Harnessing Sliding-Window Execution Semantics for Parallel Stream Processing

Gabriele Mencagli[a,*], Massimo Torquati[a], Fabio Lucattini[a], Salvatore Cuomo[b], Marco Aldinucci[c]

[a] *Department of Computer Science, University of Pisa, Italy*
[b] *Department of Mathematic and Applications, University of Naples Federico II, Italy*
[c] *Department of Computer Science, University of Turin, Italy*

Abstract

According to the recent trend in data acquisition and processing technology, *big data* are increasingly available in the form of unbounded streams of elementary data items to be processed in real-time. In this paper we study in detail the paradigm of *sliding windows*, a well-known technique for approximated queries that update their results continuously as new fresh data arrive from the stream. In this work we focus on the relationship between the various existing sliding window semantics and the way the query processing is performed from the parallelism perspective. From this study two alternative parallel models are identified, each covering semantics with very precise properties. Each model is described in terms of its pros and cons, and parallel implementations in the FastFlow framework are analyzed by discussing the layout of the concurrent data structures used for the efficient windows representation in each model.

Keywords: Data Stream Processing, Internet of Things, Continuous Queries, Sliding Windows, Parallel Computing

1. Introduction

Our world is becoming ever more hyper-connected as the number of intelligent devices installed in our everyday objects and environments is increasing at an exponential rate. An increasing number of applications require to apply Data Science techniques (e.g., data mining and machine learning algorithms) to extract insights from massive volumes of data that are often made available as transient unbounded flows of elementary items like sensor readings, stock tickers or timed events in general [1].

In recent years, Stream Processing Engines (briefly, SPEs) like Apache Storm [2], IBM InfoSphere Streams [3] and Spark Streaming [4] have become principal components in many Big Data technology stacks. Their data-flow programming style is considered a promising approach for *Internet of Things* (IoT) scenarios [5, 6], which may improve the programmability of applications with high socio-economical impact like Smart Cities, Automotive and Cultural Heritage applications [7, 8, 9].

The data-flow programming style allows high levels of parallelism by designing applications as directed graphs of computing kernels called *operators*, which consume data items from input streams and produce output results onto streams connected to other operators [10, 11]. Operators whose processing speed is not fast enough need to be internally parallelized to increase their throughput by providing timely responses to the users. Most of the existing SPEs provide low-level solutions to parallelize bottleneck operators. An approach is to replicate the operator and to schedule input items (also called *tuples*) to the replicas according to distribution policies that are safe in terms of computation semantics [10] and whose definition may be in charge of the application programmer.

To deal with the unbounded length of the stream, SPEs provide techniques to repeatedly apply the processing on the most recent tuples only. This is enabled by the so-called *sliding window* processing approach [12], where a window is a bounded set of the most recent tuples whose content is dynamically determined according to various semantics made available to the programmer (e.g., count-based, time-based and hybrid models).

A certain effort has been made over the years in order to categorize the sliding window semantics [13] and to derive properties about their computation accuracy [14]. However, *the relationship between the sliding window semantics and suitable parallelism models is still not clear*. The simplest idea is to process in parallel windows whose content is complete (i.e. all their tuples have been received). Nevertheless, different approaches can be devised based on which entities in the parallel version are in charge of managing the windows evolution and performing the processing on the triggered windows. This paper aims at providing a complete picture of the possible approaches for parallel windowed operators, which are suitable to be provided in the future as ready-to-use patterns with a user-friendly

*Corresponding author, Phone: +39-050-221-3132
Email addresses: mencagli@di.unipi.it (Gabriele Mencagli), torquati@di.unipi.it (Massimo Torquati), lucattini@di.unipi.it (Fabio Lucattini), salvatore.cuomo@unina.it (Salvatore Cuomo), aldinucci@di.unito.it (Marco Aldinucci)

interface. The main contributions can be summarized as follows:

- the *sliding-window parallelism paradigm* is proposed as an alternative to the partitioned-stateful parallelism paradigm commonly used in existing SPEs;
- the theoretical work in [13] proposed a categorization of the sliding window semantics based on the information (on past or future tuples) needed to map input items onto sliding windows. We extend such work by understanding *how the windowing semantics affects the parallelization*. Two models are derived: the *Agnostic* and the *Active Worker Models*. The two approaches are presented with their pros and cons;
- the implementation of the two models is described by focusing on the layout of the data structures that we need to represent the buffered tuples of the stream. The *Concurrent Chunk-based Window Buffer* structure is presented and designed.

The models and implementations are developed in the FastFlow streaming library [15] and the porting of such ideas on widely utilized SPEs is our future aim.

The paper is organized as follows. The next section will provide a brief background on Data Stream Processing by presenting a categorization of the different sliding window semantics. Sect. 3 proposes the two models whose implementation is presented in Sect. 4. Sect. 5 presents a set of interesting preliminary results on multicores that provide a first validation of our ideas. Finally, Sect. 6 describes the related work and Sect. 7 states the conclusion of this work and our future research directions.

2. Data Stream Processing

In this section we describe the main features of the Data Stream Processing research topic (briefly DaSP). In particular, we focus on the characteristics of the programming models provided by the most common SPEs with special attention to the windowing abstractions, which are of great importance for the goals of this paper.

Almost all the existing SPEs provide the programmer with relatively high-level constructs to implicitly or explicitly build complex streaming topologies. A *topology* is a directed graph of logic transformations called processing elements or simply *operators*. Operators can be connected arbitrarily though some SPEs constraint the form of the admissible topologies (e.g., without cycles). Operators can be *sources* that generate data flows by interfacing with external data producers like raw sensors or any data provider, *sinks* that absorb the results by eventually consolidating them into databases, and operators representing intermediate stages of the data transformation. Information exchanged within the topology assumes the form of unbounded flows of data items defined as follows:

Definition 1 (streams and tuples)**.** A data stream $\mathcal{S}$ is an unbounded sequence $\mathcal{S} = (t_0, t_1, \dots,)$ of tuples all having the same *type* $\mathcal{T}$. A tuple t is a record of $d > 0$ attributes $< a_1{:}\texttt{type}_1, \dots, a_d{:}\texttt{type}_d, t_s{:}\texttt{timestamp} >$ where $t.a_i$ is the i-th attribute (e.g., a floating point number or an integer) and $t.t_s$, also represented as $\tau(t)$, is the timestamp denoting the time at which the tuple has been produced.

While the identifiers of the tuples reflect the ordering at which they are received at the destination side, the timestamps reflect the generation time at the producer side. In general data streams are ordered, i.e. for each pair of tuples $t_i, t_j \in \mathcal{S}$, $i < j \iff \tau(t_i) < \tau(t_j)$. Instead, a stream is out-of-order if it may be possible that $\tau(t_i) < \tau(t_j)$ with $i > j$ and tuple t_i is called a *late arrival*. Disordered streams exist in the real practice and require proper mechanisms to be processed correctly (e.g., punctuations, reordering buffering [16, 17]). In the rest of this paper we will focus on ordered streams only.

Each operator executes an infinite loop where input tuples are received from its input queues, it goes over the input by eventually using its internal state and produces output tuples delivered to the operator's output queues. Most of the SPEs provide built-in operators for common data processing functions (e.g., aggregate and joins) and also allow the programmer to define customized operators performing user-defined functions. When an operator has multiple input queues (*input arity* greater than one) we can distinguish between two possible activation semantics:

- *non-deterministic*, where the internal processing logic of the operator consumes an input tuple as soon as it is available in any of its input queues;
- *data-flow*, where the operator logic is executed when at least one tuple is available in each input queue. One input tuple per queue is consumed by a single activation of the operator.

Furthermore, depending on the operator semantics, one or more input tuples can be consumed before producing an output result or alternatively more output tuples can be produced per input. In the literature this property is called *input/output selectivity* of operators [18]. Finally, if the operator has multiple output queues (*output arity* greater than one), each result tuple can be delivered to exactly one output queue selected according to a certain policy, or it can be replicated and each copy sent to each queue (data-flow semantics).

The run-time system of SPEs is responsible for executing the topologies submitted by the users, and to deploy them onto the underlying resources (e.g., cores of a multicore, nodes of a cluster) according to placement policies balancing throughput and resource consumption [19]. A topology remains in execution until the user explicitly stops it or when source operators stop producing tuples.

2.1. Windowed Operators

Specific computing models have been developed to cope with the problem of having possibly unbounded input streams that feed the running topologies. In most cases the

informational value of input tuples is time-decaying [18], and operators often require to maintain the recent history of the streams and to perform the processing task on the most recent data. The abstraction provided for doing this is based on *sliding windows* [12].

Windowed operators apply their internal processing logic on sliding windows of the input stream. A window represents a subset of the tuples belonging to the input stream that are logically grouped together. Typically, a windowed operator produces a result tuple per window where the processing logic is specific of the operator at hand. Examples are windowed operators for computing aggregates [13] (e.g., average, quantiles, standard deviation) or preference queries like skyline and top-k [20].

Existing SPEs provide some standard windowing concepts or provide the users with programming mechanisms to define custom windowing semantics. As shown in [12], the window semantics is characterized by:

- an *eviction policy*, which determines which tuples must be evicted because they are too old and they do not belong to the next window to compute;
- a *triggering policy*, which determines when a new window is ready to be computed.

