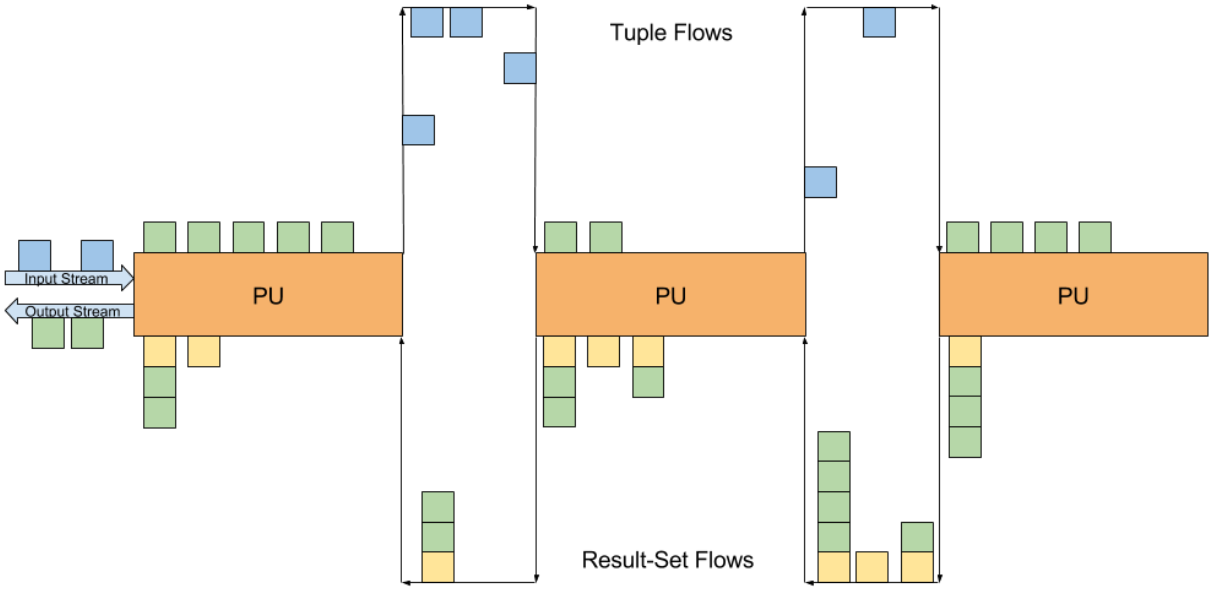
CHALMERS



Disjoint Parallelization of Sliding-Window Streaming Aggregation

Master's Thesis in Computer Science – Computer Systems and Networks

ANDREAS BEICHT

Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, February 1, 2016

Disjoint Parallelization of Sliding-Window Streaming Aggregation

ANDREAS BEICHT

Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden 2016

Faculty of Computer Science and Mathematics
Goethe University of Frankfurt
Frankfurt, Germany 2016

Disjoint Parallelization of Sliding-Window Streaming Aggregation

ANDREAS BEICHT
Examiner at Chalmers University of Technology: MARINA PAPATRIANTAFILOU
Examiner at Goethe-Universität: ULRICH MEYER

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Faculty of Computer Science and Mathematics
Goethe University of Frankfurt
DE-60629 Frankfurt
Germany
Telephone + 49 (0)69 7980

This Work has been authored within the framework of Erasmus+ at the receiving institution Chalmers University of Technology in cooperation with the sending institution Goethe University of Frankfurt.

Cover: Structure of the Handshake algorithm's communication pattern (see Figure 10 on page 22 for detailed information)

Disjoint Parallelization of Sliding-Window Streaming Aggregation

ANDREAS BEICHT
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Online analysis of data streams (with different degrees of parallelism) is becoming progressively more important as the amount of sensory data is growing. Aggregate functions represent one common example thereof. The current parallel approaches for online aggregation of data streams share one characteristic: one or several threads serve as central coordinators by distributing the work as well as the incoming data to other threads dedicated to its processing. If an approach uses centralized coordination units, its scalability is bounded to the throughput with which such coordination units can distribute work and data to processing threads. This thesis deals with the development of online analysis approaches for streaming aggregation without centralized coordination units. The coordination tasks as well as the remaining work are distributed among all participating threads. In this thesis we first introduce basics of online analysis and streaming aggregation, and then we provide several options for disjoint parallelizations of sliding-window based streaming aggregation. We study the developed approaches' runtime properties in order to maximize their throughput and minimize their latency. We also evaluate their throughput and latency in practice and discuss related work that could improve certain aspects of the approaches developed within this thesis.

Keywords: data streams, streaming aggregation, disjoint parallelization, sliding windows

Acknowledgement

I would like to thank my supervisor Vincenzo Gulisano and my examiner Marina Papatriantafilou for all the advice, comments and ideas in our frequent meetings. Our meetings' discussions led to many insights and thus were very helpful before and during the writing process of this master's thesis. It was very pleasant to work with you, since you always had an open ear to new ideas or findings as well as patience in situations, in which confusion or lack of clarity reigned.

I would also like to thank my friends and fellow students Saransh Rastogi, Martin Hühne, Tobias Schmitt, my sister Larissa and my brother Daniel for their help with proofreading this thesis.

CONTENTS

1 Introduction

1.1 Background

Data streaming analysis is becoming more and more important as the amount of sensory data is growing. Depending on the context of an application, it is often not acceptable to wait until all data that shall be processed are available and completely saved in the memory of the processing system. In some cases, it might even be impossible to store all data, since the data streams are unbounded or have an unknown size, for example in traffic monitoring or financial market applications. While some algorithms need complete knowledge about a given problem (for example all edges of a graph to calculate the shortest path between two arbitrary nodes), streams of data are often only temporarily relevant for the result of the executed algorithm, so it is possible to continuously provide analysis output for a portion of the most recent data (rather than all the data). Thus the processing of operations on large amounts of incoming data in real time is becoming a more relevant alternative to the conventional approach with all necessary data stored in ordinary databases.

The rate at which data from a data stream is reaching the system processing it is often unknown and may vary arbitrarily. If the receiving processing system is not able to handle all the incoming data at the current rate, the sender has to be informed to reduce the rate data is being sent or the processing system has to discard incoming data tuples.
Therefore, parallelism in both variants – using multiple processing cores on one machine and using multiple machines at once – is very important to guarantee that the processing system is able to handle the amount of incoming data in time and hence to ensure that it will never have to store input data outside of the main memory for later processing. In the context of e.g. continuous monitoring “in time” means that the result at any point of time is incorporating all relevant input data. Thus the processing system should be able to compute incoming data at least as fast as they are arriving.

General information about the basics of data stream processing, the requirements to use it efficiently and examples for environments which fit the requirements and others that do not, can be found in this article [1].

1.2 Overview of the thesis’ aims

One type of analysis that can be processed on streams are aggregate operations (described in more detail in Section 2.4). Aggregate operations are performed on finite portions of data of data streams, hence on a subset of the stream’s input data. The current relevant subset for an aggregate operation is defined by a “sliding window”. As soon as tuples of the input stream leave the sliding window, they can be dropped and no longer influence the result.
This master’s thesis will deal with sliding window-based aggregate operations – so for example the calculation of the median, mean value or sum of a limited amount of input tuples – and an efficient parallelization of them.

So let us assume that we have an algorithm that executes aggregate operations on a data stream concurrently on several threads to be able to compute high rates of incoming tuples. We assume that it has a central coordinator, so one processing unit that is responsible for coordinating the work of all the other processing units. It passes tasks and forwards parts of the incoming data stream to the different processing units.
A parallel program with such a central coordinator will have limited parallelization potential. As the rate of input data increases, as well as the number of processing units that are needed to meet the

ascending processing speed requirements, the work for the central coordinator will increase too. Beyond a certain limit, the coordinator will be too slow to handle all the coordination tasks in time and hence it will become the bottleneck of the system.
So what we need to avoid this limitation – and this is the thesis' goal – is a fully disjoint parallel algorithm, an approach working without a processing unit acting as central coordinator.
It shall scale with an arbitrary number of participating processing units.

1.3 Thesis organization

We shortly want to want to present the different parts of this thesis.

The thesis' goals are defined in more detail in Chapter 3 after the preliminaries chapter (Chapter 2), which explains basic knowledge that is indispensable before explaining details.

Chapter 4 shortly presents a parallel algorithm without a central coordinator which executes the join operator on data streams. Our first approach for streaming aggregation is based on this algorithm.

Within the Chapters 5-9 we develop five different parallel approaches to process aggregate operations on data streams.

Chapter 10 deals with the chosen platform and programming language for the implementations and implementation details such as classes, class interdependencies and chosen data structures.

In Chapter 11 the speed and latency of the different implementations are benchmarked and the results are presented.

Chapter 12 introduces related work; mainly articles and papers that deal with alternative approaches for streaming aggregation or data structures that could be combined with the approaches of this thesis to enhance certain aspects of it.

Finally, in Chapter 13 a conclusion is drawn. Here, we present a summary of the results along with insights of this thesis. This is followed by a discussion regarding the extent to which the goals of the thesis were met.

2 Preliminaries

This chapter explains basic terms and knowledge that are necessary to understand the thesis' exact goals, which are presented in Chapter 3.

2.1 Data streams

A data stream is defined as an unbounded, continuous flow of data packets, called *tuples* (see Figure 1). The tuples' schema is given and identical for each tuple of one data stream.
The data stream is received by a processing system that works the incoming data in some way.

Once a tuple reaches the processing system, the processing system updates the state of the computation. The processing system does not batch tuples, so it does not wait for all tuples (or any greater amount of tuples) to arrive before starting to process them.

The rate at which incoming tuples are reaching our processing system is often unknown; and may vary arbitrarily. Hence when working on data streams it is important to take care of possible congestions. If the receiving processing system is not able to handle all the incoming data at the current rate, the sender has to be informed to decrease the rate at which tuples are being sent or the processing system has to discard incoming tuples.

2.2 Tuples

A tuple is an object transporting 3 types of information: a *timestamp*, a *key* and a *value* (see Figure 1).

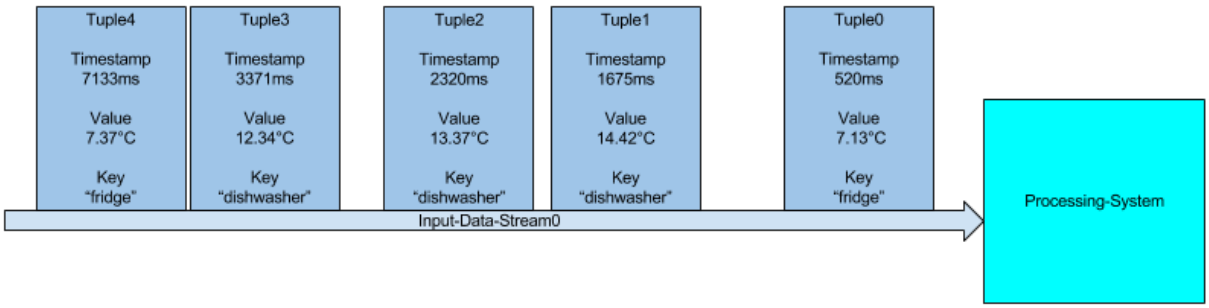


Figure 1: An input data stream and its tuples

Timestamp

A timestamp is a number giving chronological information about a tuple, for instance 520ms, 1675ms, … in Figure 1. Tuples are delivered by a data stream with non-decreasing timestamps. So after the arrival of Tuple0, we know that the next arriving tuple will have a timestamp greater than or equal to 520ms.
This property might be violated in case so called *out-of-order-tuples* are possible. Error-prone communication systems might cause tuples to arrive in a different order at the processing system than their sending order. However, the focus within this thesis will be on data streams without the possibility of out of order tuples.

Although it is possible that the timestamp of a tuple is dependent on the exact real point of time when this tuple has been created, this does not necessarily have to be the case. It gives information about the tuple order and is important to determine, how long this tuple will be relevant for the data stream operation. So Tuple1 does not have to arrive $1675ms - 520ms = 1155ms$ after Tuple0. It can also arrive in the same millisecond or 10 years later. We can only be sure that it will not arrive before Tuple0.

The key (also referred to as *group-by-value*) can be used to group tuples before processing any operations on them. The grouping of tuples works like the well-known group-by clause, commonly used when working with databases.

Finally, the value contains the tuple data itself on which we want to perform the aggregate operation. It may have an arbitrary structure. For example, it can consist only of a single integer or double, but it is also possible to store a set of complex objects within each tuple.

Having a look on Input-Data-Stream0, we could interpret its tuples like that:
520ms after starting our program the fridge sends information about its temperature, which is 7.13°C. 1675ms after the start of our program the dishwasher sends information about its temperature, which is 13.42°C. 2320ms after the start of our program the dishwasher again sends information about its temperature, which decreased to 13.37°C and so on.

2.3 Sliding windows

This chapter explains what a window is, why we need windows and how they work. There are two types of windows commonly used when working with data streams, *time-based* and *tuple-based* ones. This thesis will mainly deal with time-based windows. If not explicitly specified differently, all further chapters deal with time-based windows and we will discuss them first.

Time-based windows
We want to do complex analyzes of the incoming data of a stream, in this thesis in the form of aggregations, but the data stream is unbounded. Since aggregation is not possible on unlimited sets of data, we have to limit the scope of it to a certain portion of the stream's data.
Such a portion is called *window* and contains all tuples within a certain timestamp range. In Figure 2 Window0 contains all tuples with timestamps between 0ms and 1000ms. We call 0ms the *time start* and 1000ms the *time limit* of this window.
In this thesis timestamp ranges are including the time start but excluding the time limit. So Window0 contains all tuples with a timestamp of 0ms, 1ms, 2ms, … or 999ms.

A window has two parameters (see Figure 2), *size* (also referred to as *window size* or *WS*) and *advance* (also referred to as *window advance* or *WA*).
The window size defines the size of each window's timestamp range.
The window advance defines the distance of the timestamp range of one window to its neighbor window.
In Figure 2 each window contains tuples within a range of 1000ms and each window's timestamp range is shifted by 270ms to the right of its left neighbor.
Since windows always have the same size and are *sliding* by the same amount of time units on each step, we also call them *sliding windows*.

The timestamp ranges of two windows may overlap; i.e. one tuple can belong to several windows. In Figure 2, Window0's timestamp range (0ms-1000ms) is overlapping with Window1's timestamp range (270ms-1270ms). So Tuple0 of Figure 1 with timestamp 520ms would *contribute* (also referred to as *belong*) to both of them.

As soon as a tuple with a timestamp greater than or equal to the time limit of a window w arrives, this window w *expires*. Since the timestamps of arriving tuples only increase or remain constant (but never decrease), we know that no further tuples belonging to window w will arrive.

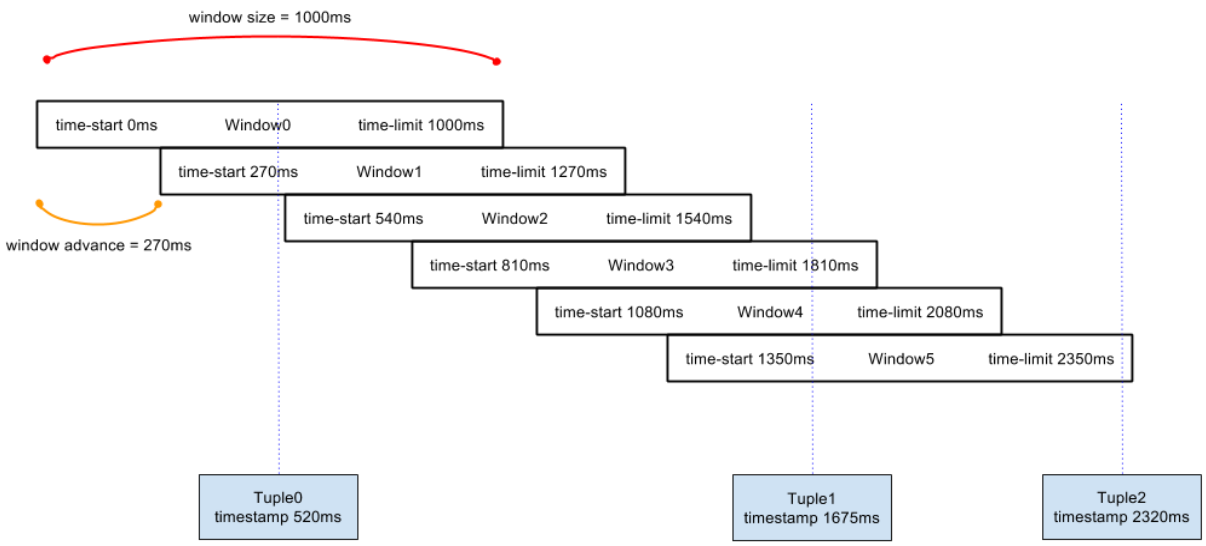


Figure 2: Window size and window advance

Let us assume that the tuples of Figure 1 are arriving at a computing system that wants to partition them into sliding windows according to Figure 2; i.e. the window size is 1000ms, the window advance is 270ms (see Figure 2) and the first incoming tuple has a timestamp of 520ms (see Figure 1).
The first window, Window0, contains all tuples having a time stamp between 0ms and 1000ms, the second window, Window1, contains all tuples having a timestamp between 270ms and 1270ms, Window2 contains all tuples having a time stamp between 540ms and 1540ms and so on.
So in Figure 2 Tuple0 belongs to Window0 and Window1, Tuple1 belongs to Window3, Window4 and Window5 (and also to the following Window6, which is not shown in this figure) and so on.

Depending on the time interval between two consecutive tuples, some windows might be empty. In the sample execution shown in Figure 2, no tuple belongs to Window2; it is empty.

The number of non-expired windows existing simultaneously is approximately $\frac{WS}{WA}$. In the current example: $\frac{WS}{WA} = \frac{1000ms}{270ms} \approx 3.7$. This means that the number of windows existing at any point of time will always be either 3 or 4 and that each tuple contributes to either 3 or 4 windows.

Window time-slots

If there is a window, to which no tuple contributes, we actually do not need this window, since it will not fulfil any function. In practice Window2 will never be created, since no tuple is contributing to it. To distinguish between windows that exist or will exist and windows that will not exist at any point of time, we introduce the term *window time-slot*. All possible window positions, no matter whether the corresponding window exists or not, are window time-slots. So each window is created within a window time-slot. But not each window time-slot has to contain a window.
In our example the time interval between 0ms and 1000ms is a window time-slot, the time interval between 270ms and 1270ms is a window time-slot, the time interval between 540ms and 1540ms is a window time-slot and so on. The window time-slot between 0ms and 1000ms contains a window: Window0. The window time-slot between 270ms and 1270ms contains a window: Window1. The window time-slot between 540ms and 1540ms is empty.

Tuple-based windows

For tuple-based windows, timestamps are irrelevant.

A tuple-based window always contains WS number of tuples and every WA number of tuple arrivals a new window will be created.
To determine which windows a tuple contributes to, each tuple gets its own unique index. The first arriving tuple gets index 0, the second tuple gets index 1 and so on.

Both tuple-based and time-based windows exhibit similar properties under certain circumstances. Let us consider two settings:

- Windows are time-based. *WS=1000ms*, *WA=270ms*. The timestamps of two consecutive tuples always have a difference of 1ms.
- Windows are tuple-based. *WS=1000 tuples*, *WA=270 tuples*.

In both situations, each window would contain 1000 tuples and after every 270 tuple arrivals one new window would be created. So applied to the same data stream each window of the time-based setting contains the same tuples as the respective windows of the tuple-based setting.

With the following settings, we have a similar situation:

- Windows are time-based. *WS=1000ms*, *WA=270ms*. The timestamps of two consecutive tuples always have a difference of 10ms.
- Windows are tuple-based. *WS=100 tuples*, *WA=27 tuples*.

In both settings, the time-based and the tuple-based one, each window contains the same 100 tuples as the respective counterpart in the other setting.

In general, if a time-based setting fulfils three conditions, there is an equivalent tuple-based setting:

(1) The **t**imestamp **d**ifference, for short TD, between two consecutive tuples has to be constant.
(2) It must be true that $WS\%TD = 0$.
(3) It must be true that $WA\%TD = 0$.

Otherwise windows with different amounts of tuples are possible, which is not allowed for a tuple-based setting.
For a time-based setting, which has a window size of WS, a window advance of WA, a constant tuple timestamp difference of TD and fulfils the three conditions, the equivalent tuple-based setting has a window size of $\frac{WS}{TD}$ and a window advance of $\frac{WA}{TD}$.

2.4 Aggregate operations

In general, an aggregate operation is an operation that groups a set of values together to form one single result value (see Figure 3).

As can be seen in Figure 3, a few examples for aggregate operations are *sum*, *average* and *count*.

In the context of streaming aggregation an aggregate operation is an operation that takes all values of all tuples belonging to one window to create a single output value. To compute an aggregate operation on tuples of a data stream, we need to define four things: the aggregate function we want to compute, the windows' size and advance and if we want to group tuples by their keys.

