Overcoming Memory Limitations in High-Throughput Event-Based Applications

Marcelo R. N. Mendes[†]
CISUC, University of Coimbra
Dep. Eng. Informática – Pólo II
Coimbra, Portugal
mnunes@dei.uc.pt

Pedro Bizarro
CISUC, University of Coimbra
Dep. Eng. Informática – Pólo II
Coimbra, Portugal
bizarro@dei.uc.pt

Paulo Marques
CISUC, University of Coimbra
Dep. Eng. Informática – Pólo II
Coimbra, Portugal
pmarques@dei.uc.pt

ABSTRACT

The last decade has witnessed the emergence of business critical applications processing streaming data for domains as diverse as credit card fraud detection, real-time recommendation systems, call-center monitoring, ad selection, network monitoring, and more. Most of those applications need to compute hundreds or thousands of metrics continuously while coping with very high event input rates. As a consequence, large amounts of state (i.e., moving windows) need to be maintained, very often exceeding the available memory resources. Nonetheless, current event processing platforms have little or no memory management capabilities, hanging or simply crashing when memory is exhausted. In this paper we report our experience in using secondary storage for solving the performance problems of memory-constrained event processing applications. For that, we propose *SlideM*, a novel buffer management algorithm that exploits the access pattern of sliding windows in order to efficiently handle memory shortages. The proposed algorithm was implemented in a real stream processing engine and validated through an extensive experimental performance evaluation. Results corroborate the efficacy of the approach: the system was able to sustain very high input rates (up to 300,000 events per second) for very large windows (about 30GB) while consuming small amounts of main memory (few kilobytes).

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design studies

Keywords

Disk, Event Stream Processing, Memory Management, Query Plan Sharing, Sliding Window.

1. INTRODUCTION

Event stream processing has recently passed from a pure academic subject to a well-established field in industry, with many of the original research projects [1][7][17] turning into fully-functional products (e.g., [9][18][23]). As the technology matures, these *stream processing engines* (SPEs) start to gain increased popularity among real-world users, finding application in the most diverse domains such as financial, telecom and sensor networks [12]. Many of these applications involve the computation of *aggregates over sliding windows*, which allow users to better measure the quality-level of their businesses and systems in real-time. For example, consider a call-center monitoring application where information about customers' calls is constantly analyzed by a stream processing engine. A typical query executing at the SPE in such scenario is:

Query 1: "*Report, every second, the average time during which customers are waiting for service, across the last hour.*"

```
SELECT AVG(waitTime)
FROM   calls [RANGE 1 HOUR SLIDE 1 SECOND]
```

The query above, expressed using the CQL language [2] syntax, specifies an AVG aggregation query over a sliding window with two parameters: RANGE, which defines the span of the window (*i.e., for how long tuples of the event stream "calls" are considered in query answer computation*); and SLIDE, which defines when tuples are expired out of the window and controls the frequency in which updated results are produced.

For a number of reasons (e.g., stringent latency requirements, transient data items, etc.), SPEs use main memory for processing their continuous queries. Unfortunately, many queries have an unbounded space cost, which cannot be determined a priori. For instance, the memory consumption of Query 1 typically depends on the event arrival rate, which might vary significantly during query execution. In those circumstances, one would expect SPEs to be prepared to deal with memory shortages. Interestingly, previous work [16] has shown that many commercial SPEs deal badly with this situation – some suffer from thrashing due to OS paging or excessive garbage collection activity while others simply crash with out-of-memory errors.

In this paper we report our experience in addressing this problem for analytic workloads composed by a large number of continuous aggregation queries. We introduce *SlideM*, a buffer management algorithm that selectively offloads sliding windows state to secondary storage when main memory becomes insufficient. Our approach is based on the observation that for most aggregation queries the memory consumption and access pattern is dictated by the sliding window, and that this access pattern can be exploited to achieve excellent performance while using small amounts of main memory.

This work was motivated by a series of real event processing applications we have been working with recently. Most of them

[†] Work partially carried out while at industrial partner FeedZai.

ICPE'13, April 21–24, 2013, Prague, Czech Republic.

involve the computation of several metrics (aggregates) over very large sliding windows, under stringent memory requirements. Throughout the rest of the paper, we use one of such use-cases, the call-center monitoring application introduced earlier and described in detail in Appendix A, to illustrate the problem. The requirements of this application include the computation of about 150 KPIs over 24-hour sliding windows, under an input rate of around 1,000 events per second, running on a machine with 2 gigabytes of main memory. Note that, given the moderate input rate, relative simplicity of the KPIs being computed (SUM, COUNT and AVG aggregates) and the large size of the window, the application tends to be memory-bound rather than CPU-bound. A simplistic calculation gives an idea on the dimension of the problem: considering that the average event size is 94 bytes, and that all tuples need to be maintained until they are expired out of the window, a single aggregation query will require at least: 1000 x 24 x 3600 x 94 bytes = 7.6 gigabytes (*we show in Section 2 that the commonly used technique of pre-aggregating data instead of storing tuples may require even more space*). Our goal is to allow such memory-intensive applications to run smoothly on an SPE whether they fit on available memory or not. In summary, we make the following contributions:

1. We analyze current proposals for executing sliding-window aggregates and show that frequently-used techniques, designed to reduce memory consumption and create opportunity for resource sharing, in many cases do not produce the desired effects (Section 2).
2. We propose an optimal buffer management algorithm to deal with memory shortages during execution of sliding-window aggregation queries. We demonstrate that, contrary to common sense, storing windows data on disk can be appropriate even for applications with very high event arrival rates (Section 3).
3. We build upon the proposed buffer management algorithm and develop a strategy to share computational resources when processing multiple aggregation queries over overlapping sliding windows (Section 4).
4. We implement our proposed techniques in a real SPE [19] and validate their effectiveness through an extensive experimental evaluation (Section 5).

2. SLIDING-WINDOW AGGREGATES

Continuous queries in SPEs are computed over infinite event streams rather than bounded datasets. However, many operations cannot be executed over infinite inputs in bounded memory, and some *blocking* operations require seeing the entire input before producing any result (e.g., join) [11]. Traditionally these two issues have been addressed in SPEs by limiting the amount of data over which operations take place through the use of *sliding windows*. A sliding window is a construct that retains only the most-recently arrived tuples of an event stream. The *size* of a sliding window determines the amount of data to be retained, and can be specified in number of tuples (*count-based* windows) or through an interval (*time-based* windows). Stale tuples are purged when window *slides*, due to arrival of new event or time passing.

A *sliding-window aggregate* (SWA) computes an aggregation function over a sliding window content and produces an updated result every time the window slides. For example, consider our motivating scenario where a stream of statistics continuously generates new information about call-center interactions with its customers. Query 1 defines a sliding window that retains the data items that arrived in the last hour, computes the average "waiting time" from this set of elements, and reports a new result every second. A common variation of this query structure is to have a *grouped aggregation*, where the input stream is logically partitioned into sub-streams, based on a grouping key, and one aggregate is produced for each partition. For instance, a GROUP-BY clause could be added to Query 1 in order to produce the average waiting time per customer region or per employee.

SWA is recognized as a fundamental operation of SPEs and has been extensively studied in previous work [3][4][12][14]. In the rest of this section we discuss how SWAs have been traditionally implemented. We present the two most frequently-used approaches, and compare how well they utilize memory resources in different workload scenarios.