Several models have been introduced in the literature based on the mechanisms adopted for controlling the two policies. The following code fragment describes the definition of a windowed operator in Apache Storm [2]:

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new MySpout());
builder.setBolt("winbolt", new MyWinBolt().
    withWindow(new Count(100), new Count(10))).
    shuffleGrouping("spout");
```

In the fragment the topology consists of two operators, a source (called *spout*) and an operator (called *bolt*) that instantiates the `MyWinBolt` class providing methods for sliding window processing (it implements the `IWindowedBolt` interface). In the syntax the windowed bolt adopts a *count-based* mechanism for controlling both the eviction and the triggering policy. In other words, the operator computation is applied over the last 100 received tuples of the stream, and the processing is repeated each time new 10 tuples received by evicting the 10 oldest tuples. These two parameters are referred to as *range* and *slide* denoted by W and S in general.

For the sake of comparison, the following code fragment reports the instantiation of the same topology in FastFlow [15], a C++11 template library for stream processing on multicores:

```
MySource source();
Win_Seq<tuple_t, output_t> winOperator(tupleInfo
    , winFunction, 100, 10, CB);
ff_Pipe<tuple_t, output_t> pipe(source,
    winOperator);
```

Where the `ff_Pipe` object consists in a pipeline where the first stage is the source and the second is the windowed operator (instantiating the `Win_Seq` template) . The constructor takes some input parameters. The `tupleInfo` and `winFunction` parameters are two functions needed for extracting the identifier from the tuple that can be a user-defined data type and the function that goes over the window tuples to produce the result. The next two parameters are the range and the slide of the windows and the last indicates that we use count-based windows (CB).

Analogously, *time-based* windows can be defined by providing the range and the slide parameters in time units instead of in terms of tuples. For example in Apache Storm this can be achieved as follows:

