
</tr>
</tbody>
</table>

**Algorithm 1:** Generic window interface and concrete implementation of a window that sums the value of attribute \( A_i \).

```java
interface Window {
    void processTuple(tupl) // update variables
    Tuple produceOutputTuple() // produce output tuple
}

class SumWindow : Window {
    int sum = 0

    void processTuple(tupl) { // update variables
        sum += tupl.A_i;
    }

    Tuple produceOutputTuple() { // produce output tuple
        return Tuple(sum);
    }
}
```
Algorithm 2: MQ and MQ$_{LF}$ – See also Fig. 2a

```java
11 Add(tuple, input) // One thread per input
12 queueinput.enqueue(tuple)
13
14 SMergeUpdateOutput() // One thread
15 if(SL : queue.input.isEmpty()) return
16 input = v: [queue, peek.ts <= queue.peek().ts]
17 tuple = queueinput.dequeue()
18 upout(tuple)
19
20 upout(tuple) // Common method for MQ and TuML$_{SC}$
21 windowTSs = getTargetWindowTSs(tuple)
22 while(windowlist.first().ts < windowTSs.first())
23     winset = windowlist.removeFirst()
24     for (window : winset)
25         if(winset == null) return
26         win.set = windowlist.removeFirst()
27         for (wts : windowTSs)
28             winlist.find(wts).put(tuple.key, win)
29         win = new Window()
30         winupout(tuple) // See L3
31
32 UpdateOutput() // TuML$_{MC}$ only
33     tuple = tgate.getNextReadyTuple() // See L59
34     upout(tuple) // See L20
35
36 Update() // Multiple threads - TuML$_{MC}$ only
37     tuple = tgate.getNextReadyTuple() // See L59
38     if (tuple == null) return
39     if (!tuple.hashToThread(threadid)) return
40     while (updateWindows(tuple)) // See L102
41         Output() // One thread - TuML$_{MC}$ only
42         winset = whive.getNextWinSet() // See L94
43         if (winset == null) return
44         forward(window, produceOutTuple()) // See L3
```

Algorithm 3: TuML$_{SC}$ and TuML$_{MC}$ – See also Fig. 2bc

```java
34 AddSMerge(tuple, input) // One thread per input
35 tgate.insertTuple(tuple, input) // See L66
36
37 UpdateOutput() // TuML$_{MC}$ only
38     tuple = tgate.getNextReadyTuple() // See L59
39     upout(tuple) // See L20
40
41 Update() // Multiple threads - TuML$_{MC}$ only
42     tuple = tgate.getNextReadyTuple() // See L59
43     if (tuple == null) return
44     if (!tuple.hashToThread(threadid)) return
45     while (updateWindows(tuple)) // See L102
46         Output() // One thread - TuML$_{MC}$ only
47         winset = whive.getNextWinSet() // See L94
48         if (winset == null) return
49         forward(window, produceOutTuple()) // See L3
```

of the former can be outpassed (L22-25). This is safe since all future tuples will have an equal or higher timestamp and will not contribute to the winset. If the new tuple contributes to a time interval that does not have a corresponding winset yet, the winset is created and added to the list (L27). If the window does not exist for the tuple’s key, it is also created (L30). Finally, the window processes the tuple (L33).

4.3 TuML$_{SC}$ and TuML$_{MC}$

These aggregate designs make extensive use of the T-Gate data structure described in the next section (API in Table 1). The T-Gate is used to pre-sort all arriving tuples and merge them into one stream. In contrast with the MQ and MQ$_{LF}$ implementations, the S-Merge operation is now executed at the first stage in the pipeline.