2.1 SWA Implementation

Many important aggregates such as AVG, SUM, COUNT, MIN and MAX can be computed incrementally, in a single-pass over data items. This, in principle, allows an aggregation operator to discard events right after they have been processed. For instance, an AVG aggregate can be computed in O(1) space by simply keeping two variables – *sum* and *count* – and updating them upon event arrivals. However, when the aggregate is applied over a sliding window, tuples are eventually expired and this tuple removal has to be reflected into the query answer – in the AVG example, this means subtracting from the *sum* variable the value of the aggregated field in the expired tuples and decrementing the *count* variable by the number of expired tuples. Therefore, an SWA operator needs to maintain information about events that arrived previously so that the result can be updated properly when they eventually leave the window.

Traditionally, two approaches have been used to keep this information about past events in an SWA. The first, simply keeps all tuples in the window until it is time to expire them [2]. Normally, the sliding window is an operator by itself which forwards incoming and expired tuples to subsequent, aggregate operators (Σ), as depicted in Figure 1. The second approach, first introduced in [14], and adopted in subsequent proposals (e.g., [12]) sub-aggregates the incoming stream using smaller windows and then aggregates these sub-aggregates into a window of the original size in order to produce the final query result (*see Figure 2*). Taking Query 1 as example, SUM and COUNT sub-aggregates are computed over a 1-second window and then aggregated over a 1-hour window, thus producing the final result. The main advantage of this *two-level aggregation* (2LA) scheme over the former *one-window* (1W) approach is that the space cost of the query no longer

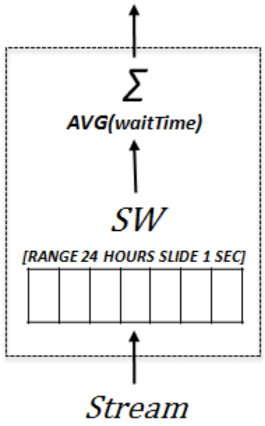

Figure 1: Plan for Query 1 using the single-window (*1W*) scheme.

depends on the input rate. However, as we are going to discuss next, it does not guarantee a bounded space cost, and for some workloads results in increased memory consumption.

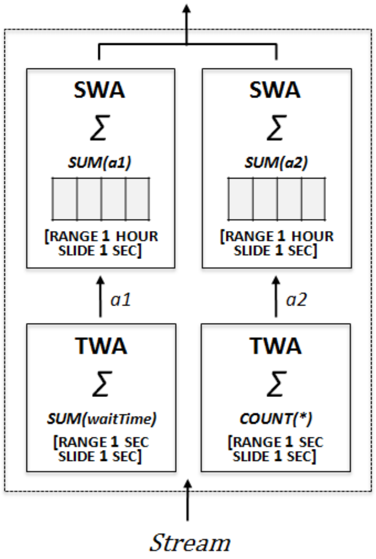

Figure 2: Execution plan for Query 1 using the *2LA* scheme

2.2 Space Cost Analysis

In this section we examine the space cost of the two widely-used SWA implementations. In particular, we demonstrate that either approach can incur in considerable memory costs, eventually bringing event processing applications to run out of memory. We also show that the *2LA* technique, originally designed to reduce space cost of SWAs, might in many cases aggravate the problem.

2.2.1 Two-Level Aggregation (2LA)

The *2LA* technique can be very useful for reducing space and computation cost of periodic sliding-window aggregates, particularly when input rates are high and/or the answer does not need to be updated often. However, there are a couple of issues that limit its effectiveness in many important scenarios. Probably the most relevant of them is that its space cost grows linearly with the number of aggregates being computed. This is particularly critical since most monitoring applications compute not one, but several aggregates – either because different aggregation functions are needed, distinct sets of attributes of the input streams are aggregated, or different grouping criteria are used. For instance, the users of the call-center monitoring application are interested not only in the *average waiting time*, but also in the *total* waiting time, the average *call time*, and the average waiting time *per region*. With the *2LA* scheme, each such aggregate results in a pair of operators, a *tumbling-window aggregate* (TWA) and a subsequent *sliding window aggregate* (SWA), as illustrated in Figure 2.

Another issue is that aggregations with a GROUP-BY clause implemented using the *2LA* scheme have their space and computation cost directly affected by the number of groups. This is because the TWA operator will produce as much aggregates as the number of distinct groups seen during the lifetime of its window. Each of these aggregates consumes space and computation in the subsequent SWA operator. Moreover, since the number of groups seen during the interval of the TWA window cannot be determined a priori, the *2LA* scheme does not guarantee a bounded space cost for grouped sliding-window aggregates. Taking into account the aforementioned factors, the total space cost when computing a set of aggregates using the *2LA* scheme is determined as follows[1].

Consider that *N* sliding-window aggregates, $\Sigma_1,\ldots,\Sigma_N$, are to be computed – *each representing a unique combination (Φ_i, F_i, P_i) of aggregation function (Φ), aggregated fields (F) and grouping criteria (P)* – over a common sliding window with size *W* and update interval *U*. The space cost of each aggregate Σ_i is given by the sum of the costs of its inner *tumbling-window* and *sliding-window* aggregates:

$$Space_{2LA} = \mathrm{N}\cdot\left(Space_{TWA} + Space_{SWA}\right)$$

The TWA operator does not keep tuples in a window and only consumes the space required to maintain an aggregation state *s'* for each of the *g* groups seen during period *U* as shown below:

$$Space_{TWA} = \mathrm{g}\cdot\mathrm{s'}$$

The SWA operator, on the other hand, keeps both the tuples produced by TWA and a per-group aggregation state:

$$Space_{SWA} = \frac{g}{U}\cdot W\cdot \mathrm{t}_{\mathrm{agg}} + G\cdot \mathrm{s}$$

In the formula above, g/U is the rate at which tuples arrive at SWA from TWA, t_{agg} is the size of the tuples produced by TWA, *G* is the total number of groups seen during *W*, and *s* is the size of the aggregation state per-group. The final space cost of the *2LA* scheme for *A* aggregates is then given by the formula below:

$$Space_{2LA} = N\cdot\left[g\cdot\left(\mathrm{s'}+\frac{W\cdot t_{agg}}{U}\right)+G\cdot\mathrm{s}\right] \tag{2.1}$$

Note that depending on the function being computed, the aggregation state sizes *s* and *s'* can be constant or grow with the number of tuples in the corresponding window. As discussed elsewhere ([3] and [14]), *subtractable* aggregates like SUM, COUNT, AVG and VARIANCE can be computed with constant storage, but *distributive* (e.g., MIN and MAX) and *holistic* (e.g., QUANTILE) functions require O(N) space.

2.2.2 To Sub-Aggregate or not to Sub-Aggregate

We now examine the space cost of the *one-window* approach. As it can be seen from Figure 1, the memory consumption for the *1W* scheme corresponds to the sum of the space required to maintain the tuples in the main sliding window, and the state of each aggregate Σ_i. The former is obtained by the product of the window size *W*, the input rate λ, and the tuple size *t*. The latter is given by the product of the number of aggregates N, the total number of groups *G* seen during *W*, and the state size of each aggregate operator *s*. Or algebraically:

$$Space_{\mathrm{1W}} = \lambda\cdot W\cdot t + N\cdot G\cdot s \tag{2.2}$$

We can see from formulas 2.1 and 2.2 that each approach is more sensitive to a given factor than the other. With the *1W*

[1] For the sake of brevity, we limit our discussion to time-based windows, but similar analysis applies to count-based windows.

implementation, the space cost will be substantial for large windows if the input rate is high. On the other hand, the *2LA* scheme is immune to the input rate[2], independently on how large the window is, but can be severely penalized if the workload has many aggregates or groups.

