University of Pisa

and

Scuola Superiore Sant'Anna

Master Degree in Computer Science and Networking
(MCSN)

Parallel processing of sliding-window queries on multicore architecture

Candidate:
Fabio Lucattini

Supervisor:
Gabriele Mencagli

Examiner:
Marco Danelutto

A.A. 2016/2017

Contents

1 Introduction

1.1 DaSP and parallelism

In recent years our ability to produce information has been growing steadily, driven by an ever increasing computing power, communication rates and hardware/software sensor diffusion. Data are often available in the form of *continuous streams*, and the development of automated tools to gather and analyze information "on the fly" in order to extract insights and detect patterns has become a valuable opportunity. [16] Consequently, an increasing number of applications (e.g., financial trading, environmental monitoring, and network intrusion detection) need continuous processing of data flowing from several sources, in order to obtain promptly and timely responses to complex queries. This information flows are called *data streams*. A data stream is an ordered sequence of data items that can be read only once possibly using limited computing and storage capabilities. Frequently such streams need to be elaborated in *real-time*, often giving more importance to the most recent data items.

With the emergence of *data streams*, many queries need to be continuously processed over unbounded sequences of transient data received at high velocity. A common approach is to process the query under a *sliding window model* [41], where the query is frequently re-evaluated over the most recent tuples received in the last time interval (e.g., expressed in seconds or milliseconds). This periodic re-evaluation further increases the computational requirements and makes the use of parallel processing techniques on today's multicores a compelling choice to run the queries in real-time efficiently and in a scalable manner.

Data stream processing (*DaSP*) is a paradigm with a lot of patterns for queries on data streams. Our work with this thesis is aimed at studying parallel models and implementations of sliding-window queries on multicores.

1.2 Windowing and parallelism

During our research we found that in literature there exist a lot of window sliding models [13]. This is due to the fact that we cannot run a traditional query on streams because we would need to have all the data available, which is unfeasible because streams may be unbounded. *Sliding windows* are an estimated solution to this problem: the results of the query are not evaluated on the *whole* stream, but only on its most recent portion. This means that the query will produce not only single of result, but instead a new updated results every time the window of the most recent data slides forward.

In literature there exist many sliding windows implementations, starting from early works in the '90s with the development of *data stream management system* (DSMS [12]) that is systems that manage continuous data streams instead of finite relations. With the DSMS introduction, the concept of sliding windows was introduced in literature with different semantics such as windows expressed in term of number of elements, or windows defined in terms of time units or also hybrid windows. Such models have been introduced to satisfy the users desires.

1.3 Implementation recap

In this work we analyze the different types of sliding windows and we classify them in order to understand how to exploit parallelism. Our model is based on the parallel execution of different windows among different in order units, because windows are usually executed independently with each other. Different types of windows have different effect on parallelism, so our implementations must be properly re-adapted to each case.

An example of classification is based on *how* each data item is assigned to the windows. There are windows where, at the arrival of an input item, it is immediately possible to define the complete list of windows that contain that item. We can distribute distinct windows among computing entities that elaborate them in parallel. So in presence of this characteristic, we are able to schedule each data item only to all the entities that will be responsible for executing a window that contains that item. For others window models the previous property is not satisfied: when the system receives a new item, it is not possible to define the complete list of windows that will contain that item because this depends on the temporal properties of the following items that will be received in the future. In that case we cannot evaluate *a-priori* the mapping between items to windows.

In this work we study a parallel pattern for computations on sliding windows, i.e. the *windows farming* [18]. It represents an implementation of a traditional *farm* pattern in the window streaming context. The basic idea is to exploit parallelism by computing different windows in parallel.

We have implemented this pattern with two different approaches: the first model (the *active workers model*) is based on the idea to completely parallelize the queries on sliding windows: that is management of the windows in terms of insertion and expiring of items, and the processing of the window data. In this case we need to know *a-priori* the mapping between items to windows, and we can use a scheduler entity that is in charge of multicasting each item to all the parallel entities (workers) that will be responsible for processing a window including that data item. This approach is suitable only for windows having this characteristic, so although it is very interesting because it enables the parallelization of all the windowing phases, it has some drawbacks such as it is not applicable to all the window types.

The other approach we implemented (*agnostic workers model*) is more "traditional": in this case the parallel entities (workers) are completely "anonymous" and they are responsible only for the windows phase execution. The windows management, in terms of window update and expiring actions, is centralized in a scheduler entity (emitter). This approach has some advantages in terms of load-balancing and it can be used for all the window types. However, it needs that the window implementation is designed to be used concurrently by multiple workers.

1.4 Results

In the thesis we have developed some benchmarks with different window types and we have evaluated the two implementation of the pattern with the *Fast-Flow* framework. It is a C++ framework for streaming programming developed by the Parallel Programming Research Group at the Computer Science Department of the University of Pisa.

For the window types that can be processed with both approaches we have analyzed and compared the two alternative approaches to discover which is eventually the best implementation with respect to the windowing configuration parameters. As expected, the two models have similar behaviors, even if the active one has an higher bandwidth due to lower internal overhead. The agnostic model remains, however, an interesting solution since it is able to achieve better load balancing and it is applicable to all the window types.

1.5 Overview

Sect. 2 is a background section where the data stream processing paradigm is introduced. We also provide a taxonomy of windowing models. Sect. 3 shows the parallel patterns, traditionally used in parallel programming.

Sect. 4 describes the *window farming* pattern, that is the parallel pattern we have developed and analyzed in detail in this thesis. In Sect. 5, we illustrate the implementation details, while Sect. 6 shows the analysis of the performances of the implemented models.

Finally, Sect. 7, provides a brief recap of the obtained results and states the conclusion of this work.

2 Background

In this chapter we analyze the technical background that we have found during the study of this topic. In particular, we will describe the main characteristics of the data stream processing paradigm and, more specifically, the use and the semantics of *window-based computations.*

2.1 Data stream processing

Owing to the steady growth of network infrastructures, many data-intensive and network-based applications (e.g., financial applications, sensor networks, social media, network analysis [38]) have gained more and more importance in the Information Technology area. For these applications the input data are not immediately available in main memory since the beginning of the computation, but rather they arrive in the form of a transient continuous data streams (e.g. ,sequences of records of attributes called *tuples*). This research field is called *Data Stream Processing*, briefly DaSP [41]

In this computing paradigm, data tuples continuously arrive in a rapid, time-varying and possibly unpredictable fashion. Furthermore, the length of the stream is potentially unbounded, or anyway it does not permit the memorization of all the data as in traditional applications. Moreover, the processing function to be applied often needs to be executed in "real-time", that is as soon as new data items arrive at the system for processing.

Another key factor of data stream processing applications is the research of strict performances requirements in terms of *high-throughput* and *low latency.* So, an important precondition is to make the application not a bottleneck, in such a way as to process data as fast as possible. For example, in on-line trading scenarios involving credit card transactions, it is imperative to design DaSP computations in order to achieve extreme low latency, because in such kind of applications performance requirements are absolutely critical and make the computation results worthless if they are not delivered in time to the users.

Due to the unbounded nature of data streams, it is possible to maintain in memory only a portion of the information received, and it is impractical to produce outputs only at the end of the stream (that actually might not exist). Therefore, a new concept has been introduced: the idea is to apply DaSP computations continuously over "portions" of the most recent data (referred to as *windows*) and defined as follows.[7]

Definition 1 (Window)**.** ***Windows*** *are data abstractions used to maintain only the most recent portion of a stream, which will be continuously updated and used to periodically compute the user-defined processing functions (also called continuous queries).*

2.2 The window-based computing paradigm

One of the main theoretical topics in this study is the formalization of window-based computations, and the definition of the semantics of the various windowing concepts that exist in literature.

We can find examples of the window-based computing paradigm in various application domain[32]. An example is a financial trading application that allows the "real-time" processing of trades and quotes from the financial markets. In order to discover trading opportunities the processing is applied only on the most recent data where importance is time-delaying(e.g., the trades and quotes received in the last 5 minutes). Another example is in network monitoring: network administrators could be interested in analyzing the incoming traffic in real-time over the traffic received in short time intervals of few milliseconds, in order to detect possible attacks by generating alerts and taking appropriate countermeasures as fast as possible [15, 23]. According to the previous examples, windows represent one of the main data abstractions in stream processing.

Definition 2 (Window extent)**.** *The extent of a window "i" (denoted by W_i) is the set of tuples that are logically associated with the i^{th} window*

In general, the query is continuously applied on each window extent, as soon as it is complete of all the tuples belonging in to it. In the ensuing description, there are tree topics that we will described in detail:

- *window semantics*, which specifies the content of the window extents;
- *window specification*, which defines how the windows are instantiated and provides a group of parameters related to how the window is utilized by the query;
- *window implementation*, which states when the window extent must be processed and when old data can be safely removed .

2.2.1 Window semantics

The first critical aspect of the window semantics is to understand which tuples belong to the various windows extents. The expression (1) describes a *Window Mapping Function*, in which T is the set of tuples of the stream and W is the set of all the existing window extents.

$$F_w : T \to W \tag{1}$$

A critical aspect is to determine which is the information needed to determine such mapping. We call this abstract information *context*. In the literature **??**, we identify two notions of context: *backward* and *forward* context. [Def. 3, 4]

Definition 3 (Backward-context)**.** *The window mapping function applied on a tuple $t \in T$ uses the* ***backward-context****. If the mapping onto the corresponding window extents exploits only the information available at the moment t has been received. In other words, it uses the knowledge on the previous tuples to determine the windows extents to which tuple t belongs to.*

Definition 4 (Forward-context)**.** *In case of a mapping function that uses the* ***forward-context****, the mapping between a tuple $t \in T$ and the corresponding window extents cannot be established by using the information available at the instant when t arrives at the system, but this mapping depends on the properties of future tuples.*

According to [37], we can distinguish between: window-semantics with a mapping function needing the knowledge about future tuples (FCA *forward-context aware*), and the ones that do not need it (FCF *forward-context free*). [Fig. 1]

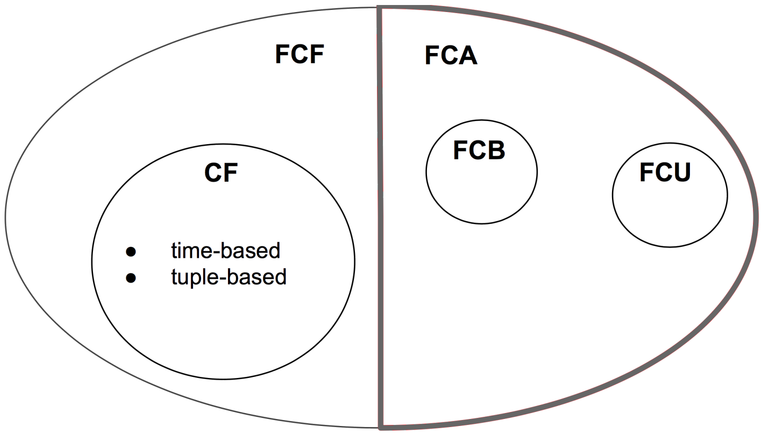


Figure 1: Window-semantics classification

FCA windows can be further categorized into two classes: forward-context *bounded* (FCB) and forward-context *unbounded* (FCU). In the case of a FCB windows-semantics, when a tuple t arrive, we can estimate the range of window extents that will contain t but we cannot establish exactly which are the extents containing t. For FCU windows, instead, it is not possible to determine the range of extents containing a tuple.

In the case of *forward-context free* (FCF) windows, it is not required the forward-context, so we can determine the window extents corresponding to an input tuple using the knowledge available at the time instant t arrived.

We can distinguish between two classes of FCF windows: the first is the so-called *context-free* (CF), in which neither the forward nor the backward context are needed. This means that can determine the windows extents of a tuple by using only the information in that tuple: the mapping function maps the tuple onto the window extents just using the properties of that tuple itself (e.g., the identifier of the timestamp of the tuple) [37].

In all the other cases the mapping function needs to know the backward context to determine which are the window extents that contain that tuple.

2.2.2 Window specification

This part is devoted to presenting how the window extents are defined and when an extent is complete and can be processed by producing a query result.

We call the window *buffer* generic data structure where the most recent tuples are stored temporarily waiting to be processed. Most of the Stream Processing Systems (SPSS) support two types of windows: *tumbling* and *sliding windows*[37]. Both of them store tuples in the window buffer in the order they arrive, but they differ in how they implement the *eviction* and *trigger* policies [Sect. 2.2.3].

Definition 5 (Tumbling Windows)**.** *Tumbling windows support batch processing: when the buffer is full its content corresponds to the extend of the activated window that can be processed. Then the buffer is emptied because window extend are by definition disjoint sets. These two operations are always executed sequentially.*

Definition 6 (Sliding Windows)**.** *Sliding windows instead, are designed to support incremental operation: according to the eviction policy, when the buffer became full only the oldest tuples in the window buffer must be evicted. In this way the eviction policy grants enough space in the buffer for the new input tuples.[39]*

2.2.3 Window implementation

In general, *window policies* define the behavior in two different cases: when the tuples can be held in the buffer (*eviction policy*) and when a window extent is complete and the corresponding computation must be fired (*trigger policy*).

Definition 7 (Eviction policy)**.** *The* ***eviction policy*** *determines the properties that the tuples must have to be held in the window buffer.*

In some cases, this behavior is guided by a property of the window itself, such as its maximum capacity. For example, if BS is the buffer size in terms of tuples, we cannot store more than BS tuples in the buffer. In other cases, the buffer size fluctuates depending on the stream rate, by containing all the tuples whose characteristics match the properties defined by the eviction policy.

Definition 8 (Trigger policy)**.** *The* ***trigger policy*** *determines when a window extent becomes ready for processing by the operator internal logic.*

In case of sliding windows, it is necessary to define both the trigger and eviction policies, instead for tumbling windows we need only the eviction policy.

As discussed in the literature [18], different policies exist and can be classified into four main categories: *count-based*, *time-based* and *delta-based* policies.

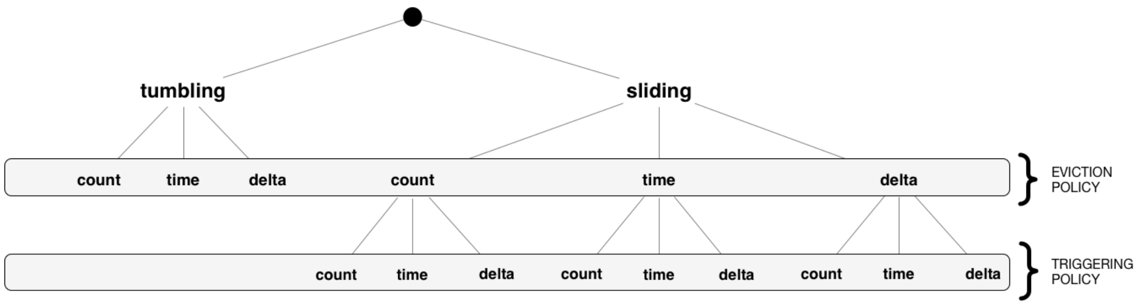


Figure 2: Taxonomy of windows

2.2.3.1 Count-based model

In the count-based policy, the eviction and/or the triggering is controlled by the number of tuples. A first "pure" model exploits the count-based policy both for the eviction and for the triggering of windows.

Fig.3 shows an example in case of tumbling windows. The value of the *window size* N is set to 4, this means that every 4 input tuples the window becomes full, so it is triggered and then evicted.

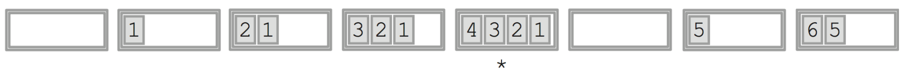


Figure 3: Count-based tumbling windows

Expression (2) is used to determine the set of tuples that belong to the window W_i $, with\ i = 1, 2, ...$ (the definition of window extents Def. 2). The set is defined as follows:

$$W_i = \{t | t \in T, (i-1) * N < t.id \leq i * N\} \tag{2}$$

Where $t.it$ is the unique identifier of tuple t (starting from one). The mapping function between tuples and windows is defined in Expression (3). For each new input tuple with id equal to $i = 1, 2, ...$ we can stationary evaluate the id of the window (starting from one) to which the tuple belongs since it is calculated as follows:

$$Win = \lceil \tfrac{i}{N} \rceil \tag{3}$$

We can use the count-based model for both the triggering and the eviction policy also in the sliding window case. In Fig.4, the *window size*$N > 0$, controls the triggering, while $S > 0$ (the *sliding value*), controls the eviction. When the number of input tuples stored in the window reach N then the window is triggered and, after the end of the processing, the S oldest tuples are evicted.

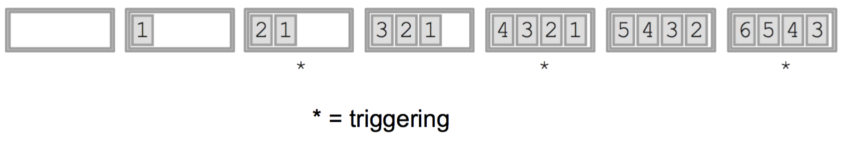


Figure 4: Count-based sliding windows

Expression (4) is used to determine the set of tuples that belong to the window W_i:

$$W_i = \{t | t \in T, (i-1) * S < t.id \leq N + (i-1) * S\} \tag{4}$$

At the arrival of an input tuple t we can statically determine the identifiers of all the window extents that contain that tuple. The evaluation is based on Expression (5):

$$\begin{aligned} first &= \begin{cases} 1 & if\ t.id < N \\ \lceil \frac{(t.id-N)}{S} \rceil + 1 & otherwise \end{cases} \\ last &= \lceil \frac{t.id}{S} \rceil \end{aligned} \tag{5}$$

The *first* and *last* formulas respectively determine the window id of the first and the last window that contain tuple t.