TuML$_{SC}$ uses a single thread to read the sorted tuples from the T-Gate, update the windows, and output the aggregated results. This is done using the UpdateOutput method, which shares much functionality with the MQ designs (L37). TuML$_{MC}$ allows multiple threads to read from the T-Gate and update the windows in parallel. This requires support for concurrent handling of the winsets. The W-Hive (API in Table 1 and further description in Section 4.5) is used to provide lock-free winset management. If the

Algorithm 4: T-Gate

```java
52 Node head, update[maxlevels] // Thread local variables
53 def Node
54     Node next[maxlevels]
55     Tuple tuple
56     int input
57
58 getNextReadyTuple()
59     next = head.next
60     if([next!=head || input #next.tuple]
61         head = next
62         return next.tuple
63     return null
64
65 insertTuple(tuple, input):
66     levels = getLevelSLDist()
67     newnode = new Node(tuple, input)
68     curnode = update[maxlevel-1]
69     for(i=maxlevels-1 downto 0)
70         next = curnode.next,
71         while(next != tail A next.tuple.ts)
72         curnode = next
73         next = curnode.next,
74         update = curnode
75         for(i=0 to levels)
76             levelinsert(update, newnode, tuple.ts, i)
77         writteninput = newnode
78         levelinsert(fromnode, newnode, ts, level)
79         while(true)
80             next = fromnode.nextlevel
81             if(next!=tail V next.ts<ts)
82                 newnode.nextlevel = next
83                 if(CAS[fromnodelevel, newnode] break
84                 else fromnode = next
```

aggregate operator’s function is order-sensitive, tuples contributing to the same window cannot be processed in parallel by multiple threads. Hence, a hash function based on the group-by attribute is used to assign input tuples to existing threads.

4.4 T-Gate

The T-Gate data structure (see Algorithm 4) maintains a merged, timestamp-ordered list of the tuples coming from the input streams.

The insertTuple method inserts a tuple at its correct position in the list, given its timestamp. Each thread has its own view of the head of the list, as all new tuples added by the same thread will be after the last one inserted by that thread. This lowers the number of nodes that a thread has to examine. When a new tuple is added, a decision is first made on how many levels the node holding the tuple should be part of. This is done in the getLevelSLDist method using the standard skip list distribution (L67). The insertion gets completed in all the needed levels (L70-75).

The node is then inserted on each level it should be part of with the use of the levelinsert helper method (L77). The helper method first checks if the node stored in the update array is still the prior node. If not, it traverses the list until it finds the right node. The next field of the prior node is then changed to point to the new node with the help of the atomic compare and swap (CAS) operation. If it fails, it means another node was inserted at the same time by another thread. In this case we need to search for the prior node again and repeat the operation.

When the node has been inserted, the written array is updated to hold a reference to the new node (L78). The index into the array is the input stream id. This is done to make sure a tuple is not read until we have received a new tuple with a higher timestamp from all the other input streams.

```java
85             writteninput = newnode
86             levelinsert(fromnode, newnode, ts, level)
87             while(true)
88                 next = fromnode.nextlevel
89                 if(next!=tail V next.ts<ts)
90                     newnode.nextlevel = next
91                     if(CAS[fromnodelevel, newnode] break
92                     else fromnode = next
```
The getNextReadyTuple method traverses the lowest level of the list to return tuples in timestamp order. A tuple can be returned if it was not the last one added by any input stream, i.e., only if it is ready. Nodes can be freed when they are no longer accessible from the nodes that are referred by head and each thread’s update[maxlevels−1]. For this reason, several memory reclamation techniques such as hazard pointers can apply [20], while also garbage collection can be exploited.

4.5 W-Hive

The W-Hive (cf. Algorithm 5) provides lock-free management of winsets. The updateWindows method adds a tuple to each window it contributes to. A reference to the earliest such window is saved in the written array for each thread (L104). This is used to keep track of when winsets are no longer being updated (L96). For each window the tuple contributes to, the method traverses the list to locate the winset with the same timestamp as the window. This is done in the same manner as when inserting a node into the T-Gate (L107–112). If the winset is found, it is searched to find the correct window for the tuple’s key (L121). If there is no window for the key, a new window is inserted into the winset with the correct key (L124). The tuple is then added to the window. If no winset is found for the timestamp, a new winset and corresponding node to hold it are created. They are inserted into the list in a similar manner to the T-Gate. The difference is that another thread might try to create a winset for the same timestamp concurrently. If this happens and the other thread manages to insert it, then the insertion must be canceled and the other winset will be used instead (L130). The getNextWinSet operation returns the next winset that is no longer being updated by input tuples. It is assumed that it will only be called by a single thread. If no thread updated any of the windows in the first winset of the list the last time it received a tuple, it can be assumed that no more tuples will contribute to the winset in the future, as each thread receives tuples in timestamp order (L96). If the new head node for the getNextWinSet operation is part of all shortcut levels, it is made the new head node for the updateWindows method. Nodes and winsets with a timestamp lower than the ones referenced by insertHead and readhead can be safely freed or automatically garbage collected.