As an example, we compare the memory consumption of the two different approaches using parameters taken from the call-center monitoring use-case. Let W=24 hours, U=10 seconds, λ=1000 events/sec, N=104 aggregates, G=10000 groups, g=1000 groups, s=s'=16 bytes[3], t=94 bytes, and t_{agg}=20bytes. The space cost of each scheme is in this case:

$$Space_{2LA} = 104 \cdot \left[1000 \cdot \left(16 + \frac{86400 \cdot 20}{10} \right) + 10000 \cdot 16 \right] \cong 17GB$$

$$Space_{1W} = 1000 \cdot 86400 \cdot 94 + 144 \cdot 10000 \cdot 16 \cong 8GB$$

As we can see, for this particular use case, performing a *two-level aggregation* in the end results in less efficient usage of memory resources than when computing the aggregates with a single window. More importantly, considering that in this application the available memory is limited to less than 2GB, neither of the two approaches allows the workload to run entirely at RAM. In this situation, it is necessary to selectively spill part of the queries state to disk so that the application does not run out of memory. Note that in either scheme the queries space cost is largely dominated by the state of the sliding window(s). For this reason, we address the problem of insufficient memory resources during computation of SWAs with an algorithm to manage the content of sliding windows.

3. *SlideM*: A BUFFER MANAGEMENT ALGORITHM FOR SLIDING WINDOWS

In this section we introduce *SlideM*, an algorithm for managing the working set of sliding windows. The proposed algorithm exploits the fact that sliding-window operators are most of the time manipulating only a small fraction of their data set and are doing so in a very predictable pattern – once a tuple is stored on the window it is not going to be accessed by the sliding-window operator until it is time to expire it, which may take long (e.g., consider a 6-hour time-based window).

SlideM is employed on a per-operator basis, that is, each window physical operator in the query plan is given a *repository* to hold its tuples. The actual location of tuples (either RAM or disk) is encapsulated by this repository, which internally implements a buffer management strategy based on *SlideM*. The repository includes a buffer pool (BP) for holding the memory-resident part of the window and a handle for accessing tuples at secondary media. Both the buffer pool and the data file at disk are divided in non-spanned blocks with a fixed block factor. These blocks are the unit of transfer between main memory and disks.

The algorithm operates as illustrated in Figure 3: when the buffer pool gets full, it first sends to disk the block containing the most recently arrived tuples because these are the ones that are not going to be needed for the longest time. Similarly, when the oldest block at RAM is expired and hence the BP has space left once more, *SlideM* brings back from disk the least recently written (LRW) block, because it contains the tuples which are going to be needed next among the ones currently at disk. This behavior ensures that memory will always contains the tuples that are going to be needed by the sliding-window operator in the shortest time. The algorithm operation is described in details in Figure 4 and Procedures 1 and 2.

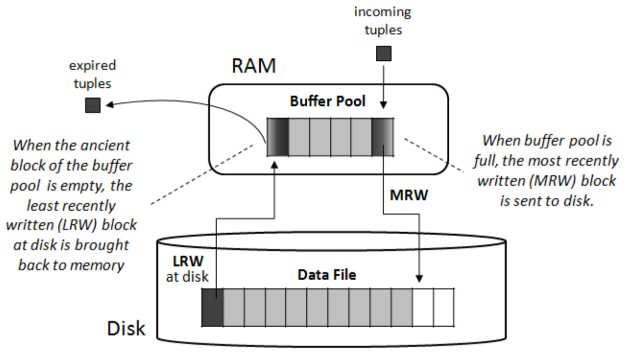
incoming tuples
expired tuples
RAM
Buffer Pool
When the ancient block of the buffer pool is empty, the least recently written (LRW) block at disk is brought back to memory
When buffer pool is full, the most recently written (MRW) block is sent to disk.
MRW
LRW at disk
Data File
Disk

Figure 3: Overview of *SlideM* operation.

Procedure 1 add(Tuple t)

```
let $recent\_block$: block at RAM holding recently arrived tuples

1:   if $recent\_block$ is full then
2:     if buffer pool is full then
3:       if there are free blocks at disk then
4:         $diskPos \leftarrow$ address of least recently released block
5:       else
6:         $diskPos \leftarrow$ end_of_file
7:       end if
8:       WRITETODISK($diskPos$, $recent\_block$);
9:     end if
10:    $recent\_block \leftarrow$ ALLOCATENEWBLOCK();
11:  end if
12:  $recent\_block$.APPENDTUPLE($t$);
```

Procedure 1 describes event arrivals: every time a new tuple needs to be stored in the window, the algorithm checks whether the block at the tail of the window still has space left. If so, it stores the tuple normally at the end of the block; otherwise it allocates a new block at the buffer pool (*as shown in step 2 of Figure 4*). If the buffer pool is full (*3*), the algorithm first spills the most recently written (MRW) block to disk (*4*) to free space for the new block that will be soon allocated. The MRW block is written at a free position on disk (*9*) or at the end of the data file, if there are no free blocks (*5*).

Tuple expiration happens as in Procedure 2: when the repository receives a request to remove a tuple at the beginning of the window it checks whether the oldest block is now empty. If so, the block is removed from the buffer pool (*as shown in step 6 of*

[2] Assuming that the number of groups *g* seen during period U is not affected by the input rate, which typically is not the case.

[3] Only *subtractable* aggregates are computed. We simplify discussion using a single value to represent the average state size of the different aggregation functions in the application.

Figure 4); if there is data at disk, the LRW disk block is brought to the buffer pool, at the position of the just-dismissed block (*7*). Then, the tuple is finally removed from the (newly arrived to memory) ancient block.

Procedure 2 expireOldest()

```
let ancient_block: block at RAM holding soon-to-expire tuples

if ancient_block is empty then
  buffer_Pool.REMOVE(ancient_block);
  if there is data at disk then
    // position at disk of least recently written block.
    lrw ← GETLRWDISKBLOCKADDRESS();
    lrwDiskBlock ← READFROMDISK(lrw);
    buffer_pool.ADD(lrwDiskBlock);
  end if
  ancient_block ← buffer_pool.GETLRWBLOCK();
end if
ancient_block.REMOVEOLDESTTUPLE();
```

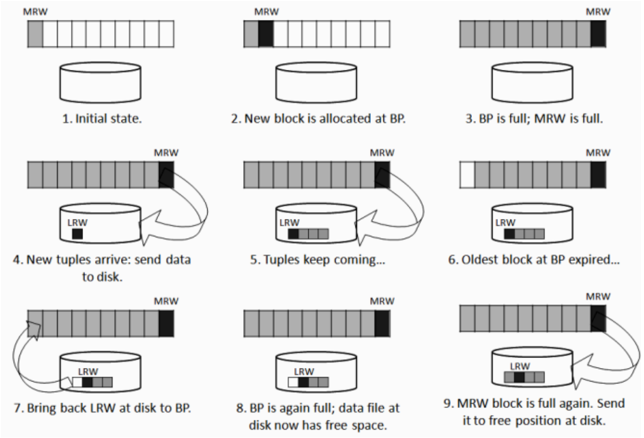
MRW
1. Initial state.
MRW
2. New block is allocated at BP.
MRW
3. BP is full; MRW is full.
MRW
LRW
4. New tuples arrive: send data to disk.
MRW
LRW
5. Tuples keep coming...
MRW
LRW
6. Oldest block at BP expired...
MRW
LRW
7. Bring back LRW at disk to BP.
MRW
LRW
8. BP is again full; data file at disk now has free space.
MRW
LRW
9. MRW block is full again. Send it to free position at disk.

Figure 4: *SlideM* algorithm in several phases of its execution.