We recall that according to the *FCF* definition [Sect. 2.2.1], a window-based computation is FCF *if and only if* we need just N, S and the tuple id, in case of sliding windows, or just the tuple id and N in case of tumbling windows to determine the window extents corresponding to the input tuple. It is worth noting that in this specific case we need neither information on future tuples (FCF case) nor to maintain information on past received tuples: so this "pure" count based model is an example of a *context-free* (CF) window-based computation [Def. 2.2.1]

2.2.3.2 Time-based model

The time-based policy controls the eviction or/and the triggering based on the notion of time, usually evaluated using a special timestamp attribute (starting from 0) applied to the tuples. We denote as $t.ts$ the timestamp of the tuple t (typically it is its generation time).

In the time-based window model the windows are composed by all the tuples received in a fixed time frame, independently of the number of input tuples.[2]

The basic "pure" model exploits the time-based policy both for the eviction and for the triggering of windows. Fig. 5 shows an example in case of tumbling windows. The value of the *window size* N is expressed in time unit, e.g., in this example the third windows $W3$ corresponds to the interval for timestamp $[10, 15)$. Assuming that the tuples are received in increasing order of timestamp, the extent of $W3$ is complete as soon as we receive the first tuple with timestamp greater or equal than 15.

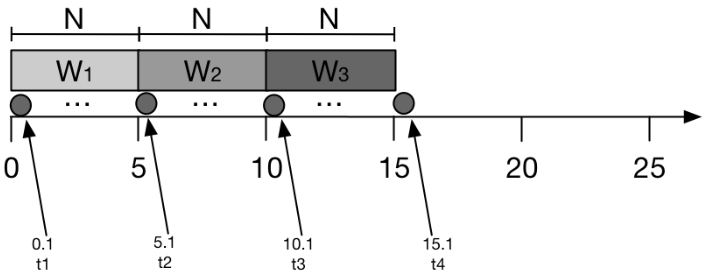


Figure 5: Time-based tumbling windows

Expression (6) is used to determine the set of tuples that belong to the window $W_i, with\ i = 1, 2, ...$ (the definition of window extents Def.2). The set is defined as follows:

$$W_i = \{t | t \in T, (i-1) * N \leq t.ts < i * N\} \tag{6}$$

The mapping function between tuples and windows is explained in Expression (7). For each new input tuple we can statically evaluate the identifier of the window (starting from 0) to which the tuple belongs. It is evaluated as follows:

$$Win \quad = \quad \lfloor \tfrac{t.ts}{N} \rfloor + 1 \tag{7}$$

In the sliding-window case we can also use a time-based model both for the triggering and for the eviction policy. Fig. 6 shows an examples with *window size* N and S as the *sliding value.* When all the tuples within the window time interval have been received, the window is triggered and, after the end of the processing, all the tuples whose timestamps are within the first S time units of the window are evicted.

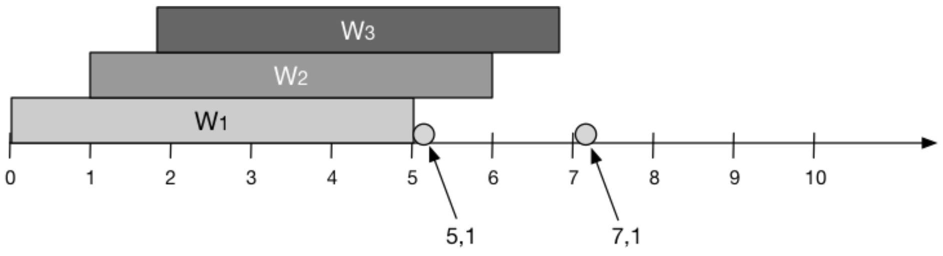


Figure 6: Time-based sliding window

Expression (8) is used to determine the set of tuples that belong to the window W_i:

$$W_i = \{t | t \in T, (i-1) * S \leq t.ts < N + (i-1) * S\} \tag{8}$$

At the arrival of an input tuple t we can statically determine to which windows the tuple belongs, and the evaluation is based on Eq.9.

$$\begin{aligned} first &= \begin{cases} 1 & if\ t.ts < N \\ \lceil \frac{(t.ts-N)}{S} \rceil + 2 & if\ \exists\ i \in \mathbb{N}\ such\ that\ t.ts = i * S \\ \lceil \frac{(t.ts-N)}{S} \rceil + 1 & otherwise \end{cases} \\ last &= \lceil \frac{t.ts}{S} \rceil \end{aligned} \tag{9}$$

The *first* and *last* formulas respectively determine the window identifier of the first and the last window to which the tuple belongs. As in the count-based model, also in this specific case we need neither information on future tuples (FCF case) nor to maintain information on past received tuples. Also this model is an example of a *context-free* (CF) window-based computation [Def. 2.2.1]

2.2.3.3 Delta-based model

The delta-based policy [15] is specified using a *delta threshold* value and a tuple attribute referred to as the delta attribute denote by a, which must be numerical and monotonically increasing. The type of the threshold value and the delta attribute must match.

When a tuple t arrives at the system, we compute $(t.a - t'.a)$ where t' is the *last tuple that triggered the preceding window*: if the difference is larger than a given threshold value $\delta > 0$, then the window is triggered. Otherwise t is inserted in the window buffer.

In the tumbling window case the application of the delta-based model both for the trigger and the eviction policy involves that when a new tuple arrives, if the difference between the delta attribute of this new tuple and the delta attribute of the oldest tuple in the current window is larger than the threshold value, then the window is processed and flushed and the new tuple is inserted into a new empty window buffer. Otherwise, the tuple is buffered as usual.

In the example of Fig.7, the window size N is set to 5 and the delta threshold Δ is set to 2. Tuple $t1$ is the "first" *marker* (it is the first input tuple) with respect to the delta threshold. At the arrival of tuple $t2$ with delta attribute 2.1 the difference between $t2.a$ and $t1.a$ is evaluated. Since this difference is equal to Δ, $W1$ is triggered and evicted [Def.5] and $t2$ is inserted in the new empty window $W2$, and $t2$ becomes the new marker.

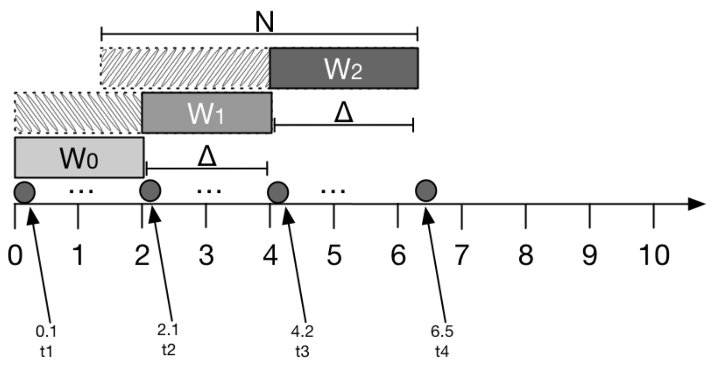


Figure 7: Delta-based tumbling window

At the arrival of $t3$, since the difference between $t3.a$ and $t2.a$ is greater than Δ, the window $W2$ have to be triggered. Now the starting point of $W2$ is evaluated as $(t3.a - N = -0.8)$. In this case, since the starting point is negative, it is set to. The previous eviction has removed all of the data in $W1$, so the available data for $W2$ are the tuples in the interval $[t2.a, t3.a)$.

In the sliding-window case, if we use a delta-based model as the trigger policy, at the arrival of a new tuple. If the difference between the new tuple delta attribute and the last triggered tuple delta attribute is larger than Δ the operator internal logic is executed. If any tuples are subject to eviction this operation can be performed only after the triggering. The last operation is the insertion of the new tuple into the buffer.

In the example of Fig.8, the delta-based model is applied both as the trigger and the eviction policy. The window size N is set to 5 and delta threshold value Δ is set to 2. The marker (M) is used for the evaluation of the delta-threshold interval k is initialized with $t1$, the first input tuple.

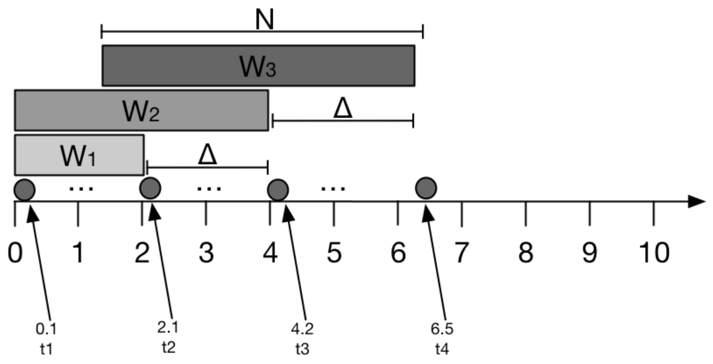


Figure 8: Delta-based sliding window

At the arrival of $t2$ we evaluate the delta-threshold as $(t2.a - M.a) = \Delta$, in this situation $W1$ is triggered and possibly evicted [Def. 6]. Because we are in a sliding window case, the evicted tuples are the ones with $t.a < (t2.a - M.a)$: in this case no tuples are evicted and the maker is reinitialized as $M = t2$.

At the arrival of $t3$, the delta threshold is equal to $(t3.a - M.a) = \Delta$ so $W2$ must be triggered. Also in this case no tuple is subjected to eviction because the starting point of $W2$ is 0. $W3$ is triggered at the arrival of $t4$. In this case the starting point is greater than 0 so the tuples to evict are the ones between 0 and the starting point $(t4.a - N)$.

This window model is quite interesting: according to the *FCA* definition (Sect. 2.2.1) we know that a window-based computation is FCA if we need information on the future tuples to determine the window extents corresponding to each input tuple. Let us go back to the example. The tuples to evict at the end of $W3$ processing depend on the value of the delta attribute of the tuple $t4$ that triggered that window. As said before, those tuples are the ones with delta attribute between 0 *and* $(t4.a - N)$ and they will not participate to the next window $W4$. Therefore, at the instant when we received a tuple such as $t3$, we do not know at that time whether it will belong to $W3$ or not since it will depends on the value of $t4$. The delta based model is the first case of FCA, because we do not know *a-priori* the markers used to evaluate Δ, so we cannot determine the starting point and the ending point of a future window at an arbitrary point in time.

2.2.3.4 Slide-by-tuple model

In the previous cases both the triggering and eviction are controlled by the same policy, i.e. according to the tuple timestamps or to the number of tuples. The case that we study in this section, represents a "hybrid" configuration, where triggering and eviction are controlled by different policies. In particular the eviction policy follows the time-based model and the trigger policy the count-based model. In this literature [13, 37] this type of windows is called *slide-by-tuple*.

It is not possible to apply the slide-by-tuple model on tumbling windows, since they need the same policy both for the triggering and the eviction. So, we focus only on the sliding window case, where N is the window length in time units and S is the slide parameter in number of tuples.

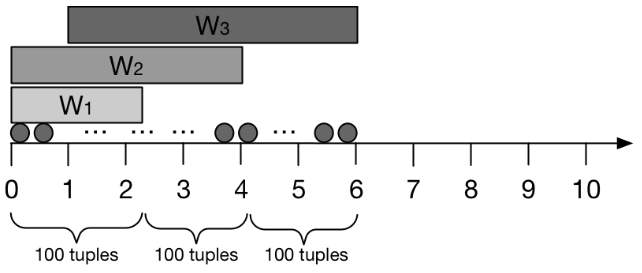


Figure 9: Slide-by-tuple with $N = 5$ and $S = 100$ tuples

Fig. 9 is a possible scenario: the window size N is set to 5 seconds and the trigger number of tuples, i.e. the sliding factor S, is set to 100 tuples.

Suppose that we complete to receive the first 100 tuple at second $2, 5$, so the window $W1$ is triggered even if the window length is less than 5 seconds. In the figure we have the same situation for the second window $W2$: at the second 4 $W2$ is triggered even if the window length is less than 5 seconds. Now suppose that we complete to receive the next 100 tuples at second 6, in this case we can obtain a window with the expected length (5 seconds) that comprises all the tuples with timestamp grater or equal than 1 and lower or equal than 6.

Interestingly, the *slide-by-tuple* model is another example of FCA window [Sect. 2.2.1]. Even if, individually, the two policies are CF if used in a "pure" model, the combination of them cannot be independent from the forward context. We know every S tuples that a new window must be triggered, and we know that the maximum size in terms of time is N, but we do not know *a-priori* the starting point and the ending point of each window, so we cannot determine which tuples correspond to the windows without the knowledge of the future tuples, that is the *forward context*.

As said, we cannot define statically the functions that rule the system

behavior based on the current context, but we can try to deduce some kind of formalization. We can determine the *theoretical* mapping between windows and tuples. We call t_i the i^{th} tuple starting from 1.
If $S = 1$ then the window W_i is triggered when we receive the tuple t_i. So, the set of tuples belonging to the window W_i, with $i = 1, 2, ...$, can be defined as follows:

$$W_i = \{t | t \in T, (t_i.ts - N) \leq t.ts < t_i.ts\} \qquad (if\ S = 1) \tag{10}$$

We can extend this expression to the general case where S can assume any value greater than 0. The set of tuples belonging to a generic window W_i, with $i = 1, 2, ...$, can stated as follows:

$$W_i = \{t | t \in T, (t_{(i*S)}.ts - N) \leq t.ts < t_{(i*S)}.ts\} \tag{11}$$

Also for slide-by-tuple windows we are able to write the function that evaluates the identifier of the first window extent containing a given tuple. As expected, we will see that this function exploits the knowledge of future tuples, i.e. the so-called *forward context* [Def. 4].

In the slide-by-tuple model, each tuple can be a triggering or non-triggering tuple:

Definition 9 (Triggering and non-triggering tuples)**.** *A tuple t_i is called a* ***triggering*** *tuple if and only if (i mod $S = 0$), otherwise it is a* ***non-triggering*** *tuple.*

In other words, a triggering tuple is a tuple that triggers the activation of a new window. Now we can establish the mapping function. We start with the funciotn of the first window. Suppose to have a triggering tuple t_i. The first window extent in which it is contained is exactly the one of the window triggered by the the arrival of this tuple. Therefore, we have:

$$first = \frac{i}{S} \tag{12}$$

In this case we need no rounding because we know t_i is a triggering tuple, so $i\ mod\ S = 0$ [Def 9]. Unfortunately, there are also tuples that do not trigger the activation of any window. Given a non-triggering tuple t_i, to establish which is the first window extent containing that tuple, we need to know information related to the *immediately successive* triggering tuple received after t_i . Let t_j be this tuple such that $j > i$. We have:

$$first \quad = \begin{cases} \frac{j}{S} & if\ t_i.t_s \geq t_j.t_s - N \\ UNDEFINED & otherwise \end{cases} \tag{13}$$

The case with *first* undefined means that the tuple t_i does no belong to any window extent: this is possible because the *slide-by-tuple* model does not guarantee that each tuple always belongs to at least one window.

Fig. 10 shows an example in which $N = 5$ and $S = 100$. We suppose that tuple t_1 arrives at second 1 and tuple t_{100} arrives at second 10. In this case the window W_1 ends at $(t_{100}.ts = 10)$ and starts at $(t_{100}.ts - N = 5)$. We can see that $t_1.ts$ is not part of that interval $[5, 10]$, so the trigger operation on W_1 does not consider this tuple: t_1 is lost.

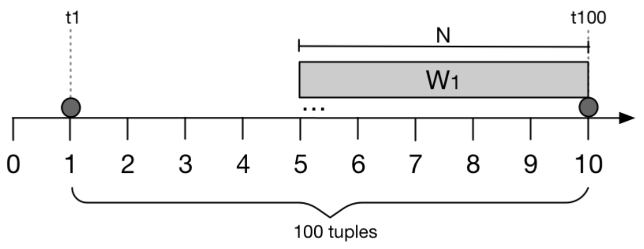


Figure 10: Slide by tuple: case of tuple not belong to any window

A similar reasoning can be developed to determine the identifier of the last window extent that contains a tuple t_i (triggering or non-triggering). Let t_k be the *last* triggering tuple that we will receive after t_i such that $t_i.t_s \geq t_k.t_s - N$. We have:

$$last \quad = \begin{cases} \frac{k}{S} & if\ t_k\ exists \\ UNDEFINED & otherwise \end{cases} \tag{14}$$

When t_k *exists* we need no rounding because t_k is a triggered tuple, so $k\ mod\ S = 0$ [Def 9]. Otherwise, the undefined case means that no window extent contains the tuple t_i.

It is easy to demonstrate that if *last* is equal to undefined, also *first* is undefined and vice-versa, i.e. the tuple does not belong to any window extent.

These explanations are valuable to understand the nature of a FCA window semantics. Different from the pure time-based and count-based models described before, in this case the *first* and *last* functions cannot be defined using only information available at the time instant when we receive a tuple t_i. In fact, we need information about future triggering tuples in order to establish the window extents that will contain a given tuple.

3 Parallel Patterns

3.1 The approach

Due to the pervasive diffusion of parallel architectures, parallel programming has become mainstream over the last years. Historically, parallel programmers rely on hand-made parallelizations and low-level libraries that give to them a complete control over parallel applications. This approach allow the programmer to manually optimize the code and to exploit at best the architecture. Besides being an impediment to software productivity and to reduced time-to-development, such low-level approaches prevent *performance portability* [40]. While *code portability* represents the ability to compile and execute the same code on different architectures, *performance portability* represents the ability to efficiently exploit different underlying parallel architectures without rewriting the application. This is an important feature that parallel programs should exhibit. Individuals and industries cannot afford the cost of re-writing and re-tuning each application for every architecture.