```
builder.setBolt("winbolt", new MyWinBolt().
    withWindow(new Duration(10, TimeUnit.SECONDS
    ), new Duration(2, TimeUnit.SECONDS))).
    shuffleGrouping("spout");
```

In this case the windowed bolt applies the computation on the tuples received in the last 10 seconds by producing a new result every 2 seconds. To move from a window to the next one the oldest tuples received in the first 2 seconds are evicted and the newest tuples received in the last 2 seconds are added to the window. In FastFlow a time-based windowed operator can be instantiated by passing the enumeration parameter "TB" to its constructor.

2.2. Windows Semantics

Different models of sliding windows are based on the mechanisms that rule the eviction and the triggering policies. Count-based and time-based windows are only two of the most common examples that belong to a more complex taxonomy. To understand the different window semantics we need to introduce a specific notation and terminology.

As originally described in [13], the window semantics can be formally specified by describing how tuples are mapped onto windows and vice-versa. We number the windows processed by an operator using a progressive identifier starting from zero, i.e. W_i is the i-th window where $W_{id} = \{0, 1, \ldots\}$ is the set of the window identifiers. We introduce two concepts:

- *window extent*: it is a function $\mathcal{E} : W_{id} \rightarrow \mathcal{P}(\mathcal{S})$ that, given the identifier of a window, returns the set of the tuples in the stream that belong to that window;
- *window mapping*: it is a function $\mathcal{M} : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{W}_{id})$ that assigns to each tuple $t \in \mathcal{S}$ the set of all the identifiers of the windows that contain that tuple.

For count-based and time-based sliding windows with range W and slide S the window extent function applied on a window with identifier $w \in W_{id}$ is defined as follows:

$$\mathcal{E}(w) = \{t_i | t_i \in \mathcal{S},\ w \cdot S \leq i < W + w \cdot S\} \quad (1)$$

$$\mathcal{E}(w) = \{t | t \in \mathcal{S},\ w \cdot S \leq \tau(t) < W + w \cdot S\} \quad (2)$$

The definition (1) is the extent function of count-based windows while (2) is the one of time-based windows.

Other windowing models have been developed. An example is represented by *slide-by-tuple* windows, which are

a sort of hybrid configuration supported by some SPEs. In this model the eviction policy is governed by a time-based mechanism (W is expressed in time units) while the triggered policy uses a count-based mechanism (S is specified in number of tuples). This reflects in sliding windows that trigger exactly every new S tuples received and each window spans from the triggering tuple t_r to the oldest tuple with timestamp greater than $\tau(t_r) - W$. Given a window $w \in W_{id}$, the window is triggered by the arrival of tuple $t_r \in \mathcal{S}$ such that $r = [(w+1) \cdot S] - 1$. The extent function is defined as follows:

$$\mathcal{E}(w) = \{t | t \in \mathcal{S},\ \tau(t_r) - W < \tau(t) \leq \tau(t_r)\} \tag{3}$$

The last example that we mention in this section consists in the *delta-based* model provided by IBM InfoSphere Streams [3]. In this model an attribute δ is chosen to be the *delta* attribute used for windowing purposes. In general, this attribute must be numeric and non-decreasing as new tuples arrive at the operator. The parameter W indicates the *eviction threshold*. When a tuple t_r triggers a new window, all the previously received tuples $t \in \mathcal{S}$ such that $t.\delta < t_r.\delta - W$ must be evicted because the difference between the delta value of the triggering tuple and the one of t is greater than the eviction threshold. The triggering policy determines when a tuple is considered a triggering one and uses a parameter S as the *triggering threshold*. Let t_r be the last tuple that triggered the window processing. A next tuple t is considered a new triggering tuple if the difference between its delta value and the one of t_r exceeds the triggering threshold, i.e. $t.\delta - t_r.\delta > S$.

To define the extent function of delta-based windows we introduce the notation t_r^i to indicate the i-th triggering tuple. Given a tuple t we indicate with $\mathcal{N}(t)$ the set of tuples with delta attribute greater than $t.\delta + S$. Hence, the i-th triggering tuple t_r^i for any $i > 0$ is defined as the tuple with the smallest identifier in the set $\mathcal{N}(t_r^{i-1})$. The extent function is defined as follows:

$$\mathcal{E}(w) \quad = \{t | t \in \mathcal{S},\ t_r^w.\delta - W \leq t.\delta < t_r^w.\delta\} \tag{4}$$

where the first triggering tuple is by default equal to the first tuple of the stream, i.e. $t_r^0 = t_0$.

So far we focused on the extent function that provides a formal definition of the content of each window. The mapping function $\mathcal{M}$ represents the inverse of $\mathcal{E}$ and plays a central role in the window semantics. The function maps a tuple onto a subset of window identifiers, i.e. the ones that contain that tuple in their extent. To formalize the definition of the mapping function we need two types of information related to either the past tuples or to future tuples with respect to a given tuple t. This is sketched in Fig. 1 where the following concepts are introduced:

- we define the *backward context* $\mathcal{B} : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{S})$ with respect to tuple $t_i \in \mathcal{S}$ the set of all the tuples that arrived at the operator *before* t_i (included), i.e. $\mathcal{B}(t_i) = \{t_j | t_j \in \mathcal{S},\ j \leq i\}$;
- we define the *forward context* $\mathcal{F} : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{S})$ with respect to tuple $t_i \in \mathcal{S}$ the set of all the tuples that will arrive *after* t_i, i.e. $\mathcal{F}(t_i) = \{t_j | t_j \in \mathcal{S},\ j > i\}$.

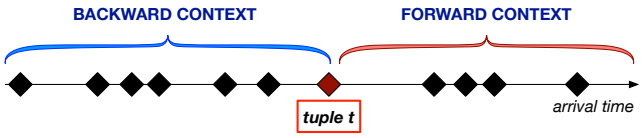


Figure 1: Representation of the backward and forward context with respect to a given tuple t.

An interesting categorization proposed in [13] is based on the type of context needed to build the mapping function. A windowing model is called *Forward Context Free* (FCF) if the mapping function does not need to access the forward context of tuples. This means that the window mapping of a tuple t can be specified by accessing to the information contained in the tuple attributes and, eventually, in the past tuples received before t. Instead, when the mapping function needs to access the forward context of tuples, the model is called *Forward Context Aware* (FCA).

The previous models can be classified in these two broad classes that have an important impact on the parallelization paradigms of windowed operators (see the next section). For the sake of brevity we will focus in detail on count-based and slide-by-tuple windows in order to exemplify the reasoning that we can apply to other models.

Definition 2 (mapping of count-based windows)**.** The mapping function of count-based windows with range and slide W and S maps a tuple $t_i \in \mathcal{S}$ onto window identifiers in the integer interval $[first\,..\,last]$, where:

$$first = \begin{cases} 0 & \text{if } i < W \\ \left\lceil \dfrac{(i - W + 1)}{S} \right\rceil & \text{if } i \geq W \end{cases} \tag{5}$$

$$last = \begin{cases} 0 & \text{if } i < S \\ \left\lfloor \dfrac{i}{S} \right\rfloor & \text{if } i \geq S \end{cases} \tag{6}$$

Proposition 1. The count-based windowing model is Forward Context Free.

The proof of the previous proposition is straightforward. According to Definition 2 the $first$ and the $last$ identifiers are expressed as a function of W and S (range and slide) that are fixed parameters of the window specification provided by the user. Furthermore, the function needs to access the identifier of the tuple and no information of future tuples is required. To be more precise, this windowing model belongs to a sub-class of FCF windows that are called *Context Free*, since only information related to the attributes of the considered tuple t is needed.

The same reasoning can be applied to the time-based windowing model which is also FCF.

Definition 3 (mapping of slide-by-tuple windows)**.** The mapping function of slide-by-tuple windows with range W (in time units) and slide S (in number of tuples) maps a tuple $t_i \in \mathcal{S}$ onto window identifiers in the integer interval $[first\,..\,last]$. We call a tuple t_i a triggering tuple if $(i+1)$ mod $S = 0$. If t_i is a triggering tuple, the first window extent in which it is contained is the one triggered by the arrival of this tuple, i.e. $first = (i+1)/S - 1$. Consider instead the case t_i is not a triggering tuple. Let $t_j \in \mathcal{S}$ the first triggering tuple after t_i, i.e. j is the smallest integer greater than i such that $(j+1) \mod S = 0$. In this case the $first$ identifier can be computed as follows:

$$first = \begin{cases} (j+1)/S - 1 & \text{if } \tau(t_i) > \tau(t_j) - W \\ \perp & \text{otherwise} \end{cases} \tag{7}$$

We use the symbol $\perp$ to refer to the case where the $first$ identifier does not exist, i.e. *the tuple t_i does not belong to any window*. Fig. 2 exemplifies two cases in order to understand the problem. Tuple t_r is the first triggering tuple after tuples t_i and t_j with $i < j$. As we can see, the first window extent that contains t_j is the one of the window triggered by t_r while t_i is not included in any window, since when the next triggering tuple t_r is received t_i is too old to be included in the triggered window.

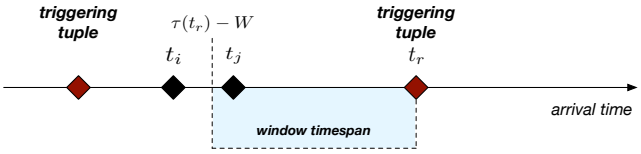


Figure 2: Example of slide-by-tuple windows with tuples included in at least one window extent and tuples that do not belong to any window extent.

A similar reasoning can be applied to find the identifier of $last$. Let t_k be the last triggering tuple that we will receive after t_i such that $\tau(t_i) > \tau(t_k) - W$. We have:

$$last = \begin{cases} (k+1)/S - 1 & \text{if } t_k \in \mathcal{S} \\ \perp & \text{otherwise} \end{cases} \tag{8}$$

If the tuple t_k does not exist, this means that no future tuple will trigger a window containing t_i, and in that case both $first$ and $last$ are $\perp$.

Proposition 2. The slide-by-tuple windowing model is Forward Context Aware.

Also for slide-by-tuple windows the proof is straightforward. To determine $first$ and $last$ we need to access to the timestamp value of tuples t_j and t_k that belong to the forward context of t_i, i.e. $t_j, t_k \in \mathcal{F}(t_i)$.

2.3. A Taxonomy of Windowing Models

It is interesting to classify the various windowing models adopted by the existing SPEs based on the context properties discussed before. Without claiming to be exhaustive, Fig. 3 shows a classification of the most common models in terms of their FCF and FCA properties. In the FCF class we have the count-based model, which is commonly adopted for applications with highly variable incoming rates (e.g., financial applications) that need a bounded memory occupation for the query processing. The time-based model is another example of FCF windows which are often easier to understand for the final users, as they can relate the windows evolution directly to time (this is useful for business applications like in e-commerce and on-line auction systems). The use of session windows [21] emphasizes the variability of the window extents that are defined based on the frequency of input tuples. Session windows are disjoint (not-overlapped) and their extents consist in a set of consecutive input tuples whose time gap (between two consecutive tuples) does not exceed a defined inactivity gap parameter. They are used when the input stream is highly irregularly distributed over the time and are useful to model the clients' behavior like in network monitoring applications [21].

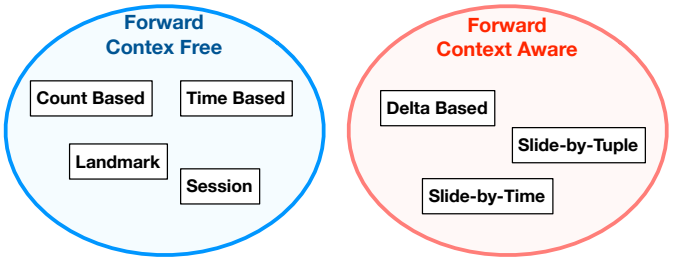


Figure 3: Categorization of common windowing models.

Hybrid models like slide-by-tuple windows and the dual *slide-by-time* (i.e. windows that slide every S time units each with a length of W tuples) are FCA models useful to combine the properties of the "pure" models described before, e.g., the user can configure the sliding factor in time units (i.e. by controlling the desired number of windows triggered per second) while the window length is expressed in terms of tuples, that is with a bounded memory occupation and a fixed cardinality.

3. Parallel Paradigms for Windowed Operators

The common approach to parallelize operators is based on *data parallelism* [10]. The idea is to replicate the operator multiple times (according to a certain *parallelism degree*) and to split the input stream by forwarding each tuple to a selected replica. In this way the same computation is applied in parallel on different stream elements.

Data parallelism must be carefully applied in case of stateful operators since the ordering of the tuples affects the computation semantics. A widely used data-parallel paradigm is called *partitioned-stateful parallelism* [18]. The precondition is that the input stream conveys tuples belonging to multiple multiplexed logical sub-streams.

Given a function $\mathcal{H}$ that maps each tuple onto a sub-stream identifier (called *key*), the assumption is that the processing of tuples of different sub-streams can be performed in parallel by different replicas, while tuples with the same key must be processed serially by the same replica. This paradigm can be expressed in most of the SPEs. In Apache Storm an operator can have multiple replicas and tuples can be forwarded using the `fieldsGrouping` or `partialKeyGrouping` scheduling. In FastFlow the paradigm can be expressed with the *Key Partitioning* pattern [22] by providing the operator to be replicated and the function $\mathcal{H}$ during the pattern construction.