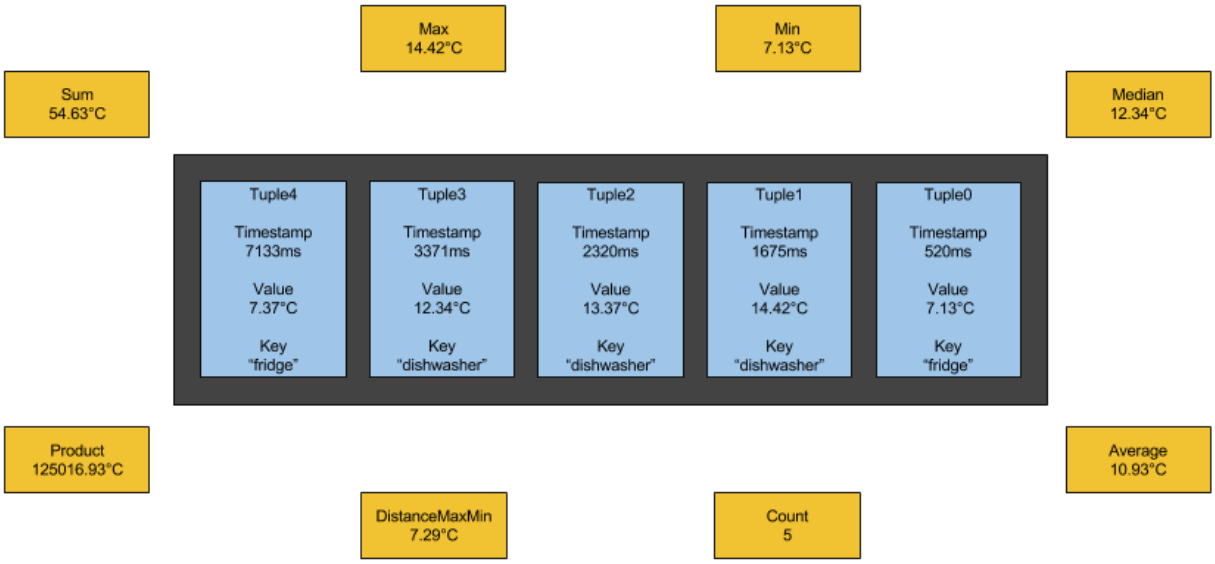


Figure 3: 8 examples for aggregate functions aggregating Tuple0, Tuple1, ..., Tuple4

Let us for example execute AggregateOperation0 of Figure 4 on the tuples of Input-Data-Stream0 of Figure 1. It processes the average aggregate function on windows of size 1000ms and advance 270ms (similar to the windows of Figure 2). The results are output into Output-Data-Stream0 (see Figure 5).

So what we are actually doing is: Every *WA=270ms* we determine the average temperature of all tuples' values that arrived during the last *WS=1000ms*. That might not seem very meaningful yet, since we are calculating the average temperature of different machines' tuples (the fridge and the dishwasher in our example), but first let us just take it as an example.

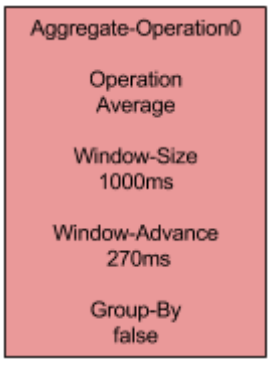


Figure 4: An example aggregate operation storing information about the aggregate function to be executed, WS, WA and if tuples are to be grouped by their key

So first, we calculate the average of all tuples' values of Window0. Window0 contains only Tuple0. Tuple0 has the value 7.13°C. So the result for Window0 is 7.13°C.
Window1 also contains Tuple0 only. Hence its result is 7.13°C.
Window2 does not contain any tuple at all. So the average of all tuples belonging to it is 0.0°C or undefined depending on how we handle such special cases.
Window3 and Window4 contain only Tuple1 with value 14.42°C. So the result for both is 14.42°C.
Window5 contains Tuple1 and Tuple2. So its result is the average of their values: $\frac{14.42°C+13.37°C}{2} = 13.895°C$.

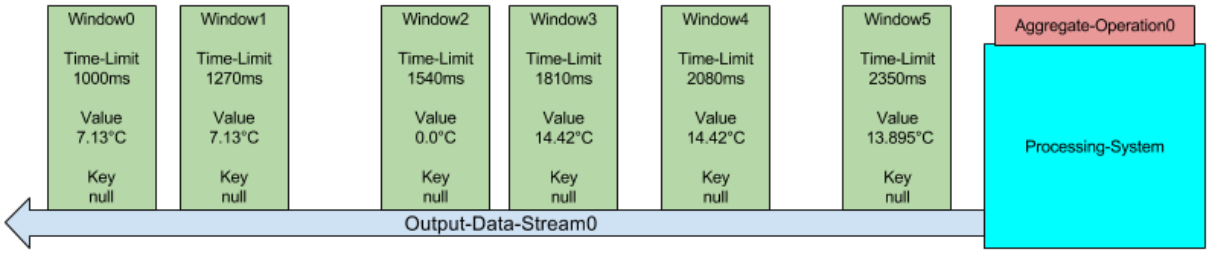


Figure 5: An Output-Data-Stream with its result tuples (expired windows) produced by a computing system computing AggregateOperation0 on Input-Data-Stream0.

Dynamic computation

When executing the average aggregate operation, we can make another observation: Whenever we get to know that a tuple contributes to a window w, we can already calculate an intermediate sum of w; a sum of the values of all tuples that are contributing to w so far. So we do not necessarily have to wait and store all incoming tuples until w has expired before we start processing w's aggregate operation. We can do that *dynamically* as the tuples are flying by. After we know that w has expired, we just divide the intermediate sum by the number of contributing tuples to get the average value.

This kind of dynamic calculation of intermediate results is possible for a lot of common aggregate operations, but not for all.
More information about computing aggregations dynamically can be found in the appendix in Section 17.1.

Order-Sensitive vs. Order-Insensitive aggregations

We need to define another property of aggregate operations that plays a role later on: the order-sensitivity. Each aggregate operation is either order-sensitive or order-insensitive.

- Order-insensitive aggregations can still dynamically compute intermediate results if the tuples are not arriving in timestamp order. All commutative operations are order-insensitive.
  For instance, when computing the sum aggregation, it does not matter in which order the tuples of a certain window are flying by. The intermediate sums might vary depending on the order of the incoming tuples; the final sum will be the same for each possible order.
- Order-sensitive aggregations cannot dynamically compute intermediate results if the tuples are not arriving in timestamp order. All non-commutative operations are order-sensitive. For example, when computing a continued fraction, it does matter in which order the tuples of a certain window are flying by. The final quotient depends on the dividend. If the dividend is chosen wrongly, because the firstly arrived tuple is not the one with the lowest timestamp, the result will be wrong.

As we see, if tuples can arrive in non-timestamp order, order-sensitive operations have to wait until a window has expired, before the aggregation can take place, while order-insensitive operations can always dynamically compute intermediate results.

Grouping tuples

As mentioned in Section 2.2, we can also group tuples by their *keys* before aggregating them. In our example grouping tuples by their keys means that we execute the aggregate operation separately for tuples of each machine (fridge, dishwasher and possible other ones).
Now the output every 270ms would not describe the average temperature of all machines during the

last 1000ms. Instead for each machine that sent at least one tuple during the last 1000ms, the average temperature of all tuples sent by this machine will be output.
So for each machine that sent a tuple during the last 1000ms, a window is being created within the same window time-slot. So Window5 with time start 1350 and time limit 2350 would not collect each tuple t having $1350 \leq t.timestamp < 2350$; instead each machine, the fridge and the dishwasher, would have its own window within the window time-slot between 1350ms and 2350ms. We can distinguish between these windows by their keys. Each window takes the respective machine's key as its own key.

2.5 Determinism of the Output Results

A deterministic algorithm outputs for certain input data always the same output results. A PU (short for **p**rocessing **u**nit) executing it will enter the same sequence of states in the same order whenever we will run it.
Working with multiple PUs can be a reason for nondeterministic behavior, e.g. if several PUs write to the same variable. Depending on the order of the writing processes, which might be depending on scheduling decisions of the operating system, the different access orders can influence the further ongoing of the calculations.

However, also nondeterministic algorithms can produce deterministic results. If we for example split our work dynamically between different PUs depending on their workload, the processing might be nondeterministic, since every PUs deals with different pieces of the work within different executions – it does not always enter the same sequence of states in the same order. But as we know it is still possible to create parallel algorithms with deterministic outputs by synchronizing nondeterministic parts between the participating PUs.

In our case – concurrent data stream applications working with sliding windows – there is another example of nondeterministic output behavior that can even be acceptable in certain cases: it could be possible that all the aggregation results of our windows are correctly output, but in the wrong order. For a lot of use cases it might be satisfactory to know the correct aggregation result of each window, even if their order is arbitrary to a certain extent, but for others it is not sufficient.

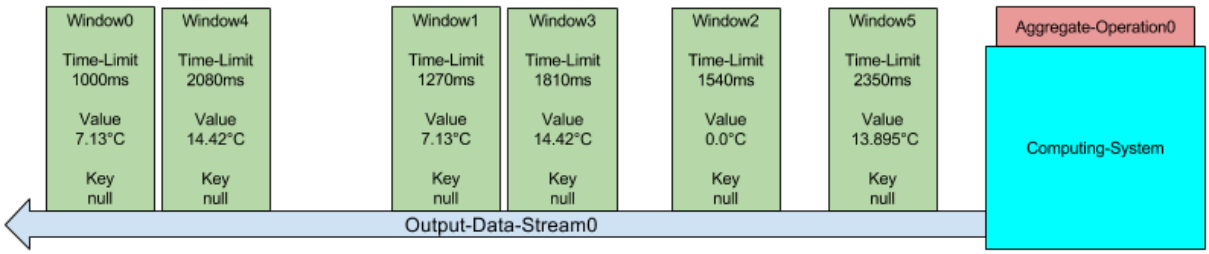


Figure 6: Equal to Figure 5, but with a different output order. Each single result tuple is correct, the order among them is not.

So for certain applications it might be viable to increase the performance at the cost of deterministic output behavior (by dropping output ordering tasks).
However, during this thesis, we will only focus on creating deterministic results: Each execution of our program shall lead to exactly the same output results in the same order.

3 Problem description & Goals of this thesis

The existing approaches for streaming aggregations are working with central coordinators, which manage the distribution of the incoming data and the merging of the results processed on the participating processing units. Central coordination of the tasks might cause bottlenecks once the managing costs exceed the coordinators computing capacity.
Therefore, this master's thesis will deal with sliding window-based aggregate operations and an efficient, fully disjoint parallelization of them. This thesis' goal is the development of algorithms for parallelized streaming aggregation without centralized coordinators.

Correctness and deterministic output behavior

We want to have deterministic output results independently of the number of participating PUs, the data stream's incoming data rate and fluctuations.
To achieve a deterministic output behavior, the aggregation result of each window has to be correct and aggregation results of windows are to be sorted in their time limit order. Windows with higher time limits are output later.

Speed and low latency

We try to find solutions that reach as high tuple throughput and as low output latency as possible.

Proceeding

We start the researching process by taking a recent approach that shares our requirements (but for a different streaming operation) and inspired by that, we design an aggregation method that uses a similar principle. This approach is called *handshake-join* and will be introduced within the next chapter.
In a similar way as the handshake-join algorithm we will only use neighbor to neighbor communication between participating PUs within our first approach. We will analyze the runtime characteristics of this first approach and think about improvements of its structure or communication pattern to develop advanced approaches featuring reduced computation and communication costs.
Each suggested approach will be implemented and benchmarked to verify the assumptions of the theoretical structure analysis.

Target platform

We mainly focus on a shared-memory solution for a single machine but preferably it should be re-usable for distributed setups as well.
More detailed information about the target platform we want to address with our approaches and their implementations are given in Chapter 10.

4 The handshake-join algorithm

In this chapter the handshake-join algorithm is presented [2]. Our first approach to deal with streaming aggregation is inspired by this algorithm.

The *handshake join* algorithm is a parallel algorithm for executing the *join* operation (described further below in this chapter) between two data streams. It is introduced in this article [2]. It avoids any centralized coordination and scales very well as we increase the number of threads executing this algorithm.

The common join database-operator takes all elements out of two groups and compares each element of the first group to each element of the second group. If any pair matches on the desired properties, it is a result of the join-operation and is to be output.
The join-operation on data streams however, is not being executed on all elements of two streams but instead on elements of sliding windows over the streams. As we see in Figure 7, one data stream is entering the window from the left side and the other data stream from the right side. Tuples of each stream are "meeting" all of the other stream's tuples they shall be joined with. During each of the sliding window's steps, all tuples with timestamps between its current upper and lower time limit are stored in it and being joined.

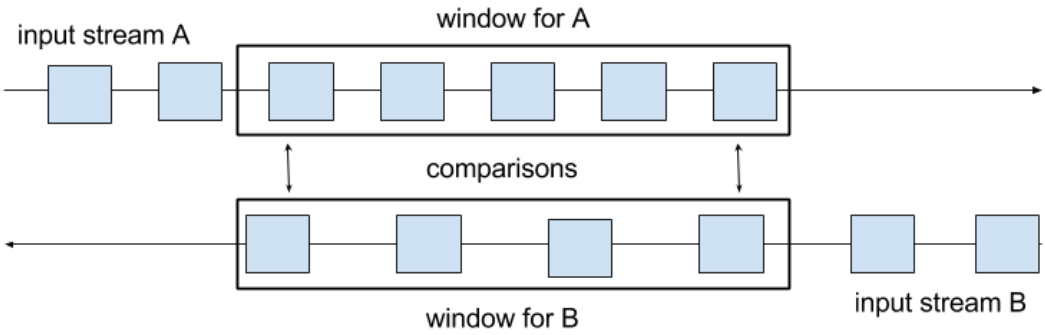


Figure 7: The join operation on data streams

To parallelize the work the handshake join algorithm uses just neighbor-to-neighbor-communication between the threads executing it. We align the threads in a line and one data stream traverses the line from the left to the right, the second data stream from the right to the left. To distribute the work now, we just split the sliding window into smaller parts and each thread takes care of one part as can be seen in Figure 8. The tuples of both streams meet each other within the sliding windows and are then compared.
However, in the parallelized version it is possible that two tuples do not meet each other, although they should: If for example one tuple of data stream A (whose tuples are traveling to the right) is located at PU 2 and one tuple of data stream B (whose tuples are traveling to the left) is located at PU 3 and both tuples are sent to their next destination at the same time, they will not meet each other, so the join result might be incomplete. In order to avoid that any tuple of stream A misses any tuple of stream B, the handshake join algorithm creates temporary copies in one of the stream to ensure that each pair of tuples between both data streams within one sliding window is being joined.
Additionally, creating a deterministic output in the parallelized version is not trivial. In contrast to the single-threaded version, we have several locations where tuples can meet each other and there is no deterministic order, in which the tuples will meet each other and get joined. The previously mentioned

paper does not handle this issue at all, so the output tuple's order is nondeterministic.

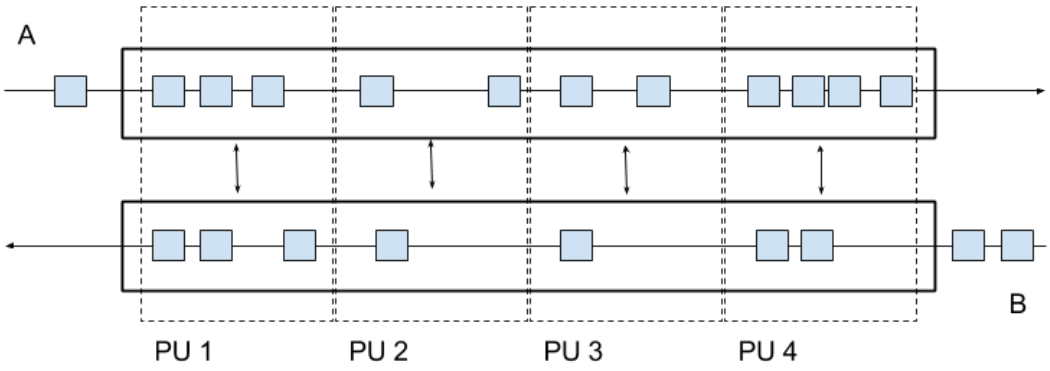


Figure 8: Parallelization pattern introduced by the handshake-join algorithm

However, there is an improved version of this algorithm by the same authors [3], which guarantees a deterministic output order and a lower latency of the join results.

In summary, this algorithm exhibits the following properties that we want to apply to our algorithm as well:

- It does not need any centralized coordination.
- It sticks to neighbor-to-neighbor communication, so only few connections are necessary. Additionally, if neighbor threads are located on neighbor processor cores, they might benefit from the cache hierarchy in modern computer systems because cores that are close to each other share faster caches.
- The "tuples meet tuples" idea can be carried into the aggregation operator. We can think of two virtual streams: one containing the tuples, the other one containing the windows. This is explained in detail in the next chapter.

5 The Handshake approach for streaming aggregation

In this chapter, we present an approach for a disjoint parallelization of sliding-window streaming aggregation. We call this algorithm the Handshake approach (for short HSA).

First (Section 5.1), we discuss the structure of the Handshake approach. We present its pseudocode and explain why it exhibits deterministic output behavior.

Next (Section 5.2), we analyze its runtime properties. Therefore, we first have to introduce definitions and denotations. We analyze the call frequency of the different parts of our algorithm as well as interdependencies between the executed aggregation and the call frequencies and costs of the algorithm's parts.

Then (Section 5.3) we discuss the determined runtime properties of the algorithm.

Finally, (Section 5.4) we consider suggestions for improvements.

5.1 Structure

Like mentioned in Chapter 3, the first aim is to develop an approach without centralized coordination to calculate aggregate operations on a data stream. It is based on the ideas of the handshake-join algorithm [2], so all participating threads will be aligned in a row and only neighbor-to-neighbor communication will be possible (see Figure 9).
Henceforward, the terms *thread* and *PU* (***p**rocessing **u**nit*) mean the same thing – one system thread working together with the others on the algorithm concurrently – and as such are used interchangeably.
However, there is a difference between these terms and a *(processing) core*. A processing core is a physical core of the processor, which can take care of an arbitrary number of threads.

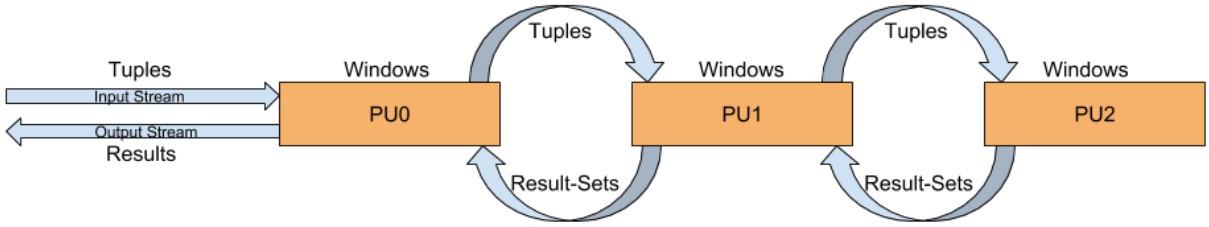


Figure 9: Communication pattern

In our approach, we will use five different types of objects called tuple, window, result set, PU and aggregate operation.
One input-stream will transport the data tuples to the leftmost PU of our processing system. One output-stream, created by the leftmost PU of our processing system, will output the results of the aggregate operation.
Within this approach each tuple will visit each thread in the row. The PUs will forward them thread by thread until each reaches the rightmost PU.

Within a single-threaded program every aggregate operation with each of its windows would be computed on the same thread. And this is the part of the calculation that we actually want to split. Each thread is only responsible for a subset of all the aggregate operations' windows.
Whenever a tuple reaches a PU p, p checks for all windows this tuple will contribute to, whether p is responsible for the calculation of this window; and this is decided by using a hash function.

The hash function that decides on which PU a certain window has to be computed, takes two parameters: the time limit of the window and, in case we want to group tuples and windows by their key, the key of the window.
Hence each pair of windows that is different in at least one of both parameters will most likely have completely different hash values. The result of a window w's hash value modulo the number of PUs determines the index of the PU on which w has to be located and processed.

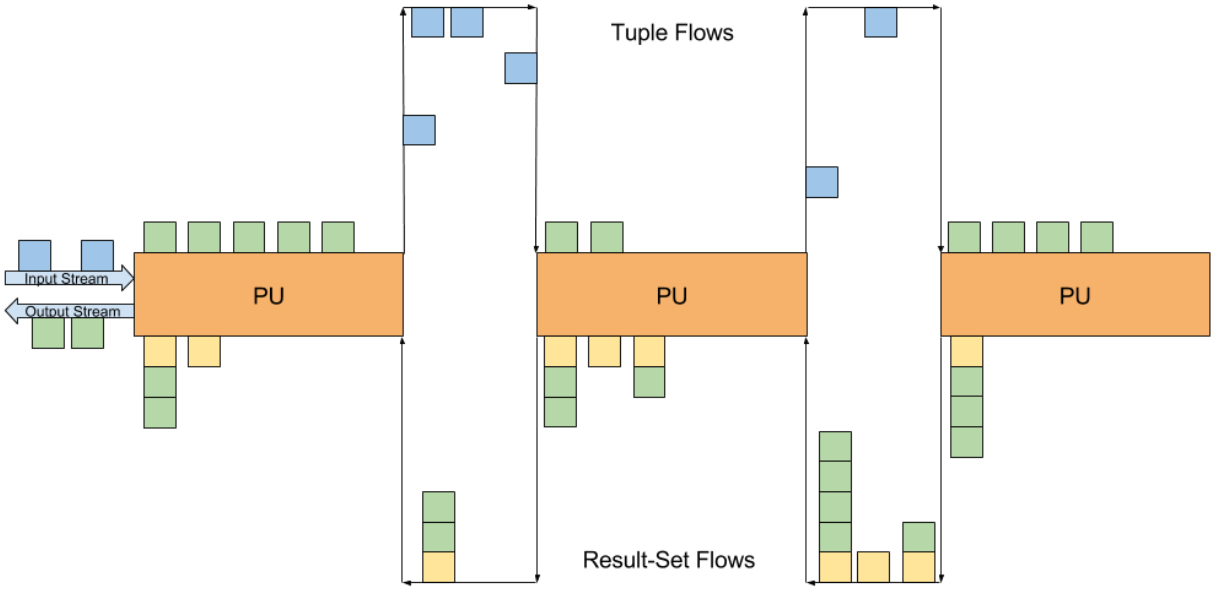


Figure 10: Algorithm logic and communication structure

Since the rightmost thread is the one that receives every tuple last – after all other threads have received, processed and forwarded it – it can decide when any window is expiring. And whenever a window (no matter which thread is actually processing it) is expiring, the rightmost PU creates a so called result set, which is forwarded from thread to thread in the left direction and collects the results of all expired windows and orders them by the windows' time limit.
When a result set r reaches the leftmost PU, this PU outputs the results of the expired windows contained within r into the output data stream.