High-level approaches to express parallel programs hide the actual complexity of the underlying hardware, providing the needed productivity and performance portability required to ensure the economic sustainability of the programming efforts. [15, 5] Consequently, programmers can focus on the computational aspects, having only an abstract high-level view of the parallel program while all the most critical implementation choices are left to the programming tool and run-time support.

This approach makes the applications independent from the architectures below, providing reasonable expectations about its performance when executed on different hardware despite different run-times can be adopted for different execution architectures. Low-level mechanisms (such as *Posix threads* [3]), programming libraries (like *MPI* [24]) or compiler extensions (such as *OpenMP* [19]) express only some characteristics of high-level parallel programming methodologies.

The main idea behind *Pattern-based Parallel Programming* [17] is to let the programmer able to define an application through parallel paradigms (also called *patterns*). *Parallel paradigms* are schema of parallel computations that recur in the realization of many real-life algorithms and applications for which parametric implementations of communications and computation patterns are clearly identified. The *FastFlow* [1] library is a pattern-based framework that we chose in this work to implement our data stream processing parallel pattern.

3.1.1 Parallel Patterns in FastFlow

FastFlow is a C++ framework for structured parallel programming targeting both shared memory and distributed memory architectures. It is composed by several layers which abstract the underlying architecture. The abstraction provided by these layers is twofold: to simplify the programming process offering high-level constructs for data parallel and stream parallel pattern creation and, at the same time, and to give the possibility to fine tune the applications using the mechanisms provided by the lower layers. The programmer can exploit *FastFlow* features by using any of the mentioned levels:

- *basic mechanisms*
- *core patterns*
- *high-level patterns*

At basic level, *FastFlow* offers single producer/single consumer (SPSC) queues which can be used as communication channels between threads[34]. These queues are characterized by the absence of locking mechanisms. SPSC queues can be classified in two main families: *bounded* and *unbounded.*

- *Bounded* SPSC queues are typically implemented using a circular buffer. They are used to limit memory usage and avoid the overhead of dynamic memory allocation;

- *Unbounded* queues are mostly preferred to avoid deadlock issues without introducing heavy communication protocols in the case of complex streaming networks, i.e. graph with multiple nested cycles.

We exploited *FastFlow* SPSC unbounded queues to implement the communication channels needed by our application. The communication channels were used to connect the *emitter* and the *collector* with the *workers* and the *workers* with each other. [Def. 3.3]

The elements passed in the queues (data items such as *tuples* in DaSP) are pointers to data-structures that are stored in shared memory.

At the *core pattern* level there are two type of patterns: *farm* and *pipeline*. They are both implemented using the basic mechanisms. *High-level patterns* are built on top of these core patterns. They are clearly characterized in specific scenarios and are targeted to the parallelization of sequential code with certain characteristics (e.g. pattern for evolutional algorithms).

3.2 Parallel paradigms

Without loss of generality, in *FastFlow* a parallel application is expressed by means of a computation graph in which nodes represent *operators*, i.e. intermediate computations in which the application can be decomposed. They communicate by means of internal data streams on which the computations are applied.

If the performance requirements are not met, operators that are *bottlenecks* have to be internal parallelized according to some parallelism paradigms. The available patterns exhibit the following features:

- they restrict the parallel computation structure to certain predefined concurrency patterns;
- they have a precise cooperation semantics;
- they are characterized by *cost models*;

- they can be composed and nested with each other to form complex computations.

In this way the programmer is free from handling details concerning the mapping between parallel computation schemes and their real implementation.

Cost models allow evaluating the performance metrics as functions of other parameters that are typical of the application (e.g., calculation time, input data size) and of the architecture (e.g., processor clock, memory access time and so forth). Some important performance measures to be considered are:

Definition 10 (Throughput)**.** *The **throughput** is the average number of stream elements (e.g., tuples) that can be completed in a time unit. It is the inverse of the "service time", that is the average time interval between the beginning of the execution on two consecutive stream elements.*

Definition 11 (Calculation Time)**.** *The **calculation time** is the average time needed to execute the computation on the single input element.*

Definition 12 (Response time)**.** *The **response time** is the time elapsed from the reception of an input element to the production of the corresponding output.*

Parallel paradigms can have different impact on these metrics, captured by their respective cost models.[15, 4, 9]

3.3 Paradigms classification

We can distinguish between two main categories of parallel patterns:

- *Stream parallel paradigms*: these patterns work on streams of homogeneous elements. Parallelism is obtained by simply processing multiple elements concurrently. They do not speed up the computation of a single element but the computation of the stream: this means that they mainly improve throughput;

- *Data parallel paradigms*: in this case the single computation (possibly also relative to a stream element) is parallelized. Usually this requires to partition the input data structure and also the computation. Paradigms that fall in this category are able to improve response time.

A parallel paradigm describes the structure and the interactions between a parametric set of entities. We can recognize different types of involved entities[15]:

- The *Emitter* receives data from input stream(s) and deals with data distribution;
- The *Workers* are executors that are in charge of performing the computation on the received data;
- The *Collector* receives the computed results from workers and transmits them onto the output streams of the parallel module.

In the next part we will review some basic parallel patterns that we will use in this work.

3.3.1 Pipeline

The *pipeline pattern* is a very simple stream parallel pattern to some parallelization problems. It assumes that the computation is expressed as a sequential composition of functions over the input elements, i.e.:

$$F(x) = F_n(F_{n-1}(...(F_1(x))...)) \tag{15}$$

In this case a possible parallelization is a linear graph of n executors, also called *pipeline stages*, each one corresponding to a specific function [Fig. 11].

This kind of solution can be adopted to increase *throughput*. The *service time* is the one of the slowest stage. *Response time* may increase due to the communications overhead between stages.

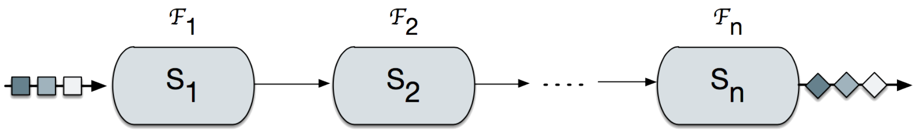


Figure 11: Pipeline parallel paradigm

3.3.2 Farm

The *farm* paradigm corresponds to the replication of a pure function among a set of identical workers. The *emitter* is responsible to send each input element to a worker, according to a certain scheduling policy ideally able to balance the workers load. If the calculation time has a low variance, a *round robin* solution for the scheduling policy can be sufficient, otherwise other sophisticate policies can be adopted.

All of the workers apply the same function F to each input element they receive from emitter. The results are sent to the collector that is in charge of collecting, eventually ordering and transmitting them onto the output stream. As for the pipeline case, this solution is able to increase *throughput* while the

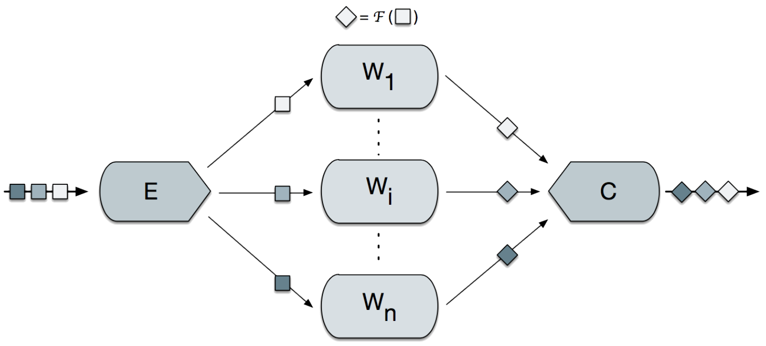


Figure 12: Farm parallel paradigm

response time cannot be improved in general [15].

3.3.3 MAP

The MAP is a *data-parallel* paradigm that consists in the partitioning (or replication) of the input data and replication of the function. In this way, distinct but functionally equivalent workers are able to apply the same operation to a distinct data partition of the current input item in parallel.

The *emitter* is responsible for distributing the various partitions (*scatter*) to the workers. Collection of the worker results is performed by the *collector* eventually exploiting a gather operation, to receive partial results and to build the final one.

In the MAP paradigm workers are independent [Fig. 13]. Each of them operates on its own local data only, without any communication during the execution.

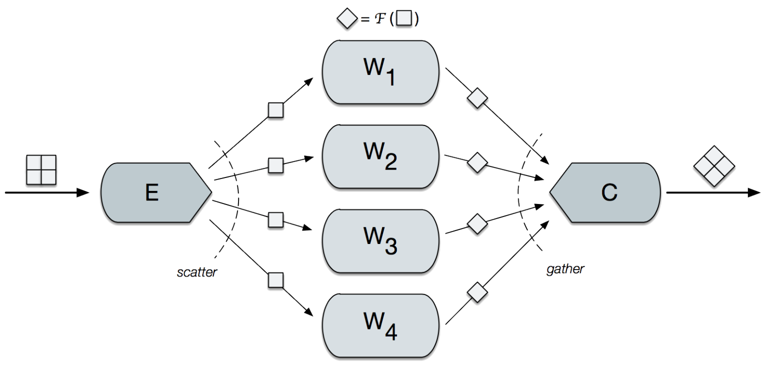


Figure 13: Map paradigm, exemplified with 4 Workers

Interestingly, a more complex data-parallel computation is characterized by cooperating workers: in order to apply a function, a worker may require to access data owned by other workers, because data dependencies are imposed by the computation semantics. In this case we speak about *stencil-based* computations, where a *stencil* is a data dependence scheme implemented by an information exchange among workers.

Data parallel paradigms (MAP and Stencils) are able to reduce the *response time* for a single input element. When applied to a stream, they can also improve *throughput*.

3.4 Towards data stream processing

When we design a stream parallel pattern we can have two possible scenarios: *stateless* or *stateful* computations.

Stateless computations are characterized by the independence from the context. For stateless operators the mentioned patterns (e.g. stream parallel ones) are valid and easily applicable. For example in the pipeline case the sequential functions can be based only on the input data to evaluate the result. Also in a farm or map paradigm the result can be evaluated only on the input element.

A *stateful computation*, instead, maintain information between computations on different input data. Such information are used to produce the result of each input. [Sect. 2.2.1] When it is necessary to parallelize a *stateful* operator, things may be more complex. Each element has own information that influences the state, so, at the arrival of each input the environment is updated and each worker must consider this change during the computation.

Stateful computations as especially important in the DaSP domain and require proper specializations and enhanced features in terms of data distribution, management policies and windowing methods that are not considered in previous traditional patterns.

In this work we study the characterizations of a parallel pattern for executing a class of data stream processing problems: *window-based streaming computations*. [33] They have two main characteristics:

- the input stream (one or more) is composed by many and relatively small in size input elements referred to as *tuples* (a record of attributes);
- tuples must be accumulated by the operator, whose logic periodically goes over the stored tuples to produce a new result.

We have focused our work on the specialization of the *Farm Pattern* adding the processing and the manage of windows. In this way we have obtained a new paradigm called *Window Farming* [18] (the original definition is in [Def. 4.3]).

To exploit farm parallelism, windows must be distributed among workers. Such distributions can be critical, since windows are data structures with particular features, and they are dynamic in the sense that their content changes over time according to their *triggering* [Def. 8] and *eviction* policy [Def. 7].

Furthermore, the cardinality of windows in terms of elements may change over time, based on the current stream arrival rate. For these reasons two problems arise:

- how to keep up-to-date the operator internal state (window buffer) used to maintain tuples for window processing. Particularly important are the distribution policies and the expiration policies. Different models can be designed depending on which entity of the parallel computation is in charge of checking the expiration of past received tuples and which entity is in charge of their removal from the state;
- how to keep the workload among different workers balanced, despite the presence of windows with different cardinalities.

These issues can still be dealt with the Pattern-based Parallel approach in order to reduce the effort and the complexity of parallel programming and simplify the reasoning about the properties of the parallel solutions in terms of throughput, response time and memory occupancy. This will require to revisit existing patterns and to propose new ones [15]. This is the goal of the next section.

4 A Parallel Pattern for Sliding-Window Queries

In this section we describe a parallel pattern for sliding-window queries. Its name is *Window Farming* [Sect. 4.3] and it is an evolution of the traditional farm pattern [Sect. 3.3.2] having, in addiction, all the features needed to manage the window-based processing model described in the past sections [Sect. 3.1].

4.1 Windowed operators

DaSP applications can be expressed as compositions of core functionalities organized in *directed flow graphs* where *nodes* are operators and *arcs* model streams, i.e. unbounded sequences of data items (*tuples*) sharing the same schema in terms of name and type of attributes.

At run-time it is possible that some operators act as *bottleneck* and the DaSP application is not able to meet the ideal performance in terms oh throughput and response time. In this case we need to optimize the computation by parallelizing all the bottleneck operators.

As said in the previous section, *stateful* operators maintain and update a set of internal data structures while processing input data, and the outputs depend not only on the present input but also on the history of past inputs (like in sliding-window). This creates dependencies between the processing of individual tuples, and consequently the parallelization of stateful operators must take into account this issue in order to preserve sequential semantics [6].

In some application cases, the physical input stream conveys tuples belonging to multiple *logical substreams* multiplexed together. Stateful operators typically maintain a separated data structure for each substream (a window for example), and update it on the basis of the computation applied only to the tuples belonging to that substream. The correspondence between tuples and substreams is usually made by applying a predicate on a partitioning attribute called *key*, e.g., it is the result of a *partition-by* predicate [18]. Examples are operators that process network traces partitioned by IP address, or trades and quotes from financial markets partitioned by the stock symbol. We refer to the case of a *partitionable state* as a *multi-keyed stateful operator* with *cardinality* $|K| > 1$.[18] The case $|K| = 1$ refers to the special case of a *single-keyed stateful operator*, which is the one studied in this work.

In this work we will focus on *windowed operators* that use sliding-windows as the predominant abstraction to implement the internal state of the operator. As said; window semantics is specified by the eviction and the triggering policies [Sect. 2.2.1]. Therefore, a window is defined by:

- a *window size* $|W|$, expressed in time units (i.e. seconds or minutes) for *time-based* windows or in number of tuples for *count-based* windows;
- a *sliding factor* Δ that expresses how the window moves. We will study cases where the sliding factor can be expressed in time units or in number of tuples.

It is important to observe that only in the sliding window case consecutive windows may have overlapping regions, i.e. the same tuple may belong to multiple consecutive windows extents.

In the discussion we will assume that the computation over the current window extent is performed at the triggering time and not incrementally as in [13]. Furthermore in the discussion we will assume that the tuples arrive ordered by their timestamp (or by a sequence number), and they must be processed in the same order of arrival. Typically, downstream operators will assume to receive the results of a parallel operator in increasing order of their identifier, so the parallel operator must produce results in the same order of the sequential implementation. This must be properly guaranteed because even if the computation on window i is triggered before the one of window j, it may be possible that the result of window j can be sent outwards before the one of window i.

4.2 Performance metrics

In the traditional farm pattern the same computing task is executed at the arrival of each input element. In contrast, in our case, a task is no more a single input element, but a segment of the input stream corresponding to *all* the tuples belonging to the same window. Given this new definition of task, the basic performance measurements [Sect. 3.2] should be revised:

Definition 13 (Throughput and service time)**.** ***Throughput*** *is now the average number of windows that the operator is able to process in a time unit. It is the inverse of the* ***service time****, that is the average time interval between the beginning of the execution of two consecutive windows.*

Definition 14 (Calculation time)**.** *The* ***Calculation time*** *is the time spent by the user function applied on the window content when the window is activated.*

Definition 15 (Response Time)**.** *The* ***response time*** *is the time elapsed from the reception of the tuple triggering a window computation to the instant when the correspondent output is produced by the operator.*

In order to define quantitatively "how good" a parallel solution is, we introduce also the concept of *scalability*:

Definition 16 (Scalability)**.** *The* ***scalability*** *provides a measure of the relative speed of the n-parallel computation with respect to the same computation with parallelism equal to 1. It is defined as the ratio of the throughput achieved with n workers to the one obtained with only one worker.*

4.2.1 Windowed Parallel Patterns

We can distinguish between two categories of parallel windowed operators, where the window farming belongs to the first one:

- *window parallel paradigms*: these patterns are capable of executing in parallel at the same time instant computations over multiple windows. They improve throughput of the whole operator but not response time;

- *data parallel paradigms* are to parallelize the computation on a single window. This require to partition each window extent among a set of identical executors. As in traditional data parallel patterns, they are able to improve both throughput and response time;

4.2.2 Distribution Phase: load balancing

In every pattern the distribution phase is important because of the dynamic nature of the windows in term of tuples contained and their cardinality. Two orthogonal aspects can be considered:

- the *granularity* at which input elements are distributed by the emitter to a set of workers, for instance the unit of distribution can be a single tuple, a group of tuples or entire windows;

- the *assignment policy*, that is how consecutive windows are assigned to the parallel workers of the pattern.

Distribution strategies lead to several possible optimizations of the same pattern or, in some cases, to the identification of new patterns. The distribution may also influence memory occupancy and the way in which the window management is performed. [Ref. [18]]

4.2.3 Workers models

By focusing on sliding-window parallel patterns we can identify two models:

- *Agnostic worker model*: in this case all the processing/storing actions needed to build and update the windows are performed by the distribution logic (i.e. by the *emitter*). Workers are agnostic of the window management and data distribution, and they are just in charge of applying the computation to the received window data;

- *Active worker model*: this approach manages partially or entirely the window semantics in the workers. Workers receive elementary stream items (tuples) from the emitter and are in charge of managing the window policies by adding new tuples and removing the expired ones. Emitter and collector act mainly as "intelligent" interfaces with respect to the workers.