It should be clear that *SlideM* operation results in a very small memory consumption. In its strict sense, the algorithm needs only the equivalent to two blocks, one for holding the oldest part of the window (*ancient block*) and another for accommodating the newly arriving events (*recent block*). In fact, the performance of *SlideM* should not be affected by buffer pool size, unless BP becomes large enough to hold the entire working set of the window – if the window does not fit at RAM the algorithm will necessarily be swapping data to/from disk. Additionally, *SlideM* is optimal in terms of the amount of generated I/O as it always evicts to disk the block that is not going to be referenced for the longest time (*the optimality of clairvoyant page replacement policies has been first established at [5]; a short proof can be found at [20]*).

3.1 Discussion: I/O Load

We now examine the I/O demand of the *SlideM* buffer management algorithm. As explained before, *SlideM* issues disk *read* requests every time blocks at RAM are expired and performs disk *write* operations whenever a new block needs to be created at buffer pool but there is no space left. Thus, the number of I/O operations requested per second (IOPS) by *SlideM* for a single sliding window operator is given by:

$$IOPS = Expired\_Blocks/sec + New\_Blocks/sec$$

Assuming that events are fixed-size and there is a balance between event arrival and expiration rates – which is always true for count-based sliding windows and is also frequently the case across a period [τ, τ+WINDOW_RANGE] of a time-based sliding window – the following property holds:

$$Expired\_Blocks/sec = New\_Blocks/sec$$

From which we derive:

$$IOPS = 2 \cdot New\_Blocks/sec$$

Now, the rate at which new blocks are produced is a function of the event arrival rate, λ, as follows:

$$New\_Blocks/sec = \lambda \,/\, block\_factor$$

Where *block_factor* represents the number of tuples stored inside a block. This relation gives us the final I/O demand:

$$IOPS = 2 \cdot \lambda \,/\, \lfloor block\_size \,/\, tuple\_size \rfloor \qquad (3.1)$$

$$IO_{bandwidth} = IOPS \cdot block\_size \qquad (3.2)$$

EXAMPLE: For an input rate of λ=1000 events/sec, 94-bytes tuples, as found in the call-center use-case, and a block size of 64KB, the IO demand of *SlideM* will be (*assuming the 1W scheme is used*):

$$IOPS = 2 \cdot 1000 \,/\, \lfloor 64 \cdot 1024 \,/\, 94 \rfloor = 2.9\ iops$$

$$IO_{bandwidth} = 2.9 \cdot 64 \,/\, 1024 = 0.18\ MB/sec$$

Note that these numbers are far less than the theoretical transfer rate of modern hard drives (e.g., up to 204 MB/sec [21]), or the maximum measured disk bandwidth achieved under workload conditions similar to the modeled application (around 25MB/sec). Therefore, *SlideM* is capable of handling much larger input rates than the ones mentioned so far or to process a much larger number of simultaneous sliding window operators before the I/O subsystem starts to become a bottleneck. Nevertheless, the scalability of the algorithm can still be greatly improved by sharing the content of overlapping windows as we discuss next.

4. SHARING STATE OF OVERLAPPING SLIDING WINDOWS

The previous section introduced *SlideM*, an efficient algorithm to manage the state of a *single* sliding window operator. Now we extend the discussion to a multi-query scenario, where *multiple overlapping* sliding windows are defined over a common event stream. The problem is of foremost importance as large-scale monitoring applications usually process several aggregation queries over different time granularities – e.g., average price of a stock in the last hour, 12 hours, last day and so on. In a naïve approach, each of these overlapping windows would be mapped into an operator inside the query execution plan. Obviously, this limits system scalability and performance since having one operator per window implies that tuples (*or pointers to tuples*) are stored multiple times at different places, thus wasting memory space. Using the algorithm of the last section only address partially this issue as the bottleneck is eventually moved from the memory system to the I/O subsystem. We then build upon the *SlideM* algorithm and propose a shared execution scheme we call *Shared SlideM* (*SSM*) to improve the usage of computational resources when processing multiple overlapping sliding windows.

4.1 Shared *SlideM* (SSM)

We consider the problem of processing a set of N aggregation queries over N sliding windows of different sizes, defined over a common event stream S. For example, assume that three SWA queries are defined over a stream "*calls*" as follows:

```
Q1:  SELECT AVG(waitTime)
     FROM calls [RANGE 1 HOUR]

Q2:  SELECT AVG(waitTime)
     FROM calls [RANGE 6 HOURS]

Q3:  SELECT AVG(waitTime)
     FROM calls [RANGE 12 HOURS]
```

A direct translation of this set of queries would result in an execution plan like the one shown in Figure 5, with aggregation (Σ) and sliding window (ω) operators being replicated for every query in the set. This naïve approach simplifies query plan generation, but wastes memory during query execution, since the tuples stored in the smaller windows are also, by definition, present in the larger windows, and could be instead maintained in a single, shared, location. This is illustrated in Figure 6: the tuples that arrived in the interval [τ_{now} - *1hour*, τ_{now}] are part of all three windows; similarly, tuples belonging to the interval [τ_{now} - *6hours*, τ_{now} - *1hour*] are shared by both the 6-hour and 12-hour windows. In this case, having one operator per window implies that roughly half of the tuples in the query set are stored more than once.

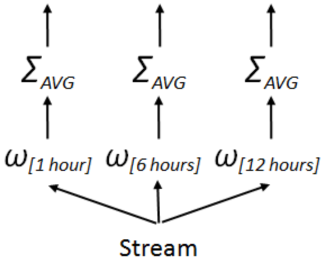

Figure 5: Unshared execution plan for three SWA queries

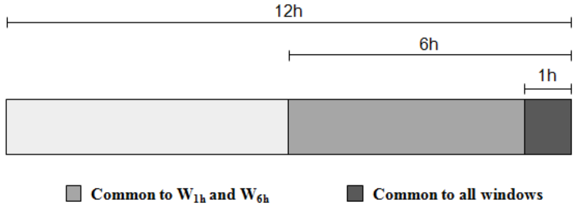

Figure 6: Three overlapping sliding windows

Using the *SlideM* algorithm reduces the pressure over main memory since portions of the windows can be offloaded to disk, but does not solve the problem of unnecessary data redundancy. Moreover, assuming that the windows do not fit in main memory, each query will produce a pair of IO operations from time to time (*read* for the ancient part of the windows and *write* for the recent segment). Eventually, as the number of queries increases, the I/O subsystem will become saturated.

To overcome these issues, we propose *SSM*, an adaptation of the *SlideM* algorithm in which multiple overlapping sliding windows are processed in a shared way. *SSM* works much like *SlideM*, in the sense that it sends parts of the window to disk when main memory is insufficient and brings data back from disk when it is time to expire them. However, contrary to *SlideM*, *SSM* manages a tuple repository that serves multiple *logical* window operators. We use the term 'logical' here because the several windows are in fact implemented by a single operator (Ω) as illustrated in Figure 7. This allows sharing computation of tuple arrivals as we explain next.

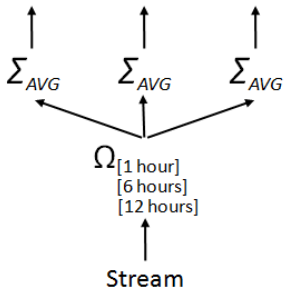

Figure 7: Shared execution plan for three SWAs, with *SSM*

A *SSM* tuple repository shared by multiple overlapping windows looks like the structure shown in Figure 8. The recent block (MRW), which stores the newly-arriving tuples, is common to all windows, but each window has its own ancient block (LRW), containing the tuples which are about to expire.