4.6 WiML

The WiML design (see Algorithm 6) is suitable only for aggregate operator’s functions \( F \) that are order-insensitive, since it does not sort the tuples prior to inserting them into their windows. When a tuple arrives it is immediately processed to update the windows it contributes to. This is done in the AddSMergeUpdate method using the W-Hive (L137). The W-Hive returns the winsets that will no longer be contributed to, which can then be forwarded (L96).

5. CORRECTNESS

In this section we sketch proofs of liveness and safety properties of the designs, namely lock-freedom and linearizability.

Theorem 1. The TuMLSC, TuMLMC and WiML aggregates and their supporting data structures (T-Gate and W-Hive) are lock-free and linearizable.

Proof sketch. All loops are bounded by a constant or by the size of a list. A list can only grow if at least one thread is making progress, which makes list traversal in accordance with the lock-free property. The only exception is the loop in the levelInsert methods. It requires a successful CAS to exit its loop. But the CAS can only fail if another thread made progress and managed to insert its node. For WiML, threads can insert tuples with the same key concurrently. In this case a lock-free hash table is used [12]. Aggregate functions can aggregate the tuples in a lock-free and linearizable way. As all loops are bounded, or are in accordance with the lock-free property, and all subcomponents are lock-free, the designs are lock-free.

TuMLSC and TuMLMC. All tuples from the same input stream are guaranteed to arrive sorted in timestamp order. When they are added to the T-Gate they cannot be read until a tuple with a higher
timestamp has been added from every input stream. When the tuples are read they have been given their predetermined position based on their timestamp. AddMerge operations will not appear to have taken effect until they are read, and therefore they can be linearized in any order relative to each other. The Update/UpdateOutput methods can only read a tuple if it is not the last one added by any of the inputs. A successful read must therefore be linearized after the written table in the T-Gate has been updated to fulfill this condition. Calls to the Update method can be linearized in any order relative to each other, as concurrent calls are never interested in the same tuples. The Output method is only called by a single thread.

**WiML.** Calls to the AddMergeUpdate method can be linearized in any order. The aggregate operator’s functions used by the windows are not order-sensitive and a window resulting output tuple will not be created until it is guaranteed that all tuples contributing to the window have been processed. The Output method is only called by a single thread. A successful call must be linearized after all threads have signaled, by writing to the written table, that they intend to contribute to a window with a higher timestamp than the first one in the W-Hive.

There is thus a valid linearization order for all methods of the aggregate operators, which makes the designs linearizable.

### 6. EVALUATION

In this section, we study the performance of the different aggregate operators presented in Section 3 in terms of throughput and latency. We discuss the improvement enabled by TuML and WiML, compared to MQ and MQ_{LF}, for different queries and varying number of inputs. Subsequently, we focus on the high scalability enabled by TuML_{MC}. Our experiments take into account the aggregate’s features that affect throughput and latency: the overall number of keys, the number of windows to which each tuple contributes and the cost of the aggregate function. For each feature, we consider two stretching points in order to show how traversing its spectrum (e.g., increasing the overall number of keys) affects the overall performance. As mentioned in Section 3.1, the evaluation of MQ_{LF} is included for the sole purpose of showing that MQ’s poor performance does not depend only on the use of locks. All the experiments represent queries that can be found in real-world applications. We take into account aggregate functions that are commonly used and also evaluate highly costly variants when studying how their cost affects the aggregate performance. Our data sets have been collected from real-world applications.