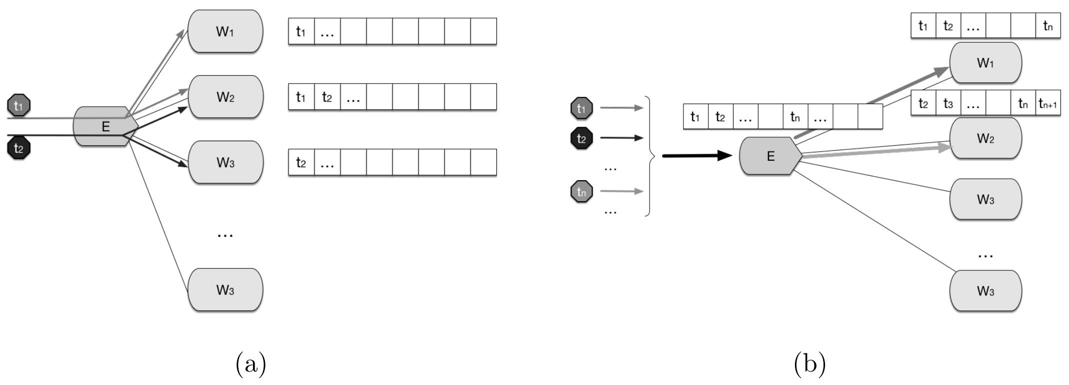


Figure 14: Pattern with active (a) and agnostic (b) workers

With different combinations of these characteristics we can obtain different implementations of the pattern. Each of them will have certain characteristics and can be suitable or not to process sliding-window queries with different semantics, as it will be described in the next part.

4.3 Window farming

In this section we describes the *Window Farming* pattern starting from a simple intuition: each window triggering implies the application of a function F on its content, and each window can be processed independently, so the result of the computation on a window does not depend on the results of any other window. Anyway, it is important to remember that F is applied on different windows that may potentially have some portions of "shared" data (this concept will be analyzed in dept in [Sect. 5]).

This idea is valid in the simplest case in which we have only one input stream and only one output stream, but also in the generic case with multiple input/output streams, each of them identified by a *key* attribute. Therefore, the new pattern can be described as follows in Fig. 15:

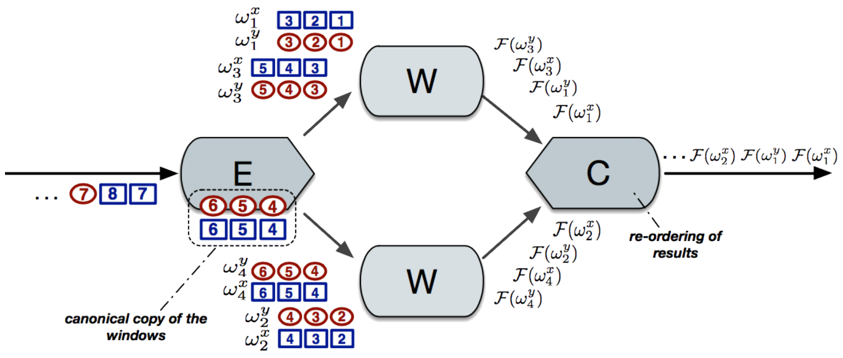


Figure 15: Window Farming with two workers. It has in input two logical sub-streams X and Y, whose elements are identified with squares and circles respectively. In the example $|W| = 3$ and $\Delta = 1$. w_i^x represents the i-th window of substream X. $F(w_i)$ represents the result of the processing function over a window

Fig. 15 assumes the *agnostic workers* model. The emitter is in charge of buffering tuples coming from the stream and builds and updates the window buffer, one for each logical sub-stream, in this case with keys X and Y.

Tuples and windows are marked with unique identifiers. The emitter inserts the tuples into the related buffer. Once a window has been completely buffered, it is transmitted to a worker. The assignment policy must grant load balancing. If the function F has a low-variance processing time, we can apply a simple *round-robin* assignment. An alternative optimized policy consist in an *on-demand* solution, where each worker signals to the emitter the availability to accept a new task. In the figure we adopt a round-robin strategy: windows w_i^x and w_i^y are assigned to worker $j = (i + 1)$ *mod* n where n is the number of workers. In the model, the emitter is in charge of removing the expired tuples according to the window semantics.

Multiple windows of the same or of different sub-streams are executed in parallel by different agnostic workers. The collector receives the results and may be responsible for reordering them. If a round robin distribution strategy is used, this will simply require to collect results from the workers in the same order in which windows are scheduled to them. Otherwise, the collector should rely on the windows sequence numbers.

A different implementation of this pattern can be obtained by adopting, a different distribution strategy by enabling the active worker model. For instance, the distribution can be performed with a finer granularity: instead of buffering entire windows and then transmit them, single tuples (or small groups of consecutive tuples) can be transmitted by the emitter to the workers as they arrive from the input stream without buffering the whole windows. Each tuple is routed to one or more workers depending on the values of the window size, sliding parameters and the distribution strategy.

To do that, the emitter must know for each tuple t which are all the window extents to which it is contained. In this way, based on the assignment policy, it can establish which are the workers that must receive that tuple. Furthermore, this activity must be performed by the emitter at the time instant the tuple t is received. This means that the *active worker model* can be used provided that the sliding-window semantics is FCF [Def. 2.2.1].

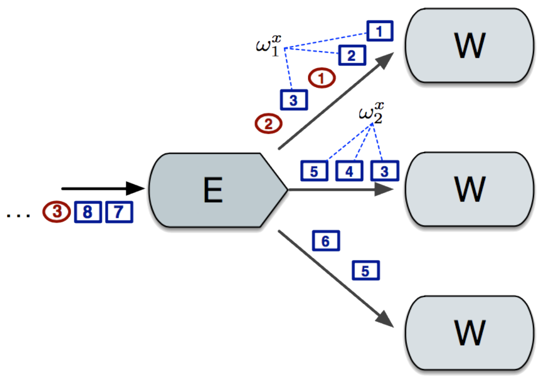


Figure 16: Window Farming with fine grained distribution. In the example $|W| = 3$ and $\Delta = 2$. Collector is omitted.

Fig. 16 shows an example with 3 active workers, $|W| = 3$ and $\Delta = 2$. Windows are assigned to the 3 workers in a round-robin strategy. Each tuple can be transmitted to *one* or *more* workers, depending to which worker(s) the corresponding windows are assigned to. For example, the tuple x_4 is part only of window w_2^x which is assigned to the second worker. Tuple x_5 belongs instead to windows w_2^x and w_3^x and thus it is multicasted to the second and the third worker.

The emitter, for each received tuple t, must be able to determine the windows on which the tuple belongs the window mapping function [Sect. 2]. The tuple is transmitted to worker j if t belongs to window w_i and w_i is assigned to worker j. [see 4.3]

Workers are now active in the management of the windows. They must be aware of the window semantics and assignment policy to manage the window triggering and tuples expiration. In fact, if a round-robin strategy is used by the emitter, then worker can handle incoming tuple like it is working on a window with size $|W|$ and slide $n\Delta$, where n is the number of workers. After the computation, the first $n\Delta$ tuples of the window can be safely discarded since they are no more needed.

For time-based window this example is still valid taking into account the tuples' timestamps instead of the number of the tuples.

5 Pattern Implementation

The implementation was developed locally on my personal machine. In order to test the code, I deployed it by *rsync*[1] into a couple of twin servers owned by the department of UniPI. Each server has the following configuration:

- **Two CPUs**: Intel(R) Xeon(R) E5-2650 @2.00GHz. Each CPU has 8 cores. Each core has a private L1d (32KB) and L2 (256KB) cache. Each CPU is equipped with a shared L3 cache of 20 MB
- **RAM**: 28 Gb
- **Operative System**: CentOS release 6.3
- **Compiler**: g++ (GCC) 4.8.1
- **User function library**: lmfit-5.1
- **FastFlow**: v.2.1.3

5.1 The FastFlow framework

As said in Sect. 3.1.1, FastFlow is the framework we have used to implement the pattern shown in the previous sections [Sect. 4.3].

FastFlow allows the customization of the farm pattern, especially we can rewrite the logic behind the single threads. Each FastFlow node, executed by a dedicated thread, can be programmed by re-defining the following methods:

- *svc_init* is the function in charge of doing the steps in the starting phase e.g., initialization of variables;
- *svc* is the "core" function, it starts every time an input tuple is received and it is in charge of applying a computation on the input data. It is

[1]**rsync** is an utility for efficiently transferring and synchronizing files across computer systems. It is commonly found in Unix-like systems and works as both a file synchronization and file transfer tool. `https://help.ubuntu.com/community/rsync`

responsible for forwarding the elaboration results to the other application stages that require them;

- *svc_end* is the ending function responsible for the “closing” of the stage (node behavior), after the processing of the last input item.

In this work we have provided two different implementations of our window farming pattern, one is based on the *Active Workers* model and the other is based on the *Agnostic Workers* model [Sect. 4.2.3]. The active implementation is applicable only to CF window models [Def. 2.2.1]. In contrast, the agnostic implementation is applicable to all the window models seen in Sect. 2.2.1, both FCA and FCF sliding and tumbling windows. One of the final goal of this work is to be able to compare the performance of the two models in some specific scenarios (studied deeply in Sect. 6) in order to study the characteristics of these approaches and to verify when the agnostic implementation is better than the active one, or vice-versa.

The project description is structured as follow: in the Sect. 5.2 we will show at a high level the stages of elaboration and we focus those with a common implementation for both the active and the agnostic workers models.
In the Sect. 5.3 and Sect. 5.4 we will present the implementation details of the Active Workers model and the Agnostic Workers model respectively.

5.2 Stages and common implementations

As we can see in Fig. 17 , in this project we can distinguish two "super-phases": *Data generation* and *Data elaboration*. Therefore the overall schema is a pipeline of two stages.

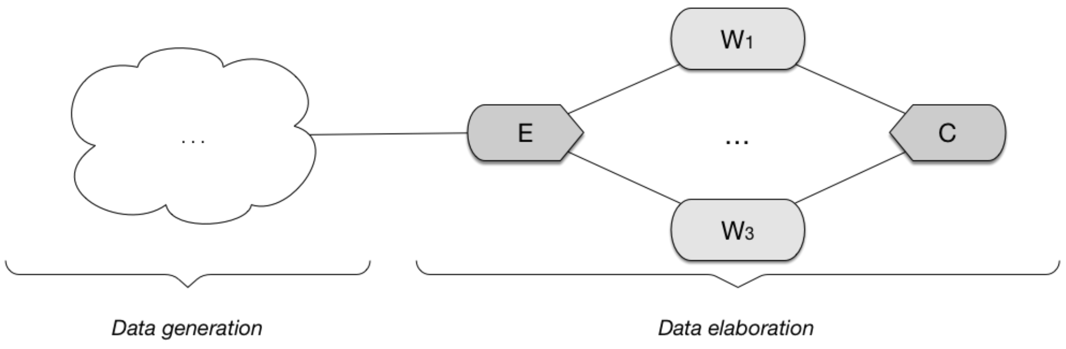


Figure 17: Elaboration stages

The first stage is the *Data generation*. It is described in Sect. 5.2.1. In this phase the data are "created", structured as *tuples*, and forwarded to the following *Data elaboration* phase.

The *Data elaboration* stage is described in Sect. 5.2.2 and it is in charge of *elaborating* the received data (tuples) by producing the final results. It consists in a farm pattern with the *Emitter* the *Workers* and the *Collector*. The Emitter and Workers have a different implementation relatively to the chosen model, so we analyze them in detail in the Sect. 5.3 and Sect. 5.4. Instead the Collector has a unique implementation for both the workers model.

5.2.1 Data generation

The goal of this stage is the generation of tuples and the transmission to the next application phase. This first phase can be decomposed into three sub-phases:

- a *dataset file* containing all the tuples in a textual format is generated by executing a Python script. This file contains for each row the values of the attributes and the application timestamp of each tuple;

- a second Python script elaborates the previous file and transforms it in binary format;
- in the last phase, starting from the data read from the binary file, the *Generator* node (a thread) generates tuples by respecting the stream speed dictated by the timestamps. The *Generator* stage is a FastFlow node connected in pipeline with window farming implementation (composed by the Emitter, Workers and Collector threads [Sect. 12]).

Data generation phase is implemented in the same way for both the active and agnostic worker model.

5.2.1.1 Input tuples

In addiction to the tuples attributes and the timestamp, we have added some further fields to the tuples. The new input structure [Code 1] is the following:

- $numWorkers$: this value represents the number of workers that need to receive that tuple, so how many distinct workers are evaluating windows in which that tuple belongs to. This field is meaningful only for the *Active Worker* implementation;
- *ts*: it is the timestamp of the tuple;
- *id*: it represents the unique identifier of the tuple (starting from 1);
- *key*: this value is the stream input identifier. It is useful in case of multiple input streams because enables us to identify the corresponding source;
- $numVal$: it is the length of the *value* array (number of attributes) per tuple;
- *emitter_ticks*: this value represents the instant when the Emitter received the tuple t, and it can be used in the evaluation of the *response time*. This value is initialized to *zero* and updated by the Emitter before sending t.

- *value*: it represents the array of attribute values used by the user function F.

Code 1: Input tuple structure

```
template <class T>
class input_struct {
        private
                std::atomic<int> *numWorkers = NULL;
                double ts;
                int id;
                int key;
                int numVal=0;
                volatile ticks emitter_ticks = 0;
        public:
                T *value = NULL;
};
```

5.2.2 Data elaboration

In this phase we describe how the computation of input tuples is performed. The elaboration phase can be decomposed into three main parts [Sect. 3.3]:

- the *Emitter* receives tuples from the *Generator* and, depending on the pattern implementation, it forwards pointers to messages to the workers. In particular, we have two scenarios [Sect. 4.2.3]: in the *Active worker* case the Emitter evaluates which Workers are interested in receiving the tuple t and then sends t to them (via a pointer). We recall that this is possible only for FCF windows, where the Emitter can determine the window mapping function by using the knowledge of the past received tuples including t (backward context [Def. 3]). In the *Agnostic worker* case instead, the Emitter maintains all of the tuples received and sends to the workers the pointer to a special window descriptor that allows the destination worker to access the tuple in the window extent. This implementation will be described in detail later;

- the main goal of the *Workers* is to apply a predefined function F on to the input tuples in the window. A worker can be *Active* or *Agnostic*

[Sect. 4.2.3]. Depending on this choice, it can receive the pointer of each tuple (*Active*) or the pointer to the window descriptor with the starting and ending point of a window extent (*Agnostic*). We recall that in the first case the workers are involved in the window management (expiring and tuple insertion) and each worker has a private window buffer, while in the latter case the window buffer is centralized in the emitter and reads directly by the workers;

- The *Collector* receives from each worker the result of its elaboration (window results), and it may reorder them based on the original arrival order. In this way the Collector ensures that the window results will be emitted in order to the final user, i.e. the result of window W_i will be produced before the one of W_j if $i < j$.

While the Emitter and Worker implementations depend on the chosen approach (Active or Agnostic), the Collector implementation is the same in both cases.

5.2.2.1 Window results

In addition to the fields specific of the query, we have added some other fields [Code 2]:

- *ts*: it is the timestamp recording the time which the result is sent from the worker to the collector;
- $tsWinStart$ and $tsWinEnd$: they are used in the time-based implementations and they are the timestamps of the starting and the ending time instant of the tuples contained in the window extent;
- *id*: it is the unique identifier of the window (starting from 1);
- *key*: it is the key corresponding to the input source used to eventually identify the logical stream source;

- *workerId*: it is the unique identifier of the worker that performs the calculation;
- *numElemInWin*: this value represents the number of tuples that participate to the calculation;
- *start_ticks*: it is the number of ticks representing the total calculation time;
- *latencyStart*: corresponds to the value *emitter_ticks* of the older input tuple belonging to the window extent, and it is necessary for the *response time* calculation.

Code 2: Window result structure

```
template <class T>
class output_struct {
        private:
                double  ts = 0,
                tsWinStart = 0,
                tsWinEnd = 0;
                int     id = -1,
                key = 0,
                workerId = 0;
                long long int numElemInWin = 0;
                volatile ticks  start_ticks = 0,
                latencyStart = 0;
        public:
                T **value = NULL;
};
```

In the next sections we describe in details the Emitter and Worker implementations in the Active and Agnostic cases.

5.3 Active worker model

As said in Sect. 4.2.3, in the *Active Worker* model the Emitter is responsible for the propagation of each tuple t to each worker W that manages at least a window where t belongs to. In the *Active Worker* model each worker maintains a “private” window buffer and it is involved in insertion and expiring activities. The emitter acts as a data forwarding entity that routes tuples to the workers in such a way as to dynamically partition the set of windows among the workers. In the ensuing discussion we will suppose that windows are distributed to the workers in a round-robin fashion, e.g. window W_i is assigned to worker j such $j = i \ mod \ N$ where N is the number of workers

5.3.1 Applicability

The *Active Worker* model cannot be applied to all the window semantics because it has some features that limit its applicability. One of its main characteristics is the delegation of the windows management directly to the Workers, so the Emitter must know *a-priori* the mapping between tuples and windows in order to forward to each worker *all and only* the tuples needed. On the one hand, this approach is very simple both from the logic and the implementation viewpoint, on the other hand its simplicity may limit the applicability. In particular, we cannot apply the *Active Worker* model in case of FCA windows like the *slide-by-tuple* model described in Sect. 9.

Another limit is about memory occupancy and traffic generation between the Emitter and the Workers. At the arrival of each tuple t, the Emitter sends t to all the “interested” workers. This is implemented as a *multicast* of the same pointer to a subset of the workers (eventually all of them). This multicast can be expensive and furthermore each Worker has to copy the tuples received into a private window buffer, and this leads to a high memory occupancy. However, this is the specificity of this implementation of the Window Farming pattern.