Figure 10 roughly depicts the algorithm's logic and communication structure. Tuples (blue quads) are forwarded PU by PU to the right. They contribute to windows (green quads) which are located on the different PUs. The rightmost PU sends result sets (yellow quads) to the left to collect expired windows. Green quads attached to yellow quads are expired windows collected by a result set.
A detailed example execution of the Handshake approach can be found in the appendix in Section 17.2.

5.1.1 Pseudocode

The corresponding pseudocode representing this approach's structure is shown in Pseudocode 1 below. It is being executed by each PU.
A PU p continuously (line 1) checks whether new tuples (line 2) or result sets (line 11) have arrived.

If a new tuple t arrives, PU p takes it and forwards it to p's right neighbor (line 3). Of course, if there is no right neighbor (that just happens if p is the rightmost PU), p does not forward anything. Next, PU p checks for each window w tuple t contributes to (line 4), whether this window is to be processed by p (line 5). This is being determined by calculating a hash function that takes window w as a parameter and returns on which PU w has to be processed.

So if p is responsible to process window w, p checks, whether it has already stored w and started processing it and if not p creates and stores w (line 6).
Then PU p adds Tuple t to window w. In case w's aggregate operation is dynamically computable (Section 2.4), p can execute the next step of the aggregate operation to create a new intermediate result (line 7).
If p is the rightmost PU in the line (line 8), it has to check if a new result set has to be created. If so, p creates the new result set with t's timestamp as time limit and inserts it to p's incoming result set queue (line 9).

If a new result set r (line 11) arrives, PU p adds all of its expired windows to r (line 12). Then it forwards r to its left neighbor (line 13). If there is no left neighbor (that just happens if p is the leftmost PU), it instead puts the results of the expired windows contained in r into the output stream.

```
1.  while (true)
2.      if ((Tuple t = incTupleQueue.first) != null)
3.          Get&ForwardTuple(t)
4.          for (each window w to which tuple t contributes)
5.              if (Prepare&ExecuteHashOperation(w) == me)
6.                  FindOrCreateWindow(w)
7.                  AggregateOperation(w,t)
8.          if (myRank == PUCount - 1)
9.              CreateNewResultSetIfNecesarry(t)
10.
11.     if ((ResultSet r = incResultSetQueue.first) != null)
12.         AddMyExpiredWindowsToResultSet(r)
13.         ForwardOrOutputResultSet(r)
```

Pseudocode 1: Pseudocode of the Handshake approach. It is executed by each PU.

5.1.2 Determinism of the output stream's results

The rightmost PU has yet another task as shown in Pseudocode 1 and mentioned in Section 5.1: It checks, whether any window on any PU has expired and if so, it creates a new result set that collects expired windows.

To guarantee deterministic behavior when collecting expired windows using result sets, we need a mechanism basing upon the following observation:
Whenever any PU p is forwarding a tuple (line 3), it immediately processes it afterwards (lines 4-9). So all PUs to the right of p can only process tuples that p has already processed (or that p is currently processing).

Observation 1: If any PU finishes processing a tuple, it has already processed all the tuples that the rightmost PU has processed so far.

Assuming that we have n PUs working on the problem, we assign indices from *0* to *n-1* to the PUs (from the leftmost to the rightmost). After processing any tuple t, the rightmost PU p_{n-1} checks, whether t led to the expiration of any window – this check is only a simple modulo calculation involving window size and window advance of the aggregate operations. If any window has expired, p_{n-1} creates a result set r. Then it adds all of its expired windows to r, if there are any (line 12), and forwards r to its left neighbor p_{n-2} (line 13).
According to Observation 1, all other PUs have already processed all tuples (including t) that p_{n-1} has processed and hence all windows with time limits smaller than or equal to t's timestamp have already expired on each PU. So r does not have to wait for windows to expire when arriving at a PU; it can

immediately collect the expired windows, sort them by their time limits and continue its way to the leftmost PU. This leads us to the next observation:

Observation 2: If a tuple t arrives at the rightmost PU and a result set r is created, all windows with time limits up to t's timestamp on each PU have expired and thus will be collected by r. So windows cannot arrive belatedly (in later result sets that will be created after r).

Let us assume p_{n-2} was much faster than p_{n-1} and has already processed a certain set of more recent tuples *SOMRT*, which led to the expiration of further windows on p_{n-2}. Now r arrives at p_{n-2}, which adds its expired windows to r. And at this point a problem arises: This set of tuples *SOMRT* that PU p_{n-2} has already processed but p_{n-1} has not yet, might lead to the expiration of windows on PU p_{n-1} too. But since r is already on its way to the left and has already left p_{n-1}, p_{n-1}'s windows that will expire by *SOMRT* will arrive in a later result set, not within r, although they might even have an earlier time limit than the windows expired by *SOMRT* on P_{n-2}. That means that the expired windows which are arriving at the leftmost PU (which sends their results into the output stream), might not be ordered by their expiration time. Hence the results in the output stream will indeed be correct, but not in a deterministic order.

There is a simple solution to this problem: Each result set r will have a time limit on its own. PUs are only allowed to add expired windows to r, whose time limits are smaller than or equal to r's time limit. So whenever the rightmost PU p_{n-1} creates a result set r, because it detects that a processed tuple t led to the expiration of any window in the processing system, p takes t's timestamp as the time limit of r. This leads to a third observation:

Observation 3: Since an expired window w cannot be added to a result set r with a time limit smaller than w's time limit, expired windows cannot arrive at the leftmost PU (and hence in the output stream) prematurely.

According to Observation 2 and Observation 3, expired windows can neither arrive too late nor prematurely. So finally we can say that each window w will arrive within the first result set, whose time limit is greater than or equal to w's time limit. Now result sets just have to sort the windows they contain by their time limit and deterministic output behavior is guaranteed.

5.2 Analysis of the algorithm's runtime properties

Now we want to analyze the runtime properties of the algorithm and its parts, which were introduced and shortly discussed in Section 5.1.1.
Firstly, we want to determine, how often a PU p has to execute the different parts (mentioned in 5.2.2) of the algorithm.
Secondly, we want to determine how expensive the execution of each part will be.
Both a part's call frequency and its execution costs are dependent on the aggregation's parameters, so we want to carve out and depict the impact of the aggregation's parameters on the runtime properties.

5.2.1 Definitions and denotations

Window size **WS** and window advance **WA** are well-known.

Number of tuples

Since in general the input stream is unbounded in its length, we only consider a prefix of **T** tuples. We name the T tuples by their indices: t_0, t_1, …, t_{T-1}.

Average timestamp difference

Two consecutive tuples have an **a**verage **t**imestamp **d**ifference of **ATD** time units.

So if we assume that t_0 has a timestamp of 0, t_{T-1} has a timestamp of $(T-1)*ATD$.

For the analysis, we assume that the tuples are distributed in some degree evenly between the first and the last tuple. That means that we expect each window to contain approximately $\frac{WA}{ATD}$ tuples. For example the analysis does not apply in a case in which the first *T-1* tuples have the same timestamp followed by a huge timestamp interval of $(T-1)*ATD$ time units without tuples before the last tuple t_{T-1} appears.

Windows per tuple

The number of windows each tuple contributes to – **WPT** (**w**indows **p**er **t**uple) – is defined by $WPT = \frac{WS}{WA}$.

Number of windows without grouping

As mentioned, the first tuple's timestamp is $(T-1)*ATD$ time units away from the last tuple's timestamp. Every WA time units a new window time-slot begins. So the overall number of windows **W** is approximately $W \approx \frac{T*ATD}{WA}$ if we do not group tuples by their keys.

There is a lower and an upper limit for W:

- If $0 \leq ATD < \frac{WA}{T}$:
  All tuples contribute to the same WPT windows. There cannot be less windows because as long as there is at least one tuple, it always contributes to WPT tuples by definition. Hence WPT is the minimum value for W.
- If $ATD = WA$:
  The first and the last tuple are covering a $T*WA$ time units lasting time span. So in case the tuples' timestamps are evenly distributed, we can approximately have up to T windows because each tuple is opening a new one (while still contributing to the $WPT-1$ old ones of the previous tuple).
- If $ATD = WS$:
  Each new tuple contributes to up to WPT new windows. If its timestamp distance to the previous tuple is equal to WS (or larger), it does not contribute to any of the previous tuple's windows, but to WPT new windows. Hence if tuples' timestamps are evenly distributed, we can have up to $T*WPT$ windows.
- If $ATD > WS$:
  The situation does not change compared to the previous case. Each tuple still contributes to up to WPT new windows, so the value of W is still at most $T*WPT$.

So without grouping the boundaries for the number of windows are $WPT \leq W \leq T*WPT$.

Number of windows with grouping

However, if we group tuples by their keys, the overall number of windows might be higher. Let us assume there are tuples with **DK** number of **d**istinct **k**eys within the whole data stream.

Basically we have to do the analysis of the non-grouping case for each distinct key, since tuples with different keys do not affect each other at all and are aggregated completely independently.

Or in other words, if we do not group tuples by their keys, each window time-slot contains up to one window that can contain tuples with any key. And if we group tuples by their keys each window time-

slot contains up to DK windows and each of them can only contain tuples with their own key.
Since the number of window time-slots remains unchanged, but only the number of windows per window time-slots can raise up to the DK-fold, there can be DK-fold as many windows if we decide to group tuples.
That means the overall number of windows W we expect is up to approximately $\frac{DK*T*ATD}{WA}$ compared to $\frac{T*ATD}{WA}$ in the non-grouping case.
To determine accurately how many windows we may expect, for each distinct key we would have to know the timestamp distribution of all tuples with this key, which might be unrealistic in many practical situations.
To avoid going beyond the scope of this section, we just note down that the boundaries for the number of windows with grouping are $DK * WPT \leq W \leq T * WPT$.

Realistic and reasonable number of windows
Independently of the fact, whether we are grouping tuples or not, especially the upper limit, which is identical in both cases, is very unrealistic, because each tuple is just contributing to its "own" WPT windows, to which no other tuples contribute, and there is no meaning in executing aggregate operations, when there is actually nothing to aggregate.
So we will define a more realistic upper limit for the amount of windows in common aggregations, where we are indeed aggregating something. Without this upper limit, meaningful analyzes of the Handshake approach and the following approaches is not possible. We define that the average timestamp distance is not allowed to be greater than the window advance: $ATD \leq WA$.
In other words: Given a data stream with an average timestamp difference of ATD, we are not allowed to execute an aggregate operator with its window advance being smaller than ATD.

There are different effects depending on whether we are grouping tuples or not:

(1) The non-grouping case:
Impact of the definition ($ATD \leq WA$) on a concrete example:
If we have one machine that sends information approximately once every second to our computing system, we are not allowed to compute an aggregation with a window advance smaller than one second. This is no limitation at all, since with a window advance of 500ms, half of all windows would return the same result as their respective predecessor, so we can forbid these redundant computations by introducing the presented limit.

In general, from this definition it follows that on average each tuple will open up to one new window and lead to the expiration of up to one window (as mentioned in the practical example, if instead ATD were significantly larger than WA, a lot of windows would contain the same tuples as their respective neighbor windows; and if ATD were larger than WS, a lot of windows would not contain tuples at all).

Since each tuple leads to the expiration of up to one window, there will be up to T result sets and each result set will contain one window. So there will be up to T windows.

(2) The grouping case:
Impact of the definition ($ATD \leq WA$) on a concrete example:
If we have 1000 machines, each sending information approximately once every 1000 seconds,

the average timestamp distance is equal to 1s as in the prior example. As in the previous example we choose the lowest value for WA which is allowed according to our definition: 1s. We are not allowed to use a window advance smaller than 1s with the same reason as in the previous example: We would just perform redundant computations if we create windows faster than tuples arrive.
So on average once every second a tuple t arrives and contributes to WPT windows with the same key. But even if we choose 1s as our window advance, 1000 aggregation results (one for each machine) will be output every second and on average only one of them will differ from the 1000 results that have been output one second ago (since only one tuple of one machine arrives per second). So we have the thousand-fold number of outputs although the rate of incoming tuples is unchanged.

In general, from the definition it follows that each tuple will start up to one new window time-slot and expire all windows of up to one window time-slot. Since each window time-slot contains up to DK number of windows (in our example one for each machine), each tuple can lead to the expiration of up to DK number of windows.
As each of the T tuples can lead to the expiration of windows of one window time-slot, it leads to the creation of up to one result set. So there will be up to T result sets.
Due to the fact that each window time-slot contains up to DK number of windows, each result set will contain up to DK number of windows. So there will be up to $T * DK$ number of windows compared to T windows in the non-grouping case.

Number of PUs
Let **P** be the number of PUs (threads) concurrently executing the algorithm.

5.2.2 Call frequency

Now we analyze how often a PU p has to execute the different parts (mentioned in Pseudocode 1) of the algorithm.

- Get & Forward Tuple: Since every tuple of the stream visits each PU, first the leftmost and lastly the rightmost, p needs to execute this part T times.
- Hash Operation: As we know, each tuple contributes to WPT number of windows. So for each incoming tuple, PU p has to find out for which of these WPT windows p is responsible. So the Hash Operation part is to be executed WPT times for each of the T tuples, so $WPT * T$ times.
- Find or Create Window: Like mentioned, each of the T tuples contributes to WPT windows, so all in all there are $WPT * T$ events where a tuple contributes to a window. However, the hash operation splits those operations among all of the P participating PUs. So if the hash function distributes the windows evenly among all P PUs, on average, p executes the Find or Create Window operation $\frac{WPT*T}{P}$ times.
- Aggregate Operation: Just as Find or Create Window, on average, p executes this part $\frac{WPT*T}{P}$ times.
- Create New Result Set If Necessary: Whenever a tuple arrives, the last PU checks, whether any windows on any PU expired and if so it creates a new result set. So this part is called T times (by the last PU).
- Add My Expired Windows to Result Set: This part is executed on p, whenever a result set reaches p. The number of result sets is depending on the number of window time-slots that

contain at least one window. As we elaborated in Section 5.2.1, the amount of windows can vary strongly. We assume that $W \leq T$ and hence that this part is not executed more often than T times. Like mentioned we cannot make any assurances, but in practical applications it might be very uncommon to have more than T windows.

- Forward Result Set: Just like the Add My Expired Windows to Result Set part, we expect that each PU is executing the Forward Result Set part between 1 and T times.

Program part	Number of executions (average) per PU
Get & Forward Tuple	T
Prepare & Execute Hash Operation	$T * WPT$
Find or Create Window	$\frac{T * WPT}{P}$
Aggregate Operation	$\frac{T * WPT}{P}$
Create New Result Set If Necessary	T (only rightmost PU)
Add My Expired Windows to Result Set	W ($W \leq T$)
Forward Result Set	W ($W \leq T$)

Table 1: Call frequency table of the HSA's program parts

According to Table 1, we create another table (Table 2) depicting the interdependencies between the aggregation parameters and the parts' call frequencies.

	Get & Forward Tuple	Prepare & Execute Hash Operation	Find or Create Window	Aggregate Operation	Create New Result Set If Necessary	Add My Expired Windows to Result Set	Forward Result Set
Increase of Thread Count							
Increase of Aggregate complexity							
Increase of Window Size							
Decrease of Window Advance							
Increase of Aggregate Operation Count							
Activation of Group by							
Colour explanation Change to part's call frequency	Locally more often globally more often		Locally unchanged globally more often		Locally unchanged globally unchanged		Locally less often globally unchanged

Table 2: Interdependencies between the aggregation parameters and the call frequency of the HSA's parts

Increase of the thread count

As we can see in Table 1, the Find Or Create Window part and the Aggregate Operation part are parallelizable (call frequency is divided by P), thus they are marked green. The Create New Result Set If Necessary part is only executed by a single PU and hence further PUs are completely unaffected by it. The remaining parts are not parallelizable at all; their call frequencies per PU are constant as the number of participating PUs grows.

Increase of the Aggregate complexity

There is no effect on the call frequency of the parts.

Increase of the window size

Increasing the window size, ceteris paribus, increases WPT, which leads to a higher call frequency of the 3 parts Prepare & Execute Hash Operation, Find or Create Window and Aggregate Operation.

Decrease of the window advance

Decreasing the window advance, ceteris paribus, increases WPT with the same effects as the increase of the window size.

Additionally, a lower window advance increases the amount of windows, which raises the call frequency of the 2 parts Add My Expired Windows to Result Set and Forward Result Set.

Increase of the number of aggregate operations executed concurrently

If x aggregate operations (instead of one) shall be executed simultaneously, the number of times we have to execute each program part is multiplied by x. The only exception is the Get & Forward Tuple part. So there still is a slight advantage of running the program once and executing x aggregate operations with it at once over executing the program x times with one aggregate operation at a time.

5.2.3 Call costs

Next, we create a table (Table 3) depicting the interdependencies between the aggregation parameters and the costs of a single call of each part.

	Get & Forward Tuple	Prepare & Execute Hash Operation	Find or Create Window	Aggregate Operation	Create New Result Set if Necessary	Add My Expired Windows to Result Set	Forward Result Set
Increase of Thread Count							
Increase of Aggregate complexity							
Increase of Window Size							
Decrease of Window Advance							(1)
Increase of Aggregate Operation Count							
Activation of Group by							(1)

Color explanation Change to part's costs	Higher	Unchanged (1): Red in case forwarded by value	Lower

Table 3: Interdependencies between the aggregation parameters and the costs of the HSA's parts

Increase of the thread count

The costs per execution of the Add My Expired Windows to Result Set part are decreasing as we increase the PU count because if we have more PUs working on an aggregation, each PU is responsible for fewer windows and hence has to locate and add fewer windows to incoming result sets.

The Find or Create Window part benefits slightly from an increasing number of participating PUs. The more PUs we are using, the less windows each PU has to store and the faster we can find and access stored windows (we will use a data structure exhibiting an *O(log(n))* runtime for all important operations (n is the number of elements in the data structure)).

Increase of the aggregate complexity

Using a more complex aggregate operation increases the cost of the Aggregate Operation part proportionally.

Increase of the window size
Changes to the window size do not affect the costs of a single call of any part of the algorithm.

Decrease of the window advance
A lower window advance leads to an increasing number of windows, so the Find or Create Window operation will need slightly more time to find and access stored windows. Accordingly, on average the Create New Result Set if Necessary part is getting slightly more expensive as well, since it is necessary to create new result sets more frequently.

Increase of the number of aggregate operations executed concurrently
If we execute separate aggregations concurrently, we store a much greater number of windows on each PU and hence the Find or Create Window part gets slightly more expensive.

Grouping of tuples by their keys
Depending on DK (the amount of distinct keys), grouping tuples may increase the number of windows drastically, which may raise the costs of the Add My Expired Windows To Result Set part to a great extent and the costs of the Find or Create Window part slightly. It also marginally affects the Prepare & Execute Hash Operation part, since if we group tuples, the hash operations take into account the windows' keys as well, making it slightly more expensive.

Summary of the interdependencies
Only the costs of two parts are significantly affected by the chosen aggregate parameters.
The Aggregate Operation part's costs are proportionally affected by the aggregation's complexity. The Add My Expired Windows To Result Sets part is affected by the number of windows per window time-slot, so the thread count and DK (the number of distinct keys).
All other parts exhibit more or less absolute independency of interdependencies between their costs and the aggregation's parameters.

Equal costs assumption
Under three conditions, we can simplify the analysis of the performance by making the assumption that all algorithm parts are equally expensive, so that one call of one algorithm part is as expensive as any other single call of any other part: We only consider cheap aggregate functions, data streams with few DK (or none at all) and single shared-memory machines (listed in more detail in the next paragraph).
Strictly speaking, this assumption is of course not true. But under those three circumstances, practical tests have shown that it is a sufficient approximation.

Three conditions for the equal costs assumption

- The computing system transports information (tuples, result sets, …) by reference, not by value. So this approximation does not apply for distributed systems. This condition is necessary to preclude that the communication network might bottleneck the processing system.
- An inexpensive aggregation is used like sum, average or min. This condition ensures that the Aggregate Operation part's costs remain comparable to the other parts' costs.
- Tuples are either not grouped by their key or if they are, DK has to be small (preferably single-digit). This condition ensures that the Add My Expired Windows to Result Set part's costs remain comparable to the other parts' costs.

5.2.4 Memory space requirements

Next, we analyze the expected memory space requirements. As we will see, they are depending on different parameters and properties of the aggregation, the data stream and the tuple structure.

There are up to $WPT * DK$ windows opened at the same time. These windows are distributed among all P participating PUs. So each PU has to store $\frac{WPT*DK}{P}$ windows.
Each window contains information about its time limit, the aggregate operation it is executing and its key which may be null; and it has a size of MW (memory window) number of bytes. Hence we need $\frac{WPT*DK*MW}{P}$ bytes of memory capacity.
In addition, a window contains either an intermediate result or tuples contributing to it, but we consider these additional memory space requirements further below in this chapter.

There is a certain amount of expired windows distributed among all P participating PUs. The number of expired windows existing at the same time depends on many different parameters: The length of the PU line P, WS, ATD (and the strength of temporary deviations from it), the size of the tuple queues, imbalances of the window responsibilities caused by the hash function and the key distribution of the incoming tuples. It is difficult to state an accurate formula for the number of expired windows existing simultaneously; practical tests have shown that in the vast majority the number of expired windows does not exceed the number of non-expired windows. Hence we expect the required memory capacity to be $\frac{WPT*DK*MW}{P}$ bytes at most.

Altogether, each tuple contributes to WPT windows.
To estimate the memory requirements for storing references to tuples or copies of tuples within windows or intermediary results within windows, we need to distinguish two types of aggregate functions:

1. **Aggregate functions that do not need to save tuples contributing to a window**
   Most aggregate functions that are computable dynamically (see Section 2.4) belong to this group, e.g. sum. After the arrival of a tuple t on a PU p, the intermediate sum of all windows t contributes to that located on p are updated and afterwards p can delete its local copy of t. So each window just stores one single intermediary result requiring MIR (memory intermediate result) bytes of space.
2. **Aggregate functions that need to save all tuples contributing to a window**
   Aggregate functions that are not computable dynamically belong to this group, e.g. median. After the arrival of a tuple t on a PU p, p stores t (or only t's value) in all windows t contributes to located on p, since for each window w the value of each tuple contributing to it is required to compute w's median after w's expiration. So each window stores all $\frac{WS}{ATD}$ number of tuples contributing to it, each requiring MT (memory tuple) bytes of space.