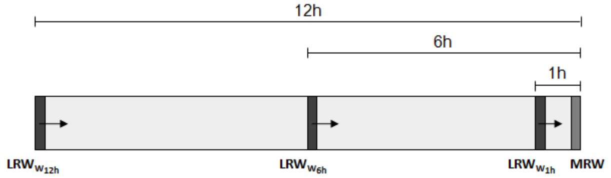

Figure 8: Shared tuple repository serving multiple windows

As the recent block is shared by all windows in the set, incoming tuples are processed only once by the shared operator Ω. Tuple arrivals in *SSM* occur essentially in the same way as in *SlideM* (see Procedure 1), with the exception that the request for adding tuples in the repository is now shared by multiple windows and the block sent to disk when the buffer pool is full is not necessarily the recent block (line 9 in Procedure 1) – a victim block must be selected instead. The major differences are, though, on the way tuple expirations are handled: first, tuples are not purged out of the repository unless the window which requested expiration is the largest one in the set. Intuitively, a tuple can only be discarded when it no longer belongs to any window, which happens when the largest window in the set requests its expiration – the same holds in a, coarser, block-level granularity. Another difference is that *SSM* does not prefetch data from disk when a block at RAM gets empty as *SlideM* does. This is because the LRW block at disk is not necessarily the block which is going to be needed next by the set of windows – with multiple windows the disk access pattern is no longer strictly sequential. Since determining which block will be required next is a potentially expensive operation, *SSM* skips prefetching and only brings data from disk when a request to a non-memory-resident block is issued. As a consequence, when the ancient block of a window gets empty it is no longer guaranteed that its "new" ancient block will be already at RAM, and as such it might be necessary to bring data from disk. Additionally, if the buffer pool is full, it will be also necessary to send a block to disk in order to open room for the upcoming block. The expiration process under the *SSM* scheme is described in detail in Procedure 3 (note that the procedure has a parameter to indicate which window the request comes from).

Procedure 3 expireOldestShared(*window_rank*)

```
let ancient_block: block at RAM holding soon-to-expire tuples of the window passed as argument
let valid_index: index of the oldest, non-expired tuple in the ancient block of the window passed as argument

1:  ancient_block ← getAncientBlock(window_rank);
2:  valid_index ← getValidIndex(window_rank);
3:  if ancient_block has only expired tuples then
4:    if window_rank is the largest then
5:      buffer_pool.remove(ancient_block);
6:    end if
7:    new_AB ← getNext(window_rank, ancient_block);
8:    if new_AB is at buffer pool then
9:      new_AB_Addr ← getBpBlockAddress(new_AB);
10:     ancient_block ← buffer_pool.get(new_AB_Addr);
11:   else
12:     new_AB_Addr ← getDiskBlockAddress(new_AB);
13:     ancient_block ← readFromDisk(new_AB_Addr);
14:     if buffer pool is full then
15:       victim_block ← getVictimBlock();
16:       buffer_pool.swap(victim_block, ancient_block);
17:       writeToDisk(new_AB_Addr, victim_block);
18:     else
19:       buffer_pool.add(ancient_block);
20:     end if
21:   end if
22:   setAncientBlock(window_rank, ancient_block);
23:   valid_index ← 0;
24: end if
25: valid_index ← valid_index + 1;
26: setValidIndex(window_rank, valid_index);
```

4.2 Discussion: I/O Load and Eviction Policy

The major advantage of *SSM* lies in a better use of memory space by avoiding that tuples are stored multiple times in the several window operators. This guarantees that no matter how many windows are defined over a given stream, the space cost will never exceed the size of the largest window in the set. As a consequence, *SSM* can handle a much greater number of queries than an unshared approach with the same amount of available memory before having to resort to secondary storage.

Now another important aspect is once the memory has been exhausted and access to disk is required, how much load *SSM* puts into the I/O subsystem. In order to determine that, consider that there are N windows of different sizes: $W_1 < W_2 < \ldots < W_N$. Let ρ be the likelihood of the next oldest block of a window being already at RAM after the current ancient block gets empty (see line 8 in Procedure 3)[4]. Assuming the buffer pool is full, the I/O pattern will be as follows: i) a write request will be issued every time a new recent block is created and ii) expiration of the ancient block of window W_i will incur, with likelihood (1-ρ): one read request, and, if $i < N$, one additional write request (for $i = N$, the ancient block is effectively removed from the buffer pool, and as such, there is no need to send data to disk to open room for the new ancient block). Algebraically:

$$\#IO = W + (1\text{-}\rho) \cdot [(N\text{-}1) \cdot (R+W) + R] \tag{4.1}$$

Note that in the limit, for ρ=0, the amount of I/O generated when processing the query set using *SSM* will be exactly the same as in *SlideM* (one pair of read and write request per window):

$$\#IO = W + (N\text{-}1) \cdot (R+W) + R = N \cdot (R+W)$$

This means that the shared execution mechanism will never perform more I/O than the unshared approach, and in the worst case the I/O pressure of the two schemes will be equivalent. For any ρ>0, *SSM* will reduce the amount of I/O, and the reduction will be as large as ρ. As discussed in previous section, the optimal eviction policy that maximizes ρ is the one that sends to disk the block that is not going to be needed for the longest time. For the single-window case, the choice is straightforward: the optimal victim block is always the most recently written block. This does not hold for multiple windows though, as the MRW block might be needed earlier by a small window than an intermediate block by larger windows. Instead, the optimal victim block in a multi-window scenario can be determined by computing the distance – in number of blocks or time units – of the candidate blocks at the buffer pool to the ancient block of each window as follows: let d_{ki} be the distance of block k at BP to the ancient block of window w_i. For any block k, ref_k denotes the next time the block will be referenced by any window, and corresponds to the minimum value in the set of distances: $ref_k = \min\{d_{ki} \mid d_{ki} > 0\}$. The victim block v is the one with the maximum value for ref_k among the B candidates at buffer pool: $v = (k \mid ref_k = \max\{ref_1, \ldots, ref_B\})$.

This *distance-based* replacement policy creates clusters of blocks in the BP, immediately after the ancient block of each window as illustrated in Figure 9 below:

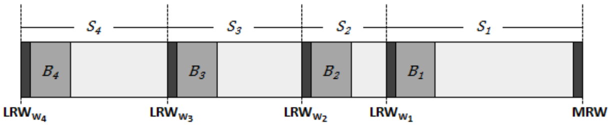

Figure 9: Typical arrangement of blocks at the buffer pool when using *SSM* block replacement policy.

Since each block is referenced only once by each window, the buffer pool hit rate of the scheme is given by the average percentage of blocks residing at memory of each segment:

$$\rho_{SSM} = \frac{1}{\mathrm{N}} \sum_{i=1}^{N} \left(B_i / S_i \right) \tag{4.2}$$

where B_i is the number of memory-resident blocks of each segment and S_i is the corresponding total number of blocks.

Clearly, the more memory is available (larger B_i) and the more overlapping the windows are (smaller S_i), the higher the buffer pool hit rate ρ_{SSM} will be.

[4] In fact, ρ represents the hit rate of the buffer pool.

5. PERFORMANCE STUDY

In this section we present an extensive experimental evaluation of the *SlideM* algorithm and the sharing scheme *SSM*. We have performed a wide variety of experiments with the objective of assessing:

1. *Effectiveness of SlideM in a real-world use-case*: we demonstrate the ability of the proposed algorithm in addressing memory shortages in a real scenario and compare its performance against the conventional memory-only implementations discussed earlier in this paper (Section 5.2).
2. *High-performance nature of SlideM*: we examine *SlideM* performance under heavy load conditions. Results reveal that the algorithm was capable of handling very high input rates for multi-gigabyte windows while keeping latency under desirable levels (Section 5.3).
3. *Performance and scalability of SSM*: we show that the sharing mechanism *SSM* scales significantly better than an unshared approach (Section 5.4).