5.3.2 Data structures

For the implementation of the private window buffer in the *Active worker* model, we decided to use a *deque*, that is a standard structure offered by the C++ standard library (STL). This is because each workers frequently adds new tuples at the end of the container and removes the oldest tuples at the beginning of the container. For this access pattern a deque is an efficent solution.

5.3.2.1 deque

The *deque* (**d**ouble-**e**nded **que**ue) is an indexed sequence container that allows fast insertion and deletion at both its beginning and its end. The storage of a *deque* is automatically expanded and contracted as needed. Expansion of a *deque* is cheaper than the expansion of a vector because it does not always involve copying all the existing elements into a new memory area.

The internal state of a deque contains a sequence of *chunks* (implemented as a map structure). In the example in Fig. 18 it is shown how the internal state of a deque is managed.

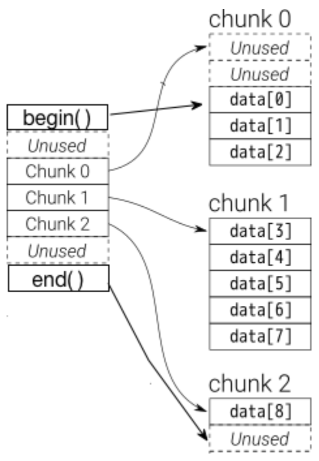


Figure 18: deque example

Each chunk is a dynamic container of *data items* that are stored in contiguous memory areas within the chunk. For example, a chunk can be implemented as a vector. Data items belonging to different chunks are commonly stored in many areas that may not be contiguous with each other.

This allows fast modifications of the deque in the last positions (*push_back* in the last chunk) as well as at the beginning of the deque (eventually by copying only the data in the first chunk). [22]

5.3.3 Time-based window model

In this work we have applied the *Active Worker* model in the case of time-based sliding window which are an example of CF and thus FCF windows. We can easily re-adapt the implementation to *count-based* windows that are in the same class.

5.3.3.1 Emitter

As said in Sect. 4.2.3, the main goal of the Emitter in the *Active Worker* model is to forward each tuple t to the "interested" workers.

When the Emitter receives t, it evaluates the mapping function to determine which are the windows containing t. After that, depending on the assignment policy (we suppose the round-robin one), the Emitter determines which Workers are responsible for elaborating those windows and it sends the pointer to t to those that are working on the windows identifiers returned from the application of the mapping function on t.
Code 3 is part of the function in charge of sending tuples to the workers.

Code 3: Active case: Emitter code

```
//evaluate the first window affected
if (ts < (wSize)) first_id = 1;
else first_id = ceil(((double)(ts - (wSize))) / ((double)slide)) + 1;

//evaluate the last window affected
last_id = ceil(((double)ts / slide));

// evaluate the workers working on the window (ROUND ROBIN)
workerId = first_id;
countWorkers = 0;

startWorkerId = (idKey - 1) % pardegree;

i = first_id;
last_id = (last_id < first_id) ? first_id : last_id;

//decide to witch worker send the data
while ((i <= last_id) && (countWorkers < pardegree)) {
        to_workers[countWorkers] = (startWorkerId + (i - 1)) % pardegree;
        countWorkers++;
        i++;
}

//set the tick before
t->setTupleEmitterTick();

//update the tuple worker number
t->setWorkers(countWorkers);
// update the statistics for the number of copies per tuple
totalCopies += countWorkers;

//send the tuple to all the interested workers
for (int i = 0; i < countWorkers; i++) {
        while (!lb->ff_send_out_to(t, to_workers[i]));
}
```

Let us have a practical example of how in our case the scheduling policy relates windows and workers and how, consequently, the Emitter selects which workers will receive the input tuples [Fig. 19 and Fig. 20]. In our example, the window size is set to 4 time units, the sliding factor Δ is set to 1 time units, the scheduling policy is round robin and we have three workers $Worker_1$, $Worker_2$ and $Worker_3$.

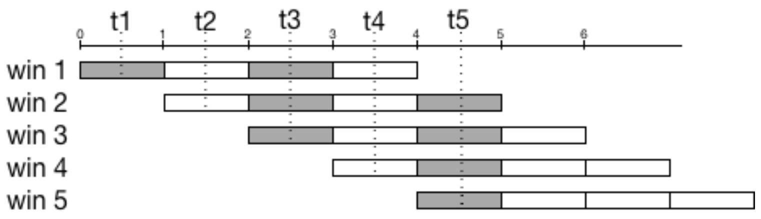


Figure 19: Active worker model: example with time-based windows.

When the Emitter receives an input tuple, it applies the mapping function to evaluate which windows will contain that tuple. Fig. 19 shows an example of mapping function between tuples and windows: t_1 belongs to the window win_1, t_2 belongs to win_1 and win_2, t_3 belongs to win_1, win_2 and win_3, t_4 belongs to win_1, win_2, win_3 and win_4 and t_5 belongs to win_2, win_3, win_4 and win_5.

The Emitter multicasts each tuple to the workers assigned to at least one

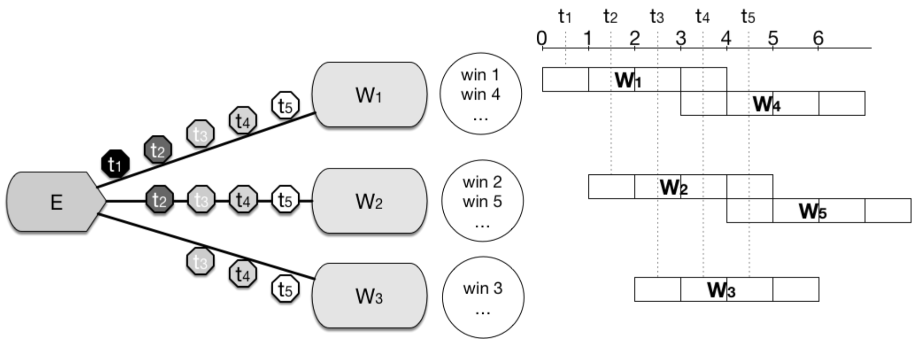


Figure 20: Active worker model: example of tuples routing

of the windows containing that tuple. With the round robin (RR) assignment policy, the correspondence between windows and workers is calculated based on the value of the *modulus* between the window identifier and the number of the workers. We can see this evaluation in the Code 3 at line 19.

Fig. 20 shows how the windows are "assigned" to the workers following the RR policy, so t_1 is send to $Worker_1$, t_2 is send to $Worker_1$ and $Worker_2$, while t_3, t_4 and t_5 are send to $Worker_1$, $Worker_2$ and $Worker_3$.

When the Emitter receives a generic tuple t from the Generator, it calculates the set of Workers that must receive t, and sets the attribute *numWorkers* of the tuple with this value. Then the Emitter sends to the "interested" Workers a pointer to the structure representing t.
When a Worker receives the pointer it copies the tuple in its "private" window buffer and decreases the attribute *numWorkers* of t: when *numWorkers* reaches zero it means that t has been already copied into the private buffers of all the "interested" workers, so the Worker that finds the counter equal to zero can deallocate the original referred tuple t. In order to be correct, this synchronization on the counter is implemented by using C++ atomic counter [28] that provides atomic increase/decrease operations on it.
The private copies of t are now stored only in the window buffers of the workers that manage the windows containing t. As shown in Sect. 5.3.3.2, the *expiring* function that concerns these copies must be called internally from the workers themselves, and it is related to private data only in this model.

5.3.3.2 Worker

Each worker is in charge of applying the function F to a window when it is triggered. At the creation of a new window win_i, a new *descriptor object* is created that contains window information, including the starting and the ending point of win_i.

When the Worker receives the pointer to t from the Emitter, it copies t in its "private" window buffer and, based on the window information, it determines whether the tuple t belongs to the current window win_i or not. If t belongs to win_i, then t is only appended to the *deque*, otherwise it means that t belongs to the next window, so t is a *triggering tuple*: in this case the current window win_i is closed and triggered by applying the function F and the tuple t is appended to the deque.

In the following code [Code 4] is shown how the creation of the window and the insertion new of the tuples are managed:

Code 4: Svc Worker (1)

```
...
queue->push_back( *(tuple) );

//if tuple ts is over the win ts and not the first
while( tuple->getTupleTimeStamp() > end_ts ){
        if ( tuple->getTupleTimeStamp() > end_ts && isFirst == false) {
                curWinDesc->isClosed = true;
                countCloseWindow++;
                startNewWin = true;
                retVal = 1;
        }
        //is a new window
        if (startNewWin) {
                countOpenWindow++;
                queueWin->push_back(*(new WinDes()));
                curWinDesc = &queueWin->back();

                //if is the first time start from 0
                if (isFirst) {
                        curWinDesc->startDesc = start_ts;
                        curWinDesc->startPos = curWinDesc->endPos = 0;
                        curWinDesc->winId = 0;
                } else {
```

```
            curWinDesc->startDesc = start_ts + sliding;
            curWinDesc->startPos = curWinDesc->endPos = queue->size();
            curWinDesc->winId = countOpenWindow-1;
        }
        curWinDesc->endDesc = curWinDesc->startDesc + winLength;

        start_ts    = curWinDesc->startDesc;
        end_ts      = curWinDesc->endDesc;
        start_pos   = curWinDesc->startPos;
        end_pos     = curWinDesc->endPos;
        isFirst     = false;
        startNewWin = false;

    } else {  //update the current window
        curWinDesc->endPos = this->queue->size()-1;
    }
}
...
```

Code 5 shows how the user function F is evaluated. F is common to all the implementations done. In fact, the function F is completely generic and must respect a predefined signature. Its implementation is responsibility of the user that instantiates our high-level pattern. F is called by the *getResult* function applied on the current window:

- *ret* is the return value of the function and its type is *output_t* (window result structure);
- *currStart_ts* is the timestamp of the instant where the worker starts the computation on the current window;
- *workerId* is the identifier of the worker;
- *tmpStart* and *tmpEnd* are the starting end ending timestamps of the window;
- *currElem* is the descriptor of the current window.

Code 5: Svc Worker (2)

```
...
ret=currStateTB->getResult(currStart_ts, workerId, tmpStart, tmpEnd, currElem);
...
```

As shown in Fig. 20, each worker has a "private" *deque* where it maintains a "private" copy of the tuples received from the Emitter.

We focus on $Worker$ 1: at the arrival of a tuple t_i, worker W_1 executes a *push-back* operation in order to insert t_i in its *deque*. Then, W_1 evaluates whether it is a *triggering* tuple or not: t_i is a *triggering* tuple *if and only if* it is the first received tuple after the end of the current window interval expected by that worker [Def. 9].
Fig 21 shows a case where t_5 is the triggering tuple for the window win_1.

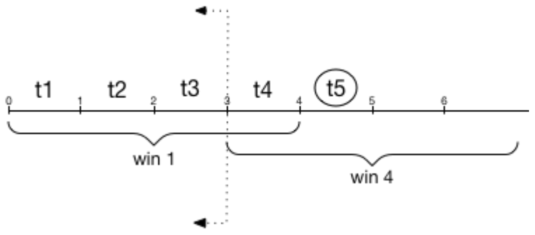


Figure 21: Active worker model: triggering and non-triggering tuples

When W_1 receives the tuple t_5, it triggers the window win_1, and it applies the function F over all the tuples belonging to the "closed" window win_1.

It is worth nothing that, in this implementation we cannot have any read-/write conflict because the deque is private and owned by the worker, and all the read/write operations are executed by the worker (the same thread) sequentially on private data only.

The worker is also responsible for managing the expiration of the oldest tuples in its private deque: all the tuples with timestamp falling in the first $\Delta * n_w$ time units of the window temporal range are expired and must be removed. We recall that Δ is the sliding factor and n_w is the number of the workers. In this case, at the end of the triggering, W_1 can expire all the tuples it does not need anymore that are the ones belonging to the first worker "sliding interval": in Fig 21 the expiring function concerns t_1, t_2 and t_3, while t_4 still belongs to win_4. In this way the processing of the windows results partitioned among workers.

5.4 Agnostic worker model

In the *agnostic worker model* the Workers are no longer responsible for the management of the windows, but the Emitter is the "*pivot*" of the elaboration. i.e. the Emitter manages and updates the window buffer in a centralized fashion while the workers do not maintain any private buffer in their internal state. An overview of this implementation is shown in Fig. 22.

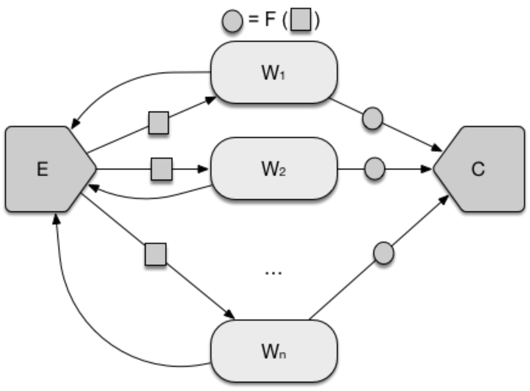


Figure 22: Agnostic model of the window farming pattern

When all the tuples belonging to window win_i are received from the Emitter, this means that win_i is "ready" to be triggered, and the Emitter selects the *first free worker* (e.g., according to the on-demand scheduling policy) and sends to it proper window metadata, such that the destination worker can execute the function F on the window content.

In the agnostic model the choice of the worker in charge of executing the current "ready" window must be based on the current *workload* of the workers, and must be a policy able to exploit at best the workers' computing capabilities, in such a way as to optimize load balancing. Our idea is to use *feedback* messages from the workers to the emitter, as shown in Fig. 22. This feature will be analyzed in Sect. 5.4.2.2 for the time-based window model and in Sect. 5.4.3.2 for the slide-by-tuple window model.

5.4.1 Applicability

In contrast to the *active workers model* [Sect. 5.3.1], in the agnostic case we do not need to evaluate the mapping function *a-priori* because this evaluation is directly done at run-time. This is the reason because this approach can be applied to all the studied window models, not only CF models. In fact, in this work we developed two implementations, one for a CF model (time-based windows) and one for a FCA model (slide-by-tuple windows). [Sect. 2.2.1]

Although compatible with any window model, the agnostic version may have some problems. The window management is assigned to the *Emitter*, so while the Workers workload can be decreased, the Emitter might become a *bottleneck* because it is in charge of managing the window buffer and tuple expiration in a centralized fashion. It must store all the input tuples, evaluate when windows are ready to be triggered, determine which worker is *available* to compute, and execute other functionalities described in Sect. 5.4.2.2. This may increase the emitter service time significantly.

Interestingly, from the traffic viewpoint we can say that, with respect to the *active workers model*, the frequency of messages generated from the Emitter to the Workers is lower than in the active case, because the Emitter does not send tuples (i.e. their pointers), but only special messages providing information about which window is triggered and how and where the worker can find the tuples of the corresponding window extent [Sect. 5.4.2.2]. This happens every window triggering and not at every tuple arrival.

5.4.2 Time-based window model

As in the active case [Sect. 5.3.3], we start the agnostic analysis and implementation with a time-based window model. During this work we incurred in some problems related to the use of the *deque* data structure [Sect. 5.3.2.1]: in the following section we show in detail those issues and the new data structure we have created for this model. Its name is *fqueue* [Sect. 5.4.2.1].

5.4.2.1 fqueue data structure for time-based windows

In the *active workers model* the data structure *deque* [Sect. 5.3.2.1] was utilized to store the tuples into the private worker window buffer, because there are no shared data and thus read/write conflicts between workers do not take place.

Instead, in the *agnostic* case, as specified in detail in Sect. 5.4.2.2, we have only one shared buffer (the Emitter buffer) that contains all the tuples received. A single deque cannot be shared and used concurrently by the emitter and the workers. For example, it may happen that while the Emitter writes on a chunk, one or more workers try to read other tuples of the same chunk. The chunk memory may be relocated by the deque implementation [Sect. 5.3.2.1], and this may cause an incorrect behavior and potential run-time error (segmentation fault). For this reason, we introduce a new special container for our purposes, the *fqueue.*

The *fqueue* implementation is based on the *deque* principles: as in a *deque*, we have a list of chunks but we manage differently the size and the internal structures of the chunks and the memory allocation areas.

In a *deque* we are able to know neither the size of chunks, nor the mapping function between tuples and chunks. This situation denies us the possibility to manage the read/write conflicts.

In this implementation of the *fqueue*, we do not know *a-priori* the size of each chunk (that is the number of tuples belonging to it), but we associate to each chunk a fixed temporal interval. Let T_i be the temporal interval of the i^{th} chunk. Based on its timestamp, each tuple will be stored in a specific chunk, i.e. the one whose temporal interval contains the timestamp of the tuple [36]. In this way, we can guarantee that a chunk cannot contain in the middle the ending point of a window and the starting point of the next one.

This is achieved by properly choosing the size of the chunk time interval which is as follows:

$$end\_time = start\_time + GCD(winSize, slideSize) \tag{16}$$

That is all the chunks have the same temporal length.

In this way, until the Emitter writes on a chunk, the tuples in that chunk are not needed by any worker because they do not belong to any triggered windows in execution somewhere. A chunk will be read by a worker only when it is closed (all the tuples within to its interval have been received) and it will not be further modified by adding new tuples by the Emitter.

Furthermore, we can be sure that also the expiring operation generates no read/write conflicts, because the chunks do not interfere with each other (only the last and the first chunk can be modified by the emitter). In particular, the emitter will delete the first (oldest) chunks only when all the windows using them have been computed by the workers. This aspect will be further commented later in this section.