Required space per window in the first case: $MW + MIR$ bytes.
Required space per window in the latter case: $MW + \frac{MT*WS}{ATD}$ bytes.

The space requirements of the tuples and result sets being forwarded in the PU chain depend on the size of the queues used by the participating PUs. One result set requires MRS bytes and one tuple requires MT bytes of disc space. If we consider queues containing up to QS (queue size) number of

tuples and result sets respectively (before forcing the sending PU to pause sending further objects), the required disc space is $MT * QS + MRS * QS$ per PU.

So the maximum overall space requirements per PU that we expect excluding lower amounts of constant overhead costs are:
In the first case: $\frac{2*(MW+MIR)*WPT*DK}{P} + QS * (MT + MRS)$ bytes.
In the latter case: $\frac{(MW+\frac{MT*WS}{ATD})*WPT*DK}{P} + \frac{(MW+MIR)*WPT*DK}{P} + QS * (MT + MRS)$ bytes.

It should be noted that in both cases expired windows only contain a final result, which requires as much memory capacity as an intermediate result. So expired windows do never store any tuples.

5.2.5 Latency

In this section, we shortly introduce the definition of latency and factors influencing it.

Granted that there is one tuple t that leads to the expiration of a window w. Then the latency is defined as the real-time difference between the point of time when t entered the processing system (from the input data stream into the leftmost PU) and the point of time when w's result leaves the processing system (from the leftmost PU into the output data stream).
So during the latency time, t travels through the PU chain from the leftmost PU to the rightmost PU contributing to WPT number of windows on its way. On the rightmost PU t causes the creation of a new result set r, which moves through the PU chain from the rightmost PU to the leftmost PU collecting all expired windows having time limits up to r's time limit (window w is among them) and outputs them into the output stream.

Altogether, tuple t contributes to WPT number of windows by either updating intermediate results or being saved within them (see Chapter 5.2.4, the two different cases). So the higher WPT and the higher the complexity of update or contribution steps, the higher the latency.
When t on its way to the rightmost PU - or r on its way to the leftmost PU - arrive at a PU p, p might already have tuples and result sets in its queues that will be treated before t. Hence the larger QS (the size of the queues) and the higher P (the number of participating PUs), the higher the latency can get.
On distributed systems, the transfer of a tuple or result set to the next PU may cost a significant amount of time as well depending on latency and throughput of the communication network and the size of tuples (MT) and result sets (MRS); that increases the latency as well.

In summary, changes to the following parameters affect the latency: WPT, the complexity of aggregation steps, QS, P, the latency of the communication network, the throughput of the communication network, MT, MRS.

5.3 Discussion of the approach's runtime properties

Due to Table 1 and the equal costs assumption, the Prepare & Execute Hash Operation part is the bottleneck of the algorithm. The bigger WPT is, the more the Hash Operation part tends to be the limiting factor for the performance of the Handshake approach.
We can reduce the computation time each thread has to invest into the Find or Create window part and the Aggregate Operation part just by increasing P (see Table 1).
We cannot reduce the computation time needed for the other parts at all.

All in all, the Handshake approach has 5 different parts that are not or not entirely parallelizable.

- **Get & Forward Tuple, Hash Operation and Forward Result Set parts**
  They are not parallelizable at all.
- **Create New Result Set If Necessary part**
  This part is not parallelizable. It is executed by the rightmost PU; additional PUs working on the program are not influenced by it at all.
  It is very cheap (3 arithmetical operations and up to one constructor call) and called as often as the forwarding parts and the Add My Expired Windows to Result Set part, so the rightmost PU does not have a significant amount of extra work. As a thought experiment, we could consider this part as a subpart of the Get & Forward Tuple part which is called equally often but much more expensive. Then the rightmost PU just has a negligibly more expensive Get & Forward Tuple part than the other PUs.
- **Add My Expired Windows To Result Set part**
  It is partially parallelizable. As we increase the number of PUs working on the program, each PU still has to execute this part with the same frequency, but since the average amount of expired windows each PU has to add to the result set decreases (because they are distributed among all PUs), the work each PU has to invest decreases. As there are some constant costs in this part, the local costs are not converging to 0, so we cannot call this part entirely parallelizable.

Differences if not all three conditions are true
We shortly discuss the differences in situations in which at least one of our conditions is not fulfilled. Comparable costs of the parts are not guaranteed in these situations.

- **Using distributed systems instead of single machines**
  Depending on the speed of the communication system, the Get & Forward Tuple part or the Forward Result Set part might limit the performance.
- **Using an expensive aggregate function**
  For an aggregate function requiring *x* times as much computation time as a cheap aggregation (sum, min, average, …), we could multiply the call frequency of the Aggregate Operation part by *x* to get a realistic value estimating the computation time requirements compared to the other parts.
- **Using a data stream sending tuples with a large number of distinct keys**
  For data streams sending tuples with DK number of distinct keys, we could multiply the call frequency of the Add My Expired Windows To Result Set part by DK to get a realistic value estimating the computation time requirements compared to the other parts.

In either case, the forwarding parts, the Aggregate Operation part or the Add My Expired Windows to Result Set part can become the most time consuming part as well. E.g. a very expensive aggregate function could hide the computation costs of the hash operations up to a certain number of participating PUs.

5.4 Suggestions for improvement

If we want to increase the scalability, we need to think about alternatives:

- Can we accelerate the non-parallelizable parts?
- Can we reduce the number of times we need to execute the non-parallelizable parts?
- Can we completely avoid the non-parallelizable parts by structural changes?

Let us have a look at the 5 parts that are not or not entirely parallelizable:

I. **Get & Forward Tuple**

It is not possible to speed-up the process of getting and forwarding a tuple itself. We just get and forward a reference to a certain object, placed within the shared-memory. Also there is no way to inform n PUs with less effort than n sending steps, at least as long as we do not have a broadcast function at hand.
However, we could reduce the number of times we need to forward a tuple if we use a window distribution pattern, in which not each PU needs to process each tuple. In our distribution pattern each PU needs to know each tuple because a hash function decides about the responsibilities of the PUs over the windows. But then we also need a different communication pattern. Sticking only to neighbor-to-neighbor-communication makes it necessary to send a tuple through all PUs if for example we want to reach the last PU in the line.
One approach that is introduced later (the Round-Robin-Cyclic Pane-based approach), deals with this idea.

II. **Prepare & Execute Hash Operation**

According to our analysis, this is by far the most expensive part because as Table 1 depicts, in most settings it is being executed much more often than the other parts.
Actually it might be possible to speed-up this part in all 3 ways mentioned:

We could probably accelerate it by developing the fastest applicable hash function that still offers the necessary randomness to guarantee a random window distribution for arbitrary input data, instead of choosing a simple built in one of the chosen language's standard library.

We could also store information about the responsible PUs for each tuple's windows within this tuple. Then we would only have to calculate the hash function once for all windows a tuple contributes to. That means, whenever a tuple arrives at a PU p, p only needs to look up in the tuple's stored data, whether p is responsible for any windows this tuple contributes to and if so for which. But within the Handshake approach all tuples are entering the processing system on the same PU (the leftmost one), which would have to do all the computations to find out which PU is responsible for which windows of a tuple and store this information. So in order to avoid the leftmost PU to become the bottleneck of the system, it would be necessary to change the whole communication pattern of the algorithm to allow the input data stream to insert data tuples on each PU.
And this is exactly what we are doing within a new approach explained later on (the Hash-Once approach).

We could also completely drop the hashing part and think about other ways to distribute windows to PUs. For example we could assign windows in a round-robin fashion to the PUs. For instance, taking Figure 2 as an example and assuming 3 PUs are working on the program, we could say: Each PU p_i, $i \in \{0,1,2\}$, is responsible for all windows having an index of $3j + i, j \in \{0,1,2, \dots\}$.
That means PU0 is responsible for Window0, Window3, Window6, Window9 and so on, PU1 is responsible for Window1, Window 4, Window7, Window10 and so on and PU2 is responsible for Window2, Window5, Window8, Window11 and so on. So whenever a tuple visits a new

PU, with few calculations this PU could iterate through windows, for which it is responsible and to which this tuple contributes.
This is what we are doing within a new approach explained later on (the Round-Robin approach and also the Round-Robin-Cyclic approach).

III. **Add My Expired Windows to Result Set, Forward Result Set and Create New Result Set If Necessary:**
As long as the windows are distributed randomly over the PUs, to ensure a deterministic output order of the windows, each result set has to visit each PU within the current approach (see Section 5.12). A change to the window distribution pattern combined with another communication pattern described in the previous parts might limit the number of PUs each result set has to visit. But then one PU would have to merge the expired windows of the different result sets by their timestamps; and this is what we want to avoid, and the reason behind the result sets, where the synchronization is managed by all PUs together.

However, the Forward Result Set part is very cheap anyway, at least as long as we forward result sets just by reference and not by value. So for the Handshake approach there is no necessity to focus on accelerating this part.

As discussed in the previous section, the Add My Expired Windows to Result Set part is partially parallelizable. For each result set, each PU has to call this part once, but the amount of work is reduced as the number of participating PUs increases because of the decreasing amount of expired windows each PU has to add to the result set.

The Create New Result Set If Necessary part is responsible for creating the result sets. As long as we use a line as our network topology with the leftmost PU outputting all results and as long as each result set has to visit all PUs, there is no other way than having the rightmost PU to create result sets and send them into the left direction if we want to avoid additional network traffic.
Even with different network topologies, the creation of result sets on different PUs would require additional synchronization effort to order the result sets after visiting all PUs; this synchronization has to be done by the outputting PU. That is necessary because as result sets are not taking the same routes to visit all PUs, we cannot know in which order they will arrive at the PU that is outputting their expired windows.
However, the costs of the part are not high, since checking the necessity to create a new result set requires only two arithmetical operations; and also creating a new result set only requires one cheap call of the result set constructor.

6 The Round-Robin approach for streaming aggregation

In this chapter, we present another approach for a disjoint parallelization of sliding-window streaming aggregation. We call this algorithm the Round-Robin approach (for short RRA). The Round-Robin approach is based on the Handshake approach but changes certain aspects of it to gain advantages under certain circumstances.

First (Section 6.1), we discuss the structure of the Round-Robin approach. We present its pseudocode and explain why it exhibits deterministic output behavior.

Next (Section 6.2), we analyze its runtime properties. We analyze the call frequency of the different parts of our algorithm as well as interdependencies between the executed aggregation and the call frequencies and costs of the algorithm's parts.

Then (Section 6.3) we discuss the determined runtime properties of the algorithm. We compare them with the HSA's results.

Finally, (Section 6.4) we consider suggestions for improvements.

6.1 Structure

The next approach will still stick to the handshake communication pattern and was mentioned in Section 5.4.
While during the previous chapter a hash function decided, on which PU a window is to be processed, we will now completely drop the hash function. Instead we distribute the windows in a round-robin-fashion to the PUs to exceedingly reduce the costs to compute the window responsibilities. Now consecutive PUs are responsible for windows of consecutive window time-slots.
We pay for the saving of computation time with a disadvantageous work distribution in certain situations in which we group tuples by their keys. This is discussed in more detail in Section 6.3.

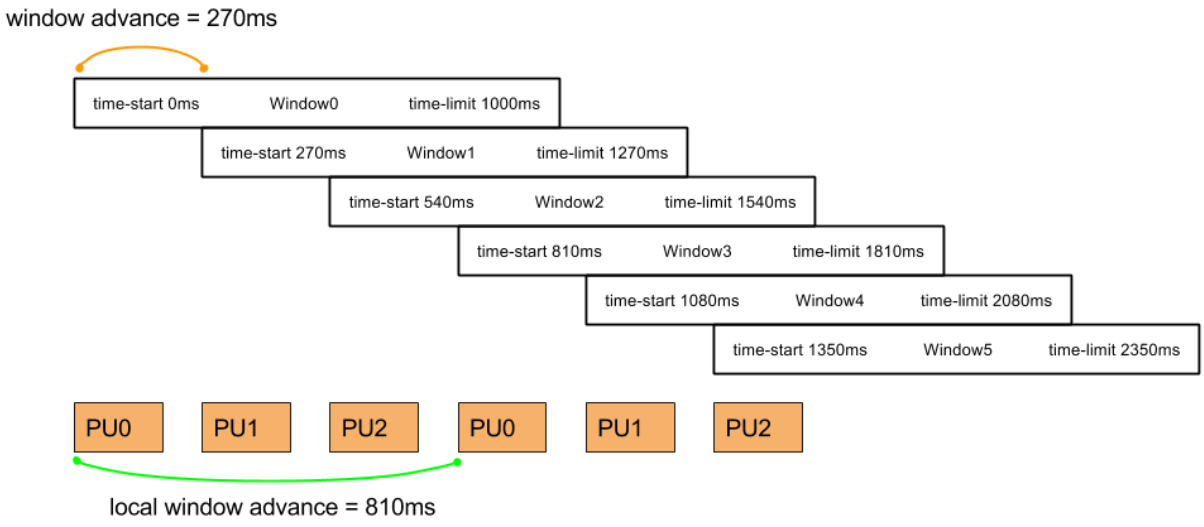


Figure 11: The Round-robin window distribution among the participating PUs

Let us assume we have n PUs working on the program.
Each PU p_i, $i \in \{0,1,\dots,n-1\}$, is responsible for all windows with the indices $j * n + I, j \in \{0,1,2,\dots\}$.

So taking AggregateOperation0 from Figure 4 as an example and using 3 PUs, we make the following observation (see Figure 11):

- PU0 is responsible for Window0, Window3, Window6…
  So in Figure 2 PU0 is responsible for the window within the window time-slot from 0ms to 1000ms, the one from 810ms to 1810ms, the one from 1620ms to 2620ms and so on.
- PU1 is responsible for Window1, Window4, Window7…
  So in Figure 2 PU1 is responsible for the window within the window time-slot from 270ms to 1270ms, the one from 1080ms to 2080ms, the one from 1890ms to 2890ms and so on.
- PU2 is responsible for Window2, Window5, Window8…
  So in Figure 2 PU2 is responsible for the window within the window time-slot from 540ms to 1540ms, the one from 1350ms to 2350ms, the one from 2160ms to 3160ms and so on.

So whenever a tuple t visits a new PU, with few calculations this PU can iterate through windows, for which it is responsible and to which t contributes.

To allow each PU to determine all windows it has to take care of, it just needs to know the window advance and window size of the aggregate operation, in our example 270ms and 1000ms, the number of PUs working on the program, we call it P and it is equal to 3 in our example, and its own rank, so 0, 1 or 2.
While each window has a time limit difference of 270ms to its two neighbors, on each single PU all pairs of neighbored windows have a distance of $P * WA$, so in our example $3 * 270ms = 810ms$. Window0, Window3, Window6… which are all computed by PU0 have a distance of 810ms to each other. The same applies to Window1, Window4, Window7… which are all computed by PU1 and Window2, Window5, Window8... which are all computed by PU2. We name this distance *local window advance* (or *LWA* for short).

Concerning the communication structure, Figure 9 and Figure 10 of the Handshake approach are still correct and applicable for the Round-Robin approach, since the pattern has not changed.

6.1.1 Pseudocode

The overall structure of the program compared to the previous approach and its pseudocode - see Pseudocode 1- changes only in one detail: The Hash-Operation part is omitted.

```
1.  while (true)
2.      if ((Tuple t = incTupleQueue.first) != null)
3.          Get&ForwardTuple(t)
4.          for (each window w to which tuple t contributes that I have to compute)
5.              FindOrCreateWindow(w)
6.              AggregateOperation(w,t)
7.          if (myRank == PUCount - 1)
8.              CreateNewResultSetIfNecesarry(t)
9.
10.     if ((ResultSet r = incResultSetQueue.first) != null)
11.         AddMyExpiredWindowsToResultSet(r)
12.         ForwardOrOutputResultSet(r)
```

Pseudocode 2: Pseudocode of the Round-Robin approach. It is executed by each PU.

Assumed a tuple named *curTuple* arrives at an arbitrary PU p. An appropriate piece of pseudocode used by p for determining all windows *curTuple* contributes to that p has to compute could look as depicted in Pseudocode 3.

```
1.  long localWindowAdvance = windowAdvance * puCount
2.  long localWindowOffset = myRank * windowAdvance
3.  long tempLong = curTuple.time % localWindowAdvance
4.  long firstWindowTimeLimit = curTuple.time - tempLong + localWindowOffset
5.  if (firstWindowTimeLimit <= curTuple.time)
6.      firstWindowTimeLimit += localWindowAdvance
7.  long lastWindowTimeLimit = curTuple.time + windowSize;
8.
9.  for (long curWindowTimeLimit = firstWindowTimeLimit; curWindowTimeLimit <= lastWindo
    wTimeLimit; curWindowTimeLimit += localWindowAdvance)
10.     //proceed by processing the parts FindOrCreateWindow & AggregateOperation
```

Pseudocode 3: Pseudocode of the window responsibility calculations

PU p searches the first window *curTuple* contributes to that p has to process (*firstWindowTimeLimit*). After each for-loop iteration, p increases the current window's time limit by LWA number of time units (see Pseudocode 3) to get the next window w of which p has to take care.
As soon as w is out of range to *curTuple*, so as soon as w's time limit is more than WS number of time units away, p has finished processing all windows it is responsible for.

6.1.2 Determinism of the output stream's results

Nothing has changed to the communication pattern compared to the HSA in the previous chapter. The result sets being sent from the last PU in the line still collect expired windows up to a certain time limit and order them by their expiration time.
So all information concerning the determinism of this approach can be found in Section 5.1.2.

6.2 Analysis of the algorithm's runtime properties

Similar to Section 5.2, we want to analyze the runtime properties of the algorithm and its parts.

6.2.1 Definitions and denotations

The definitions and denotations of this approach are identical to the ones of the Handshake approach (Section 5.2.1).

6.2.2 Call frequency

The call frequencies are equal to the ones of the HSA with the exception that there is no Prepare & Execute Hash Operation part. So the call frequency table as well as the parameter interdependencies table look similar to the tables of the previous approach (see Table 1) and the justifications can be read in Section 5.2.2.

Program part	Number of executions (average) per PU
Get & Forward Tuple	T
Find or Create Window	$\frac{T * WPT}{P}$
Aggregate Operation	$\frac{T * WPT}{P}$
Create New Result Set If Necessary	T (only rightmost PU)
Add My Expired Windows to Result Set	W ($W \leq T$)
Forward Result Set	W ($W \leq T$)

Table 4: Call frequency table of the RRA's program parts

The whole window responsibility calculations presented in Pseudocode 3 consist of 6-7 arithmetical operations per arriving tuple. Because of the negligible costs that are not influenced by any aggregation parameter but that are constantly very low, we did not create a new part for them. We could just add

them to the Get & Forward Tuple part. So whenever a PU gets and forwards a tuple, it additionally calculates the window responsibilities.

	Get & Forward Tuple	Find or Create Window	Aggregate Operation	Create New Result Set if Necessary	Add My Expired Windows to Result Set	Forward Result Set
Increase of Thread Count						
Increase of Aggregate complexity						
Increase of Window Size						
Decrease of Window Advance						
Increase of Aggregate Operation Count						
Activation of Group by						
Color explanation Change to part's call frequency	Locally more often globally more often		Locally unchanged globally more often		Locally unchanged globally unchanged	Locally less often globally unchanged

Table 5: Interdependencies between the aggregation parameters and the call frequency of the RRA's parts

6.2.3 Call costs

The call costs are equal to the HSA with the exception that there is no Prepare & Execute Hash Operation part. So the parameter interdependencies table looks similar to the one of the previous approach and the justifications can be read in Section 5.2.3.

	Get & Forward Tuple	Find or Create Window	Aggregate Operation	Create New Result Set if Necessary	Add My Expired Windows to Result Set	Forward Result Set
Increase of Thread Count						
Increase of Aggregate complexity						
Increase of Window Size						
Decrease of Window Advance						(1)
Increase of Aggregate Operation Count						
Activation of Group by						(1)
Color explanation Change to part's costs	Higher		Unchanged (1): Red in case forwarded by value		Lower	

Table 6: Interdependencies between the aggregation parameters and the costs of the RRA's parts

6.2.4 Memory space requirements

We expect memory requirements equivalent to the ones of the HSA (see Chapter 5.2.4) due to the identical structure.

6.2.5 Latency

We do not expect significant latency differences between both approaches, since their structures are similar (see Chapter 5.2.5). In many situations the latency can be slightly lower because of the faster tuple processing rate (no hash operations), especially if WPT is large.

6.3 Discussion of the approach's runtime properties

The Round-Robin approach still has 4 different parts that are not or not entirely parallelizable. The Prepare & Execute Hash Operation part that was the bottleneck of the Handshake approach has been omitted, the rest remains mainly unchanged.

With the same definitions and prerequisites as stated in Section 5.2.1 and the assumption that one call of each part costs the same amount of computing time, we want to figure out the program parts that are limiting the execution speed and parallelization potential.
In the Round-Robin-Handshake approach there are two possible situations with different parts being the bottleneck (see Table 4):

- *WPT>P*: If one tuple contributes to more windows than there are PUs, than the most frequently called parts are the Find or Create Window part and the Aggregate Operation part. We could add more PUs to our processing system to reduce the number of executions of the Find or Create Window part and the Aggregate Operation part each PU has to do.
- *P≥WPT*: Else, we cannot significantly accelerate the algorithm by adding more PUs, since the 4 parts Get & Forward Tuple, Create New Result Set If Necessary, Add My Expired Windows to Result Set and Forward Result Set still have to be executed T times each by every PU, no matter how many PUs we are using.

So the 4 latter parts hinder us from efficiently using more than WPT PUs for this approach.

Comparison with the Handshake approach
Let us compare the expected throughput of the Round-Robin approach with the Handshake approach. Since the most frequently called part of the algorithm is omitted and because there are only minor changes to the other parts, one could expect that the Round-Robin approach is always faster than the Handshake approach, which is not correct. There are some cases, when the Handshake approach is superior to the Round-Robin approach.