This paradigm suffers from some limitations. First, load balancing is negatively influenced in case of skewed key distributions and the maximum parallelism is limited by the number of existing keys. Furthermore, this paradigm does not exploit the definition of sliding windows. We observe that sliding windows represent a very special kind of state corresponding to a "partial view" of the stream, that is a segment of the input tuples logically grouped together and that can be processed independently. This can be exploited in a new data-parallel paradigm called *sliding-window parallelism*, where the idea is to execute in parallel different windows independently of whether they belong to the same key or not. Therefore, parallelism is not limited by the number of keys but the paradigm is also effective in case of streams without keys, i.e. when all the tuples belong to the same logical stream.

This new paradigm can be instantiated as a parallel streaming program composed of several concurrent entities. The custom distribution logic of inputs to the replicas is executed by the *Emitter* entity. In Apache Storm this can be implemented by extending the `CustomStreamGrouping` interface while in FastFlow the emitter can be explicitly programmed by rewriting proper virtual methods of the `ffnode` class. Each replica is executed by a *Worker* entity which has all the processing capabilities of the original operator. Finally, a *Collector* is responsible for gathering the results of the computed windows and to emit them in the output stream in increasing order of the window identifier. We distinguish two conceptual models that represent different implementation strategies of the paradigm. The distinction depends on which concurrent entities are involved in the following activities: data distribution, window triggering and tuples eviction policies, and the processing of triggered windows.

In the *agnostic worker model* depicted in Fig. 4 the workers are not aware of the sliding window semantics nor of the fact that the operator function is applied to partially overlapped segments of the input stream. The windowing logic is *centralized* in the emitter, which is responsible for storing the most recent tuples in the so-called *window buffer* and to determine when a new window is triggered.

The workers do the processing on different windows in parallel, therefore they must be able to read the extents of the windows to compute which are dispatched by the emitter (with different implementation choices as we will see in Sect. 4). So, the workers are anonymous and agnostic of the window management, i.e. they are just in charge of applying the computation on the received window extents. The distribution by the emitter can be performed using load balancing policies to exploit at best the workers processing capabilities such as by using an on-demand distribution or any other load-aware strategy.

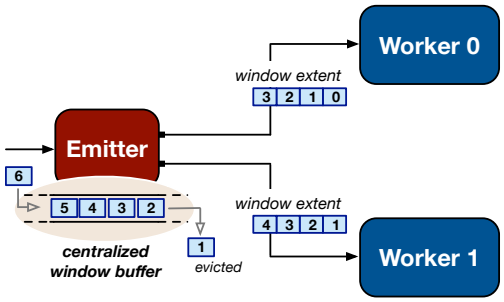


Figure 4: Agnostic model: example with count-based sliding windows with $W = 4$ and $S = 1$ tuples and two workers.

An alternative approach is based on the *active worker model* depicted in Fig. 5, which is a fully distributed implementation of the paradigm since all the window management activities are now entirely delegated to the workers. They receive from the emitter single tuples and maintain *private* window buffers where the received tuples are stored and evicted according to the sliding window specification.

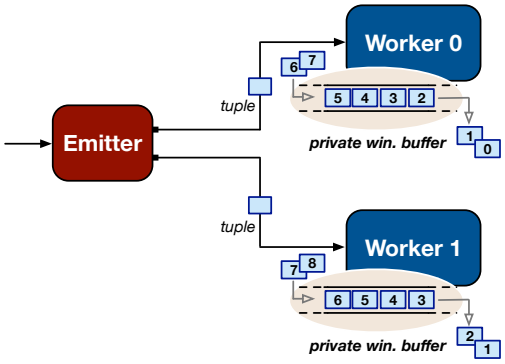


Figure 5: Active model: example with count-based sliding windows with $W = 4$ and $S = 1$ tuples and two workers.

In order to increase the query throughput, the workers must apply the processing on different windows. The assignment is the *round-robin* one, i.e. window $i \in W_{id}$ is assigned to worker $j = i \mod N$, with N the parallelism degree. This has two consequences exemplified in Fig. 6:

- each worker receives only the tuples belonging to the extents of the assigned windows. For example, in the figure the worker with identifier one should not receive tuple t_0 since this belongs only to the extent of window win_0 which is not assigned to that worker. Instead, the same tuple must be scheduled to worker

zero which is responsible for processing that window. In general, the same tuple can be multicasted to a subset of the workers. For instance, in the figure the tuple t_1 is transmitted to both the workers;
- to correctly process all the assigned windows the workers use proper private sliding window parameters. In the example the operator sliding window specification is $W = 4$ and $S = 1$ tuples. Instead, each worker uses a private sliding parameter of $S_w = 2$. In general this parameter is determined using the formula $S_w = \min\{W, S \cdot N\}$.

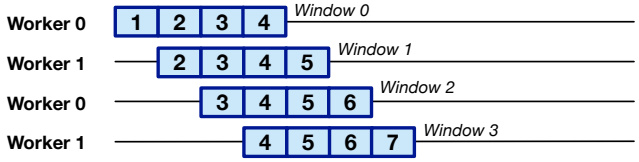


Figure 6: Example of active worker model: count-based sliding windows with $W = 4$ and $S = 1$ tuples and two workers.

3.1. Discussion

In this part we will discuss the pros and cons of the two models. They are summarized in Tab. 1.

In the active worker model the emitter must be able to forward tuples to the workers on-the-fly according to their mapping on window extents. Upon receiving a tuple $t \in \mathcal{S}$, the emitter executes the following steps: *1)* the window mapping function is evaluated in order to determine all the window extents that contain t; *2)* the tuple is forwarded to all the workers w such that there exists a window identifier $j \in \mathcal{M}(t)$ such that $w = j \mod N$. This means that the mapping function must be computed by the emitter using the knowledge available at the time tuple t is received. As discussed in Sect. 2.2, this is possible in FCF windowing models only. Instead, in FCA models the identification of the window extents containing t depends on the properties of future tuples that will be received after t, thus the emitter is not able to determine completely which are the window extents containing t. The agnostic model does not suffer from this constraint since the windowing logic is centralized and executed by the emitter without needing the on-the-fly distribution of tuples among the workers.

Another important aspect is the type of query supported. We distinguish between:

- *non-incremental queries* are queries whose processing function takes as an input argument the whole window extent (set of tuples) and applies the processing on the whole set to produce the window result;
- *incremental queries* are expressed by an online processing algorithm that incrementally updates the window result as soon as a new tuple belonging to the window extent is ready to compute.

The active worker model can be adopted for both the query types. The active workers either buffer all the tuples of their assigned window extents and start the processing when an extent is complete (non-incremental case), or they update incrementally the results of their assigned windows as soon as a new tuple is scheduled by the emitter. Since in this model each worker knows which windows are statically assigned to it (i.e. according to the private sliding parameter S_w), optimizations are possible. As an example, the active workers can independently adopt the pane-based approach [23], which logically divides each window in disjoint panes such that intermediate results of panes in common to consecutive windows of that worker can be reused to save computation time.

The agnostic model with FCF windows allows using both non-incremental queries and incremental queries. The first case is the one described in the previous section: the emitter schedules to the workers whole window extents to compute. In case of an incremental query the model can be easily extended: the emitter is now in charge of scheduling to the workers (on-demand) messages containing the partial result of a window and the tuple used to update such result. Instead, for the FCA windowing semantics, the agnostic model is the unique solution and the query processing must be expressed necessarily in a non-incremental fashion. To summarize, Fig. 7 shows a flowchart describing the selection between the two models

	Active Model	**Agnostic Model**
Pros	**1.** Suitable for *incremental* and *non-incremental* queries. **2.** Window buffer implemented as a *private* data structure per worker. **3.** Optimizations exploiting *computation sharing* among consecutive windows are possible.	**1.** Suitable for both *FCF* and *FCA* windowing models. **2.** *Load balancing* can be easily implemented with an on-demand distribution. **3.** Tuples are never replicated among workers, thus allowing *memory saving*.
Cons	**1.** Suitable for *FCF* windowing models only. **2.** Tuples are *replicated* in the private window buffers of the workers. **3.** Potential *load imbalance* in case of window extents with different cardinalities (due to the static window assignment).	**1.** A *concurrent* implementation of the centralized window buffer is needed. **2.** *Incremental* queries are supported only if the underlying windowing model is FCF. **3.** No easy chance to exploit *computation sharing* among consecutive windows.

Table 1: Pros and cons of the two implementation models of the sliding-window parallelism paradigm.

in different scenarios of windowing model and query type.

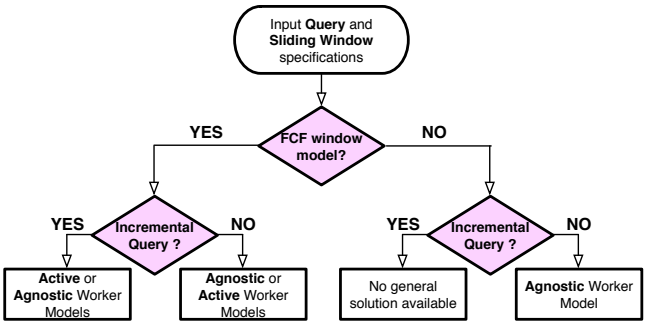


Figure 7: Worker models supporting different windowing semantics and query types.

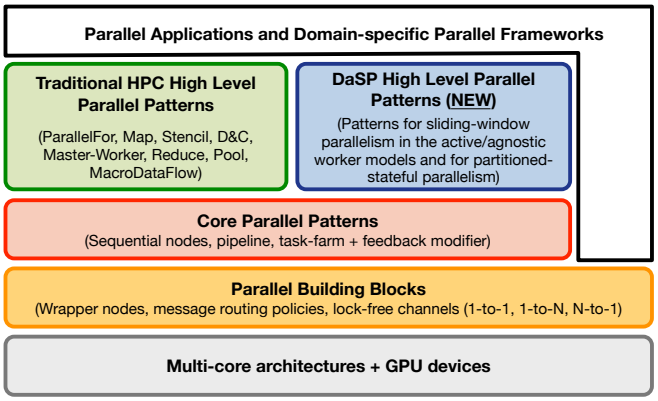


Figure 8: Layered FastFlow design: the blue box is the part of the third layer that will made available in the next release.

Other aspects of the parallelization are *load balancing* and *memory usage*, which are both in favor of the agnostic model. Windows are distributed to workers by balancing the workload also in case of extents with significantly different cardinalities. Furthermore, efficient implementations of the agnostic worker model (as discussed in the next section) should avoid copying the window extents to the workers for non-incremental queries, but the emitter distributes special meta-data that make the worker able to access the right set of tuples. Although this avoids replication of tuples (as in the active model) it introduces a higher implementation complexity since the internal windowing structures are now accessed concurrently by the emitter and the workers threads. Efficient solutions for this problem will be presented in the next section.

4. Implementation on Multicores

In this section we describe the implementation of the two alternative models of the sliding-window parallelism paradigm. Before presenting the implementations, we first provide a brief overview of the FastFlow environment for efficient data streaming on multicores, which is the framework that we used in this paper.

4.1. The FastFlow Framework

FastFlow is an open-source, structured parallel programming framework supporting highly efficient stream parallel computation on heterogeneous multi-core platforms [15]. It is realized as a C++11 header-only template library that allows the programmer to simplify the development of parallel applications modeled as directed graphs of processing nodes (operators). FastFlow provides a set of ready-to-use, parametric algorithmic skeletons modeling the most common parallelism exploitation patterns, which may be freely nested to model arbitrarily complex graph topologies. Its design is layered as shown in Fig. 8.

The lower layer, called *Parallel Building Blocks*, provides wrapper components, i.e. a control flow realized with POSIX threads; a set of communication channels realized with lock-free single-producer/single-consumer FIFO queues [24], and a set of policies for routing/gathering messages exchanged between different nodes. Above these mechanisms, the second layer (called *Core Parallel Patterns*) provides the FastFlow node abstraction (*ff_node*), i.e. the basic unit of parallelism that encapsulates the user's business logic code, and the most well-known streaming parallel patterns such as the *pipeline* (*ff_pipe*) to build tandem networks of operators and the *task-farm* (*ff_farm*) to replicate the same node. At this level, a pattern modifier called *feedback* can be used to build cyclic streaming networks.

The third layer (called *High-Level Parallel Patterns*) provides specialized patterns built on top of the two lower layers. The patterns are: *pipeline*, *farm*, *map*, *map+reduce*, *stencil*, *master-worker*, *Divide&Conquer* and *PoolEvolution*. For the purpose of this paper, the patterns for modeling sliding-window parallelism (both the agnostic and the active models) have been implemented on top of the Core Parallel Patterns level and they will be released as patterns of the third level in the next FastFlow release (the blue box in Fig. 8).

Each node is used to run a concurrent activity in a single sequential component (by default it is mapped onto one runtime thread), and it has associated two channels: one used to receive input data (pointers to data) to be processed and one to deliver the (pointers to the) results. The key idea underneath the FastFlow model is that ease-of-development and runtime efficiency can both be achieved by raising the level of abstraction of the design phase.

From the programmer viewpoint, the FastFlow parallel patterns can be used by instantiating proper objects from the FastFlow classes and by implementing precise class methods: the *svc* method that encapsulates the computation to be performed on each input datum to obtain the output result; the *svc_init* and *svc_end* methods that are executed once when the application is started and before it is terminated. Only the *svc* method must be provided by the programmer in order to instantiate a FastFlow pat-

tern. In the Listing 1 we reported a snippet of a FastFlow code implementing a three-staged pipeline:

Listing 1: A FastFlow three stage pipeline example.