For the first set of experiments, we used queries and stream definitions taken from the real use-case, as described in Appendix A. For the other two sets, we used synthetic queries and datasets. Tests setup and methodology are described next.

5.1 Experimental Setup and Methodology

We implemented our proposed techniques in Pulse [19], a Java-based stream processing engine from industrial partner FeedZai. Experiments were conducted on a server with two Intel Xeon E5420 2.50 GHz Quad-Core processors, 4 GB of RAM, and 4 SATA-300 disks distributed in two RAID-0 arrays, running Windows Server 2008 x64 and Hotspot x64 JVM (*configured with a 1 GB heap size*). One RAID array was used to host the OS while the other was used to hold window data during tests. Measurements were taken as follows:

- A single Java application was responsible for generating, submitting and consuming tuples during the performance runs. Input data was submitted to the SPE through local method calls using its API.
- Tests consisted in a *warmup phase*, during which the SPE was brought to a steady state, and a subsequent *measurement interval* (MI), when the performance of the system was measured. The duration of both warmup and MI was set to the time necessary for traversing 1.5 times the window – *e.g., an experiment with a 6-hour window ran for 18 hours (9h of warmup plus 9h for MI)*.
- We collected both application-level and system-level metrics. Average throughput was computed as the ratio between processed tuple count and elapsed time. Latency was computed through the `nanoTime()` method of the Java runtime, called immediately before and after sending tuples to the SPE. Memory consumption was computed by the end of tests using standard Java SDK methods. CPU, disk, and process metrics were collected using the *System Monitor* tool of MS-Windows.

All experiments with *SlideM* and *SSM* used a fixed block size (64 kilobytes).

5.2 Call-Center Use-Case Results

Our first set of experiments mimics the workload conditions of a real event processing application, and consists in processing a number of aggregations like Query 2 below:

Query 2 : *Compute call-center statistics in the last day*

```
SELECT   COUNT(*),
         SUM(busyTime),
         AVG(busyTime)
FROM     calls [RANGE 24 HOURS SLIDE 10 SECONDS]
GROUP BY serviceId
```

On total, the application computes 144 aggregates, resulting in 104 distinct SWA operators – *the query above is replicated for 6 different fields and 8 distinct grouping criteria, with the COUNT aggregate being shared by the queries with different fields; a more detailed description of the use-case can be found in Appendix A*. We then compare the performance of memory-only SWA implementations against application performance when paging sliding window content to disk through the *SlideM* algorithm. Both the *1W* and the *2LA* SWA approaches were tested in each case. Results are presented in Figure 10.

The three uppermost graphs show the performance of the two non-managed implementations, and illustrate what typically occurs with memory-constrained event processing applications in most Java-based SPEs: as the application working set approaches the available memory threshold, the system spends progressively more time with garbage collection, increasing the tuple processing latency and preventing the SPE to cope with the data input rate. Since there is no data to be purged until the sliding window closes, the application eventually crashes with an out-of-memory error. In our tests this happened before 3 hours for the *1W* implementation and before 1 hour when using the *2LA* scheme, as signalized in Figure 10.

The results above contrast with application behavior when the *SlideM* algorithm is employed to manage the state of the sliding windows, as illustrated in the bottom part of Figure 10. Using our proposed algorithm allowed the experiments to complete without errors, while keeping performance metrics in desirable levels. As expected, memory consumption and tuple processing latency was larger when using the *2LA* scheme than with the *1W* approach in this use-case.

5.3 Performance of *SlideM*

Many event processing applications, like those found in the financial trading environment, require that SPEs be able to process a considerable volume of data within very short periods of time. Using disks in these cases might be inadequate if they are not able to cope with the very high arrival rates and stringent latency requirements. In this section we examine how *SlideM* performs in such critical scenarios. For that, we employ a simple microbenchmark, which consists in computing one or more aggregations over a stock market data stream. Each input tuple has 4 attributes: *Timestamp*, *Symbol*, *Price* and *Volume* (about 28 bytes). We fill tuples by repeatedly cycling through a list of 100 stock symbols and assigning the tuple creation time to the timestamp field and random values to the other two. The workload consists in computing the volume-weighted average price (VWAP) of each stock over the last hour, as shown in Query 3. Six runs of this experiment are performed,

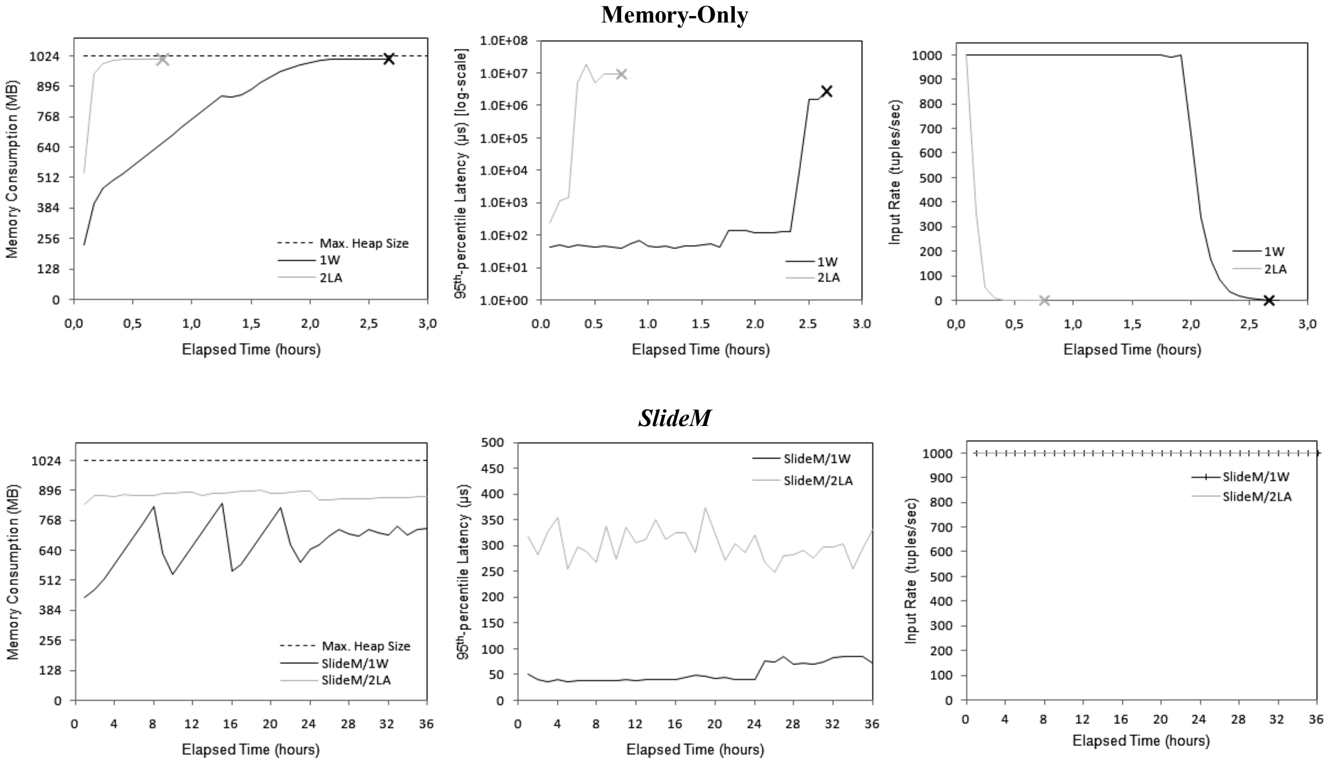

Figure 10: Comparison: performance of *SlideM* vs. memory-only implementations in real-world-based workload conditions

progressively scaling the injection rate from 50,000 up to 300,000 tuples per second.