Let us assume we want to execute a very expensive aggregation. So the assumption of equal costs of each part does no longer apply. We assume that one execution of the Aggregate Operation part is as costly as 1000 executions of the Prepare & Execute Hash Operation part (and 1000 times as expensive as the other parts as well). We use 10 PUs. The window size be 1000 and the window advance be 500, which means that without grouping $WPT = \frac{WS}{WA} = 2$ non-expired windows are existing all the time. Looking into Table 4, we can see that the Prepare & Execute Hash Operation part is called $T * WPT$ times on each PU, so in our case 2T times. The Aggregate Operation part is called $\frac{T*WPT}{P}$ times, so 0,2T times. Since we are executing a very expensive aggregation, costing as much as 1000 Hash Operation executions, the execution time of the Aggregate Operation part is $\frac{1000*0,2T}{2T} = 100$ times higher than the execution time of the Hash Operation. So in this case the Hash Operation part is not the limiting factor and its costs are negligible.
Now we use both approaches to execute the aggregate operator. We execute it two times. The first time we do not group tuples by their key, the second time we do.

- **Without grouping**
  The Handshake approach assigns both windows via a hash function randomly to PUs (possibly both to the same, which would be bad), while the Round-Robin approach assigns both

consecutive windows to consecutive PUs.
So no matter if we are using the Handshake approach or the Round-Robin approach, at each point of time 8 PUs will be almost completely unemployed because they have no window to take care about.
As long as the Handshake approach does not assign both windows to the same PU, both approaches have a similar throughput. As mentioned, the additional Hash Operations of the Handshake approach carry no weight, since the aggregation we are executing is by far more expensive.

- **With grouping**
  Let us assume a lot of tuples with $DK = 200$ different keys are arriving with timestamps between 500 and 1000. Without grouping we would have $WPT = \frac{WS}{WA} = 2$ open windows now: One in the window time-slot from 0 to 1000, one in the window time-slot from 500 to 1500. With grouping, each window time-slot contains one window for each distinct key. Since there are 200 different keys, we have $WPT * DK = 2 * 200 = 400$ open windows.
  The Round-Robin approach still assigns each of the 200 windows in the window time-slot from 0 to 1000 to one PU and each of the 200 windows in the window time-slot from 500 to 1500 to the following PU. Hence 2 PUs are processing 200 windows each, while 8 PUs are mostly unemployed.
  The Handshake approach assigns all 400 windows randomly to the 10 PUs. Thus on average all 10 PUs are processing 40 windows each.
  Since the Hash Operations of the Handshake approach still carry no weight because of the expensive aggregation, the Handshake approach will be up to 5 times faster than the Round-Robin approach.

So only under three conditions that have to be true at the same time, the Handshake approach might be superior: As long as one of those conditions is not fulfilled, the Round-Robin approach is faster.

- The Aggregate Operation part has to be significantly more expensive (approximately at least P times more expensive) than the Hash Operation part to make the Aggregate Operation part the limiting factor
- The tuples must be grouped by their key and there have to be at least as many distinct keys that $WPT * DK > P$ holds
- $WPT = \frac{WS}{WA} < P$
  The number of simultaneously existing (non-expired) consecutive windows must be smaller than the number of participating PUs (so that the Round-Robin approach does not distribute the work among all participating PUs).

6.4 Suggestions for improvement

Partially, the suggestions for the Handshake approach (Section 5.4) still apply for the Round-Robin approach. In the Round-Robin approach, the most frequently used operations (compare Table 1 and Table 4) are called T times instead of $T * WPT$ times on each PU, as long as at least WPT number of PUs are working concurrently.

In order to avoid that each PU has to call the non-parallelizable part Get & Forward Tuple T times, so once for each arriving tuple, we have only one possible option: We need to change the structure of the algorithm and also the communication pattern in a way that not each tuple has to visit each PU.

Similarly, in order to avoid that each PU has to call the operations Add My Expired Windows to Result Set and Forward Result Set up to T times, so once for each arriving result set (and as mentioned in Section 5.2.1, we do not expect there to be more than T result sets and window time-slots), we need to change the structure of the algorithm and also the communication pattern in a way that not each result set has to visit each PU or we need to reduce the number of result sets which increases the latency and raises the memory requirements of the processing system because more expired windows need to be stored until a result set collects them.

The approach introduced in Chapter 9 deals with both of these suggestions; neither of them will be part of the next approach.

7 The Hash-Once approach for streaming aggregation

In this chapter, we present another approach for a disjoint parallelization of sliding-window streaming aggregation. We call this algorithm the Hash-Once approach (for short HOA). The Hash-Once approach is based on the Handshake approach but changes certain aspects of it to gain advantages under certain circumstances.

First (Section 7.1), we discuss the structure of the Hash-Once approach. We present its pseudocode and explain why it exhibits deterministic output behavior.

Next (Section 7.2), we analyze its runtime properties. We analyze the call frequency of the different parts of our algorithm as well as interdependencies between the executed aggregation and the call frequencies and costs of the algorithm's parts.

Then (Section 7.3) we discuss the determined runtime properties of the algorithm. We compare them with the results of the previous approaches (HSA and RRA).

Finally, (Section 7.4) we consider suggestions for improvements.

7.1 Structure

The Hash-Once approach is based on an idea explained in Section 5.3.
In short, it breaks partially with the principles of the previous approaches by allowing certain further communication links (not only neighbor-to-neighbor communication) and changing the structure of tuples by adding additional information to them.

Within the Handshake approach each tuple causes WPT Hash-Operation executions on each PU, where WPT is the number of windows each tuple contributes to (Section 5.1). All of these WPT Hash Operation part executions among the PUs are identical.
As the Round-Robin approach shows, we can completely avoid using hash functions, which accelerates the speed in many cases considerably. But in certain cases presented in Section 6.3.1 this approach cannot distribute the work evenly among the PUs. For those cases we try to improve the Handshake approach by reducing the number of Hash Operations the PUs have to execute to determine window responsibilities.

To significantly decrease the amount of Hash Operations we need to execute, we could store information about the responsible PUs for each window to which a tuple t contributes within this tuple t. Then for each tuple t we would only have to calculate the hash function once for each window to which t contributes. That means, whenever a tuple arrives at a PU, this PU only needs to look up in t's stored data, whether it is responsible for any windows this tuple contributes to and if so for which.

Figure 12 depicts the first three tuples and the aggregation used in Chapter 2. Now, they additionally contain responsibility information.
Each tuples carries P number of lists. In each list we store the window start time of all windows the respective PU is responsible for.
Here we assume (it is an arbitrary definition) that the hash function is mapping Window0 (starting at time 0ms) to PU0, Window1 (starting at time 270ms) to PU1, Window2 to PU1, Window3 to PU2, Window4 to PU0 and Window5 to PU2.

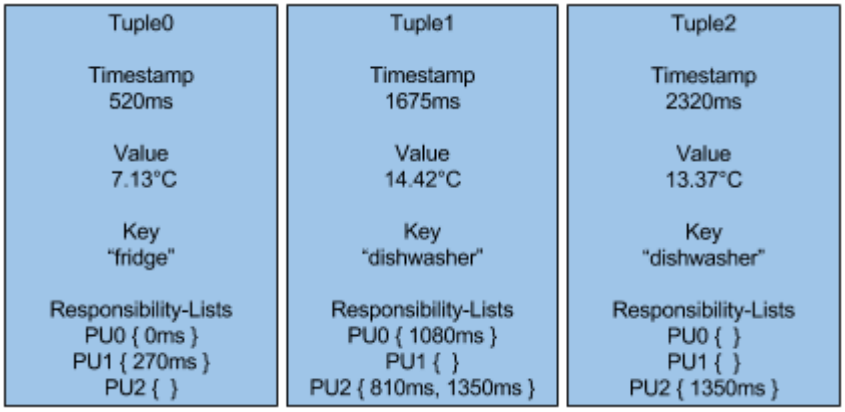


Figure 12: Tuples and their responsibility information in the HOA

However, if each PU reads the tuple's data to determine, whether it is responsible for any window this tuple contributes to, one PU has to store this information first. And to make sure that the information is available on each PU, the PU on which the tuples enter the processing system, so the leftmost PU, has to execute the Hash Operations and write the results to the tuples.

So within the Handshake approach the leftmost PU would have to do all the work to find out which PU is responsible for which windows of a given tuple and store this information.
That means that the leftmost PU has to do a lot of extra work. Even if we lay down that PU0 drops all other tasks and is merely responsible for executing the Hash Operations, it will bottleneck the system once a certain rate of necessary Hash Operations per second is exceeded. The rate at which Hash Operations are to be computed is defined by the number of tuples per second multiplied with WPT, the amount of windows each tuple contributes to.

So in order to avoid the leftmost PU to become the bottleneck of the system, we could distribute the necessary Hash Operation executions among all PUs. But then it would be necessary to change communication pattern of the algorithm to allow the input data stream to insert data tuples on each PU. And this is exactly what we are doing within this approach (see Figure 13).

Assumed we have 4 PUs. Each of the 4 PUs receives about a quarter of the input stream's tuples and is responsible to calculate the responsibilities of the windows they are contributing to using the Hash Operation.
However, it is still necessary that each tuple visits each PU. That means that a PU in the middle of the PU line (PU1 or PU2 in our example) needs to send its tuples in both directions unless we change another property of our communication pattern: We connect the rightmost with the leftmost PU so we connect the PU line to a PU ring. Now the PUs can stick to sending tuples to their successor. Tuples will need another parameter informing about where this tuple entered the PU ring so that we get to know as soon as the tuple has visited all PUs; then we can delete it.

Taking Figure 13 as an example, let us call the left upper PU PU0 (or *first PU*) and the other PUs in clockwise order PU1, PU2 and PU3.
If the input data stream inserts a tuple t into PU1, we store in t the information that it entered the processing system on PU1. After that PU1 executes the Hash Operations for all windows t contributes to and stores the information about the responsible PUs within t. Now t is being forwarded and processed PU by PU in clockwise order. Each PU p checks for incoming tuples, whether p is the last PU

this tuple has to visit. In case of t, PU0 is the last PU, which t has to visit. So after processing t, PU0 discards t instead of forwarding it.

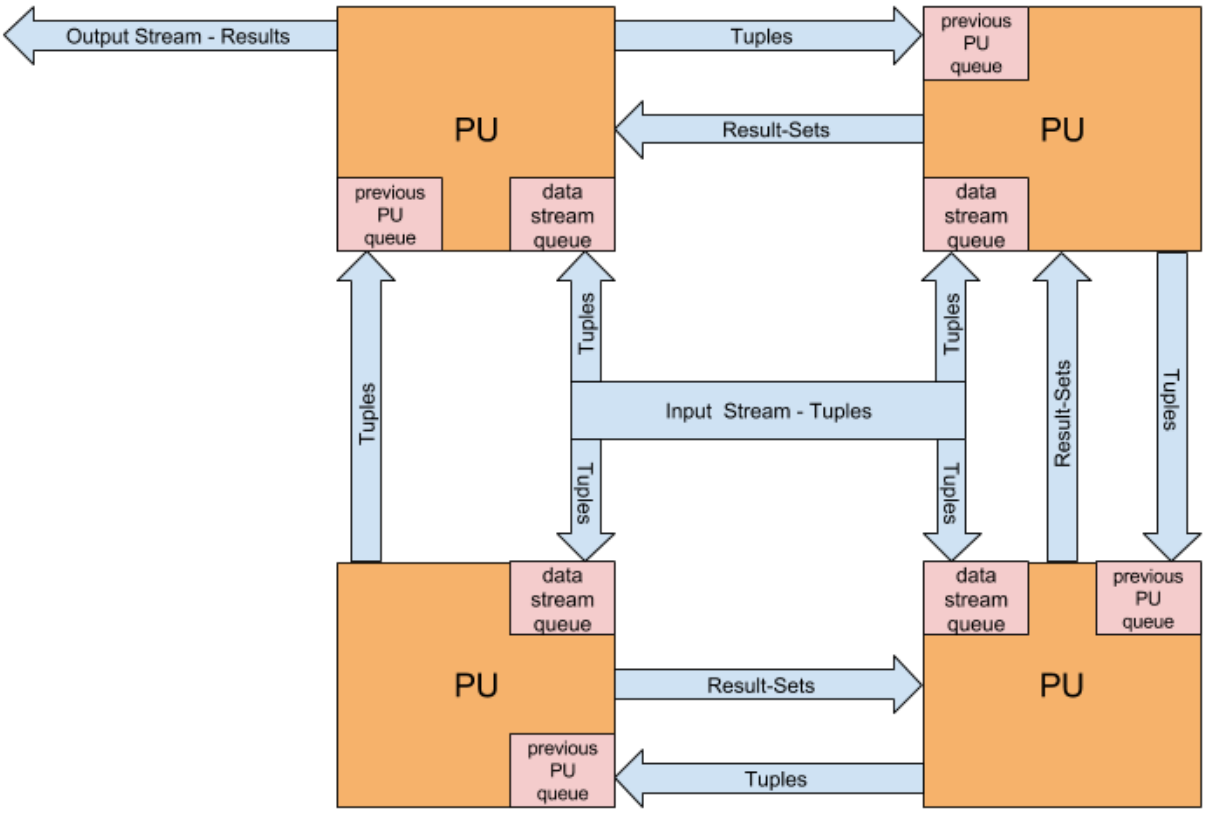


Figure 13: Structure of the Hash-Once approach with 4 PUs.

The save-timestamp

It is important for each PU to know the so called *save-timestamp*. A PU needs to know the timestamp up to which it has already processed all tuples, no matter on which PU they have been inserted. If, for example, the save-timestamp of a PU is equal to 1000, this PU knows that no further tuples with a timestamp lower than 1000 will arrive. If a result set arrives now with a time limit of 900, the PU knows that it definitely has already received and computed all tuples that might contribute to the windows the result set is collecting.

Since the input data stream sends new tuples to each PU, tuples are not necessarily arriving in timestamp order at the PUs.
Hence we need to deploy certain rules for forwarding tuples to use such a safe-timestamp which is necessary to ensure deterministic output behavior.

First of all, each PU uses two different queues to receive tuples (see Figure 13):
One queue accepts tuples coming from the input data stream. We call it *data stream queue*.
Another queue accepts tuples arriving from the previous PU. We call it *previous PU queue*.

The rules for each PU for forwarding tuples are as follows.

1. **If my data stream queue is empty and my previous PU queue is empty**
   Then I wait for incoming tuples

2. **If my data stream queue is empty and my previous PU queue is filled**
   Then I wait for incoming tuples
3. **If my data stream queue is filled and my previous PU queue is empty**
   Then I take the first tuple of the data stream queue, then I forward it to my successor and then I process it.
4. **If my data stream queue is filled and my previous PU queue is filled**
   Then I compare the timestamp of the first tuples in both queues and take the tuple with the lower one.
   If I am not the last PU processing this tuple, then I forward it to my successor and then I process it.

Using these rules, a PU can maintain a save-timestamp.
Let us assume the input data stream sends a tuple t to PU0.
All further tuples the input data stream sends to PU0 will have a higher timestamp.
PU3 is the last PU tuple t will visit. When t visits PU3, it has already visited PU0 (where it entered the processing system), PU1 and PU2.
All tuples that have entered the processing system on PU0 and that will arrive at PU3 after tuple t, have a higher timestamp because they were sent to PU0 by the input data stream after t was sent. To be forwarded by PU1, t needed to have a lower timestamp than the tuples entering the processing system on PU1 (rule 4). If there are currently no tuples in PU1's data stream queue, we wait until there are some (rule 2) to ensure that t is only being forwarded by PU1 if we can be sure that all further tuples PU1 sends, have a higher timestamp.
PU2 acts similarly.
So when t enters PU3, we know that no further tuples with a lower timestamp than the timestamp of t will arrive, neither from PU0, nor from PU1 or PU2. So whenever a tuple that was inserted on PU0 enters PU3, PU3 can take its timestamp to update its save-timestamp, since it knows that further arriving tuples will have higher timestamps.
Similarly, whenever a tuple that was inserted on PU1 enters PU0 (the last PU it visits), PU0 takes its timestamp to update its save-timestamp. PU1 updates its save-timestamps whenever tuples arrive that were inserted at PU2 and PU2 updates its save-timestamp whenever tuples arrive that were inserted at PU 3.
So now each PU has a save-timestamp and will add windows to an arriving result set not before its save-timestamp is greater than or equal to the result set's time limit.

Result sets
The creation of result sets is still solely being handled by the last PU in the ring, in Figure 13 the left lower one, so in the same manner as in the Handshake approach. Result sets are being forwarded counterclockwise PU by PU. Before forwarding it, each PU waits until its save-timestamp is greater or equal than the result set's time limit and then it adds its expired windows to the result set.

The first PU in the ring, PU0, will produce output tuples whenever a result set arrives.

Order-sensitive aggregations
The communication pattern has changed compared to the approach introduced in the previous chapters. Tuples are entering the processing system on arbitrary PUs but still visit all PUs. They are, in contrast to the previous approaches, not necessarily arriving in timestamp order at PUs.
When processing order-sensitive aggregate operations, we cannot dynamically compute them without

further effort if it is not guaranteed that tuples are arriving with non-decreasing timestamps (discussed in detail in Section 2.4).

7.1.1 Pseudocode

The corresponding pseudocode representing this approach's structure is shown in Pseudocode 4. It is being executed by each PU.

A PU p continuously (line 1) checks, whether there are tuples in one or both of the tuple queues (line 2) or result sets (line 24).

```
1.  while (true)
2.      if (datastreamQueue.isFilled() or prevPUQueue.isFilled())
3.          Tuple tupleToProcess = null
4.          if (datastreamQueue.isEmpty() and (Tuple t = prevPUQueue.first) != null)
5.          else
6.              if ((Tuple t = datastreamQueue.first) != null and prevPUQueue.isEmpty())

7.                  Prepare&ExecuteHashOperation&StoreResultsInTuple(t)
8.                  tupleToProcess = t
9.              else
10.                 if ((Tuple t1 = datastreamQueue.first) != null and (Tuple t2 = prevP
    UQueue.first) != null)
11.                     if (t1.timestamp < t2.timestamp)
12.                         Prepare&ExecuteHashOperation&StoreResultsInTuple(t1)
13.                         tupleToProcess = t1
14.                     else
15.                         tupleToProcess = t2
16.
17.         if (tupleToProcess != null)
18.             if (tupleToProcess.lastPUVisited != me)
19.                 Get&ForwardTuple(tupleToProcess)
20.             else
21.                 mySafeTimestamp = tupleToProcess.timestamp
22.             for (each Window w, Tuple tupleToProcess contributes to that I have to c
    ompute)
23.                 FindOrCreateWindow(w)
24.                 AggregateOperation(w,tupleToProcess)
25.             if (myRank == PUCount - 1)
26.                 CreateNewResultSetIfNecesarry(tupleToProcess)
27.
28.     if ((ResultSet r = incomingResultSets.first) != null and mySafeTimestamp >= r.ti
    meLimit)
29.         AddMyExpiredWindowsToResultSet(r)
30.         ForwardOrOutputResultSet(r)
```

Pseudocode 4: Pseudocode of the Hash-Once approach. It is executed by each PU.

If there are tuples in at least one of both incoming tuple queues (the data stream queue or the previous PU queue), p applies the 4 forwarding rules described in the previous section. If a tuple of the data stream queue is being chosen, p has to calculate the responsibilities and saves them within the tuple (lines 7 or 12).

Now, if a tuple was chosen (line 17), p forwards it to the next PU (line 19) unless p is the last PU this tuple has to visit (line 18). Otherwise p can update its safe-timestamp (line 21).

Next p processes the tuple. It determines all windows to which this tuple contributes that p has to compute just by looking up the responsibility information stored within this tuple (line 22).

So if p is responsible to process a window w, it checks, whether it has already stored w and started processing it and if not it stores w. Then PU p adds tuple t to window w (line 23).

In case w's aggregate operation is dynamically computable (Section 2.4), we can execute the next step

of the aggregate operation to create a new intermediate result (line 24).
Next, the last PU (line 25) has to check, whether any windows have expired and if so, it creates a new result set (line 26).

If there arrived a new result set r and p's safe-timestamp is greater than or equal to r's time limit (line 28), PU p adds all of its expired windows to r (line 29). Then it forwards r to its left neighbor (line 30). If p is the PU with index 0, it instead puts the results of the expired windows contained in r into the output stream.

7.1.2 Determinism of the output stream's results

The save-timestamp explained in the previous section ensures that a window will never leave a PU until the PU knows that no further tuples contributing to that window will arrive. It forces incoming result sets to wait until all windows contributing to this result set have expired.
So as in the previous approaches the result sets being sent from the last PU contain all expired windows up to a certain time limit ordered by their expiration time. Hence the output behavior is deterministic in this approach too.

7.2 Analysis of the algorithm's runtime properties

Similar to Section 5.2, we want to analyze the runtime properties of the algorithm and its parts.

7.2.1 Definitions and denotations

The definitions and denotations of this approach are identical to the ones of the Handshake approach (Section 5.2.1).

7.2.2 Call frequency

The call frequencies are equal to the ones of the HSA with the exception that the Prepare & Execute Hash Operation & Store Results in Tuple part substitutes the Prepare & Execute Hash Operation part of the Handshake approach. This part is now also accountable for storing the determined window responsibilities within the respective tuple. By storing the responsibilities within the tuples, we reduce the amount of executions of the Hash Operation part by the factor P; we divide the remaining work ($T * WPT$) evenly among all PUs, which leads to $\frac{T*WPT}{P}$ calls per PU.

Program part	Number of executions (average) per PU
Get & Forward Tuple	T
Prepare & Execute Hash Operation & Store Results In Tuple	$\frac{T * WPT}{P}$
Find or Create Window	$\frac{T * WPT}{P}$
Aggregate Operation	$\frac{T * WPT}{P}$
Create New Result Set If Necessary	T (only last PU in ring)
Add My Expired Windows to Result Set	W ($W \leq T$)
Forward Result Set	W ($W \leq T$)

Table 7: Call frequency table of the HOA's program parts

The call frequency table as well as the parameter interdependencies table look similar to the tables of the Handshake approach (see Table 1) and the justifications can be read in Section 5.2.2.