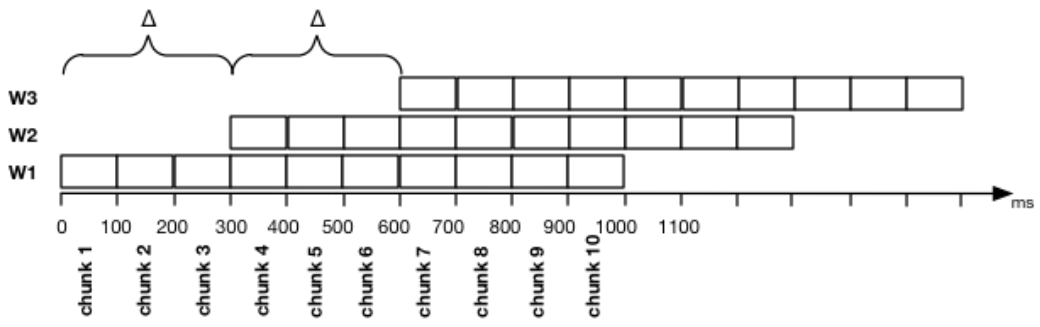


Figure 23: fqueue example

Fig 23 shows the management of the chunks in a *fqueue*: the window size is set to 1 *sec* and the sliding value is set to 300 *ms*, so the $GCD(1000, 300) = 100$ *ms*. By applying the formula (16) starting from 0 we obtain: chunk 1 $[0 - 100)$, chunk 2 $[100, 200)$ and so forth.

Since we do not know how many tuples will be stored in a chunk, it is implemented by a dynamic container, i.e. a standard C++ STL *vector*.

5.4.2.2 Emitter

The emitter is in charge of collecting and storing input tuples into the fqueue. In this way there is always one single copy of each tuple and it resides in the

Emitter fqueue shared with the workers.

At the arrival of a new tuple, the Emitter adds it to the *fqueue* [Sect. 5.4.2.1]. After that, it checks if the tuple timestamp is greater than the timestamp of the ending point of the window: if true it closes the current window and it "signs" it as *triggered*, then it updates the number of windows closed and starts (logically) a new window.

The triggered window has to be sent to a worker in charge of applying the F function. The emitter chooses the first available worker which is currently idle. To this end, the emitter maintains for each worker a flag stating whether it is available to compute or not. Such flags are set to false when a window is scheduled to a worker, and marked as true when a feedback from that worker is received by the emitter.

Each sliding window shares a group of chunks with previous and next windows. When a windows is triggered, the next window slides forward of one or more chunks, depending on how many chunks are needed to cover the sliding interval (in the example of Fig. 23 three chunks):

$$num\_chunks = \frac{sliding\_value}{GCD(winSize, slideSize)} \tag{17}$$

The Emitter non-deterministically receives two different types of input messages: the first is a tuple received from the *generator*. The second are *feedbacks* sent from the workers, which notify the end of the elaboration of a window by the sending worker. The feedback contains the window starting timestamp and the worker identifier.

At the arrival of a feedback message, the Emitter has to update the list of idle workers and starts the *expiring* phase: this phase is critical because we must evaluate with attention which tuples have to be expired in order to avoid deleting tuples still in use by some workers.

The use of *fqueue* [Sect. 5.4.2.1] grants us to do this evaluation. If the worker's feedback contains a window starting point greater than ending point of a chunk, we are sure that all the tuples in that chunk are no more required, and it can be safely removed.

However, when more than one worker is active, it may be possible that feedback massages are received by the emitter out-of-order. To manage those cases, the emitter stores the "out of order" feedbacks [Code 6 at line 18] received in a *priority queue* data structure (implemented by a deque), and it maintain a variable *minOnTheFly* where it records the smaller window starting timestamp.

Code 6: Agnostic expiring function

```
 1  if ((*t).isSpecialTuple()) {
 2          workerStatus[idWorkerSpecialTuple] = 0;
 3          nextWorker--;
 4
 5          if(minOnTheFly == ((*t).getTupleValue())[0]){
 6                  minOnTheFly = cleanInConfirmedTsWorker(minOnTheFly);
 7
 8                  if(currentQueue->size() > 0) {
 9                          lastDeleted = (*currentStateTB->getTupledeque())->
                                erase(minOnTheFly);
10                          lastDeleted = currentStateTB->evictionUpdate(
                                lastDeleted, minOnTheFly);
11                  }
12                  if (lastDeleted > 0 && (startValue - lastDeleted) > 0) {
13                          startValue -= lastDeleted;
14                          lastDeleted = 0;
15                  }
16                  minOnTheFly+=slide;
17          } else {
18                  insertInConfirmedTsWorker(((*t).getTupleValue())[0]);
19          }
20  }
21  ...
```

After inserting an "out of order" feedback in the deque, when we receive the feedback with window starting timestamp corresponding to *minOnTheFly* we must run a preliminary expiring operation: we check if in the "out of order" feedbacks queue there are some window starting timestamps contiguous to the current *minOnTheFly* and we update *minOnTheFly* with the greater value we find [Code 6 at line 6]. Then, we are ready to run the expiring function based on the updated value of *minOnTheFly* [Code 6 at lines 9 and 10]. We can remove all the chunks with starting time smaller than *minOnTheFly*.

5.4.2.3 Worker

The worker receives from the Emitter a window descriptor data structure (a *task*) containing the information about the window to elaborate, in particular a *window tuples iterator* and a *window descriptor*.

The *window tuples iterator* allows a simple access to all and only the tuples belonging to the window. Since the tuples are stored in the *fqueue*, we cannot use a standard iterator, but we must build a custom iterator in charge of scanning the *fqueue* data structure, in particular the portion of the *fqueue* containing the tuples belonging to that specific window.

The *window descriptor* includes information such as the window identifier, the starting and the ending timestamps and other variables used to check the state of the window in order to perform the elaboration correctly.

Code 7: Agnostic Svc Worker function

```
output_t *svc(TaskTBL *t) {
...
        #if STATS == 1
        TICKS_START = (*t).getTicksFireWin();
        #endif
...
        queueType::iterator *tmp = (*currentStateTB->getTupledeque())->
            getITListChunk(startTs, endTs);
        if( tmp != NULL ) {
                FUN_ITERATOR_TYPE _begin = *tmp->beginIT();
                FUN_ITERATOR_TYPE _end   = *tmp->endIT();
                ticks CALCULATION_START = getticks();

                size = distance(_begin,_end);
                ret = currentStateTB->getResult(currentStartWinVal, workerId,
                    _begin, _end, currentElem);

                ret->setCalculationTicks(getticks()-CALCULATION_START);
                lastIdRes = ret->getTupleId();
                delete tmp;
        }else{
```

```
20                  ++lastIdRes;
21                  ret = new output_t(0, lastIdRes, currentStateTB->getKey(),
                        workerId, startTs, endTs);
22          }
23  ...
24          if (limitUP != -1 && ret != NULL && (*ret).getTupleEndInterval() <=
                limitUP) {
25                  ret->setNumElemInWin(size);
26                  ret->updateWorkerId(workerId);
27                  ret->updateCurrTs((*ret).getTupleStartInterval());
28                  #if STATS == 1
29                  ret->setEmitterLatencyTicks(TICKS_START);
30                  #endif
31
32                  ff_send_out_to(ret, 1);
33                  ff_send_out_to(createSpeciaTuple(0, workerId,
                        currentStartWinVal), 0);
34                  operationDone = true;
35                  if (receiveEOS == true) {
36                          ff_send_out_to(EOS, 1);
37                  }
38          }
39          delete t;
40          return GO_ON;
41  }
```

As we can see in Code 7 at line 14, we apply the F function to the window through the *getResult* function. At the end of the elaboration, if the result *ret* is not null, the worker sends *ret* to the Collector [Code 7 at line 32], then it also send a feedback to the Emitter [Code 7 at line 33].
As said in Sect. 5.4, in the agnostic worker model the workers essentially are in charge of applying F to the windows. Therefore, they do not run the expiring function because the tuples are managed by the Emitter only.

5.4.3 Slide-by-tuple model

As discussed in Sect. 5.1, the *agnostic worker model* is applicable both to CF window models (e.g., time-based windows [Sect. 5.4.2]) and to FCA window models [Def. 2.2.1]. In this section we show how to adapt the agnostic worker model for a class of FCA windows, e.g., slide-by-tuple windows.

5.4.3.1 fqueue data structure for slide-by-tuples windows

Although slightly different from the previous example (time based model), the slide-by-tuple model needs to address further issues. Fig. 24 shows an example in which the windows size is set to 1000 ms, the sliding value is set to 10 tuples and each chunck is a vector that contains the tuples in an interval of 250 ms (we can use any other temporal interval in this example).

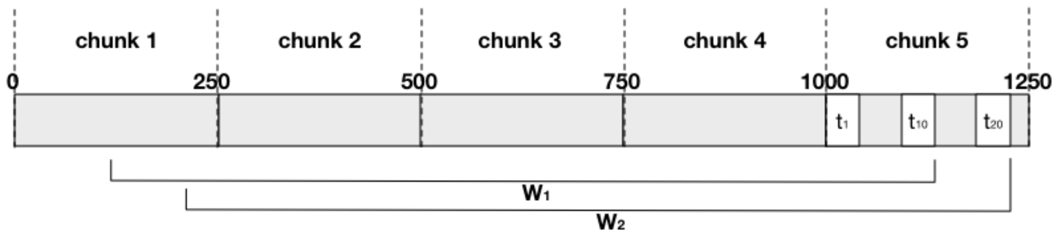


Figure 24: Example of fqueue. Window size= 1000 ms and Δ = 10 $tuples$. We suppose to use chucks with fixed temporal interval as previously. Suppose each chunk is associated with an interval of 250 ms

.

Suppose that t_1 is the first tuple of *chunk* 5 and t_i is the i^{th} tuple. We can observe that the *fifth* chunk contains tuples belonging both to window W_1 and W_2.
At the arrival of tuple t_{10}, that is the triggering tuple for W_1, the worker assigned to W_1 ($Worker1$) starts to read the tuples belonging to that window, where some of those tuples are contained in *chunk* 5 (t_1-t_{10}). In the meanwhile, the emitter modifies *chunk* 5 at the arrival of $t_{11} - t_{20}$ tuples. It is possible that the vector implementing the chunk does not have enough space to contain these new tuples, so it may be relocated. In this case the pointers of $Worker1$ to the tuples t_1-t_{10} may become invalid.

For this reason the fqueue with chunks implemented by vectors is not suitable for this model. We solved this issues by implementing chunks with a predefined size statically fixed: e.g. we used a C++ STL *array*. In this way we are sure that the pointers to the tuples do not change because the array is never relocated.

In the array version, each chunk has a static size in terms of number of tuples, which is a fqueue configuration parameter. For this reason, each chunk is now associated with a temporal interval of variable length which is not known statically. Fig. 25 shows the use of a fqueue with the array version. In this example we set the array size to 10 tuples.

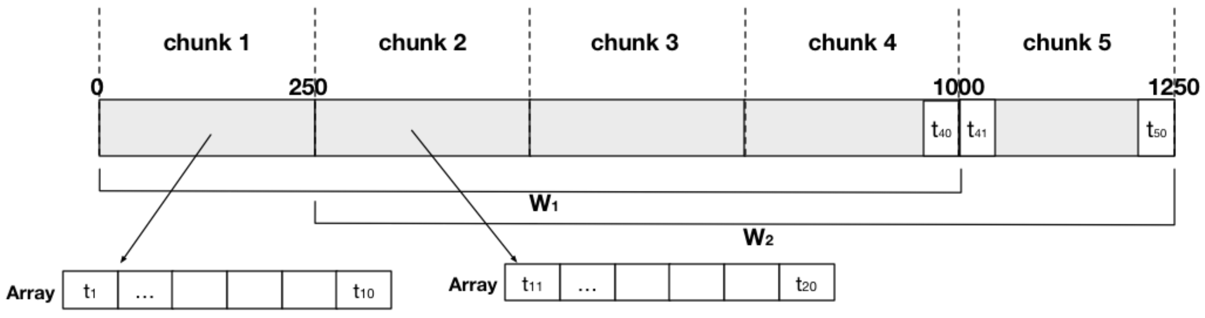


Figure 25: Example of fqueue with chunks implemented as arrays of size = 10. Window size = 1000 ms and $\Delta = 10$ $tuples$

The smaller is the array size the higher is the *fragmentation*. It means that the tuples belonging to a window extent are not stored in a unique data structure, but they are *fragmented* into different arrays.
The steps of the tuple insertion in the fqueue are:

1. If the last chunk has enough space, the tuple is appended there. Otherwise a new array (chunk) is created and appended to the chunks list;
2. the starting point of the new array is set with the timestamp of the new input tuple t_i (we also maintain for each chunk its starting and ending time for reasons that will be described later);

Fig. 26 shows an example. In this example, the array size is set to 3 *tuples* and the sliding value Δ is set to 10 *tuples*:

At the arrival of t_1 the tuple is inserted in the first array (chunk), and the same is done for t_2 and t_3. But what happen when t_4 arrives? The tuple

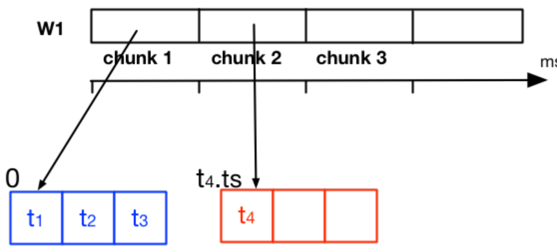


Figure 26: fqueue implementation with arrays.

belongs to the current window, but the first array (blue) is full. We need more space, so we create a new array (red one) in which we insert the new input tuple t_4.

At the arrival of tuple t_{10}, that is the triggering tuple for the current window, we must scan the tuples to search the the starting point of the window, $Tws = t_{10}.ts - window\_size$.
The easier mechanism is based on the *tuple-by-tuple scan*, but it can be expensive. To optimize this research, we set additional information on the chunks, i.e. the *ending point*; which is set to the timestamp of the last tuple when it is full. In this way, we can compare Tws with the starting point (Ts) and the ending point (Te) of each chunk: if Tws is between Ts and Te we know that the first tuple of the window belongs to this chunk, so we scan tuple-by-tuple only this chunk. Otherwise, we continue to compare Tws with the previous chunks ends up to the first one. If the Tws value is negative (only for the first windows) we choose as the window starting point the first tuple of the first chuck.

We must consider that typically the arrays of the fqueue implementation are stored in memory areas *not* contiguous. This may cause a certain overhead for reading the tuples that will be discussed in the experimental section.

5.4.3.2 Emitter

The Emitter has a very similar implementation as the one for the time-based model [Sect. 5.4.2.2]. The main difference is the different fqueue implementation previously discussed. Both models use the same interface, i.e. from the user perspective the operations are the same, but internally they are implemented in a different way. More specifically:

- *Insertion operation*: if a chunk is implemented by a vector, this operation is a simple *push_back*. In the array case, instead, the insert operation consists adding the tuple to the current array (chunk), or eventually on the next chunk [Code 8 at line 8].

Code 8: Array push_back

```
1  if ((*it).getNumElem() < (*it).getLimitRange()) {
2          (*it).add(elem);
3          ...
4  } else {
5          ...
6          (*it).setEndTs(lastElemet.getTupleTimeStamp());
7          ...
8          myList->addSpecialChunkARRAY(elem->getTupleTimeStamp());
9          ++it;   //move to the new chuck
10         (*it).add(elem);
11         ...
12 }
```

- *Eviction operation*: In the slide-by-tuples model, that is in the array case, a chunk can cover more than one sliding interval. In this case we must wait for the end of elaboration of all the windows whose starting point belongs to the chunk. After that, the chunk can be safely removed.

5.4.3.3 Worker

The worker implementation is exactly the same of the one adopted for the time based model [Sect. 5.4.2.3]. As in the Emitter case, some small differences exist and are related to the operations on the fqueue because the two models use different versions of the data structure. The main difference is the reading operation of a window extent that is based on an *iterator* implemented in different way in the two fqueue versions.

Even if we have two different versions of the fqueue, the signature of the user function F is the same, as it is based on a general *iterator* concept. In this way we allow the user to write the F function once and to use it independently of the fqueue implementation and the underlying parallel pattern implementation.

6 Experiments

In this section we will describe the evaluation of the proposed alternative implementations of the window farming pattern on a shared-memory machine. The goal of this evaluation is to assess the performance of the proposed versions in order to evaluate which is the best solution and to understand their differences.

For the CF-window model [Def.2.2.1] (e.g., time-based windows) we can adopt both the active and the agnostic version, so we can compare the two implementations by evaluating different performance metrics that they are able to achieve. In particular, the metrics considered in this work are: the throughput, calculation time and scalability.

In the following experiments we will compare the different implementations based on these performance metrics. We use the following basic configuration for all the tests:

- *Number of workers*: from 1 to 13. In our machine we have 16 cores, three of them are reserved to execute the generator, the emitter and the collector threads, while the other 13 cores are available for the workers.
- *Tuples rate*: 100.000 tuples/sec. The inter-arrival times of the tuples (i.e. the average difference between two timestamps of consecutive tuples) are exponentially distributed. We consider an average rate of 100.000 tuples per second, because it is an intensive rate that allows us to analyze our application in a high-frequencing context, such as the one of a financial application [14].

In our tests the other parameters such as the *window size* and the *sliding value*, are not fixed. In this way we are able to compare the implementations under different configurations.

6.1 Benchmark query

In our work we use a benchmark application in order to study the sliding window pattern. The F function is based on *lmfit*, a C++ library that provides a high-level interface to non-linear optimization and curve fitting problems. The library, based on the Levenberg-Marquardt regression algorithm, provides a number of useful enhancements to optimization and data fitting problems, and it has been used in past similar problems [42].