Query 3 : *Compute the VWAP of each stock over the last hour*

```
SELECT   Symbol, SUM(Volume*Price)/SUM(Volume)
FROM     Stock [RANGE 1 HOUR]
GROUP BY Symbol
```

Note that the window definition in the query above does not include a SLIDE, which means that the result must be updated whenever a new tuple arrives at the *Stock* stream[5]. In all experiments, the buffer size was set to the minimum, 2 blocks (128KB), so that system performance is measured under maximum I/O pressure. Results are shown in Table 1.

As we can see, the system was able to handle up to 300,000 tuples per second, with the CPU being the limiting factor at that point. Average processing latency was fairly low in all experiments (a few microseconds), and even the absolute maximum latency remained under acceptable levels as the load was increased. Disk utilization was also quite low in all runs, as it can be seen from the average disk queue length (ADQL) metric in Table 1. The reason is that the disk bandwidth required by *SlideM* at the maximum load of 300,000 tuples per second in this benchmark is only 16 MB/sec, which is still far from the maximum measured disk transfer rate, as discussed in Section 3.1. This moderate load posed by *SlideM* into the I/O subsystem was crucial to remove a bottleneck (memory) without creating a new one, thus allowing the SPE to fully exploit the available CPU power.

Table 1: *SlideM* Performance, scaling injection rate

Injection Rate (tuples/sec)	Space Cost (GB)	Avg. Latency (ms)	Max. Latency (ms)	% CPU	ADQL
50,000	4.7	0.009	2.9	5%	0.016
100,000	9.4	0.012	3.6	13%	0.035
150,000	14.1	0.007	3.9	23%	0.056
200,000	18.7	0.012	16.7	37%	0.071
250,000	23.5	0.008	14.4	69%	0.103
300,000	28.2	0.008	18.9	100%	0.145

5.4 Performance of *SSM*

We now examine the performance of the shared execution scheme *SSM* and quantify to which extent it scales better than an unshared approach. Experiments here consist in processing N instances of Query 3, using either the unshared *SlideM* algorithm or its shared counterpart, *SSM*, under an input rate of 5,000 tuples per second. The number of queries in each experiment, N, took the following values: N = {2, 4, 8, 16, 32}. All N queries in the set have different window sizes, uniformly distributed in the interval [3600, 7200] seconds. Available memory was set to

[5] An aggregation query without a SLIDE clause would probably make little sense if its result were to be output (i.e., used for monitoring purposes). In many cases, however, the result of an aggregation is used as input for further processing (e.g., pattern detection), and updated results must be produced as soon as new data is available.

512MB in all experiments (for the non-shared version, this amount was equally divided among the N buffer pools). We then measured for each algorithm the total space cost and the amount of pressure put onto the I/O subsystem. Results are depicted in Figure 11:

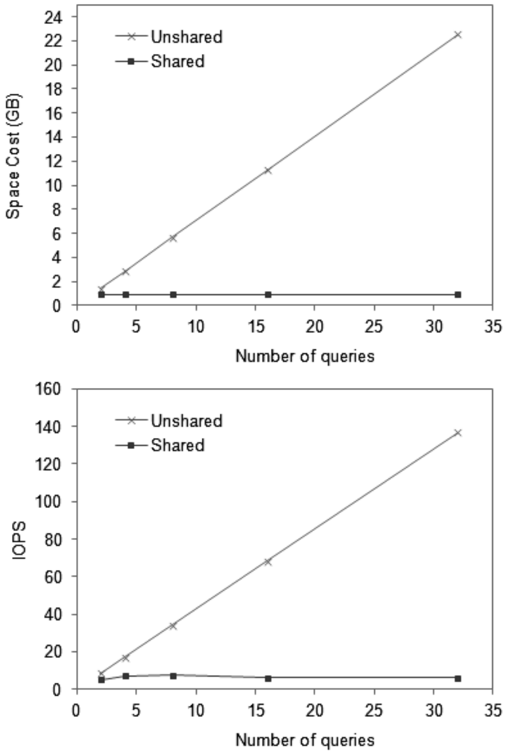

Figure 11: Performance of *SlideM* (unshared) and *SSM* (shared) in a multi-query scenario.

As expected, the total space cost (*memory and disk*) of the unshared approach grows linearly with the number of windows while with the shared strategy the space cost remains constant (*it is bounded to the size of the largest window*). *SSM* was also significantly more I/O-efficient, issuing up to 22 times less disk requests than the unshared implementation. The explanation for this remarkable difference in the number of I/O requests between the two algorithms is that *SSM* benefits from the fact that adding more queries to the set reduces the distance between the overlapping windows, thus increasing the hit rate of the buffer pool as expressed in formula 4.2. This way, while the I/O pressure of the unshared implementation consistently grows as more windows are used, with *SSM* it tends to stabilize since the increase on buffer hit rate compensates for the increased number of simultaneous queries. The result is a much better scalability as we can see in Figure 11.

6. RELATED WORK

There has been considerable work on resource management in stream processing systems [4] [11] [17]. For dealing with memory shortages, two approaches have been widely employed. The first consists in providing approximate answers by shedding load [8] [22] [24], with research on this area focusing essentially on minimizing error of approximations. However, many event processing applications rely on exact answers to perform complex data analysis and support real-time decision making. In these cases, techniques such as load shedding or approximation are not applicable. The alternative, then, is to use secondary storage as an extension of main memory. Indeed, such disk-based approach has been adopted in a number of proposals such as [6], [10] and [15]. The focus of those works, though, is on processing of *join queries*. Liu et al [15] consider queries with multiple operators and propose strategies to choose which part of the operator states to spill during query execution in order to maximize the overall throughput. As discussed before, this is not an issue for sliding-window aggregates since the data access pattern can be accurately predicted. Farag and Hamad [10] propose a two-phase external-memory algorithm that joins the arriving tuples of one stream with the memory-resident data of the other streams, and postpones matching with the disk-resident portion until the stream runs out-of-space or arrival of new tuples stalls. Chakraborty and Singh [6] propose an *Exact Window Join* algorithm that deals with memory shortages by deferring the load during high workload, and processing the deferred load during the period of low workload. This strategy, however, results in high delays (> 5 seconds) even for moderate data input rates (450 tuples per second). To the best of our knowledge, our work is the first to address the problem of exact answer computation of aggregations in memory-limited, high-throughput, environments.

Shared processing of sliding-window aggregates has been previously explored in a couple of proposals. Arasu and Widom [3] devise two algorithms for sharing execution of multiple sliding-window aggregates, where a common aggregation function is computed over different window sizes. These algorithms assume an *aperiodic* scenario, where results are produced on-demand (when user polls a query). Our proposed strategy, on the other hand, is for periodic aggregates and applies even when different aggregation functions are used. In [12] Krishnamurthy proposes a strategy for sharing the execution of multiple periodic sliding-window aggregates implemented under the *2LA* scheme. The strategy focuses on *computation sharing*, and consists in computing the partial aggregates with only one shared operator, rather than using one operator per query. It does not address, however, the *space sharing* problem introduced in this paper, as the partial aggregates are still stored several times at the main window.

7. CONCLUSIONS

In this work we introduce techniques for overcoming the traditional memory limitations faced by stream processing engines when processing aggregation queries over sliding windows. We address the problem by proposing a novel buffer management algorithm, *SlideM*, which offloads sliding window state to disk during memory shortages. In order to further increase algorithm scalability, we also proposed *SSM*, a query sharing strategy that prevents explosion of space cost by storing the state of multiple overlapping sliding windows in a single, shared, repository. Experimental results demonstrated that the two techniques together provide significant performance and scalability benefits. With *SlideM* the system was able to handle up to 300,000 events per second for multi-gigabyte windows while consuming only 128 kilobytes of main memory. In a scenario with multiple simultaneous queries, *SSM* reduced space cost by a factor of up to 24, issuing up to 22 less disk requests.