	Get & Forward Tuple	Prepare & Execute Hash Operation & Store Results in Tuple	Find or Create Window	Aggregate Operation	Create New Result Set if Necessary	Add My Expired Windows to Result Set	Forward Result Set
Increase of Thread Count							
Increase of Aggregate complexity							
Increase of Window Size							
Decrease of Window Advance							
Increase of Aggregate Operation Count							
Activation of Group by							
Color explanation Change to part's call frequency	Locally more often globally more often		Locally unchanged globally more often		Locally unchanged globally unchanged		Locally less often globally unchanged

Table 8: Interdependencies between the aggregation parameters and the call frequency of the HOA's parts

7.2.3 Call costs

The call costs are equal to the HSA with the exception that the communication costs increase, especially if we transport data by value, since all tuples carry additional information about the window responsibilities. That can increase the size of tuples by several magnitudes depending on WPT. The rest of the parameter interdependencies table looks similar to the one of the Handshake approach and the justifications can be read in Section 5.2.3.

	Get & Forward Tuple	Prepare & Execute Hash Operation & Store Results in Tuple	Find or Create Window	Aggregate Operation	Create New Result Set if Necessary	Add My Expired Windows to Result Set	Forward Result Set
Increase of Thread Count							
Increase of Aggregate complexity							
Increase of Window Size	(1)						
Decrease of Window Advance	(1)						(1)
Increase of Aggregate Operation Count	(1)						
Activation of Group by							(1)
Color explanation Change to part's costs	Higher		Unchanged (1): Red in case forwarded by value			Lower	

Table 9: Interdependencies between the aggregation parameters and the costs of the HOA's parts

7.2.4 Memory space requirements

We expect memory requirements similar to the ones of the HSA (see Section 5.2.4). They can be slightly higher due to the usage of two tuple queues instead of one and the higher latency (see the next section, Section 7.2.5), which can lead to larger numbers of expired windows that are simultaneously stored in the processing system waiting to be output.

7.2.5 Latency

While we expect very few circumstances, under which the Handshake approach will be superior to the Hash-Once approach in terms of throughput (see Table 7), we expect the latency to be much lower in the Handshake approach.
In the Handshake approach, the tuple that causes the creation of a new result set at the rightmost PU has already passed all PUs in the line, so the result set can quickly pass the PU line because all PUs have processed the necessary tuples already.
In the Hash-Once approach, the tuple that causes the creation of a new result set will only update the safe-timestamp of one PU. The result set can only pass the whole PU ring, as soon as each PU has a safe-timestamp greater than or equal to the result set's time limit.
Depending on the input stream's properties, this synchronizing behavior may introduce a huge delay. For example, if the input stream stops sending tuples for a certain amount of time, some PUs might still have a safe-timestamp that is not high enough to allow result sets to pass.
Possible imbalances of the work distribution or other external influencing factors (e.g. scheduling decisions of the operating system) between the participating PUs may lead to PUs exhibiting incoming tuple queues that are filled with many elements; this can cause delayed updates of the safe-timestamp on those PUs and hence hinder result sets from quickly passing the PU ring to be output.
However, it is difficult to quantify the latency.
In an ideal case, if the input stream is sending tuples continuously at a constant rate and evenly distributed among the participating PUs and there are no imbalances between the participating PUs' workloads, then we do not expect long latencies. If the number of participating PUs is single-digit or double-digit and the tuple processing speed is not exceptionally slow (e.g. because of very expensive aggregations or very large numbers for WPT), we expect latencies of up to one second.

Since the Round-Robin approach has the same communication structure as the Handshake approach, the argumentation here is similar and we expect the Round-Robin approach's latency to be much lower than the Hash-Once approach's latency.

7.3 Discussion of the approach's runtime properties

The Hash-Once approach still has 4 different parts that are not or not entirely parallelizable. The Prepare & Execute Hash Operation part that was the bottleneck of the Handshake approach has been substituted by a parallelizable part, the rest remains mainly unchanged.

With the same definitions and prerequisites as stated in Section 5.2.1 and the assumption that one call of each part costs the same amount of computing time, we want to figure out the program parts that are limiting the execution speed and parallelization potential.
In the Hash-Once approach there are two possible situations with different parts being the bottleneck (see Table 7):

- WPT>P: If one tuple contributes to more windows than there are PUs, than the most frequently called operations are the Hash Operation, the Find or Create Window operation and the Aggregate Operation.
  We could add more PUs to our processing system to reduce the number of executions each PU has to perform.
- P≥WPT: Else, we cannot significantly accelerate the algorithm by adding more PUs, since the 4 parts Get & Forward Tuple, Create New Result Set If Necessary, Add My Expired Windows to

Result Set and Forward Result Set still have to be computed T times each by every PU, no matter how many PUs we are using.

So the 4 latter parts hinder us from efficiently using more than WPT PUs for this approach.

Comparison of the throughput with the Handshake approach and the Round-Robin approach
Next we compare the performance of the Hash-Once approach with the previous approaches.

Since the most frequently called part of the Handshake approach was substituted by a cheaper part, we expect the Hash-Once approach to be much faster than the Handshake approach. The Hash-Once approach also performs well under the 3 conditions that are unfavorable for the Round-Robin approach mentioned in Section 6.2.4.

The Round-Robin approach, which completely dropped the Hash Operation part instead of just substituting it with a cheaper part, will in most settings perform better than the Hash-Once approach, but only slightly better, since the costs of the Hash-Once approach's Hash Operation part are asymptotically equal to the Find or Create Window part and the Aggregate Operation part. If we choose a very expensive aggregation, the advantage of the Round-Robin approach might even become negligible.
However, if the aggregation's parameters are as stated in Section 6.2.4, the Hash-Once approach will be superior to the Round-Robin approach, just like the Handshake approach, because the Round-Robin approach is not able to distribute the work among all participating PUs

Advantages of the structural changes
There are two further advantages of the Hash-Once approach over the previous approaches resulting from the structural changes:

- It gives us the opportunity to use multiple input streams at the same time; as for one input stream, each stream has to guarantee timestamp order of its own tuples, but timestamp order between the different streams is not necessary. Each input stream sends data to one of the participating PUs and by using the 4 synchronization rules explained in Section 7.1, all input streams get synchronized and a deterministic output is ensured. This way the synchronization's costs are distributed evenly among all PUs.
  In the Handshake approach and the Round-Robin approach when operating with several input streams, it would be necessary to charge one extra thread with the task of merging all input streams before forwarding it to the first PU in the line.
- The Hash-Once approach may in some situations be more resilient to short variations of the participating PUs' speed. In the previous approaches each PU that works slightly slower for a short period of time – for example due to temporary imbalance of the work distribution – slows all the following PUs in the line because it forces them to wait for further tuples to be forwarded. In the Hash-Once approach, each tuple has its own input stream and as we know tuples from the own input stream can be processed and forwarded if the previous PU queue is empty, which means that no PU has to wait for the previous PUs, as long as there are tuples incoming from the input stream.
  Hence we expect a slightly higher processor utilization and accordingly a somewhat better performance.

Disadvantages of the structural changes

There are four disadvantages of the Hash-Once approach resulting from the change of the communication structure:

- If we want to apply this approach to a distributed system, tuples have to be transferred physically. It is not sufficient to transport references like in a shared-memory based version, since all participating PUs will possibly need access to all of the tuples data. Since the Hash-Once approach stores information about the window responsibilities within the tuples, the size of tuples might increase dramatically, depending on the WPT parameter and the "default" size of a tuple. This can increase the network traffic by several magnitudes.
  In contrast, in the Handshake approach and the Round-Robin approach the tuples' size always remains constant.
- The connections are not limited to neighbor-to-neighbor communication. Hence in case we want to use this approach on distributed systems, we need to maintain twice as many connections: one from each PU to both of its neighbors and one from each PU to the input thread.
- We already discussed that the latency will always be higher due to the necessity for result sets to wait until each PU on its way has a sufficiently high safe-timestamp.
  If the input streams do not send tuples for a longer period of time to one PU p (for example because there is a temporary congestion or problem with the connection), p's neighbor in counterclockwise direction (the last PU p's tuple are visiting on their way through the PU ring) will not be able to update its safe-timestamp during this time. P's neighbor can neither process and forward tuples (see the forwarding rules in Section 7.1) nor result sets.
  So in order to work properly and with reasonable latency, each PU needs incoming tuples in short intervals.
- As discussed in Section 7.1, without further effort it is not possible to dynamically compute order-sensitive aggregate operations, since in general tuples are not arriving in timestamp order at the different PUs.

7.4 Suggestions for improvement

In the Hash-Once approach, the most frequently used operations (compare Table 1, Table 4 and Table 7) are also called T instead of $T * WPT$ times on each PU, as long as at least WPT number of PUs are working concurrently. So the suggestions for improvement for the Hash-Once approach are similar to the ones for the Round-Robin approach (Section 6.3). We need to change the structure of the algorithm and also the communication pattern in a way that not each tuple or each result set has to visit each PU.

8 The Round-Robin-Cyclic approach for streaming aggregation

In this chapter, we present another approach for a disjoint parallelization of sliding-window streaming aggregation. We call this algorithm the Round-Robin-Cyclic approach (for short RRCA). The Round-Robin-Cyclic approach is a combination of elements of the Round-Robin approach and the Hash-Once approach to exploit advantages of both.

First (Section 8.1), we discuss the structure of the Round-Robin-Cyclic approach. We present its pseudocode and explain why it exhibits deterministic output behavior.

Next (Section 8.2), we analyze its runtime properties. We analyze the call frequency of the different parts of our algorithm as well as interdependencies between the executed aggregation and the call frequencies and costs of the algorithm's parts.

Then (Section 8.3) we discuss the determined runtime properties of the algorithm. We compare them with the results of the previous approaches (HSA, RRA and HOA).

Finally, (Section 8.4) we consider suggestions for improvements.

8.1 Structure

The Round-Robin-Cyclic approach is basically a combination of the Round-Robin approach and the Hash-Once approach.

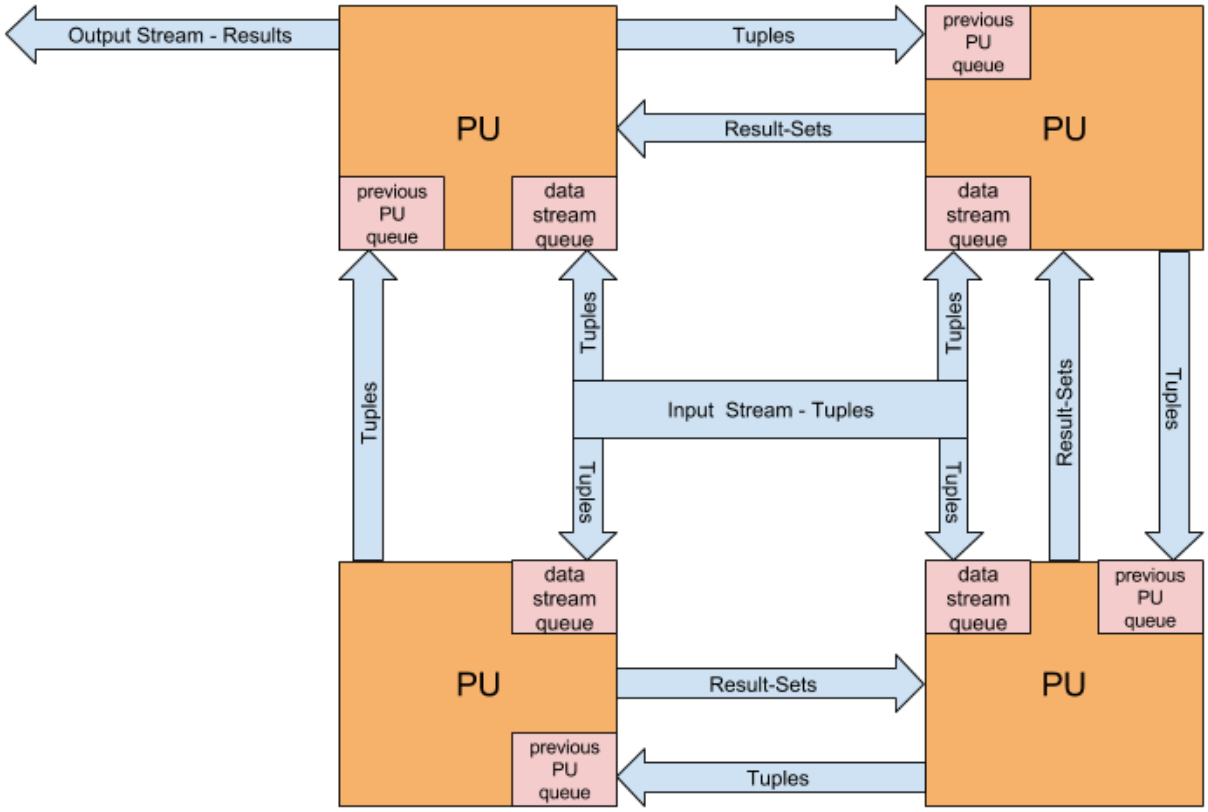


Figure 14: Structure of the Round-Robin-Cyclic approach with 4 PUs.

It takes the window distribution pattern of the Round-Robin approach and the communication structure and synchronization rules of the Hash-Once approach.
So as the Hash-Once approach, it breaks partially with the neighbor-to-neighbor-limited

communication structure by allowing certain further communication links. It uses the 4 presented tuple forwarding rules to synchronize the PUs' input streams and guarantee deterministic output behavior.
But unlike the Hash-Once approach, it does not add further window responsibility information to the tuple, since the responsibilities are still resulting from the Round-Robin schema. The only additional information a tuple needs is the knowledge about the PU on which it entered the processing system.

The Round-Robin-Cyclic approach is meant to grant the Round-Robin approach the advantages that come along with the cyclic communication structure of the Hash-Once approach at the cost of some of the downsides of the Hash-Once approach.

8.1.1 Pseudocode

The corresponding pseudocode representing this approach's structure is shown in Pseudocode 5. It is being executed by each PU.

The pseudocode is very similar to the one of the Hash-Once approach but the Prepare & Execute Hash Operation & Store Results in Tuple part is dropped. Instead we identify the window for a given tuple by calculating the 6-7 arithmetic operations presented in Section 6.1.1 and Pseudocode 3.

```
1.  while (true)
2.      if (datastreamQueue.isFilled() or prevPUQueue.isFilled())
3.          Tuple tupleToProcess = null
4.          if (datastreamQueue.isEmpty() and (Tuple t = prevPUQueue.first) != null)
5.          else
6.              if ((Tuple t = datastreamQueue.first) != null and prevPUQueue.isEmpty())

7.                  tupleToProcess = t
8.              else
9.                  if ((Tuple t1 = datastreamQueue.first) != null and (Tuple t2 = prevP
    UQueue.first) != null)
10.                     tupleToProcess = t1.timestamp < t2.timestamp ? t1 : t2
11.
12.         if (tupleToProcess != null)
13.             if (tupleToProcess.lastPUVisited != me)
14.                 Get&ForwardTuple(tupleToProcess)
15.             else
16.                 mySafeTimestamp = tupleToProcess.timestamp
17.             for (each window w, tuple tupleToProcess contributes to that I have to c
    ompute)
18.                 FindOrCreateWindow(w)
19.                 AggregateOperation(w,tupleToProcess)
20.             if (myRank == PUCount - 1)
21.                 CreateNewResultSetIfNecesarry(tupleToProcess)
22.
23.     if ((ResultSet r = incomingResultSets.first) != null and mySafeTimestamp >= r.ti
    meLimit)
24.         AddMyExpiredWindowsToResultSet(r)
25.         ForwardOrOutputResultSet(r)
```

Pseudocode 5: Pseudocode of the Round-Robin-Cyclic approach. It is executed by each PU.

A PU p continuously (line 1) checks, whether there are tuples in one or both of the tuple queues (line 2) or result sets (line 23).

If there are tuples in at least one of both incoming tuple queues (the data stream queue or the previous PU queue), p applies the 4 forwarding rules as described in Section 7.1 of the Hash-Once approach. Now, if a tuple was chosen (line 12), p forwards it to the next PU (line 14) unless p is the last PU this

tuple has to visit (line 13). Otherwise p can update its safe-timestamp (line 16).
Next p processes the tuple. It determines all windows to which this tuple contributes that p has to compute by calculating the arithmetic operations presented in Section 6.1 (line 17).
So if p is responsible to process a window w, it checks, whether it has already stored w and started processing it and if not it stores w. Then PU p adds tuple t to window w (line 18).
In case w's aggregate operation is dynamically computable (Section 2.4), we can execute the next step of the aggregate operation to create a new intermediate result (line 19).
Next, the last PU (line 20) has to check, whether any windows have expired and if so, it creates a new result set (line 21).

If there arrived a new result set r and p's safe-timestamp is greater than or equal to r's time limit (line 23), PU p adds all of its expired windows to r (line 24). Then it forwards r to its left neighbor (line 25). If p is the PU with index 0, it instead puts the results of the expired windows contained in r into the output stream.

8.1.2 Determinism of the output stream's results

Nothing has changed to the communication pattern compared to the Hash-Once approach introduced in the previous chapter. The result sets being sent from the last PU in the line still collect expired windows up to a certain time limit and order them by their expiration time.
The 4 forwarding rules that are granting us the safe-timestamp ensure that all windows w which will be added to a result set r are not added to r before all tuples that might contribute to these windows have arrived.
So all information concerning the determinism of this approach can be found in Section 7.1.2.

8.2 Analysis of the algorithm's runtime properties

Similar to Section 5.2, we want to analyze the runtime properties of the algorithm and its parts.

8.2.1 Definitions and denotations

The definitions and denotations of this approach are identical to the ones of the Handshake approach (Section 5.2.1).

8.2.2 Call frequency

The call frequencies are exactly the same as in the Round-Robin approach. The justifications can be read in Section 5.2.2 and 6.2.2.

Program part	Number of executions (average) per PU
Get & Forward Tuple	T
Find or Create Window	$\frac{T * WPT}{P}$
Aggregate Operation	$\frac{T * WPT}{P}$
Create New Result Set If Necessary	T (only last PU in ring)
Add My Expired Windows to Result Set	W ($W \leq T$)
Forward Result Set	W ($W \leq T$)

Table 10: Call frequency table of the RRCA's program parts

The program parts that are limiting the execution speed and parallelization potential are equal to the ones in the Round-Robin approach. For more information and a short discussion, we refer to Section 6.2.2.

	Get & Forward Tuple	Find or Create Window	Aggregate Operation	Create New Result Set if Necessary	Add My Expired Windows to Result Set	Forward Result Set
Increase of Thread Count						
Increase of Aggregate complexity						
Increase of Window Size						
Decrease of Window Advance						
Increase of Aggregate Operation Count						
Activation of Group by						
Color explanation Change to part's call frequency	Locally more often globally more often		Locally unchanged globally more often		Locally unchanged globally unchanged	Locally less often globally unchanged

Table 11: Interdependencies between the aggregation parameters and the call frequency of the RRCA's parts

8.2.3 Call costs

The call costs are equal to the RRA. So the parameter interdependencies table looks similar to the one of the RRA and the justifications can be read in Section 5.2.3 and 6.2.3.

	Get & Forward Tuple	Find or Create Window	Aggregate Operation	Create New Result Set if Necessary	Add My Expired Windows to Result Set	Forward Result Set
Increase of Thread Count						
Increase of Aggregate complexity						
Increase of Window Size						
Decrease of Window Advance						(1)
Increase of Aggregate Operation Count						
Activation of Group by						(1)
Color explanation Change to part's costs	Higher		Unchanged (1): Red in case forwarded by value		Lower	

Table 12: Interdependencies between the aggregation parameters and the costs of the RRCA's parts

8.2.4 Memory space requirements

As for the HOA (see Section 7.2.4), we expect memory requirements similar to the ones of the HSA (see Section 5.2.4). They can be slightly higher due to the usage of two tuple queues instead of one and the higher latency (see the next section, Section 7.2.5), which can lead to larger numbers of expired windows that are simultaneously stored in the processing system waiting to be output.

8.2.5 Latency

The latency behavior of the RRCA is akin to the one of the HOA (see Section 7.2.5). We expect the approaches with a cyclic communication structure (the HOA and the RRCA) to perform worse than the approaches with a PU line, since the four tuple forwarding rules and the maintenance of the safe-timestamp delays the output of created result sets.

8.3 Discussion of the approach's runtime properties

The Round-Robin-Cyclic approach still has 4 different parts that are not or not entirely parallelizable.

Comparison of the throughput with the previous approaches

We compare the expected performance of the Round-Robin-Cyclic approach with the previous approaches.

Because of the absence of hash functions to determine window responsibilities, both the Round-Robin approach and the Round-Robin-Cyclic approach have advantages over the hash-based approaches (HSA and HOA). As long as the 3 aggregation parameters mentioned in Section 6.2.4 do not apply, we expect the RRCA as well as the RRA to be significantly faster than the HSA and slightly faster than the HOA.

Advantages of the structural changes

As explained in the previous chapter, the cyclic communication structure grants us two advantages over the non-cyclic approaches (a more detailed discussion can be found in Section 7.2.4):

- It gives us the opportunity to use multiple input streams at the same time.
- The RRCA can be more resilient to short variations of the participating PUs' speed, which may lead to a higher throughput rate compared to the RRA due to higher processor utilization.

Disadvantages of the structural changes

Of the four disadvantages introduced by the cyclic communication structure, one becomes obsolete: Since we do not store additional information within tuples, there is not more network traffic than in the Handshake and Round-Robin approach, when using the RRCA on distributed systems (see Section 7.2.4). However, two possible disadvantages remain:

- The connections are not limited to neighbor-to-neighbor communication. Hence in case we want to use this approach on distributed systems, we need to maintain twice as many connections.
- If a PU does not receive incoming tuples for a longer period of time, its neighbor in counterclockwise direction cannot process and forward tuples and result sets and update its save-timestamp.
- As discussed in Section 7.1, without further effort it is not possible to dynamically compute order-sensitive aggregate operations, since in general, tuples are not arriving in timestamp order.