At each activation of a new window, the operator executes the *Levenberg-Marquardt* regression algorithm over all the points (tuples) within the window, and produces a polynomial fitting.

6.2 Time-based windows analysis

For the time-based window model we can compare the performance metrics achieved by both the active and the agnostic implementation in order to understand the differences in terms of scalability, bandwidth and performance, so in general both relative and absolute metrics. We see in detail three comparisons, one for each distinct metric.

6.2.1 Processing bandwidth

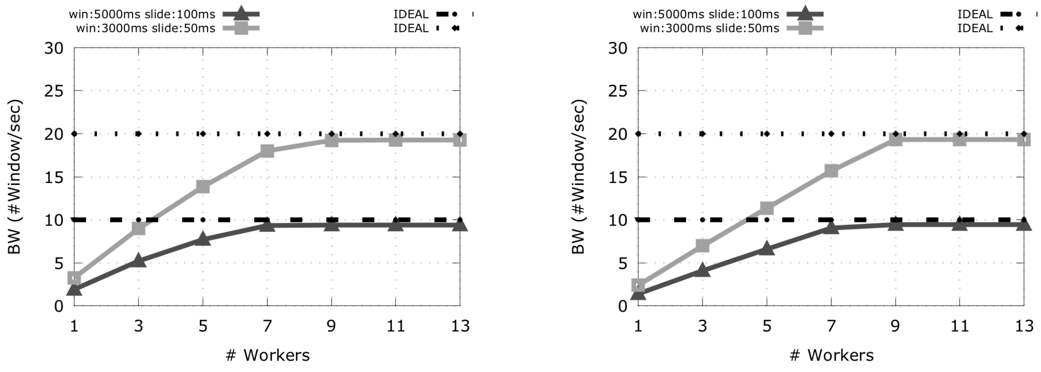
A sequential execution may not be able to elaborate the input stream in real-time because this could be faster with respect to its processing capability. In this case we have a *bottleneck*. In order to avoid this situation we need to increase the parallelism degree.

We want to understand if there are some performance differences between the active and the agnostic approach. Considering an equal value of the parallelism degree, which version is faster? Which version has a higher scalability? This is the goal of the bandwidth analysis.

In the first two graphs inf Fig. 27 we compare the bandwidth values of the active and the agnostic implementations. For each execution we can calculate the ideal value of the bandwidth according to the chosen configuration parameters (*window_size* and *sliding_value*). The *ideal bandwidth* is the maximum number of windows that can be processed per second. For example, if we have *window_size* of 1000 *ms* and *sliding_value* of 100 *ms* in the ideal case we elaborate 10 windows per second: we cannot elaborate more than 10 windows, but a smaller number in case the parallelization is a bottleneck.

Fig. 27 plots the behavior of the active (left) and the agnostic model (right) on two different window configurations. In both cases we reach values of the bandwidth close to the ideal value. We use two different sliding-window configurations:

- (1) - *window_size* = 5000 *ms* and *sliding_value* = 100 *ms*
- (2) - *window_size* = 3000 *ms* and *sliding_value* = 50 *ms*



(a) (b)

Figure 27: Active (a) and agnostic (b) model bandwidth values

Using the configuration parameters (1) the active and the agnostic models have a similar behavior: they reach the ideal value of the bandwidth with at least 7 workers activated. Actually, in the agnostic case with 7 workers we are very close to the ideal bandwidth that can be exactly achieved with 8 workers. In the scenario (2) the active and the agnostic models both need 9 workers instead of 7 to reach a value close to the ideal bandwidth because the slide factor is smaller, so they are triggered more frequently.

In both cases, we observe that the active model achieves a greater value of bandwidth with parallelism degrees smaller than the one required to remove the bottleneck in that configuration. Therefore, the active model is in general faster than the agnostic implementation. The main reason is that the fqueue implementation is less efficient with respect to the data structure used in active model (deque). The fqueue memory layout is influenced by the window definition, in particular by the number of chunks participating to a window: the higher is the number of chunk the worse is the agnostic calculation time because the fqueue is slower. In contrast, the active model is independent of the window definition.

Now, we show an example with two configurations where both the models are unable to reach the ideal bandwidth:

- (3) - $window\_size = 5000$ ms and $sliding\_value = 50$ ms
- (4) - $window\_size = 1000$ ms and $sliding\_value = 10$ ms

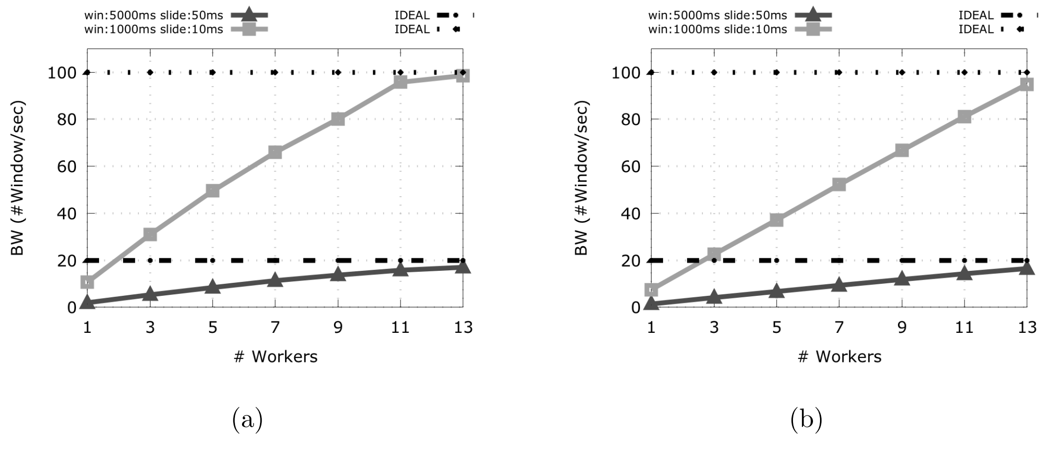


Figure 28: Active (a) and agnostic (b) model bandwidth

In case the $windows\_size$ is 1000 ms and the $sliding\_value$ is 10 ms the ideal bandwidth value is 100. While the active model reaches a throughput very close to the ideal bandwidth with 13 workers, the agnostic model does not reach the ideal bandwidth because it needs more than 13 workers that are unavailable on that machine. In case of $windows\_size$ of 5000 ms and $sliding\_value$ of 50 ms instead, both the models do not achieve the ideal bandwidth. The agnostic model reaches 16, 50 window/sec. with 13 activated workers while, in correspondence of 13 workers the bandwidth value of the active one is of $17,02$ window/sec.

6.2.2 Completion time

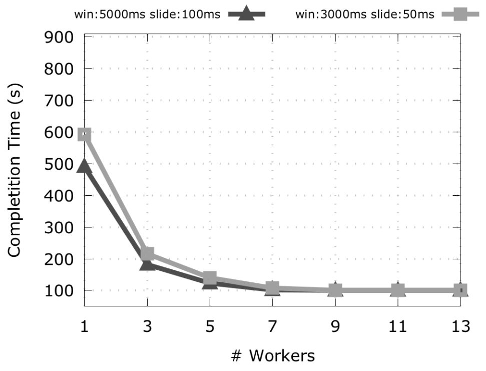
T_{COMP} is a metric that represents the time needed for the elaboration of all the input tuples, i.e. the completion time of the entire stream.

In our experiments the generator sends tuples with a rate of 100.000 $tuples/sec$ and the total tuples generated are 10 *millions*. Therefore, the stream generation time is of 100 seconds. In case the computation is not a bottleneck, we expect that the completion time will require a little bit more than 100 seconds (we pay the latency of the last window).

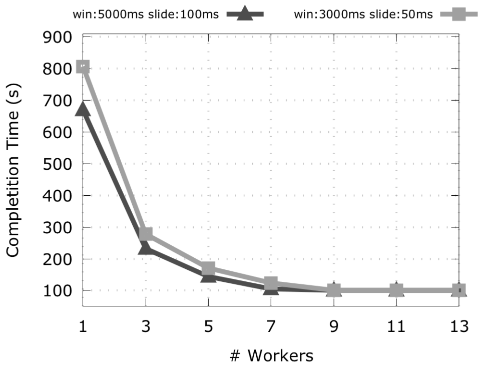
It is obviously possible to correlate the completion time and the scalability. The completion time decreases until the scalability increases, and this happens as long as we reach a completion time close to 100 seconds (if we are able to remove the bottleneck in that configuration).

In the completion time analysis we use the same configurations studied for the bandwidth analysis:

- (1) - $window\_size = 5000\ ms$ and $sliding\_value = 100\ ms$
- (2) - $window\_size = 3000\ ms$ and $sliding\_value = 50\ ms$



(a)



(b)

Figure 29: Active (a) and agnostic (b) model completion time

Fig. 29 shows two cases where T_{COMP} decrease up to the parallelism degree that eliminates the bottleneck, that corresponds to the decrease with the highest scalability value. Fig. 30 plots the two cases:

- (3) - $window\_size = 5000\ ms$ and $sliding\_value = 50\ ms$
- (4) - $window\_size = 1000\ ms$ and $sliding\_value = 10\ ms$

In these configurations the two models do not reach the ideal completion time of 100 *sec*, but T_{COMP} continuously decreases by increasing the workers number.

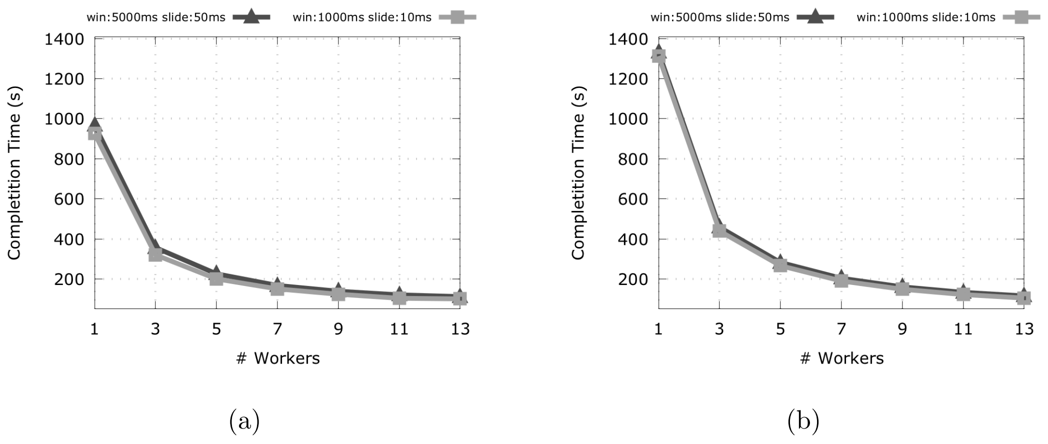


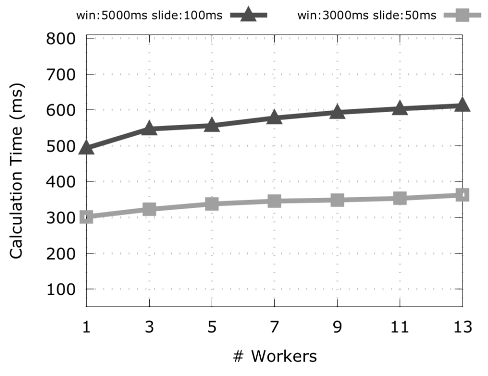
Figure 30: Active(a) and agnostic (b) model completion time

6.2.3 Calculation time

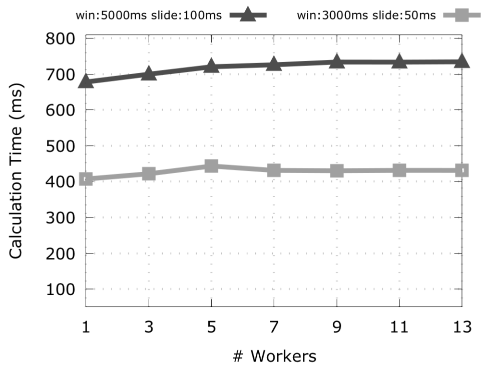
The calculation time (T_{CALC}) represents the average time spent by a worker for a window evaluation. In the ideal case the T_{CALC} value should be constant and independent of the workers number. Fig. 31 plots our experiments with the configuration (1) and (2):

- (1) - $window\_size = 5000\ ms$ and $sliding\_value = 100\ ms$
- (2) - $window\_size = 3000\ ms$ and $sliding\_value = 50\ ms$

In Fig. 31 we note that in the agnostic case [Fig. 31(b)] the calculation time is higher than in the active one. The reason is due to the fqueue implementation [Sect. 5.4.2.1], in particular the iterator implementation for accessing the tuples of the window.



(a)



(b)

Figure 31: Active (a) and agnostic (b) model calculation time

We also observe that in the agnostic case the T_{CALC} value is more stable with respect to the active one. In the active case shown in Fig. 31(a) with the configuration (1) the initial value is 500 ms, then it increases slowly up to 600 ms. Instead in the agnostic case the T_{CALC} variation is smaller.
Why this variation is higher in the active model with respect to the agnostic one? In the agnostic case the window buffer is *centralized*. The emitter writes the tuples and the workers read and elaborate the results on the windows without modifying the centralized window buffer managed by the emitter. In the active case instead, the memory hierarchy is more "stressed" because *each* tuple is *copied* in more than one private worker deque, so the memory hierarchy is likely more stressed. This is one of the main reasons for the increase in the calculation time by adding more workers especially in the active model.

6.2.3.1 Fragmentation analysis

The calculation time of the agnostic model is higher. The main reason is due to the windows fragmentation. *Fragmentation* is an important aspect in our experiments and it is directly connected to the calculation time. We know that T_{CALC} depends on the "length" of the window, in particular on how many tuples belong to the window. We observe that while in the active case all the tuples of a window are likely contiguous in memory (it depends on the internal deque policy but this is out of our control), in the agnostic case a window is for sure divided into several chunks.

Fig. 32 presents a comparison between the active case, and the agnostic case that suffers from fragmentation induced by our fqueue. The T_{CALC} values are evaluated with different window configurations. The window size is fixed 5000 ms while the sliding value assumes different values. Fragmentation is evaluated as: $\frac{window\_size}{GCD(window\_size, sliding\_value)}$ [Eq. 16], so with different sliding values we have:

- $sliding\_value = 500\ ms \ \rightarrow \ fragmentation = 10\ chunks$
- $sliding\_value = 250\ ms \ \rightarrow \ fragmentation = 20\ chunks$
- $sliding\_value = 100\ ms \ \rightarrow \ fragmentation = 50\ chunks$
- $sliding\_value = 50\ ms \ \rightarrow \ fragmentation = 100\ chunks$
- $sliding\_value = 10\ ms \ \rightarrow \ fragmentation = 500\ chunks$

In the deque implementation the internal fragmentation depends on the number of tuples in the window while in the fqueue case it depends on the sliding factor. This results in a higher impact of fragmentation in the calculation time of the agnostic version.

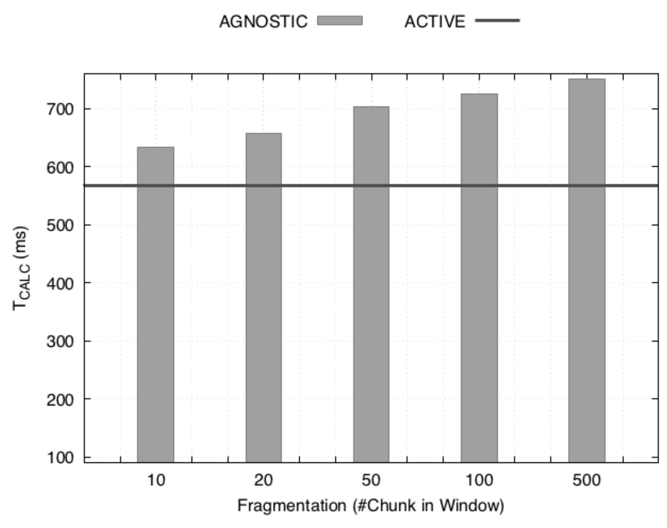


Figure 32: Fragmentation analysis: agnostic case with different sliding values compared to the active case

As shown in the Fig. 32, a higher number of chunks corresponds to a higher T_{CALC} value because the window buffer is more fragmented and the calculation time results higher.

6.2.4 Scalability

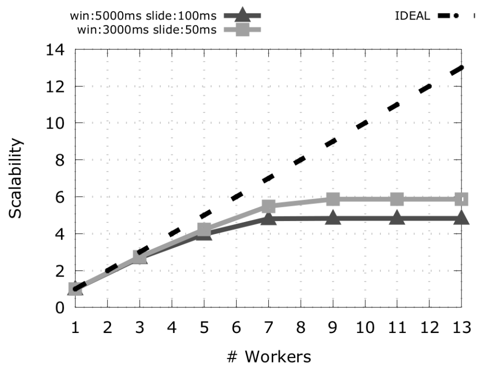
The value of scalability is indicative of how the system is able to exploit parallelism. We recall that, the scalability value with parallelism degree n is evaluated as the ratio of the bandwidth value obtained with parallelism degree n to the bandwidth value obtained with one worker:

$$S^{(n)} = \frac{B^{(n)}}{B^{(1)}} \tag{18}$$

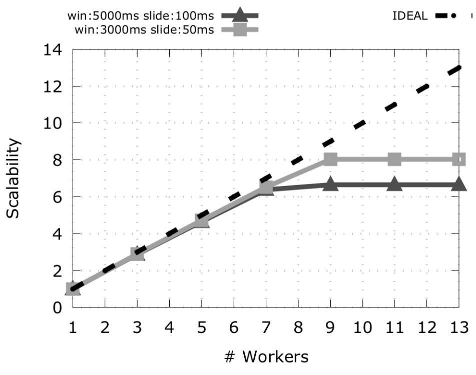
For the scalability analysis we use the same configurations studied for the bandwidth analysis:

- (1) - $window\_size = 5000\ ms$ and $sliding\_value = 100\ ms$
- (2) - $window\_size = 3000\ ms$ and $sliding\_value = 50\ ms$

The dashed line in Fig. 33 represents the ideal scalability. As we can observe, in the two graphs both the models scale until they reach the ideal bandwidth. Once the ideal bandwidth is reached, the scalability stops to increase because adding more workers has no effect on throughput.