There are a couple of issues we did not explore in this paper and constitute interesting avenues for future work. First, the techniques proposed here are designed to exploit the access pattern of sliding window operators during event arrivals and expirations, which allows excellent performance for SWA queries. Our implementation currently supports random access to tuples inside the window, but probably in a way far from optimal for operations such as joins. Therefore, we intend to investigate state-spilling mechanisms that work well for a more diversified gamma of queries. A promising direction is to combine per-operator *SlideM* repositories with a global buffer manager responsible for serving queries with less predictable access patterns. Another possible direction for future work is to extend the proposed algorithms and develop new disk-based techniques in order to enable SPEs to recover their state in the advent of system failures. Finally, the results presented in this work were obtained with conventional hard drives. It shall be interesting to observe the behavior of the proposed techniques in conjunction with faster storage technologies like solid-state disks (SSDs) or phase-change memories (PCMs).

Acknowledgements

We would like to thank the development team at FeedZai for all the feedback and valuable discussions during the implementation of the query execution engine of the Pulse SPE. This research has been supported in part by the Portuguese Science and Technology Foundation (FCT), under grant Nº 45121/2008, and by the industrial partner FeedZai.

8. REFERENCES

[1] Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S.B. Aurora: a new model and architecture for data stream management. In *Proceedings of VLDB Journal*. 2003, 120-139.

[2] Arasu, A., Babu, S., and Widom, J. The CQL continuous query language: semantic foundations and query execution. In *Proceedings of VLDB Journal* Vol. 15 Issue 2, 2006, 121-142.

[3] Arasu, A. and Widom, J.: Resource Sharing in Continuous Sliding-Window Aggregates. In *Proc. of the 30th VLDB Conference* (Toronto, Canada, September 2004), 336-347.

[4] Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J.: Models and Issues in Data Stream Systems. In *Proceedings of the 21st Symposium on Principles of Database Systems*, pages 1–16, June 2002.

[5] Belady, L.A.: A Study of Replacement Algorithms for Virtual-Storage Computer. In *Proceedings of IBM Systems Journal*. 1966, 78-101.

[6] Chakraborty, A., and Singh, A.: Processing Exact Results for Sliding Window Joins over Time-Sequence, Streaming Data Using a Disk Archive. In *Proceedings of the 1st Asian Conference on Intelligent Information and Database Systems* (Vietnam 2009), 196-201.

[7] Chandrasekaran, S., et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of CIDR 2003* (Asilomar, California, USA).

[8] Dobra, A., Garofalakis, M., Gehrke, J., Rastogi, R.: Processing Complex Aggregate Queries over Data Streams. In *Proceedings of the 2002 ACM SIGMOD*, Madison, Wisconsin, USA.

[9] Esper: http://esper.codehaus.org/

[10] Farag, F., and Hamad M.A: Adaptive Execution of Stream Window Joins in a Limited Memory Environment. In *Proc. of the 11th International Database Engineering and Applications Symposium* (Banff, Canada, 2007), 12-20.

[11] Golab, L., and Ozsu, M.: Issues in Data Stream Management. SIGMOD Record, Vol. 32, Issue 2, 5–14.

[12] Hinze, A., Sachs, K., Buchmann, A.P. Event-based applications and enabling technologies. In *Proceedings of DEBS* 2009 (New York, USA), Art. 1, 15 pages.

[13] Krishnamurthy, S. *Shared Query Processing in Data Streaming Systems*. Ph.D. Thesis, University of California, Berkeley, 2006.

[14] Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P.A. No Pane, no Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams. SIGMOD Record (2005), Vol. 34, Issue 1, 39-44.

[15] Liu, B., Zhu, Y., and Rundensteiner, E.A.: Run-Time Operator State Spilling for Memory Intensive Long-Running Queries. In *Proceedings of the 2006 ACM SIGMOD*, (Chicago, Illinois, USA), 347-358.

[16] Mendes, M.R.N., Bizarro, P., and Marques, P.: A Performance Study of Event Processing Systems. In *Proceedings of TPCTC 2009* (Lyon, France), 221-236.

[17] Motwani, R., and et al.: Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *Proceedings of CIDR 2003* (Asilomar, California, USA).

[18] Oracle CEP: http://docs.oracle.com/cd/E13157_01/wlevs/docs30/

[19] Pulse: http://www.feedzai.com/products/pulse

[20] Roy, B.V. A Short Proof of Optimality for the MIN Cache Replacement Algorithm. In *Proceedings of Information Processing Letters*, 2007, 72-73.

[21] Seagate Cheetah hard disk Data Sheet: http://www.seagate.com/files/docs/pdf/datasheet/disc/cheetah-15k.7-ds1677.3-1007us.pdf

[22] Srivastava, U., and Widom, J.: Memory-Limited Execution of Windowed Stream Joins. In *Proc. of the 30th VLDB Conference* (Toronto, Canada, September 2004), 324-335.

[23] StreamBase: http://www.streambase.com/

[24] Tatbul, N., Etintemel, U., Zdonik, S. B., Cherniack, M., and Stonebraker M.: Load shedding in a Data Stream Manager. In *Proceedings of the 29th VLDB Conference* (Berlin, Germany, September 2003), 309-320.

APPENDIX

A. USE-CASE DETAILS

In this section we present some of the details of the call-center monitoring application that we referred to throughout this paper. The description provided here is for a proof-of-concept prototype, which corresponds to a simplified version of the full application, still under development.

As mentioned before, the purpose of the application is to provide a real-time view of the operation of a large call center chain. The company is spread over 20 geographical sites and has around 12,000 agents serving more than 3 million customer requests per day. A statistical module collects information about the calls and produces a stream of data items describing each step of the interactions between the call center and its customers. This data stream, whose schema is shown in Figure 12, is then fed into the stream processing engine, where 144 KPIS are continuously computed. Each KPI corresponds to a 24-hour sliding-window aggregate over a given field of the stream. Specifically, three functions (SUM, COUNT and AVG) are applied over 6 attributes of the stream (*alertingTime*, *busyTime*, *wrapUpTime*, *waitTime*, *helpTime*, and *availableTime*), using 8 different grouping keys (*instance*, *serviceId*, *agentId*, *mediaId*, *interactionLegId*, *agentSite*, *callSite*, and *direction*).

In our experimental evaluation we filled the tuples of the stream *AgentInteractions* with synthetic data since real datasets were not available due to confidentiality issues. The generated data, however, respected the critical properties of the original input stream, such as the cardinality of the attributes used as grouping key in the queries and the distribution of these groups over time. We did not replicate eventual oscillations on tuple arrival rate though, keeping the injection rate fixed in 1,000 tuples per second. All the tests were performed in a virtual machine with 8 cores, 2GB of RAM, and running Window Server 2008, as found in the production environment.

```
AgentInteractions (
   timestamp                long,
   instance                 int,
   start                    long,
   sessionId                int,
   serviceId                int,
   agentId                  int,
   interactionLegId         int,
   alertingTime             int,
   busyTime                 int,
   wrapUpTime               int,
   waitTime                 int,
   direction                int,
   mediaId                  int,
   helpTime                 int,
   agentReleased            bool,
   mediaOutcome             int,
   finalSegment             bool,
   agentSite                int,
   callSite                 int,
   availableTime            int,
   availableTimeByService   int,
   held                     int,
   help                     int
)
```

Figure 12: Schema of the input data stream in the call-center monitoring application