#### 6.1 Evaluation setup

The evaluation has been conducted with an Intel based workstation with two sockets of 6-core Xeon E5645 (Nehalem) processors with Hyper Threading (24 logical cores in total) and 48 GB DDR3 memory at 1366 MHz. The prototype has been implemented in Java and experiments have been run using the OpenJDK Runtime Environment (IcedTea 2.3.9).

We use two datasets that we refer to as SoundCloud (SC) and Energy Consumption (EC). SC has been collected from the online audio distribution platform SoundCloud from a subset of approximately 40,000 users exchanging comments about 250,000 songs between 2007 and 2013. Tuples contain comments sent by users in relation to songs and are composed by the attributes \{ts, user, song, cmt\}. EC contains energy consumption readings collected from a set of 243 smart meters between May 2012 and June 2013. Tuples are composed by attributes \{ts, meter, cons\}.

All experiments start (resp. end) with a warm-up (resp. cool-down) phase, presented results are averaged over 10 runs. In all experiments, we process data from the EC and SC datasets, modifying only the rate at which tuples are injected. Table 2 presents the parameters of the queries used in the evaluation: identifier ID, dataset DS, window size WS, window advance WA, group-by parameter K and aggregate function F. We measure throughput as the tuples/second (t/s) processed by an aggregate operator and latency as the timestamp difference between an output tuple and the latest input tuple that produced it. The T-Gate and W-Hive use a maximum of 3 levels in all experiments.

#### 6.2 Baseline and new designs comparison

In this set of experiments, we measure the maximum throughput and the minimum latency of the MQ, MQ_{LF}, TuML_{SC} and WiML implementations, quantifying the improvement enabled by the use of concurrent data structures while using the same number of input and output threads.

**Parallelization benefit.** We first focus on the average duration of the main operations performed by \(I_t\) and \(O_t\) threads for queries K-OS and K-OI and 20 input streams. Results for the query K-OS are presented in Fig. 3a (we use logarithmic scale to better appreciate the different orders of magnitude). Both \(I_t\)’s and \(O_t\)’s operations are faster for MQ_{LF} compared to MQ since the former relies on lock-free queues. \(I_t\)’s duration increases while \(O_t\)’s decreases for TuML_{SC} since the S-Merge operation is performed by \(I_t\). \(O_t\) threads, which constitute the bottleneck, will reach 100,000, 130,000 and 160,000 t/s for MQ, MQ_{LF} and TuML_{SC}, respectively.

Figure 3b compares the latency evolution of the different implementations for an increasing input rate. In all experiments, the latency initially decreases with the increasing rate (lower inter-arrival times at the inputs result in shorter queuing times for input tuples) while it explodes upon saturation of the operator (that is, when the injected load exceeds its maximum throughput). The throughput achieved by each implementation is close to the expected one (from Fig. 3a). MQ_{LF} and TuML_{SC} achieve a throughput 1.3 and 1.6 times higher than MQ, respectively. Average durations for query K-OI are shown in Fig. 3c. It can be noticed that \(I_t\)’s duration increase is greater for WiML than TuML_{SC} since both
Table 2: Parameters for queries used in the evaluation. Identifiers ID are composed by 3 letters. The first letter (small or capital) represents the aggregate parameter being studied (k - overall number of keys, w - window size, f - applied function). The last two letters specify whether F is order-sensitive (OS) or order-insensitive (OI).