(a)

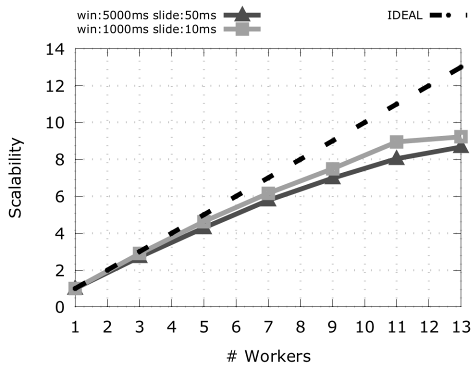


(b)

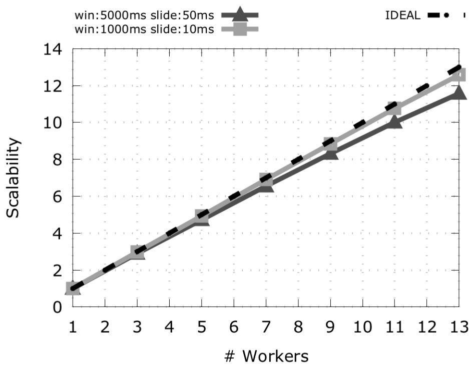
Figure 33: Active (a) and agnostic (b) model scalability

Fig. 34 shows the behavior from the scalability viewpoint of the configurations (3) and (4):

- (3) - $window\_size = 5000\ ms$ and $sliding\_value = 50\ ms$
- (4) - $window\_size = 1000\ ms$ and $sliding\_value = 10\ ms$



(a)



(b)

Figure 34: Active (a) and agnostic (b) model scalability

We recall that with those configurations both the models do not reach the ideal bandwidth value, so as expected the scalability value keeps increasing by adding more workers.

As we can see in [Fig. 34], the agnostic model scalability is closer to the ideal one respect to the active case. One of the reasons is that the agnostic model always guarantees load balancing because we use an *on-demand* window distribution. Instead, the active model has a window-workers assignment which is statically fixed in a round-robin fashion. In case of windows with large variations in their cardinality (caused by "oscillation" in the tuples arrival rate), the agnostic model still provides load balancing while the active model may be unable to provide it and thus its scalability may be sub-optimal. This aspect deserves to be better investigated in our future work.

6.3 Slide-by-tuple windows analysis

For the slide-by-tuple window model we can only implement the agnostic approach (as described in detail in Sect. 2.2.3.4). So, we study only this model in this part. Some of the basic windows configurations are the same used in the previous [Sect. 6]:

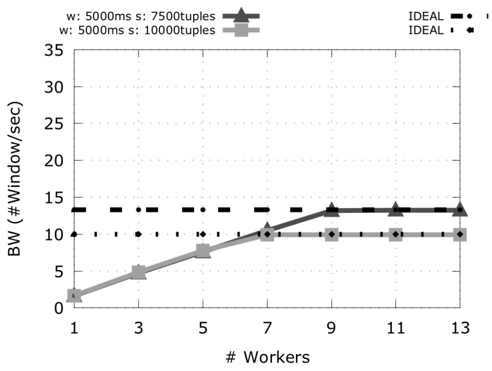
- *Number of workers*: from 1 to 13
- *Tuples rate*: 100.000 tuples/sec.

Also in this case the parameters *window_size* and *sliding_value* are not fixed, but we chose different values in order to show the model behavior in different situations.

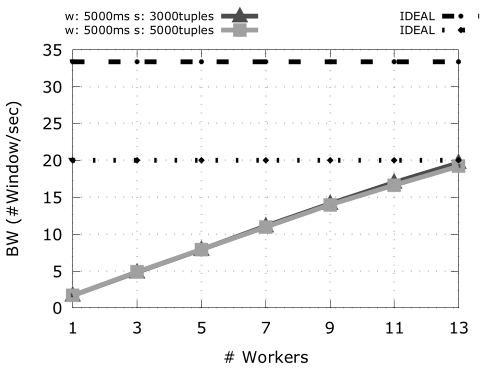
6.3.1 Processing bandwidth

We analyze our agnostic implementation with the following configurations:

- (1) - *window_size* = 5000 *ms* and *sliding_value* = 7500 *tuples*
- (2) - *window_size* = 5000 *ms* and *sliding_value* = 10000 *tuples*
- (3) - *window_size* = 5000 *ms* and *sliding_value* = 5000 *tuples*
- (4) - *window_size* = 5000 *ms* and *sliding_value* = 3000 *tuples*



(a) Configurations (1) and (2)



(b) Configurations (3) and (4)

Figure 35: Agnostic model bandwidth

In the figures the dotted lines represent the ideal bandwidth. In this case we have evaluated it as:

$$BW_{ideal} = \frac{tuples\_rate}{sliding\_value} \tag{19}$$

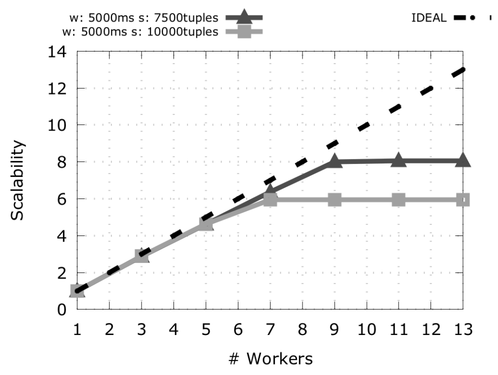
We recall that the *tuples_rate* is the number of input tuples per second and the *sliding_value* represent the number of tuples we have to count between two consecutive windows, so the ratio in Eq. (19) gives us the number of windows triggering per second.

Fig. 35(a) shows two cases in which the system can reach the ideal bandwidth (configurations (1) and (2)), so it is not a bottleneck. Clearly the configurations have different ideal bandwidth values. In the first case we reach the ideal value with at least 9 workers, while in the second case we are able to reach the ideal bandwidth with 7 workers.

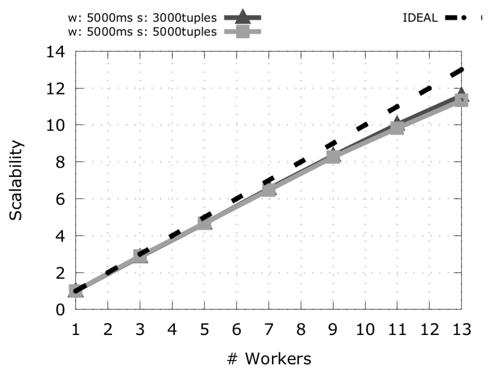
We tested the system also with configurations (3) and (4). In these cases [Fig. 35(b)] the system is not able to reach the ideal bandwidth, so it remains a bottleneck even if we use all the 13 workers. We can explain this by observing the *sliding_value*: the smaller is the *sliding_value* the faster is the windows triggering and the higher is the number of windows that each workers have to process. In this case the number of available workers is not enough to reach the ideal bandwidth.

6.3.2 Scalability

Fig. 36 shows the scalability in the same windows scenarios studied before. The results with the configurations (1) e (2) are shown in Fig. 36(a) and represent the cases where the system is able to reach the ideal bandwidth value so the scalability stops increasing by adding more workers. The results with the configurations (3) e (4) are shown in Fig. 36(b) and represent the cases where the system cannot reach the ideal bandwidth value. In those cases the system continues to scale because the available number of cores is not enough to remove the bottleneck.



(a) Configurations (1) and (2)



(b) Configurations (3) and (4)

Figure 36: Agnostic model scalability

6.3.3 Fragmentation analysis

The agnostic model for slide-by-tuple windows uses the fqueue implemented with arrays, so we can analyze the fragmentation value in case of different (chunk) array size. Fig. 37 shows the comparisons between T_{CALC} with different values of the array sizes.

In this test the *sliding value* is fixed to 5000 *tuples*, instead the *array size*, which is a configuration parameter, assumes different values:

- $array\_size = 313 \rightarrow fragmentation = 16\ chunks$
- $array\_size = 200 \rightarrow fragmentation = 25\ chunks$
- $array\_size = 152 \rightarrow fragmentation = 33\ chunks$
- $array\_size = 100 \rightarrow fragmentation = 50\ chunks$
- $array\_size = 50 \rightarrow fragmentation = 100\ chunks$

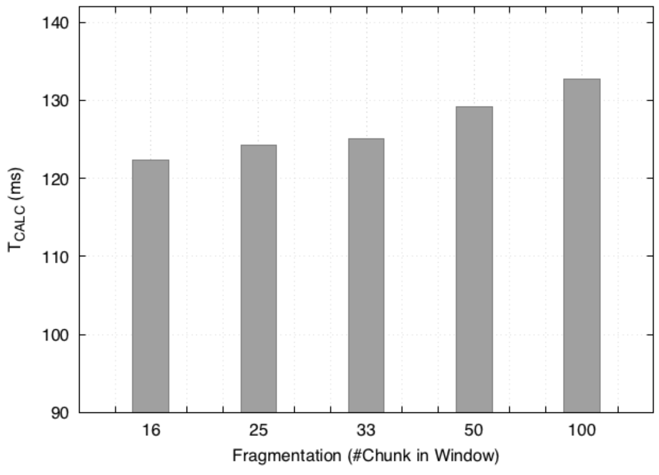


Figure 37: Fragmentation analysis: agnostic case with different array size

Also in this case with more chunks the window context is more fragmented and the cost to iterate overall the window tuples results higher.

7 Conclusion

This work has been very interesting and it has allowed me to learn and to understand the principles of computing parallel computations which is, nowadays, of great importance because of the increasing amount of data to be processed in real time and the diffusions of devices generating hugs sequences of data.

The agnostic approach has been developed not only as an alternative to the active one, but also as an "integration" to it because it is able to process all types of windows, even those the active approach is not able process.

From the comparisons between agnostic and active worker models for those window types that can be processed with both of them, it emerges that the active worker model has better performances in terms of bandwidth and completion and calculation times. This result is not a surprise and it is related to the data structures used and the scheduling policy. Instead, the data structures and the scheduling policies chosen for the agnostic implementation represented one of the most challenging parts of my work and allow the system to process all types of windows.
This work can be extended in several directions. The active model, although faster and applicable to some window definitions, may suffer of load balancing issues especially in streams exhibiting burstiness. This aspect deserves further investigation in the future.

Bibliography

[1] M. Aldinucci M. & Danelutto M. & Kilpatrick P. & Torquati. “Fastflow: high-level and efficient streaming on multi-core”. In: (2014).

[2] Andrea Cicalese Andrea Bozzi. “Parallel paradigms for Data Stream Processing”. Dipartimento di informatica di Pisa, 2012.

[3] David R. Butenhof. “Programming With Posix Threads”. 1997.

[4] Bertolli C. “Fault tolerance for high-performance applications using structured parallelism models. PhD thesis, University of Pisa”. In: (2008).

[5] J. Darlington J. & Guo Yi-ke. & To H. W. & Yang. “Parallel skeletons for structured composition. In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming”. In: (1995).

[6] C. Fetzer & E Lusk & P. Felber. “Speculation in Parallel and Distributed Event Processing Systems”. In: (2010).

[7] Salvatore Di Girolamo. “Skyline on sliding window data-stream: a parallel approach”. Dipartimento di informatica di Pisa, 2013.

[8] M. Liu & M. Li & D. Golovnya. “Sequence pattern query processing over out-of-order event streams”. In: (2009).

[9] L. Veraldi M. Vanneschi. “Dynamicity in distributed applications: issues, problems and the ASSIST approach”. In: (2007).

[10] J. Li & K. Tufte & V. Shkapenyuk & V. Papadimos & T. Johnson & D. Maier. “Out-of-order processing: a new architecture for high-performance stream systems”. In: (2008).

[11] E. Tiakas & A. N. Papadopoulos & Y. Manolopoulos. “Skyline queries: An introduction”. In: (2015).

[12] G. Cugola & A. Margara. “Processing flows of information: From data stream to complex event processing”. In: (2012).

[13] G. Mencagli & M. Torquati & M. Danelutto & T. De Matteis. "Parallel Continuous Preference Queries over Out-of-Order and Bursty Data Streams". In: (2017).

[14] G. Mencagli & M. Torquati & M. Danelutto & T. De Matteis. "Parallelizing High-Frequency Trading Applications by Using C++11 Attributes". In: (2015).

[15] Tiziano De Matteis. "Parallel Patterns for Adaptive Data Stream Processing". Dipartimento di informatica di Pisa, 2016.

[16] T. De Matteis & S. Di Girolamo & G. Mencagli. "Continuous Skyline Queries on Multicore Architectures". In: (2016).

[17] Tiziano De Matteis & Gabriele Mencagli. "Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing". In: (2016).

[18] Tiziano De Matteis & Gabriele Mencagli. "Parallel Patterns for Window-Based Stateful Operators on Data Streams: An Algorithmic Skeleton Approach". In: (2016).

[19] L. Dagum & R. Menon. "OpenMP: an industry standard API for shared-memory programming". In: (1998).

[20] Streams. Yuanzhen Ji & Hongjin Zhou & Zbigniew Jerzak & Anisoara Nica. "Quality-driven processing of sliding window aggregates over out-of-order data streams". In: (2016).

[21] C. Balkesen & N. Tatbul & M. T. Ozsu. "Adaptive input admission and management for parallel stream processing". In: (2013).

[22] Konrad Rudolph. What's really is a deque in STL? 2017. URL: `http://stackoverflow.com/a/6292437`.

[23] K. A. Scarfone and P. M. Mell. "guide to intrusion detection and prevention systems (idps)". In: (2007).

[24] W Gropp & E Lusk & N Doss & A Skjellum. “A high-performance, portable implementation of the MPI message passing interface standard”. In: (1996).

[25] Giuliano Casale & Ningfang Mi & Ludmila Cherkasova & Evgenia Smirni. “How to Parameterize Models with Bursty Workloads”. In: (2008).

[26] N. Mi & G. Casale & L. Cherkasova & E. Smirni. “Injecting realistic burstiness to a traditional client-server benchmark in Proceedings of the 6th International Conference on Autonomic Computing, pp. 149–158”. In: (2009).

[27] Ningfang Mi & Giuliano Casale & Ludmila Cherkasova & Evgenia Smirni. “Injecting Realistic Burstiness to a Traditional Client-Server Benchmark”. In: (2009).

[28] std::atomic. 2017. URL: `http://it.cppreference.com/w/cpp/atomic/atomic`.

[29] Cagri Balkesen & Nesime Tatbul. “Scalable Data Partitioning Techniques for Parallel Sliding Window Processing over Data Streams”. In: (2011).

[30] S. Krishnamurthy & M.J. Franklin & J. Davis & D. Farina & P. Golovko & A. Li & N. Thombre. “Continuous analytics over discontinuous streams”. In: (2010).

[31] G. Cormode & F. Korn & S. Tirthapura. “Time-decaying aggregates in out-of-order streams”. In: (2008).

[32] M. Danelutto & T. De Matteis & G. Mencagli & M. Torquati. “Data Stream Processing via Code Annotations”. In: (2016).

[33] Marco Danelutto & Gabriele Mencagli & Massimo Torquati. “Efficient Dynamic Memory Allocation in Data Stream Processing Programs”. In: (2016).

[34] Massimo Torquati. “Harnessing Parallelism in Multi/Many-Cores with Streams and Parallel Patterns”. Ph.D. Thesis Proposal Dipartimento di informatica di Pisa, 2016.

[35] J. Li & D. Maier & K. Tufte & V. Papadimos & P. A. Tucker. “Semantics and evaluation techniques for window aggregates in data streams”. In: (2005).

[36] Jin Li & David Maier & Kristin Tufte & Vassilis Papadimos & Peter A. Tucker. “No Pane, No Gain: Efficient Evaluation of Sliding-Window Aggregates over Data Streams”. In: (2005).

[37] Jin Li & David Maier & Kristin Tufte & Vassilis Papadimos & Peter A. Tucker. “Semantics and Evaluation Techniques for Window Aggregates in Data Streams”. In: (2005).

[38] Henrique C. & M. Andrade & B. Gedik & D. S. Turaga. “Fundamentals of Stream Processing, Application Design, Systems, and Analytics”. In: (2014).

[39] Henrique C.M. Andrade & Bugra Gedik & Deepak S. Turaga. “Fundamentals of stream processing”. Cambridge University Press, 2014.

[40] D. Buono & G. Mencagli & A. Pascucci & M. Vanneschi. “Performance Analysis and Structured Parallelization of the STAP Computational Kernel on Multi-core Architectures”. In: (2017).

[41] Brian Babcock & Shivnath Babu & Mayur Datar & Rajeev Motwani & Jennifer Widom. “Models and issues in data stream systems”. In: (2002).

[42] J. Wuttke. Lmfit a c library for levenberg-marquardt curve fitting. 2015. URL: `http://apps.jcns.fz-juelich.de/lmfit`.

[43] Chee-Yong Chan & H.V. Jagadish & Kian-Lee Tan & Anthony K.H. Tung & Zhenjie Zhang. “Finding k-Dominant Skylines in High Dimensional Space”. In: (2006).