```
struct stageA:ff_node_t<Type1, Type2> {
   int     svc_init() { ....; return 0;}
   OUT_t *svc(IN_t *in) { return FA(in); }
};
struct stageB:ff_node_t<Type2, Type3> {
   OUT_t *svc(IN_t *in) { return FB(in); }
};
struct stageC:ff_node_t<Type3, Type4> {
   OUT_t *svc(IN_t *in) { return FC(in); }
   void   svc_end() { ... };
};
ff_Pipe<> pipe(stageA, stageB, stageC);
pipe.run_and_wait_end();
```

The *task-farm* pattern has been extensively utilized and customized for implementing the active and the agnostic worker model versions of the sliding-window parallelism paradigm. This has been possible because pattern allows the programmer to customize the behavior of the scheduling node (emitter) and of the gathering node (collector) while any node can be passed as an input parameter of the pattern constructor and indicates the operator to be replicated (the replicas are called workers).

4.2. Implementation of the Active Worker Model

As discussed in Sect. 3, the active worker model is valid for FCF windowing models only, and consists in the pre-assignment of consecutive windows to the workers according to a round-robin policy. Tuples are scheduled and possibly multicasted to the workers by evaluating the window mapping function $\mathcal{M}$ on-the-fly at each new tuple arrival. Each worker is in charge of managing a *private* window buffer and executes its own insertion and eviction actions based on the window range W and the private sliding S_w parameter. Since the window buffer is private, no consistency and correctness problem arises in its management from the concurrency perspective because just one thread will access such data structure (i.e. the one executing the corresponding worker).

The emitter algorithm pseudo-code is reported in Alg. 1. When the emitter receives the tuple t, it evaluates the mapping function to determine which are the window extents containing the tuple t. Then, according to the round-robin assignment, the emitter determines which workers need to receive the tuple. Then, the emitter multicast the input tuple to those workers. It is worth noting that in the FastFlow runtime only a memory pointer to the tuple structure is communicated to the destination workers.

As we can observe, the evaluation of the window mapping function is performed by calling the `IdentifyWindowIndexes` routine at line 3. The *fist* and the *last* indexes are computed as described in Sect. 2.2 for count-based and time-based windows. The pseudo-code is exemplified by passing the timestamp of the tuple to the routine (in case of count-based windows we use the tuple identifier). Again, the *fist* and the *last* indexes must be computed explicitly by the emitter and this must be performed using the knowledge available at the time tuple t is received (without information about the timestamps of the future tuples), implying that a FCF model must be used to adopt the active worker model.

Algorithm 1: Emitter algorithm in the Active Worker Model

Input: an input tuple t
Result: tuple t is dispatched to *all* the workers needing it