<table>
<thead>
<tr>
<th>ID</th>
<th>DS</th>
<th>WS</th>
<th>WA</th>
<th>A</th>
<th>F</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-OS</td>
<td>EC</td>
<td>30</td>
<td>3</td>
<td>meter</td>
<td>first()</td>
<td>Forward the first consumption reading, group by meter (243 distinct keys)</td>
</tr>
<tr>
<td>w-OS</td>
<td>EC</td>
<td>20</td>
<td>2</td>
<td>meter</td>
<td>first()</td>
<td>Forward the first consumption reading, group by meter (each tuple contributes to 20 windows)</td>
</tr>
<tr>
<td>W-OS</td>
<td>EC</td>
<td>40</td>
<td>2</td>
<td>meter</td>
<td>first()</td>
<td>Forward the first consumption reading, group by meter (each tuple contributes to 20 windows)</td>
</tr>
<tr>
<td>f-OS</td>
<td>SC</td>
<td>20</td>
<td>2</td>
<td>song</td>
<td>first-mail/IP(cmt)</td>
<td>Forward the first comment containing a mail address, group by song</td>
</tr>
<tr>
<td>F-OS</td>
<td>SC</td>
<td>20</td>
<td>2</td>
<td>song</td>
<td>first-mail/IP(cmt)</td>
<td>Forward the first comment containing a mail or an IP address, group by song</td>
</tr>
</tbody>
</table>

| Order-sensitive (OS) functions (MQ, MQ,F, TuMLSC) |

**Order-insensitive (OI) functions (MQ, MQ,F, WiML)**

<table>
<thead>
<tr>
<th>ID</th>
<th>DS</th>
<th>WS</th>
<th>WA</th>
<th>A</th>
<th>F</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>k-OI</td>
<td>EC</td>
<td>30</td>
<td>3</td>
<td>meter</td>
<td>count()</td>
<td>Count the number of consumption readings, group by meter (243 distinct keys)</td>
</tr>
<tr>
<td>K-OI</td>
<td>SC</td>
<td>30</td>
<td>3</td>
<td>song</td>
<td>count()</td>
<td>Count the number of comments, group by song (40,000 distinct keys)</td>
</tr>
<tr>
<td>w-OI</td>
<td>EC</td>
<td>20</td>
<td>2</td>
<td>meter</td>
<td>avg(coms)</td>
<td>Compute the average consumption, group by meter (each tuple contributes to 20 windows)</td>
</tr>
<tr>
<td>W-OI</td>
<td>EC</td>
<td>40</td>
<td>2</td>
<td>meter</td>
<td>avg(coms)</td>
<td>Compute the average consumption, group by meter (each tuple contributes to 20 windows)</td>
</tr>
<tr>
<td>f-OI</td>
<td>SC</td>
<td>20</td>
<td>2</td>
<td>song</td>
<td>count-mail(cmt)</td>
<td>Count the number of comments containing a mail address, group by song</td>
</tr>
<tr>
<td>F-OI</td>
<td>SC</td>
<td>20</td>
<td>2</td>
<td>song</td>
<td>count-mail/IP(cmt)</td>
<td>Count the number of comments containing a mail or an IP address, group by song</td>
</tr>
</tbody>
</table>

S-Merge and Update operations are performed by the I thread. With WiML, each I thread is able to process 22,000 t/s in parallel if less than 2 million t/s are produced by O1. As shown in Fig. 3d, WiML achieves a higher throughput and a lower latency compared to MQ,F and MQ (approximately 400,000 t/s), independently of the input rate (tuples are processed independently by each I thread).

In the remaining of this section, we study the performance of TuMLSC and WiML with respect to MQ and MQ,F for the different queries presented in Table 2. In all experiments, we experience the same performance behavior for each distinct implementation. As the number of inputs increases, the latency of multi-queue implementations (MQ and MQ,F) increases while their throughput decreases linearly. This pattern is broken by the TuMLSC, whose throughput is rather stable as the number of inputs increases. The pattern is even reversed by the WiML, whose throughput actually increases as the number of inputs increases (due to the enhanced parallelism of the underlying W-Hive) while achieving the lowest and almost constant latency.

**Varying number of keys.** The overall number of keys in the data affects both the operator’s throughput and latency since the higher the number of keys, the higher the number of tuples produced for all windows starting at the same timestamp. Results highlight that TuMLSC and WiML perform better than both MQ and MQ,F, whose throughput decreases linearly with the increasing number of inputs.