8.4 Suggestions for improvement

The suggestions for improvement of this approach are identical to the ones for the Round-Robin approach and the Hash-Once approach (see Section 6.4 and Section 7.4). To accelerate the algorithm, we need to handle tuples and result sets in a way such that each tuple and each result set do not have to visit each PU.

9 The Round-Robin-Cyclic Pane-based approach for streaming aggregation

In this chapter, we present another approach for a disjoint parallelization of sliding-window streaming aggregation. We call this algorithm the Round-Robin-Cyclic Pane-based approach (for short RRCPA). The Round-Robin-Cyclic Pane-based approach rests upon the Round-Robin-Cyclic approach but introduces a new object type – the pane – to improve its performance.

First (Section 9.1), we discuss the structure of the Round-Robin-Cyclic Pane-based approach. We present its pseudocode and explain why it exhibits deterministic output behavior.

Next (Section 9.2), we analyze its runtime properties. We analyze the call frequency of the different parts of our algorithm as well as interdependencies between the executed aggregation and the call frequencies and costs of the algorithm's parts.

Then (Section 9.3) we discuss the determined runtime properties of the algorithm. We compare them with the results of the previous approaches (HSA, RRA, HOA and RRCPA).

Finally, (Section 9.4) we consider suggestions for improvements.

9.1 Structure

The Round-Robin-Cyclic Pane-based approach (also referred to as *RRCPA* or *pane-based approach*) rests upon an idea presented in this article [4]. It breaks with the last four approaches' principle that each tuple has to visit each PU and is under certain circumstances able to outperform the previous approaches by several orders of magnitude.

Redundant aggregations in the previous approaches

Let us imagine the following scenario: We have one input stream and want to execute an aggregation on it, for example we want to calculate the sum. We let the window size be one hour and the window advance one minute and we do not group tuples by their key.

All tuples with timestamps of the same minute, for example all tuples with timestamps between 70:00 minutes and 70:59 minutes, contribute to the same 60 windows. In our example all those tuples contribute to the window from 11:00 minutes to 70:59 minutes, to the window from 12:00 minutes to 71:59 minutes, to the window from 13:00 minutes to 72:59 minutes and so on. The last window to which those tuples contribute is the one from 70:00 minutes to 129:59 minutes.

All those 60 windows sum up the values of the respective tuples they contain, which also means that the tuples with timestamps between 70:00 minutes and 70:59 minutes are summed up 60 times. Theoretically, it would be sufficient to sum up the values of the tuples with timestamps between 70:00 minutes and 70:59 minutes once and reuse the sum whenever it is needed, so altogether 60 times. This would reduce the number of aggregation steps by the factor 60, so about 98,3%. And this is what the new approach is doing.

Panes and their properties

The new approach avoids the repeated calculations of the same operations.

The sum of every single minute is calculated only once and on one single PU.

We call one of those one-minute lasting windows *pane*.

A pane is a usual window; the only difference to a regular window is that its time span **PNS** (**p**a**n**e **s**ize) is not WS number of time units (60 minutes), but WA time units (1 minute). Its advance **PNA** (**p**a**n**e

advance) is equal to WA as well. So one window consists of **p**anes **p**er **w**indow $PPW = \frac{WS}{WA}$ number of panes. We assume that WS is divisible by WA without a remainder. Since PNS=WA, panes are always non-overlapping, so each tuple is just contributing to one single pane.

So in our example, a window consists of $PPW = \frac{WS}{WA} = \frac{60\ minutes}{1\ minute} = 60\ panes$.

The number of windows to which one pane contributes (**WPP**, **w**indows **p**er **p**ane) is always equal to PPW.

Within our new approach the results for each 60 minutes lasting window are aggregated from the results of the respective 60 one-minute lasting panes.

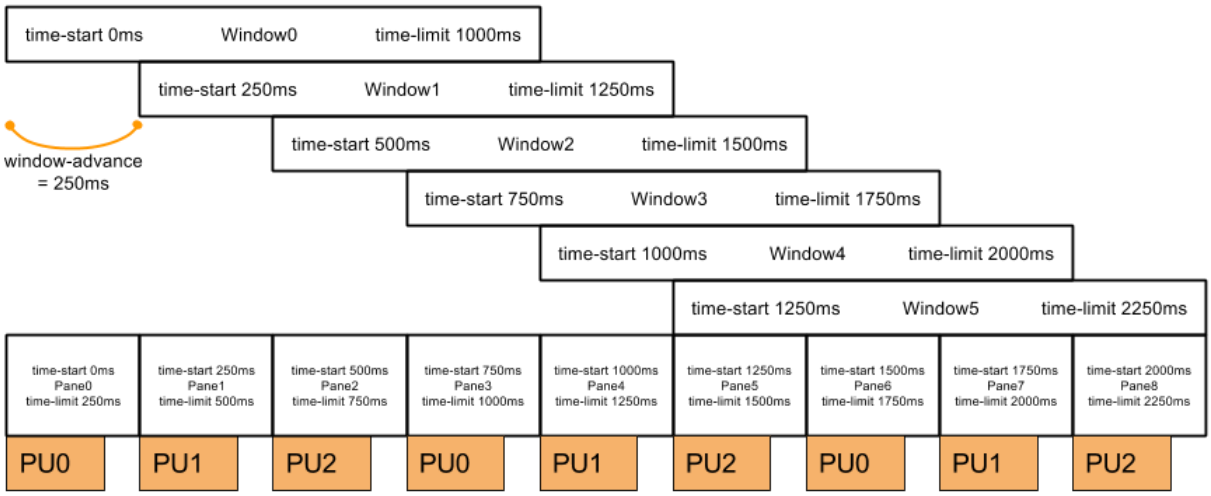


Figure 15: Panes and windows; WS=1000ms, WA=250ms

In the appendix in Section 17.4 there is a figure depicting a concrete example for streaming aggregation explaining the differences between the non-pane-based approaches and the pane-based one.

Pane responsibilities

The responsibilities for the panes are distributed in Round-Robin-fashion among the participating PUs, as we can see in Figure 15. The aggregation for all tuples contributing to the first pane (tuples with timestamps between 0 and 250 time units) is computed by the first PU (PU0), the aggregation for all tuples contributing to the second pane (tuples with timestamps between 250 and 500) is handled by the second PU (PU1) and so on.

Since the panes are non-overlapping and each tuple contributes to only one single pane, tuples do not need to visit all PUs. For each tuple it is sufficient to visit one single PU.

Unfortunately, without further effort, the input stream does not know to which PU it has to send a given tuple t. To find out which PU is responsible for t, it needs to calculate $responsiblePU = \frac{t.timestamp\ \%\ (P*WA)}{WA}$, so $responsiblePU = \frac{t.timestamp\ \%\ 750}{250}$ in the example in Figure 15.

This means that in each 750 time units lasting interval, PU0 is responsible for all tuples with timestamps between 0 and 249, PU1 is responsible for all tuples with timestamps between 250 and 499 and PU2 is responsible for all tuples with timestamps between 500 and 749.

So in this approach, the input stream (or in other words the machine that is sending its tuples) has to execute additional computations to determine the responsible PU for a given tuple. Depending on the sending machine's speed and the rate tuples shall be sent, the sending machine might bottleneck the system. The sending machine has quasi centralized coordination tasks here and in this point the RRCPA

infringes the aim of a fully disjoint parallelization. However, there are other articles that deal with the parallelization of input streams as well (see Chapter 12 – Related Work).

Pane expiration

Each pane expires under the same circumstances, like regular windows do. In Figure 15, Pane0 on PU0 expires, as soon as PU0 receives a tuple with a higher timestamp than Pane0's time limit. Since PU0 does not receive any tuples for Pane1 or Pane2, the lowest possible timestamp of an incoming tuple that causes the expiration of Pane0 is 1000, which contributes to Pane3 on PU0.

In general, when using n PUs to process our program, each PU p_i is responsible for $pane_i$, $pane_{i+n}$, $pane_{i+2n}$, $pane_{i+3n}$ and so on. So between two consecutive panes of one PU there are *n-1* other panes. Thus the more PUs we are using, the more panes are between two consecutive panes of one PU and hence the longer it takes, until one PU receives the first tuple of its next pane that leads to the expiration of its current pane. So an increasing amount of PUs leads to an ascending latency that panes need to expire, independent of the rate tuples are incoming or the processor utilization.

If for example we execute an aggregation with a window size of 60 minutes and a window advance of 1 minute and use 20 PUs to work on the problem, we have a latency of 20 minutes until the first pane expires.

It would be possible to send dummy tuples or messages to inform other PUs about arrived tuples in order to shorten the delay after which windows expire. But we have to bear in mind that in case WA≈ATD, the number of required dummy tuples is as high as the number of tuples itself, causing much additional work. So using additional messages is basically a trade-off between throughput and latency.

Window responsibilities

The responsibilities of windows are also distributed in round-robin fashion: PU0 is responsible for Window0, Window3, Window6, …, PU1 is responsible for Window1, Window4, Window7, …, and PU2 is responsible for Window2, Window5, Window8, … (see Figure 15).

Aggregating panes

In the previous approaches, to calculate the final aggregation result of a window, we aggregated all the tuples contributing to the window.

In the pane-based approach, to calculate the final result of a window, we aggregate the aggregation results of all panes of which this window consists. Taking Figure 15 as an example, to compute the aggregation result of Window0, we need to aggregate the aggregation results of Pane0, Pane1, Pane2 and Pane3, to compute the aggregation result of Window1, we need to aggregate the aggregation results of Pane1, Pane2, Pane3 and Pane4 and so on.

In general, a PU has not access to all panes that it needs to compute the aggregation result of a window. Taking Figure 15 as example, PU0 needs access to Pane0, Pane1, Pane2 and Pane3 to calculate the aggregation result of Window0. Since PU0 is only responsible for Pane0, Pane3, Pane6 and so on, it needs access to panes that have been computed on other PUs, in our example Pane1 and Pane2.

Transferring expired panes

So in the pane-based approach panes (and not tuples) are being forwarded PU by PU, until every PU that needs access to it has received and processed it.

For the pane transport we have to distinguish two cases:

- If $PPW \geq P$, so if one window consists of at least as many consecutive panes as there are PUs, each pane needs to visit each PU.

Taking Pane4 of Figure 15 as example, we see that Pane4 contributes to Window4, Window3, Window2 and Window1. As we know, consecutive windows are being handled by consecutive PUs, so if there are more windows to which each pane contributes (WPP=PPW=4 in our example) than PUs (P=3 in our example), a pane has to visit all PUs.

- If $PPW < P$, so if one window consists of fewer consecutive panes than there are PUs, each pane does not need to visit each PU.
  Let us assume, the example of Figure 15 is processed by 100 instead of 3 PUs. Taking Pane4 as example, we see that Pane4 contributes to Window4, Window3, Window2 and Window1. Window4 is being processed by PU4, Window3 is being processed by PU3, Window2 is being processed by PU2 and Window1 is being processed by PU1. So we forward Pane4 into counterclockwise direction to PU3, PU2 and PU1.
  There is no reason to forward Pane4 further into counterclockwise direction to PU0 (which is processing Window0, Window100, Window200, …), PU99 (which is processing Window99, Window 199, Window 299, …) because as mentioned Pane4 only contributes to Window4, Window3, Window2 and Window1.

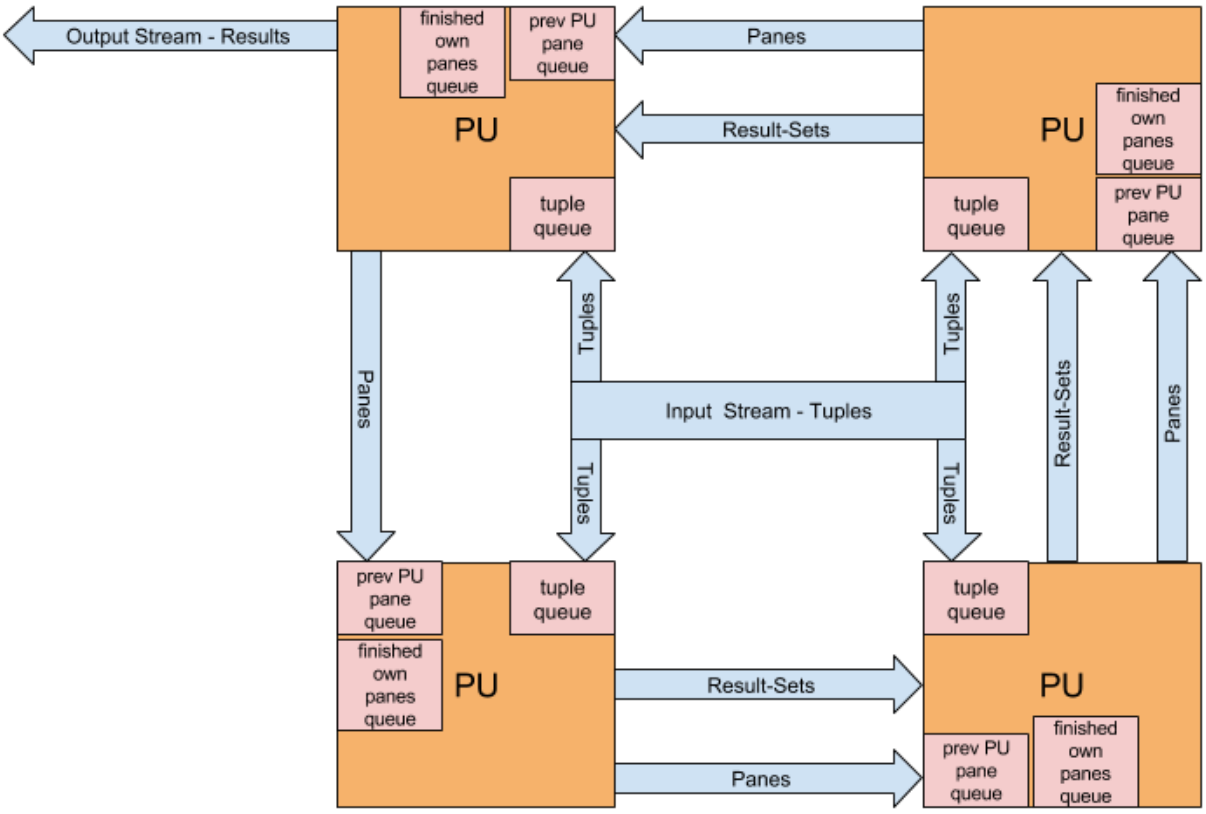


Figure 16: Structure of the Round-Robin-Cyclic Pane-based approach with 4 PUs

Pane queues and forwarding rules

Similar to the two different queues for tuples in the previous approaches, each PU p has two queues for panes: one for panes processed on p itself and one for panes received by the neighbor in clockwise direction.

The rules for forwarding and processing panes in the queues are equal to the rules for forwarding and processing tuples of the non-pane-based approaches, just with the difference that the forwarding direction is counterclockwise, not clockwise. This way PUs can maintain a safe-timestamp like in the non-pane-based approaches, which is necessary to know, when all panes that might contribute to a

window have arrived.
In the non-pane-based approaches PUs can update the safe-timestamp whenever a tuple from the right neighbor (neighbor in clockwise direction) arrives because those tuples went through the whole PU ring and ensure that no further tuples with lower timestamps will arrive. Afterwards the PU can delete this tuple.
In our pane-based approach, there are two different cases:

- If $PPW \geq P$, then each pane has to visit all PUs. Since the forward direction is counterclockwise, PUs may update their safe-timestamp, whenever a pane from the left neighbor (neighbor in counterclockwise direction) arrives.
- If $PPW < P$, then each pane has to visit PPW PUs only as explained earlier in this section. Hence each PU may update its safe-timestamp, whenever a pane from the PU PPW-1 steps in clockwise direction arrives.

Result sets
There is still only one PU that is outputting results into the output stream whenever a result set arrives. We call it PU p_0. The indices of the other PUs are rising in clockwise order around the PU ring.
Result sets are handled a bit differently to the non-pane-based approaches. The last PU p_{n-1} (in Figure 16 the PU in the bottom left corner) creates a new result set, whenever a tuple arrives that updates its safe-timestamp and contributes to a new pane.
Like in the previous approaches, all windows are added to a result set, whose time limit is smaller than or equal to the time limit of the result set. A window can only be added to a result set if it is already expired, so if the respective PU knows for sure that no more panes will arrive that contribute to it. Otherwise the PU has to wait for the windows to expire before forwarding the result set.

9.1.1 Pseudocode

The corresponding pseudocode representing this approach's structure is shown in Pseudocode 6. It is being executed by each PU.

The pseudocode has certain similarities to the cyclic approaches, hence the Hash-Once approach and the Round-Robin-Cyclic approach. The lines 6-24 in Pseudocode 6 are comparable to the lines 2-26 of Pseudocode 4 of the Hash-Once approach and the lines 2-21 of Pseudocode 5 of the Round-Robin-Cyclic approach. Instead of forwarding, choosing and aggregating tuples, we execute these steps with incoming panes.

A PU p continuously (line 1) checks, whether there are tuples incoming from the input stream (line 2), panes in one or both of the pane queues (line 6) or result sets (line 26).

If there arrived a tuple t (line 2), we find the pane which t contributes to (line 3). If there is no such pane or the current pane has expired, we push the expired pane in the *finishedOwnPanesQueue* and create a new pane (still part of line 3). Then we process the next aggregation step of the current pane (line 4).

If there are panes in at least one of both incoming pane queues, the *finishedOwnPanesQueue* or the *panesOfPrevPUQueue* (line 6), p applies the four forwarding rules, described in Section 7.1 of the Hash-Once approach, to the incoming panes (lines 7-14).
Now, if a pane was chosen (line 15), p forwards it to the next PU (line 17) unless p is the last PU this pane has to visit (line 16). Otherwise p can update its safe-timestamp (line 19).

Next, p processes the pane. It determines all windows to which this pane contributes that p has to compute (line 20). Determining those responsibilities is very cheap and consists of only few arithmetic operations similar to the ones used and present in Section 6.1 of the Round-Robin approach.
So if p is responsible to process a window w, it checks, whether it has already stored w and started processing it and if not it stores w. Then PU p adds tuple t to window w (line 21).
In case w's aggregate operation is dynamically computable (Section 2.4), we can execute the next step of the aggregate operation to create a new intermediate result (line 22).
Next, the last PU (line 23) has to check, whether its safe-timestamp was updated by the current pane and if so, it creates a new result set (line 24).

```
1.  while (true)
2.      if ((Tuple t = incTupleQueue.first) != null)
3.          Pane p = FindOrCreatePane&PushExpiredPanesIntoOwnQueue(t)
4.          AggregateOperation(p,t)
5.
6.      if (finishedOwnPanesQueue.isFilled() or panesOfPrevPUQueue.isFilled())
7.          Pane paneToProcess = null
8.          if (finishedOwnPanesQueue.isEmpty() and (Pane p = panesOfPrevPUQueue.first)
    != null)
9.          else
10.             if ((Pane p = finishedOwnPanesQueue.first) != null and panesOfPrevPUQueu
    e.isEmpty())
11.                 paneToProcess = p
12.             else
13.                 if ((Pane p1 = finishedOwnPanesQueue.first) != null and (Pane p2 = p
    anesOfPrevPUQueue.first) != null)
14.                     paneToProcess = p1.windowTimeLimit<p2.windowTimeLimit ? p1 : p2

15.         if (paneToProcess != null)
16.             if (paneToProcess.lastPUVisited != me)
17.                 Get&ForwardPane(paneToProcess)
18.             else
19.                 mySafeTimestamp = paneToProcess.windowTimeLimit
20.             for (each window w, pane paneToProcess contributes to that I have to com
    pute)
21.                 FindOrCreateWindow(w)
22.                 AggregateOperation(w,paneToProcess)
23.             if (myRank == PUCount - 1)
24.                 CreateNewResultSetIfNecesarry(paneToProcess)
25.
26.     if ((ResultSet r = incResultSetQueue.first) != null and mySafeTimestamp >= r.tim
    eLimit)
27.         AddMyExpiredWindowsToResultSet(r)
28.         ForwardOrOutputResultSet(r)
```

Pseudocode 6: Pseudocode of the Round-Robin-Cyclic Pane-based approach. It is executed by each PU

If there arrived a new result set r and p's safe-timestamp is greater than or equal to r's time limit (line 26), PU p adds all of its expired windows to r (line 27). Then it forwards r to its left neighbor (line 28). If p is the PU with index 0, it instead puts the results of the expired windows contained in r into the output stream.

9.1.2 Determinism of the output stream's results

The communication pattern has changed compared to the approaches introduced in the previous chapters.
As discussed in Section 9.1, the difference of the RRCPA to the non-pane-based approaches is that the windows are computed differently: In the RRCPA, they are calculated by aggregating panes, which are

calculated by aggregating tuples. In the other approaches, they are calculated by aggregating tuples (without the intermediate step using panes).

If we ignore tuples and panes for a moment and just look at windows and result sets, the RRCPA is completely similar to the RRCA. Consecutive windows are handled by consecutive PUs in a round-robin fashion. Result sets are being created by the last PU in the ring, forwarded PU by PU in counterclockwise direction, collecting all windows on their way with a time limit which is smaller than or equal to their own time limit (waiting for all unexpired windows among them to expire, with the aid of the safe-timestamp) until they reach the first PU in the ring, which outputs the results. Since the output behavior of the RRCA and the HOA (see sections 7.1.2 and 8.1.2) was deterministic, the RRCPA's output behavior has to be deterministic too if we assume correct computations of the windows themselves.

Observation 1: The result set management and thus the output behavior is still deterministic if the windows themselves are processed correctly.

Each pane is either visiting each PU or WPP number of PUs (whichever is smaller, see Section 9.1), to contribute to all WPP number of windows the pane belongs to. Windows are not leaving PUs prematurely due to the safe-timestamp and our pane forwarding rules, so it is guaranteed that each pane will always contribute to all WPP number of windows. So windows are processed correctly as long as the panes they consist of are processed correctly.

Observation 2: The window computations are correct if the panes themselves are processed correctly.

Each pane p remains as long at a PU until an arriving tuple leads to p's expiration.
Thus if tuples are arriving at each PU in timestamp order, it is guaranteed that no more tuples will arrive to contribute to p after p's expiration.