```
1: procedure SVC(t)
2:   ▷ Return the first, last identifiers of the windows containing t
3:   (first, last) ← IdentifyWindowIndexes(τ(t), W, S)
4:   ▷ Determine the destination workers
5:   Workers ← ToWorkers(t, first, last)
6:   for each Worker_k ∈ Workers do
7:     send t to Worker_k
```

The worker algorithm pseudo-code is shown in Alg. 2. The worker receives the tuple t from the emitter, it copies t in its private window buffer and, based on the window specification, it determines whether the tuple is a triggering tuple, i.e. one or more window extents can be complete and the query can process the window tuples.

Algorithm 2: Worker algorithm in the Active Worker Model

Input: an input tuple t
Result: an output result for each triggered window (none if no window is triggered)

```
1: procedure SVC(t)
2:   ▷ Insert the tuple in the private window buffer
3:   insert(winBuf, t)
4:   ▷ Determine the window extents that are complete
5:   winsTriggered ← checkWinsTriggered(winBuf, τ(t))
6:   for each w ∈ winsTriggered do
7:     result ← computeFunction(winBuf, w)   ▷ user function call
8:     send result to Collector              ▷ produce the result
9:   purgeExpiredTuples(winBuf, ...)
```

At line 3 the worker inserts the tuple into its private window buffer called `winBuf`. The buffer is implemented as a dynamic container (as the ones available in the C++ STL library or using customized containers). Then, the worker determines whether some window extents are now complete. For the time-based windowing model the tuple timestamp is used to check this, see line 5. For each triggered window the user function `computeFunction` is called by passing the window buffer and the index of the triggered window as input parameter. The index is used by the run-time system to pass to the user a C++ iterator to the tuples that belong to the extent of the window to compute. Finally, the results are sent to the collector and the window buffer is purged of the expired tuples (line 9).

The collector (pseudo-code reported in Alg. 3) collects the results produced by the workers in a first-come-first-served discipline. To produce the results in output in the proper order (i.e. according to the window identifiers), the collector maintains an ordered data structure (a priority queue called `resultsQueue`) where it inserts the incoming

results and extracts them in the output order.

Algorithm 3: Collector algorithm in both the models

Input: a *result* from one of the workers
Result: zero, one or more results are transmitted in output

```
1: procedure SVC(result)
2:   insert(resultsQueue, result)
3:   r ← extractNextInOrder(resultsQueue)
4:   while r ≠ nil do
5:     send r to the next stage
6:     r ← extractNextInOrder(resultsQueue)
```

4.3. Implementation of the Agnostic Worker Model

We recall that in the agnostic worker model each worker is anonymous and unaware of the window semantics. The workers are responsible for applying the user's function to the received window data and for producing a result for each window. The entire windowing management is in charge of the emitter, which: *i)* stores the most recent tuples; *ii)* determines when a new window is ready to be computed, and *iii)* which tuples must be evicted.

When all the tuples belonging to a given window are received by the emitter, it delegates the task of computing the user function on the window extent to an available worker. Copying the whole window extent in a private worker buffer is too costly and infeasible for performance reasons. Therefore, the emitter sends to the worker a structure containing the meta-data for accessing the window content. To allow the access to the right tuples of the window extent, all the input tuples received by the emitter are stored in a *concurrent* data structure shared between the emitter itself and the pool of workers.

The design of such data structure is critical for implementing the correct semantics of the computation and for minimizing the runtime overhead for reading and writing the data. The objective is to design it in a way that concurrent read/write accesses do not hinder the performance of the system. By leveraging the specific access pattern of the window buffer and the information gathered at runtime regarding which parts of the data structure is currently accessed, we are able to avoid costly synchronization primitives to protect concurrent accesses.

From the emitter perspective, the data structure is logically a *queue*: input tuples are added to the tail of the queue, instead expired tuples are removed from the head of the queue. On the worker side, the data structure has to appear as a contiguous sequence of tuples where it reads all the tuples corresponding to the assigned window extent. The expiring of a tuple and its eviction from the queue can in principle produce a read/write conflict between the emitter and the workers. To clarify things, consider the queue implemented as a dynamic container whose internal representation can be occasionally reallocated for growth. A concurrent access by a thread (emitter) that adds/removes elements while other threads (workers) read some portions of the queue is not thread-safe in general (e.g., in case of standard C++ STL containers).

Such potential conflicts are efficiently avoided by our `CC-WBuf` data structure (*Concurrent Chunk-based Window Buffer*) internally maintained and modified by the emitter and read by the workers. The first property that we exploit is that the emitter can evict a tuple only if all the windows whose extents contain that tuple have been already computed by the workers. This is implemented in FastFlow by using the *feedback* modifier applied to a *task-farm* pattern. Essentially, it allows workers to send back messages to the emitter as shown in Fig. 9. As soon as the workers complete the processing on a sliding window, a message is sent to the emitter via the feedback channels in order to inform it of the window identifier that has been computed. These feedback messages are used by the emitter for two reasons:

1. to know which windows have been elaborated in order to evict from the `CC-WBuf` those tuples that will not belong to any future window;
2. to provide load balancing among workers by scheduling triggered windows to ready workers, i.e. those workers that have completed the computation on a window and that have already sent the feedback.

The `CC-WBuf` structure is organized as a list of *chunks* as shown in Fig. 9. In the general case, each chunk contains a fixed number of tuples (this value denoted by $\mathcal{C}_{size}$ is a configuration parameter of the system). Chunks are static containers (e.g., arrays) allocated and released disjointly. The emitter adds new tuples to the first (pointed by the *tail* pointer) chunk until there is enough space. When the tail chunk is filled out, the emitter allocates another chunk moving the tail pointer to the new chunk. Each sliding window shares a group of chunks with the previous and next windows. When a window is triggered, the next window slides forward of a number of tuples according to the sliding parameter S. When the window slides, zero, one or more chunks slide forward as well, and so they can be considered for removal as soon as proper feedback messages are received from the workers.

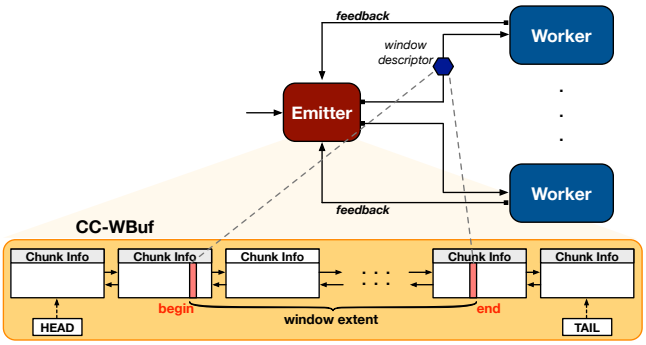


Figure 9: `CC-WBuf` data structure shared between the emitter and the workers in the agnostic worker model.

The workers receive from the emitter a data structure (*window descriptor*) containing the information for access-

ing the window extent to elaborate. In particular, the structure contains a *window iterator* that allows accessing to all (and only) the tuples belonging to the window that has to be computed. It is used by the user function to scan the data structure by providing a logical view as a contiguous sequence of tuples. The signature of the user function becomes something very high level like:

```
result_t computeFunction(Win_Iterator begin,
    Win_Iterator end, size_t w_id, ...);
```

Therefore, from the workers viewpoint the internal data layout of the `CC-WBuf` structure is completely transparent and they are unaware of the fact that the tuples belonging to that assigned window might span several disjoint chunks. The window descriptor also includes information such as the window identifier, the starting timestamp (for time-based windows) of the window and other information used to check the state of the window.

The emitter algorithm pseudo-code is reported in Alg. 4 for time-based windows. It non-deterministically receives two different types of input messages: a tuple (t) or a feedback message sent by one of the workers (fb) which notifies the end of the elaboration of a window and provides information for the expiring phase (described later).

At the arrival of a new tuple t (**Case 1** in Alg. 1), the emitter inserts t in the tail chunk or allocates a new chunk (line 3). Then, according to the sliding window semantics, it checks whether new windows are triggered at line 4. The emitter maintains for each worker a flag stating whether it is ready to process a window or not (initially all the workers are ready). For each triggered window (none, one or many) the emitter looks for a ready worker (line 6) and in case it exists the window descriptor is created and sent to it (the `send` call automatically unsets the ready flag of the worker). Otherwise, the identifier of the triggered window is stored in a data structure `readyWindows` at line 11.

In case the emitter receives a feedback message from a worker (**Case 2** in Alg. 1), it extracts from the message the identifier of the sending worker (line 13) and tries to pop the identifier of a previously triggered window that has not been assigned to a worker yet (line 14). If such window exists, a new window descriptor is created and sent to that worker. Otherwise, the worker is set as ready again at line 19.

Then, the emitter handles the expiring phase which must be performed carefully since some chunks from the head must be released when all the tuples contained are no longer necessary for the workers computation. The specific expiring logic depends on the sliding window semantics. In the following we describe the case of time-based sliding windows shown in Alg. 4.

We associate with each chunk additional information related to the smallest and the highest timestamps of the tuples that it contains. Furthermore, each feedback message conveys the starting timestamp of the computed window (shortly called *window timestamp*). If that timestamp is greater than the highest timestamp of the head chunk, the

Algorithm 4: Emitter algorithm in the Agnostic Worker Model

Input: a tuple t or a feedback message fb from a worker
Result: window descriptors W_d are dispatched to ready workers