With respect to order sensitive functions, we compare queries k-OS (Fig. 4a) and K-OS (Fig. 4b). The query k-OS uses the EC dataset (243 distinct keys) while the query K-OS uses the SC dataset (40,000 distinct keys). The upper part of each figure presents the throughput while the bottom part presents the latency (in logarithmic scale). For 20 inputs, TuMLSC provides the highest throughput (2.9 times better than MQ’s for query k-OS) and the lowest latency. With respect to order insensitive functions, we compare queries k-OI (Fig. 4c) and K-OI (Fig. 4d). As for order sensitive functions, both MQ’s and MQ,F’s throughput decreases for increasing number of inputs. On the other hand, WiML throughput increases accordingly to the number of inputs, achieving a maximum throughput of 2.6 million t/s and 430,000 t/s, respectively. MQ’s and MQ,F’s latencies increase with the number of inputs while WiML’s one remains approximately constant.

**Varying number of windows to which tuples contribute.** The rationale for this experiment is that the higher the number of windows to which each input tuple contributes, the higher the duration of the Update operation. Also in this case, our enhanced implementations outperform both MQ and MQ,F. An increasing number of windows to which tuples contribute results in an overall throughput breakdown and latency increase.

We first focus on order sensitive functions with queries w-OS (Fig. 5a) and W-OS (Figure 5b). For query w-OS, each tuple contributes to 10 windows. For query W-OS, each tuple contributes to 20 windows. With respect to order insensitive functions, we compare queries w-OI (Fig. 5c) and W-OI (Fig. 5d). Also in this case, WiML outperforms MQ and MQ,F. For query w-OI, WiML’s maximum throughput is of approximately 2.4 and 1.4 million t/s, 19 and 16 times better than MQ, respectively.

**Varying function cost.** In this set of experiments, we study how throughput and latency evolve with respect to different function costs. We expect the throughput to decrease and the latency to increase accordingly to the increasing cost of the aggregation function. As observed before, TuMLSC performs better than multi-queue implementations, although this improvement becomes smaller when running very expensive aggregate functions. WiML outperforms multi-queue implementations both in terms of throughput and latency independently of the aggregate function cost.
Figure 5: Varying number of windows to which tuples contribute - evaluation.

With respect to order sensitive functions, we compare queries f-OS (Fig. 6a) and F-OS (Fig. 6b). When increasing the function cost (query F-OS), TuML\textsubscript{MC}’s throughput and latency become really close to MQ’s and MQ\textsubscript{L,F}’s ones. Throughput and latency evolution for order insensitive functions are evaluated for queries f-OI (6c) and F-OI (6d). For both experiments, WiML achieves a throughput of approximately 615/100 t/s, 9 times better than MQ.

6.3 TuML\textsubscript{MC} scalability evaluation

In this section, we focus on the scalability of the TuML\textsubscript{MC} implementation. We execute all the previous queries for order sensitive functions using the TuML\textsubscript{MC} implementation for an increasing number of threads (up to 12, the physical number of cores of the machine used in the evaluation). For each query, we present how the throughput and the latency evolve when considering 5 and 20 inputs streams.

Figures 7a and 7b present the throughput and latency evolution for queries k-OS and K-OS. It can be noticed that K-OS scales better than k-OS for an increasing number of threads due to the increase of O\textsubscript{t} operations’ duration caused by the higher number of keys. Figures 7c and 7d present the throughput and latency evolution for queries w-OI and W-OI. In this experiment, throughput and latency behave similarly despite the increased duration of O\textsubscript{t} operations. This is because the increased O\textsubscript{t} operations duration is actually caused by an higher number of windows updated by each tuple, resulting in an higher contention in the underlying W-Hive. Finally, Figures 7e and 7f present the throughput and latency evolution for queries f-OIS and F-OIS.

Summary of results. Comparing the implementations that rely on one I\textsubscript{t} thread per input and a single output thread O\textsubscript{t}, both TuML\textsubscript{MC} and WiML perform better than MQ and MQ\textsubscript{L,F}, enabling to cope with streams of higher speed. The improvement enabled by TuML\textsubscript{MC} is more sensitive to the aggregate parameters than WiML, which clearly outperforms MQ and MQ\textsubscript{L,F}. When increasing the number of processing threads, TuML\textsubscript{MC}’s performance increases both in terms of throughput and latency. Moreover, its scaling does not degrade when increasing the number of threads above the number for which the highest rate is achieved.

7. RELATED WORK

Parallel execution of data streaming operators has been addressed mainly by means of partitioned parallelism [8, 4], where multiple instances of an operator are assigned to distinct partitions of a given stream. The way tuples are routed to instances (round-robin, hash-based [8] or pane-based [4] routing is used) depends on the operator’s semantic. It should be noticed that partitioned parallelism is orthogonal to our parallelization technique since we focus on the performance improvement of individual instances of an operator. The work presented in [15] discusses a multi-threaded streaming protocol that adjusts the number of processing threads depending on the system load. Similarly to our TuML\textsubscript{MC} implementation, the protocol defines a single work queue from which multiple worker threads consume tuples. Nevertheless, that does not take into account sorting of input tuples, which is one of the key challenges in our protocol. Moreover, the authors do not discuss improvements enabled by concurrent data structures in the multi-threaded environment. With respect to parallel data streaming in the context of multi-core CPUs and GPUs, the authors of [16] present a parallel implementation of the aggregate operator and study how it performs on distinct parallel architectures. The aggregate model discussed by the authors differs from ours since windows are tuple-based and the overall number of distinct group-by values is known before-hand and does not vary over time. Moreover, no discussion is provided about deterministic processing in the context of multiple input sources. Parallel processing in multi-core CPUs and GPUs is also discussed in [5] but, differently from us, the authors focus on pattern detection rather than data aggregation and rely on automata-based incremental processing (AIP).

As discussed in section 3, one of the challenges in providing deterministic processing is the merging of multiple timestamp sorted input streams. This has been discussed in the context of parallel-distributed SPEs [18, 8] and replica-based fault tolerance protocols for data streaming [3]. All the existing approaches rely on separated input queues (similarly to the M\textsubscript{Q} protocol). As shown in our evaluation, this merging is not efficient and implementations such as TuML\textsubscript{MC} can drastically improve the overall operator’s performance.
Figure 7: TuML\textsubscript{MC} scaling for increasing number of threads.

8. CONCLUSIONS AND FUTURE WORK

Providing the appropriate data structures that best fit the needs of a concurrent application is a key research issue [13]. In this paper, we show how important this is in the context of streaming aggregation. We propose new data structures for managing tuples and windows (T-Gate and W-Hive). Their operations and their lock-free implementations enable better interleaving and hence improve the balancing and the parallelism of the aggregate operator’s processing stages. As shown in the extensive evaluation based on real-world datasets, our enhanced aggregate implementations outperform existing ones both in terms of throughput and latency, and are able to handle heavier streams, increasing the processing capacity up to one order of magnitude. Another fundamental class of data streaming operators we plan to investigate in the future are the join aggregation. We propose new data structures for managing tuples and list-based sets. These structures can improve their performance by reducing their sequential number of comparisons.

9. REFERENCES


APPENDIX

A. SKIP LIST AND T-Gate COMPARISON

In this experiment, we evaluate the performance of a lock-free skip list and the T-Gate, measuring the maximum throughput with which EC tuples coming from multiple input streams can be sorted. For this comparison we used the ConcurrentSkipListMap from Java’s java.util.concurrent package, an implementation based also on [20].

Results for 5, 10, 15 and 20 input streams are presented in Fig. 8. It can be noted that, when using a skip list, checking for ready tuples (done explicitly since the skip list does not differentiates between tuples that are ready or not) results in a cost linear to the number of input streams. Thus, the skip list’s throughput degrades while T-Gate throughput grows. For 20 input streams, the skip list is able to sort approximately 1.5 million t/s while the T-Gate reaches approximately 2.2 million t/s (1.5 times better). The T-Gate also achieves a lower sorting latency, approximately 1 ms for 20 input streams against the 1.6 ms latency of the skip list.

![Figure 8: T-Gate and Skip List comparison.](image-url)