```
1:  procedure SVC(t or fb)
2:    if t is a tuple then                                   ▷ Case 1
3:      insert(ccwinBuf, t)
4:      winsTriggered ← checkWinsTriggered(ccwinBuf, τ(t))
5:      for each w ∈ winsTriggered do
6:        Worker_i ← getReadyWorker() ▷ get a ready worker, if any
7:        if Worker_i ≠ nil then
8:          W_d ← prepareWindowMetadata(ccwinBuf, w)
9:          send W_d to Worker_i
10:       else
11:         add(readyWindows, w)        ▷ store that window w is ready
12:   if t is a feedback then                                ▷ Case 2
13:     worker_fb ← getWorker(fb)
14:     w ← get(readyWindows)      ▷ select a ready window, if any
15:     if w ≠ nil then
16:       W_d ← prepareWindowMetadata(ccwinBuf, w)
17:       send W_d to worker_fb
18:     else
19:       setReadyWorker(worker_fb)
20:     ▷ Expiring phase
21:     ts ← getWindowTimestamp(fb)
22:     insert(winTsQueue, ts)
23:     ts ← nextWinTs(winTsQueue)
24:     while minWinTs == ts do
25:       ts ← nextWinTs(winTsQueue)
26:       minWinTs ← minWinTs + S
27:     purgeExpiredTuples(ccwinBuf, minWinTs)
```

chunk can be released and the same reasoning can be applied to the next chunk that becomes the new head and so forth. This technique has a subtle pitfall. When there are more workers running it may be possible that feedback messages could be received out-of-order with respect to the window order. To avoid this problem, the emitter stores and re-orders all the window timestamps received from the workers in a priority queue (`winTsQueue` in the pseudo-code) and maintains the the smallest timestamp of the window that is currently under computation (in the `minWinTs` variable). When the window timestamp of the received feedback message is equal to `minWinTs` (line 24), we can start the expiring operations: *i)* we first check whether in `winTsQueue` there are some window timestamps contiguous to the current value of `minWinTs`[1] and if yes we update `minWinTs` with the greater value we have found; *ii)* on the base of the updated value of `minWinTs` computed at the previous point, we remove all the chunks with highest timestamp smaller than `minWinTs`. When chunks are evicted from the `CC-WBuf` (line 27), all the contained tuples are deleted.

4.3.1. Optimized layout for time-based windows

The size $\mathcal{C}_{size}$ of the chunks is an important parameter that can affect the performance of the parallelization. The smaller the chunk size the higher the number of chunks

[1] A window timestamp ts precedes ts' if $ts + S = ts'$ where S is the slide parameter.

used for storing each window extent. A small chunk size may increase substantially the cost of iterating across all the tuples. In contrast, the larger the chunk size the higher the amount of memory used by the system since chunks are evicted less frequently. The impact of the fragmentation on the performance will be analyzed in Sect. 5.

Therefore, the choice of $\mathcal{C}_{size}$ is an important problem whose optimal value may depend on run-time parameters of which the user can be aware of (e.g., the average arrival rate of the input stream). In this part we propose an optimization of the `CC-WBuf` layout for time-based windows which implicitly extracts the chunk-based organization from the window specification.

Fig. 9 shows the case where chunks have a static fixed size in terms of tuples while they span over a different time range (depending on the arrival rate). Another approach consists in associating to each chunk a fixed temporal length while the size in tuples may be different between chunks. The idea is to choose the time length of each chunk as $\mathcal{C}_{len} = GCD(W, S)$. Fig. 10 shows an example with windows of $W = 6$ seconds sliding every $S = 2$ seconds (chunks of 2 seconds). Each chunk is implemented as a dynamic container that can be dynamically resized.

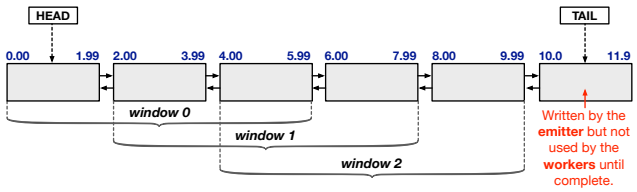


Figure 10: Layout of the Concurrent Chunk-based Window Buffer for time-based sliding windows.

This layout has an important implication that makes it very effective and flexible. Each window extent will span in exactly $W/\mathcal{C}_{len}$ chunks, i.e. a chunk cannot contain both the last tuple of a window extent and the first one of the next window. Consequently, read/write conflicts on the same chunk cannot happen because the workers only read chunks that are *complete*, i.e. the emitter never modifies those chunks by adding new tuples. Instead, the tail chunk modified by the emitter is never read by the workers before it is complete (i.e. all the tuples in its time range has been received). Consequently, standard dynamic containers (like the ones in the C++ STL library) can be employed to implement the chunk internals.

5. Experiments

In this part, we first describe the application use case used for the tests, then we analyze how the active and agnostic worker models behave in various conditions.

All tests have been executed on an Intel Xeon Server equipped with two Intel E5-2695 Ivy Bridge CPUs running at 2.40GHz and featuring 24 cores (12 per socket). Each hyper-threaded core has 32KB private L1, 256KB private L2 and 30MB of L3 shared cache. The machine has 64GB of DDR3 RAM, running Linux 3.14.49 x86_64. The compiler used is `gcc` version `4.8.5`. The code has been compiled with the -O3 optimization flag.

5.1. Application Description

The use case is a financial trading application previously described in [25]. It consists in a pipeline of three components: *1)* a *source* operator that generates the stream of data by interfacing with some external data producers; *2)* a parallel *algotrader* operator that applies a user-defined algorithm on sliding windows of the stream, and *3)* a *sink* operator that consolidates the received results in a DB. The stream conveys financial quotes, i.e. buy and sell proposals (bid and ask) represented by a record of attributes such as the proposed price, volume and the stock symbol. The query is provided with a non-incremental algorithm that estimates the future price of stock symbols by computing the Levenberg-Marquardt regression to produce a fitting polynomial. For the regression we used the regression provided by the C++ library lmfit[2].

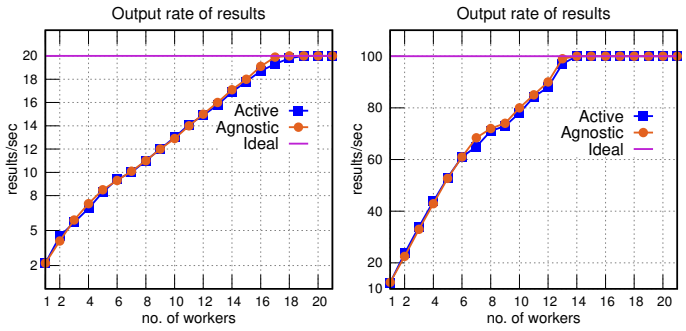
5.2. Results with Time-based Windows

In this part we consider time-based windows, for which we can adopt both the active and the agnostic worker model. The aim is to compare the two implementations by evaluating their capacity to sustain the input rate and the time required to compute a single window (latency).

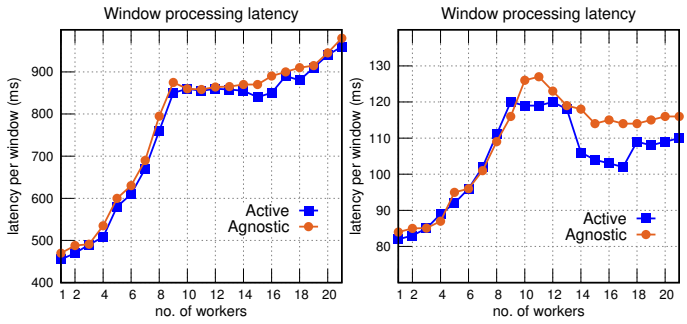
Fig. 11a shows the number of results produced per second with an input rate of $200K$ tuples/s for the active and the agnostic worker model by varying the number of workers and considering two distinct configurations: *i)* $W = 1$ second and $S = 50$ milliseconds; *ii)* $W = 1$ second and $S = 10$ milliseconds. As we can see from the figure, the two models achieve almost the same output rate and both of them are able to reach the ideal output rate with the same number of resources, i.e. 17 and 14 workers for the two configurations considered, respectively.

Fig. 11b shows the execution latency for computing one window (called $Tcalc$) for the two models in the two configurations described previously. We can see that the agnostic worker model has a higher window latency (of 3.65% on average). This is mainly due to the higher cost of accessing to the tuples in the shared `CC-WBuf` data structure used in the agnostic implementation. In fact, in the tests we used a fixed chunk size of $5,000$ tuples, therefore a window extent is stored in approximately 200 and 40 chunks for the two window sizes respectively. Those chunks have been allocated independently, so they can be released as soon as they are no longer necessary for the computation. Likely, they are not allocated contiguously in the memory and so the time for iterating across all tuples of the window is higher with respect to the active model.

[2]lmfit library: http://lmfit.github.io/lmfit-py/



(a) Output rate: W=5s, S=50ms (**left**); W=1s, S=10ms (**right**).



(b) Worker time: W=5s, S=10ms (**left**); W=1s, S=10ms (**right**).

Figure 11: Active *vs* Agnostic worker model considering two different window lengths (W) and slides (S). The input rate is 200K tuples/s, the chunk size for the agnostic model is set to 5,000 tuples.

It is worth noting that, despite the $Tcalc$ is slightly higher in the agnostic model, both the agnostic and the active worker models have the same output rate. This can be explained by the fact that the implementation of the agnostic model obtains a better workload balancing among workers (thus a better workers' utilization) because it uses an on-demand scheduling policy for the windows that is implemented exploiting the feedback information coming from the workers. Instead, the active worker model uses a round-robin scheduling that has a lower overhead but is not able to perfectly balance the workload of the workers.

Fig. 12 shows for the agnostic worker model the impact of the chunk size on the output rate (left-hand side of the plot) and on the $Tcalc$ (right-hand side of the plot). As we can see, small chunk sizes introduce higher overhead (of 10% at most) both for the output rate and for the $Tcalc$. On the contrary, small chunks allow releasing memory more quickly thus reducing the memory footprint required by the application. Since the smaller the chunk size the higher the number of fragments used to store the window extent in memory, the worker has to iterate on more non-contiguous fragments increasing the computation time and eventually lowering the output rate. However, starting from a "large enough" value (approximately from 5,000 tuples in the plot), the difference on the performance with different chunk sizes is very limited. For this reason, we decided to use a fixed value for the chunk size set to 5,000 tuples.

As discussed in 4.3.1, for time-based windows it is possible to use a fixed temporal length for the chunk size instead of a fixed number of tuples. Fig. 13 compares the performance of the agnostic worker model when the chunk size is statically fixed to 5,000 tuples and when it is automatically assigned by the runtime system to a value equal to $GCD(W, S)$ units of time (milliseconds in our case). For the configuration considered, this means that the chunk size for the automatic case is set to 10 ms, so a single chunk contains all the tuples received in a time range of 10 ms. As shown, the output rate obtained by the two versions is almost the same while $Tcalc$ is slightly lower (2.8% on average) when the chunk size is set to 5,000. In fact, for the

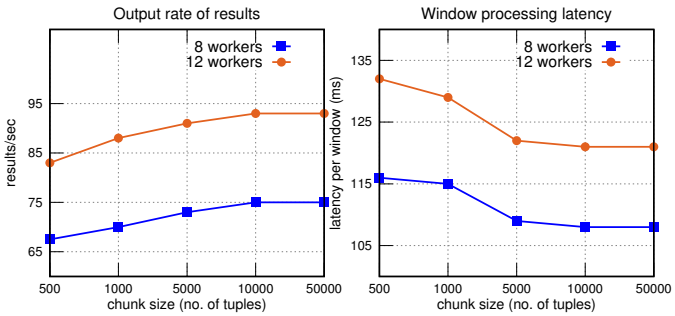


Figure 12: Impact of the chunk size on the output rate (**left**) and on the window latency (**right**) with 8 and 12 workers. Input rate 200K tuples/s, W=1s and S=10ms.

window length and slide size considered, the fixed temporal length chunk size case (automatic in the figure) stores in one chunk approximately 2,000 tuples (because of the considered input rate). So, the two values are very close and the performance obtained is quite similar. This result confirms that this optimization can be used for time-based windows providing a good tradeoff between performance and reduced implementation complexity.

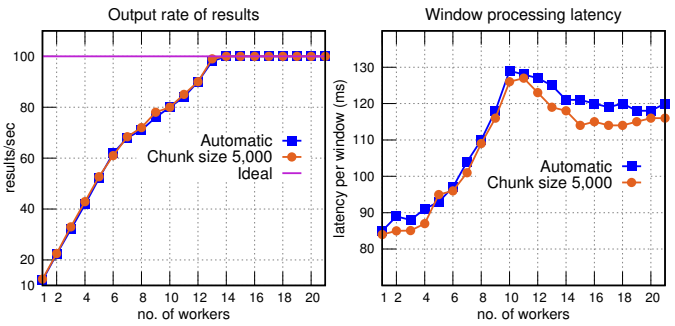


Figure 13: Static *vs* automatic assignment of the chunk size for time-based windows in the agnostic worker model. Output rate (**left**) and window latency (**right**). The input rate is 200K tuples/s, W=1s and S=10ms.

Finally, Fig. 14 shows which is the number of workers required to reach the optimal output rate considering different input rates (left-hand side of the plot) and the corre-

sponding $Tcalc$ value (right-hand side of the same figure). In the test we use time-based windows with $W = 1$ sec and $S = 25$ ms and a static chunk size of $5,000$ tuples. To sustain an input rate of $600K$ tuples/s the algotrader operator has to use all available cores of the machine (i.e. 21 workers). The $Tcalc$ value is about $600ms$. Similar results have been obtained by the active worker model with a lower value for $Tcalc$. Such results are not shown here for space constraints, however for the $600K$ tuples/s the active model has a $Tcalc$ of about 500 ms using 21 worker replicas.

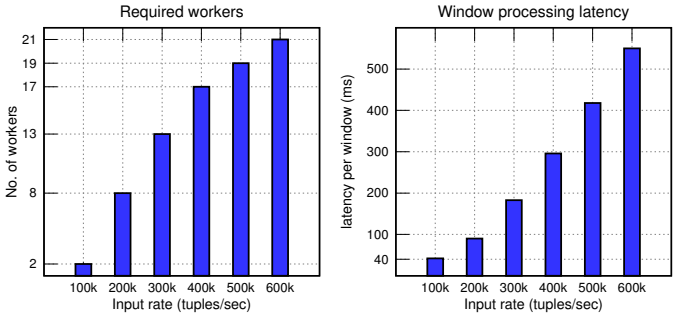


Figure 14: Number of workers required for sustaining the input rate (**left**); window latency (**right**).

Summarizing, the active and agnostic worker models have very similar performance metrics concerning the number of resources used and the output rate. In general, the agnostic worker model has a higher $Tcalc$ value due to the higher costs for accessing window's tuples.

5.3. Results with Slide-by-tuple Windows

Slide-by-tuple windows are FCA, so we can only use the agnostic worker model. Fig. 15 shows the impact of the window length on the output rate when the slide is fixed to $4,000$ tuples (approximately 20 ms with an input rate of $200K$ tuples/s). As expected, the longer the window length the more the workers needed to reach the optimal output rate. In this test, we need five workers for a window length of 1 sec and 14 workers in the case of W=2 seconds. For the case W=3 seconds, with 21 workers we are able to reach only half of the optimal output rate.

Fig. 16 presents the results obtained with W fixed to 3 seconds and the slide size varies from $4,000$ to $6,000$. As expected, with higher values of the slide fewer workers are required to sustain the input rate. Interestingly, the user can play with the values of W and S to obtain the desired level of the output rate and $Tcalc$ according to the number of resources available on the machine at hand.

6. Related Work

SPEs have been developed in order to facilitate the definition of complex topologies of operators that are run transparently over a distributed system. While this programming approach fosters inter-operator parallelism

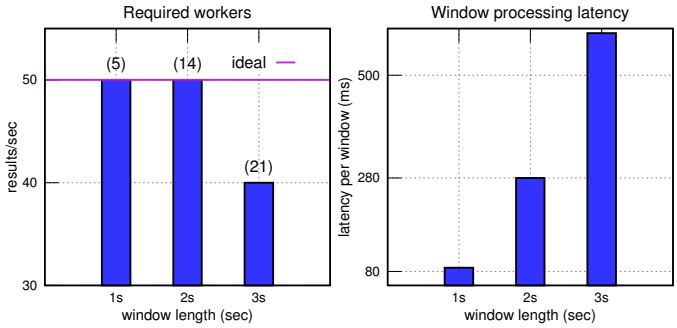


Figure 15: Impact of the window length on the output rate (**left**) and on the window latency (**right**) when the slide is fixed to $4,000$ tuples. Input rate of 200K tuples/s, chunk size of $5,000$ tuples.

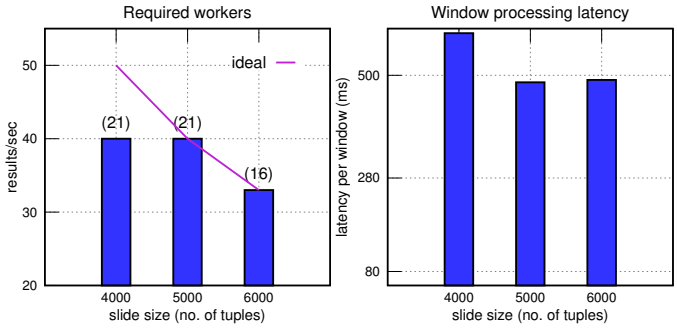


Figure 16: Impact of the slide on the output rate (**left**) and on the window latency (**right**) with window length of 3 sec. The input rate is 200K tuples/s, chunk size of $5,000$ tuples. The numbers on top of the bars are the workers used.

within a single query through *data pipelining* [26], the runtime system is responsible for scheduling the query execution in an efficient way by possibly sharing the computation of operators in common with other active queries [27].

Intra-operator parallelism is aimed at increasing the throughput via operator replication and data partitioning [28]. Expressing such kind of parallelism for stateful operators is not easy because it employs various forms of dynamic partitioning and data placement that need specific algorithms and depends on the type of internal state maintained by the operators [29]. Consequently, programming such parallel operators is cumbersome in the existing SPEs because they still provide low-level interfaces for this [30].

The most common technique to parallelize stateful operators is to leverage the existence of a partitionable state [18]. One reason for its success is that this approach decouples the scheduling logic from the type of state information maintained by the replicas. In fact, it can be used for any kind of state (e.g., sliding windows, stream sketches or synopsis). However, performance problems manifest in real applications, like load imbalance and state transfer overhead in case of elastic supports able to dynamically change the number of replicas [31, 32, 33]. The first problem has been studied by relying of efficient consistent hash functions [34] that allow good load balance

and low migration cost under skewed workloads. The second has led to the definition of seamless state migration techniques [35, 36, 25].

The type of state information plays a crucial role to find parallelization opportunities that deviate from the classic partitioned-stateful paradigm. In [28, 37] a description of the suitable techniques is provided by focusing on locking algorithms and scheduling strategies that preserve the computation correctness for different kinds of stateful computations. Although interesting such papers do not focus on the nature of the state, but are based on generic descriptions of what the computation semantics expects from the output results. In this paper, instead of looking for general optimizations, we focus on a very specific but commonplace type of state represented by sliding windows [12].

Various sliding window models have been proposed for continuous queries [12] in order to accommodate different user's requirements. A very detailed study of the window semantics in terms of information needed to build the window extents and their mapping has been proposed in [13]. However, this work does not discuss the implications of the sliding window semantics on the parallelism paradigm but focuses on the sequential execution model only. The classification of windowing models in the two broad FCF and FCA classes helps in building the bridge between the type of information needed to define the window extents and their temporal properties. This has practical implications on the implementation model of intra-operator parallelism which have not been studied before. In fact, prior work like in [22] focuses more on the type of query algorithm to execute and its properties rather than on the sliding window semantics. Recently, the work in [38] studied the problem of batch-scheduling in windowed parallel operators, where windows are assigned to replicas in batches (set of consecutive windows) to reduce the number of tuples to be multicasted to the replicas. This approach represents an interesting optimization of our active work model, however, it is limited to FCF windows for which the tuple scheduling can be easily performed on-the-fly by inspecting the timestamp fields of the tuples.

7. Conclusions and Future Work

In this paper we studied the implications of different sliding window models on parallel patterns for data stream query processing. The study demonstrated that there is a strict relationship between the sliding window semantics and the way in which the parallel processing can be performed in terms of distribution policy and in terms of which parts of the computation (i.e. window management and processing) can be executed in parallel. Starting from previous work on sliding window semantics, we proposed two models for parallel windowed operators and we developed them on the FastFlow framework targeting multicores. The implementations provide a ready-to-use and effective set of patterns for data streaming applications.

Our future goal is to study the applicability of a wide set of optimizations, like the automatic selection of the best chunk size according to the query setting (e.g., arrival rate, window length and slide parameters) and to introduce the support for the elastic scaling of the number of workers, in order to achieve desired tradeoffs between performance and resource/energy consumption.

Acknowledgments

This work has been partially supported by the EU H2020 project RePhrase (EC-RIA, ICT-2014-1).

References

[1] G. De Francisci Morales, A. Bifet, L. Khan, J. Gama, W. Fan, Iot big data stream mining, in: Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16, ACM, New York, NY, USA, 2016, pp. 2119–2120. doi:10.1145/2939672.2945385.
URL http://doi.acm.org/10.1145/2939672.2945385

[2] A. Jain, A. Nalya, Learning Storm, Packt Publishing, 2014.

[3] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, C. Moran, Ibm infosphere streams for scalable, real-time, intelligent transportation services, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 1093–1104. doi:10.1145/1807167.1807291.
URL http://doi.acm.org/10.1145/1807167.1807291

[4] Z. Nabi, Pro Spark Streaming: The Zen of Real-Time Analytics Using Apache Spark, 1st Edition, Apress, Berkely, CA, USA, 2016.

[5] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, V. Vlassov, Spanedge: Towards unifying stream processing over central and near-the-edge data centers, in: 2016 IEEE/ACM Symposium on Edge Computing (SEC), 2016, pp. 168–178. doi:10.1109/SEC.2016.17.

[6] C. Hochreiner, M. Vogler, P. Waibel, S. Dustdar, Visp: An ecosystem for elastic data stream processing for the internet of things, in: 2016 IEEE 20th International Enterprise Distributed Object Computing Conference (EDOC), 2016, pp. 1–11. doi:10.1109/EDOC.2016.7579390.

[7] A. Chianese, F. Marulli, V. Moscato, F. Piccialli, A "smart" multimedia guide for indoor contextual navigation in cultural heritage applications, in: International Conference on Indoor Positioning and Indoor Navigation, 2013, pp. 1–6. doi:10.1109/IPIN.2013.6851448.

[8] M. Hong, J. J. Jung, F. Piccialli, A. Chianese, Social recommendation service for cultural heritage, Personal and Ubiquitous Computing 21 (2) (2017) 191–201. doi:10.1007/s00779-016-0985-x.
URL https://doi.org/10.1007/s00779-016-0985-x

[9] S. Cuomo, P. D. Michele, F. Piccialli, A. Galletti, J. E. Jung, Iot-based collaborative reputation system for associating visitors and artworks in a cultural scenario, Expert Systems with Applications 79 (2017) 101 – 111. doi:http://dx.doi.org/10.1016/j.eswa.2017.02.034.
URL http://www.sciencedirect.com/science/article/pii/S0957417417301240

[10] S. Schneider, M. Hirzel, B. Gedik, K. L. Wu, Safe data parallelism for general streaming, IEEE Transactions on Computers 64 (2) (2015) 504–517. doi:10.1109/TC.2013.221.

[11] G. Mencagli, M. Vanneschi, Towards a systematic approach to the dynamic adaptation of structured parallel computations using model predictive control, Cluster Computing 17 (4) (2014) 1443–1463. doi:10.1007/s10586-014-0346-3.
URL http://dx.doi.org/10.1007/s10586-014-0346-3

[12] B. Gedik, Generic windowing support for extensible stream processing systems, Software: Practice and Experience 44 (9) (2014) 1105–1128. doi:10.1002/spe.2194.
URL http://dx.doi.org/10.1002/spe.2194

[13] J. Li, D. Maier, K. Tufte, V. Papadimos, P. A. Tucker, Semantics and evaluation techniques for window aggregates in data streams, in: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, SIGMOD '05, ACM, New York, NY, USA, 2005, pp. 311–322. doi:10.1145/1066157.1066193.
URL http://doi.acm.org/10.1145/1066157.1066193

[14] L. Chen, G. Lin, Extending sliding-window semantics over data streams, in: 2008 International Symposium on Computer Science and Computational Technology, Vol. 2, 2008, pp. 110–113. doi:10.1109/ISCSCT.2008.187.

[15] M. Danelutto, M. Torquati, Structured parallel programming with "core" fastflow, in: V. Zsók, Z. Horváth, L. Csató (Eds.), Central European Functional Programming School, Vol. 8606 of LNCS, Springer, 2015, pp. 29–75. doi:10.1007/978-3-319-15940-9_2.
URL http://dx.doi.org/10.1007/978-3-319-15940-9_2

[16] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, C. Fetzer, Quality-driven processing of sliding window aggregates over out-of-order data streams, in: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS '15, ACM, New York, NY, USA, 2015, pp. 68–79. doi:10.1145/2675743.2771828.
URL http://doi.acm.org/10.1145/2675743.2771828

[17] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, D. Maier, Out-of-order processing: A new architecture for high-performance stream systems, Proc. VLDB Endow. 1 (1) (2008) 274–288. doi:10.14778/1453856.1453890.
URL http://dx.doi.org/10.14778/1453856.1453890

[18] H. Andrade, B. Gedik, D. Turaga, Fundamentals of Stream Processing, Cambridge University Press, 2014, cambridge Books.

[19] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Optimal operator replication and placement for distributed stream processing systems, SIGMETRICS Perform. Eval. Rev. 44 (4) (2017) 11–22. doi:10.1145/3092819.3092823.
URL http://doi.acm.org/10.1145/3092819.3092823

[20] S. Borzsony, D. Kossmann, K. Stocker, The skyline operator, in: Data Engineering, 2001. Proceedings. 17th International Conference on, 2001, pp. 421–430. doi:10.1109/ICDE.2001.914855.

[21] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, S. Whittle, The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing, Proc. VLDB Endow. 8 (12) (2015) 1792–1803. doi:10.14778/2824032.2824076.
URL http://dx.doi.org/10.14778/2824032.2824076

[22] T. De Matteis, G. Mencagli, Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach, International Journal of Parallel Programming (2016) 1–20doi:10.1007/s10766-016-0413-x.

[23] J. Li, D. Maier, K. Tufte, V. Papadimos, P. A. Tucker, No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams, SIGMOD Rec. 34 (1) (2005) 39–44. doi:10.1145/1058150.1058158.
URL http://doi.acm.org/10.1145/1058150.1058158

[24] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, M. Torquati, An efficient unbounded lock-free queue for multi-core systems, in: Proceedings of the 18th International Conference on Parallel Processing, Euro-Par'12, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 662–673. doi:10.1007/978-3-642-32820-6_65.
URL http://dx.doi.org/10.1007/978-3-642-32820-6_65

[25] T. De Matteis, G. Mencagli, Proactive elasticity and energy awareness in data stream processing, J. Syst. Softw. 127 (C) (2017) 302–319. doi:10.1016/j.jss.2016.08.037.
URL https://doi.org/10.1016/j.jss.2016.08.037

[26] I.-T. A. Lee, C. E. Leiserson, T. B. Schardl, Z. Zhang, J. Sukha, On-the-fly pipeline parallelism, ACM Trans. Parallel Comput. 2 (3) (2015) 17:1–17:42. doi:10.1145/2809808.
URL http://doi.acm.org/10.1145/2809808

[27] Y. Ji, A. Nica, Z. Jerzak, G. Hackenbroich, C. Fetzer, Quality-driven disorder handling for concurrent windowed stream queries with shared operators, in: Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems, DEBS '16, ACM, New York, NY, USA, 2016, pp. 25–36. doi:10.1145/2933267.2933307.
URL http://doi.acm.org/10.1145/2933267.2933307

[28] S. Wu, V. Kumar, K.-L. Wu, B. C. Ooi, Parallelizing stateful operators in a distributed stream processing system: How, should you and how much?, in: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, DEBS '12, ACM, New York, NY, USA, 2012, pp. 278–289. doi:10.1145/2335484.2335515.
URL http://doi.acm.org/10.1145/2335484.2335515

[29] G. Mencagli, M. Vanneschi, E. Vespa, Control-theoretic adaptation strategies for autonomic reconfigurable parallel applications on cloud environments, in: 2013 International Conference on High Performance Computing Simulation (HPCS), 2013, pp. 11–18. doi:10.1109/HPCSim.2013.6641387.

[30] C. Misale, Pico: A domain-specific language for data analytics pipelines, Ph.D. thesis, Computer Science Department, University of Torino (May 2017). doi:10.5281/zenodo.579753.
URL https://iris.unito.it/retrieve/handle/2318/1633743/320170/Misale_thesis.pdf

[31] C. Bertolli, G. Mencagli, M. Vanneschi, A cost model for autonomic reconfigurations in high-performance pervasive applications, in: Proceedings of the 4th ACM International Workshop on Context-Awareness for Self-Managing Systems, CASEMANS '10, ACM, New York, NY, USA, 2010, pp. 3:20–3:29. doi:10.1145/1858367.1858370.
URL http://doi.acm.org/10.1145/1858367.1858370

[32] C. Bertolli, G. Mencagli, M. Vanneschi, Analyzing memory requirements for pervasive grid applications, in: 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 2010, pp. 297–301. doi:10.1109/PDP.2010.71.

[33] G. Mencagli, M. Vanneschi, Qos-control of structured parallel computations: A predictive control approach, in: Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science, CLOUDCOM '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 296–303. doi:10.1109/CloudCom.2011.47.
URL https://doi.org/10.1109/CloudCom.2011.47

[34] B. Gedik, Partitioning functions for stateful data parallelism in stream processing, The VLDB Journal 23 (4) (2014) 517–539. doi:10.1007/s00778-013-0335-9.
URL http://dx.doi.org/10.1007/s00778-013-0335-9

[35] B. Gedik, S. Schneider, M. Hirzel, K.-L. Wu, Elastic scaling for data stream processing, IEEE Trans. Parallel Distrib. Syst. 25 (6) (2014) 1447–1463. doi:10.1109/TPDS.2013.295.
URL http://dx.doi.org/10.1109/TPDS.2013.295

[36] C. Hochreiner, M. Vögler, S. Schulte, S. Dustdar, Elastic stream processing for the internet of things, in: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), 2016, pp. 100–107. doi:10.1109/CLOUD.2016.0023.

[37] M. Danelutto, P. Kilpatrick, G. Mencagli, M. Torquati, State access patterns in stream parallel computations, The International Journal of High Performance Computing Applications 0 (0) (0) 1094342017694134. arXiv:http://dx.doi.org/10.1177/1094342017694134, doi:10.1177/1094342017694134.
URL http://dx.doi.org/10.1177/1094342017694134

[38] R. Mayer, M. A. Tariq, K. Rothermel, Minimizing communication overhead in window-based parallel complex event processing, in: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS '17, ACM, New York, NY, USA, 2017, pp. 54–65. doi:10.1145/3093742.3093914.
URL http://doi.acm.org/10.1145/3093742.3093914