Observation 3: The pane computations are correct if tuples are arriving in timestamp order at each PU.

It is our basic assumption that tuples are arriving in timestamp order at each PU. It follows from this that panes are processed correctly (observation 3). Therefore, we can deduce that windows are processed correctly (observation 2). From this we infer that the result set management and thus the output behavior is deterministic (observation 1).

9.2 Analysis of the algorithm's runtime properties

Similar to Section 5.2, we want to analyze the runtime properties of the algorithm and its parts.

9.2.1 Definitions and denotations

We are using the same definitions and denotations as the ones of the Handshake approach in Section 5.2.1. The definitions and properties for **T**, **ATD**, **WS**, **WA**, **WPT**, **W**, **DK**, and **P** were given and discussed in this section.

In addition, we need to introduce a few further definitions.
In Section 9.1 we established two new definitions:
PPW (**p**anes **p**er **w**indow): the amount of panes of which each window consists. Since each window has a size of WS and all panes have a size and advance of WA, $PPW = \frac{WS}{WA}$.
WPP (**w**indows **p**er **p**ane): the amount of windows to which each pane contributes. Since each pane

has a size of WA and windows with a size of WS are sliding with intervals of the size WA, $WPP = \frac{WS}{WA}$ and hence *PPW=WPP*.
Taking Figure 15 as an example, we can see that all panes that are not in the border areas of the stream contribute to $WPP = \frac{WS}{WA} = 4$ panes. Furthermore, each window consists of $PPW = \frac{WS}{WA} = 4$ panes.

Assuming that $ATD \leq WA$, the number of panes **PN** is approximately equal to the number of windows W because both panes and windows are sliding with an advance of WA. However, we have a few additional panes because the first window consists already of $PPW = \frac{WS}{WA}$ number of panes. Hence $PN = W + PPW - 1$. So as long as $PPW << W$, which might be the case for most practical applications, $PN \approx W$ is a good approximation.

9.2.2 Call frequency

Now we want to analyze how often a PU p has to execute the different parts (mentioned in Pseudocode 6 and Section 9.1.1) of the algorithm.

- Find or Create Pane & Push Expired Panes into Own Queue: This part is responsible to find or create the appropriate pane for incoming tuples. In case an incoming tuple leads to the expiration of p's panes, it pushes them into the finished-own-panes-queue. Every tuple of the stream visits one PU to contribute to a pane. So in average we expect p to call this part $\frac{T}{P}$ times.
- Aggregate Operation (tuples to pane aggregation): After finding the appropriate pane for an incoming tuple, the tuple takes part in the pane's aggregation. So this part is also called $\frac{T}{P}$ times by p.
- Get & Forward Pane: There are PN panes.
  In case $PPW \geq P$, each pane has to visit each PU, which means that this part is called PN times by p.
  In case $PPW < P$, each pane visits PPW PUs, so p calls this part $PN * \frac{PPW}{P}$ times.
  Hence we can combine both terms in the following one: $min\left(PN, PN * \frac{PPW}{P}\right)$
- Find or Create Window: Each of the PN panes contributes to WPP windows, so all in all there are $WPP * PN$ events where a tuple contributes to a window. However, the windows are distributed evenly among the P participating PUs in Round-Robin-fashion. So on average, p executes the Find or Create Window operation $\frac{WPP*PN}{P}$ times.
- Aggregate Operation (panes to window aggregation): After finding the appropriate window for an incoming pane using the Find or Create Window part, the pane is aggregated with the window. So, just as the Find or Create Window part, p executes this part $\frac{WPP*PN}{P}$ times.
- Create New Result Set If Necessary: Whenever the last PU p receives a pane that updates its safe-timestamp, p creates a new result set. According to the Get & Forward Pane part, $min\left(PN, \frac{PN*PPW}{P}\right)$ panes will arrive at each PU. However, since all panes are distributed evenly among the PUs, only every P-th of all PN panes will have the last PU as its final destination, which is necessary to update its safe-timestamp.
  So this part is called $\frac{PN}{P}$ times and only by the last PU.

- Add My Expired Windows to Result Set: This part is executed on p, whenever a result set reaches p. As presented in the Create New Result Set If Necessary part, we expect up to $\frac{PN}{P}$ result sets to be created. Each of them has to visit each PU, so this part is called $\frac{PN}{P}$ times by p.
- Forward Result Set: Just like the Add My Expired Windows to Result Set part, we expect that p is executing the Forward Result Set part $\frac{PN}{P}$ times.

Program part	Avg. number of executions per PU
Find or Create Pane & Push Expired Panes into Own Queue	$\frac{T}{P}$
Aggregate Operation (tuples to pane aggregation)	$\frac{T}{P}$
Get & Forward Pane	$min\left(PN, \frac{PN * PPW}{P}\right)$
Find or Create Window	$\frac{PN * WPP}{P}$
Aggregate Operation (panes to window aggregation)	$\frac{PN * WPP}{P}$
Create New Result Set If Necessary	$\frac{PN}{P}$ (only last PU in ring)
Add My Expired Windows to Result Set	$\frac{PN}{P}$
Forward Result Set	$\frac{PN}{P}$

Table 13: Call frequency table of the RRCPA's program parts

As we see, all program parts are parallelizable in the Round-Robin-Cyclic Pane-based approach. The call frequency of all parts is inversely proportional to the number of participating PUs.

According to Table 13, we create another table (Table 14) depicting the interdependencies between the aggregation parameters and the parts' call frequencies.

	Find or Create Pane & Push Expired Panes into Own Queue	Aggregate Operation (tuples to pane)	Get & Forward Pane	Find or Create Window	Aggregate Operation (panes to window)	Create New Result Set if Necessary	Add My Expired Windows to Result Set	Forward Result Set
Increase of Thread Count			(1)					
Increase of Aggregate complexity								
Increase of Window Size			(2)					
Decrease of Window Advance								
Activation of Group by								
Color explanation Change to part's call frequency	Locally more often globally more often (2): yellow in case P<PPW		Locally unchanged globally more often (1): green in case P>WWP		Locally unchanged globally unchanged		Locally less often globally unchanged	

Table 14: Interdependencies between the aggregation parameters and the call frequency of the RRCPA's parts

Increase of the thread count

As mentioned in the previous section, all parts are either in general or at least under certain circumstances scaling with an increasing number of PUs.

If before and after an increase of the thread count *PPW>P* still holds, the local call frequency of the Get & Forward pane part remains constant and hence the global call frequency rises. This is because under this circumstance each pane has to visit each PU anyway and if there are more PUs, the Get & Forward Pane part has to be called more frequently globally.

The amount of created result sets is depending on the amount of updates of the safe-timestamp on the last PU because the last PU creates a new result set, whenever its safe-timestamp is updated. An update can take place whenever a pane arrives, whose last destination is the last PU. An increase of the number of PUs decreases the fraction of panes whose final destination is the last PU. Hence the amount of created result sets decreases as the number of participating PUs rises.
The output delay also increases as we increase the number of participating PUs because fewer result sets will be created, while the overall amount of windows remains constant. So each result set transports more windows of an increasing number of window time-slots.

The decreasing number of result sets negates the effect of a globally increasing number of calls of the two parts Add My Expired Windows to Result Set and Forward Result Set per result set, caused by the increasing number of PUs each result set has to visit. Therefore, the global call frequency remains constant while the local call frequency decreases.

Increase of the aggregation's complexity
There is no effect on the call frequency of the parts.

Increase of the window size
Increasing the window size also increases the number of panes per window as well as the number of windows per pane. As Table 13 depicts and Section 9.2.2 discusses, this increases the call frequency of the Find or Create Window part and the Aggregate Operation part. If *PPW<P*, the call frequency of the Get & Forward Pane part will also increase, since each pane has to visit an increasing amount of PUs. If *PPW≥P*, each pane visits each PU anyway, so there is no change to that part's call frequency.

Decrease of the window advance
Decreasing the window advance does not only increase WPP and PPW as an increase of the window size; it also increases the number of panes. So basically the call frequency of all parts are affected by changes to the window advance. The parts Find or Create Window and Aggregate Operation – and under certain circumstances the Get & Forward Pane part as well – even exhibit a quadratic correlation, since they are affected linearly by two factors, PN and WPP (PN and PPW for the Get & Forward Pane part). For instance, a decrease of the window advance by 10 would increase the call frequency of these parts by $10^2=100$.

Increase of the Aggregate Operation count
In the four previous approaches, processing several aggregations on tuples of the same data stream separately has a disadvantage compared to processing them concurrently because in the latter case it is not necessary to send all tuples of the data stream once for each aggregation. Sending all tuples once was sufficient, since every tuple had to visit each PU anyway.
In the pane-based approach each tuple visits only one PU because it contributes to exactly one pane; and the window advance parameter determines which pane that is and where it is located. If we compute several aggregations at once, depending on the aggregation's window advance, in general the pane to which a tuple contributes is not located on the same PU, so the data stream would have to send the tuples for each aggregation again. Since there is no other advantage in computing several

aggregations within the same program instance, we can just run several instances of our whole algorithm independently at the same time.

Group tuples by their key

If we group tuples by their key, for each distinct key falling into one pane time-slot (the equivalent to a window time-slot), there will exist one pane collecting all tuples with this key. That increases the overall amount of panes and windows depending on the amount of distinct keys DK, which again increases the call frequency of the Get & Forward Pane part.

9.2.3 Call costs

Next, we create a table (Table 15) depicting the interdependencies between the aggregation parameters and the costs of a single call of each part.

	Find or Create Pane & Push Expired Panes into Own Queue	Aggregate Operation (tuples to pane)	Get & Forward Pane	Find or Create Window	Aggregate Operation (panes to window)	Create New Result Set if Necessary	Add My Expired Windows to Result Set	Forward Result Set
Increase of Thread Count								
Increase of Aggregate complexity								
Increase of Window Size								
Decrease of Window Advance								
Activation of Group by								(1)
Color explanation Change to part's call frequency	Higher			Unchanged (1): red in case forwarded by value (not reference)		Lower		

Table 15: Interdependencies between the aggregation parameters and the costs of the RRCPA's parts

Increase of the thread count

By increasing the number of threads the windows are distributed among more PUs. So each PU takes care of less windows which decreases the costs of the Find or Create Window part and the Add my Expired Windows to Result Set part.

Increase of the aggregation's complexity

As discussed in the previous section, within the pane-based approach there are two different aggregation parts: One aggregates tuples to panes, the other one aggregates panes to windows. Actually they are not working completely identically. There is a detail about aggregate operations in general that was not important for the previous approaches, but is important for the pane-based approach and to determine the influence of an increase of the aggregation's complexity, we need to take care of it. We have to distinguish between two different parts of an aggregation: the *tuple pre-processing* and the *aggregation* itself.

Let us assume that for each window we want to compute the sum of the squared values of its tuples.

The previous approaches (HSA, HOA, RRA, RRCA) proceeded like this: When a tuple t is incoming on a PU p and p is responsible for a window w to which t contributes, then the Aggregate Operation part squares t's value – which is the tuple pre-processing subpart – and adds the squared value to the

intermediate sum of w – which is the aggregation subpart. So each call of the Aggregate Operation part pre-processes tuple t (depending on the chosen aggregate operation, it is possible that the tuple pre-processing subpart is not doing anything at all) and aggregates it afterwards with window w. As presented in the tables 1, 4, 7 and 10, the previous approaches execute the Aggregate Operation part $\frac{T*WPT}{P}$ times per PU, so $T * WPT$ times altogether. Since as mentioned each Aggregate Operation part consists of a tuple pre-processing subpart and an aggregation subpart, both of them are called $T * WPT$ times altogether as well.

The Tuples to Pane Aggregate Operation part of the pane-based approach works similarly. The value of each arriving tuple t is being squared (the tuple pre-processing subpart) and then added to the intermediate sum of the corresponding pane p (the aggregation subpart). As presented in Table 13, the pane-based approach executes the Tuples to Pane Aggregate Operation part $\frac{T}{P}$ times per PU, so T times altogether. So the tuple pre-processing subpart and the aggregation subpart are called T times altogether as well.

However, the Panes to Window Aggregate Operation part of the pane-based approach operates differently. It just aggregates the intermediate results of tuples to a window and hence does not contain the tuple pre-processing subpart. As presented in Table 13, the pane-based approach executes the Panes to Window Aggregate Operation part $\frac{PN*WPP}{P}$ times per PU, so $PN * WPP$ times altogether. So this part calls the aggregation subpart $PN * WPP$ times altogether.

So overall the previous approaches call the tuple pre-processing subpart $T * WPT$ times and the aggregation subpart $T * WPT$ times.
The pane-based approach calls the tuple pre-processing subpart T times and the aggregation subpart $T + PN * WPP$ times.

Thus in summary, to determine the influence on the algorithm's runtime, we need to define what exactly is getting more expensive: the tuple pre-processing subpart or the aggregation subpart itself. While this question is less important for the previous approaches, since both subparts are called equally often, it matters for the pane-based approach.
Using a more complex tuple pre-processing function affects only the costs of the Tuples to Pane Aggregate Operation part. Using a more complex aggregation subpart affects the costs of both the Tuples to Pane Aggregate Operation part and the Panes to Window Aggregate Operation part.

Group tuples by their key
The Find or Create Pane & Push Expired Panes into own Queue part will become more expensive, since working with one pane at each point of time is less costly than organizing the access, storage and expiration of many panes existing simultaneously.

Similarly, the Find or Create Window part as well as the Add My Expired Windows to Result Set part are affected by the increasing amount of windows. Finding a window among an increasing amount of stored windows gets more expensive as well as adding a rising number of windows to each result set.

9.2.4 Memory space requirements
The memory space requirements differ from the demands of the previous approaches.
As in the HSA, we have to distinguish between two types of aggregate functions: Functions belonging to the first type do not need to save copies or references of tuples within windows because the

creation of intermediate results supersedes the necessity to store tuples. Aggregate functions belonging to the latter type do need to save copies or references of tuples within windows (see Section 5.2.4). The same applies to panes – which basically are sub-windows – as well. Hence for aggregate functions of the first type, panes have a constant size; it is independent of the number of tuples they contain and thus it is equal to the size of windows. When computing aggregate functions of the second type, panes have a size depending on the number of tuples they contain.
According to Section 9.1, we expect that there are approximately as many panes as there are windows (there are always at least $PPW - 1$ less windows than panes but on the long run this difference becomes insignificant).

First, we contemplate aggregate functions of the first type: Panes and windows have the same constant size. Since there are approximately as many panes as windows, the space needed by panes is approximately equal to the space needed by windows.
Next, we consider aggregate functions of the second type: Since panes are in most cases smaller than windows – or equal if *WS*=*WA* – they will demand less or the same amount of memory space as windows. To be accurate, we expect windows to consume WS/WA times the memory space of panes.

Due to the possibly extremely high latency (see the next section, Section 9.2.5), the number of expired windows can be much higher than in the previous approaches. Since the latency depends on many different factors, it is difficult to specify the exact number of expired windows. According to observations of Section 9.2.5, the number of expired windows can be three times higher than the number of active windows. We use this ratio as an approximation.
As in the HSA, there is one tuple queue, one queue for result sets and two queues for panes on each PU.

As in Section 5.2.4, we define that a tuple needs MT bytes of memory capacity, a result set needs MRS bytes, a window's intermediate result needs MIR bytes, a window or a pane (excluding intermediate results or stored tuples) needs MW bytes. Queues contain QS number of elements at the most.

So the maximum overall space requirements per PU that we expect excluding lower amounts of constant overhead costs are:
For aggregate functions of the first type: $\frac{5*(MW+MIR)*WPT*DK}{P} + QS * (MT + MRS + 2 * (MW + MIR))$ bytes.

For aggregate functions of the latter case: $\frac{(MW+\frac{MT*WS}{ATD})*WPT*DK}{P} + \frac{3*(MW+MIR)*WPT*DK}{P} + \frac{(MW+\frac{MT*WA}{ATD})*WPT*DK}{P} + QS * (MT + MRS + 2 * (MW + MIR))$ bytes.

Interdependencies between queue size and scalability
In certain cases, there is an additional factor concerning the size of the main memory that affects the scalability potential of the RRCPA. It is of importance in cases in which $ATD << WA$, so if a larger number of tuples contributes to each pane.
In these cases when using the pane-based approach, the input thread will send a large number of tuples to one pane on a PU p_i, then after WA time units a large number of tuples to the next pane on $p_{(i+1)\%P}$ (the neighbor PU in clockwise direction) and so on. Having a look at Table 13 (due to our assumption it holds that $PN << T$), the PU that currently receives tuples will have significantly more work than all other PUs because it additionally has to process many calls of the Tuples to Pane

Aggregate Operation part.
According to our definition of streaming aggregation, we want to avoid storing incoming data but process them while flying by instead. In our case using the RRCPA this can unfortunately lead at each point of time to up to one saturated PU and *P-1* PUs that exhibit a low utilization. So by using the pane-based approach, compared to the non-pane-based approaches, the overall amount of work may be reduced, but due to the low processor utilization, the eventual performance can still be worse in certain cases.

If the amount of tuples per pane is nearly or entirely constant among all panes and if we can afford storing larger amounts of tuples, we could alternatively proceed as follows:
Instead of sending only at a rate of r tuples per second that the receiving PU can manage without storing a lot of data, we allow the data stream to send tuples at a higher rate of up to $r * P$ tuples per second. So the input stream sends tuples up to P times faster than a receiving PU could process them, but since each PU receives tuples only during $\frac{1}{P}$ of the time, we expect a receiving PU p to need exactly as much time to compute all received data until the input thread will send tuples to p again. Hence the latency introduced at this point will be up to $WA * P$ time units (if the tuple sending rate is equal to $r * P$) because this is the amount of time a PU needs to process all received tuples for one pane. The crucial point here is that in order to send tuples faster than PUs can process them, we need enough main memory storage. More precisely, if we allow the PUs' incoming tuple queues to store at least $\frac{WA}{ATD}$ number of tuples, which is the number of tuples that contribute to each pane, the input stream can send all tuples that contribute to a pane to the responsible PU and then the stream can immediately continue sending tuples to the next pane on the consequent PU. If the PUs' tuple queues would be smaller, the input stream would repeatedly have to wait on PUs which may prevent an efficient parallelization of the tuple aggregation part.
If we assume no fluctuations of the ATD, each PU needs enough space to store at least $\frac{WA}{ATD}$ tuples in its tuple queue. Hence when using P number of PUs, the machine needs to have enough space to store $\frac{P*WA}{ATD}$ number of tuples. In cases in which we expect larger fluctuations of the tuples' timestamp distance around the ATD, we should consider to increase the size of the tuple queues even more.

In our evaluations later on, we will consider both cases: in the one there is enough memory to store all tuples contributing to one pane, in the other there is not. If we do not have sufficient memory the Tuples to Pane Aggregate Operation part will be computed serially and not in a parallelized manner.

9.2.5 Latency

After a PU p received the last tuple for a pane m, it takes at least $WA * (P - 1)$ time units, until PU p will receive another tuple that leads to the expiration of m. Afterwards m will be forwarded to up to *P-1* number of further PUs. According to the four forwarding rules, m can only be forwarded by a PU if this PU has an expired pane in its own-pane-queue, whose timestamp is higher than m's timestamp. Since the pane time-slot is shifted by WA time units per PU, it will take another $WA * (P - 1)$ time units until m reaches its final destination, where it updates the destination PU's safe-timestamp. Afterwards we still need to wait for the last PU to create a result set and for all PUs to have a safe-timestamp greater than or equal to the PU which is m's final destination.
If m's final destination is the last PU in the ring, p_{n-1}, then a result set will immediately be created that collects windows to which m contributed.
Else, if m's final destination is x PUs left of p_{n-1}, we need to wait another $x * WA$ time units ($1 \leq x \leq$

$n-1$), until the safe time stamp of p_{n-1} will be updated as well, which leads to the creation of a result set that collects windows to which m contributed.
So the overall latency (in time units) is at least: $2 * WA * (P-1) \leq latency \leq 3 * WA * (P-1)$. It may be even higher, depending on the time needed to merge panes to windows or possible imbalances between the PUs that lead to pane queues filled with a lot of elements causing additional delay.

As a demonstration we inspect the following example: We compute an arbitrary cheap aggregation (for example the sum) with an arbitrary window size, a window advance of 1 minute and use 24 PUs participating in the execution. Then the latency between a tuple leading to the expiration of a pane m and the first windows that contains m being output is at least 46-69 minutes, depending on the index of the destination PU of m.

Compared to the descriptions of the previous approaches' analysis sections, this is an extreme increase of the latency. Instead of a few milliseconds or in the worst case seconds, the latency of the pane-based approach bases almost entirely on the window advance and the number of participating PUs.

9.3 Discussion of the approach's runtime properties

To simplify the comparison of the approaches' performance, we again make a simplistic assumption: All algorithm parts of all approaches are equally expensive. So one call of one algorithm part (no matter of which approach) is as expensive as any other single call of any other part of any approach. Strictly speaking, this is still not true, but *under certain circumstances*, practical tests have shown that this assumption is a sufficient approximation (also see Chapter 5.3):

- The computing system transports tuples, panes and windows by reference, not by value; so this approximation does not apply for distributed systems.
- A "cheap" aggregation is used like the sum, the average or the minimum aggregation.
- Tuples are either not grouped by their key or if they are, the number of distinct keys DK has to be small (preferably single-digit, since according to the *Group tuples by their key* paragraph of Section 9.2.3 many distinct keys increase some parts' costs heavily).

Definitions

To classify the numbers of the call frequency table (Table 13) better, let us recall some definitions:

- $PPW = WPP = \frac{WS}{WA}$
  For explanations of the constants **p**anes **p**er **w**indow and **w**indows **p**er **p**ane see Section 9.1 and Section 9.2.1.
- $WPT = \frac{WS}{WA}$
  In the call frequency tables of the previous approaches, we use the constant **w**indows **p**er **t**uple, which is defined by the number of windows to which each tuples contributes.
- $W \leq T$
  Another assumption we made and discussed in detail in Section 5.2.1 was that there are always at least as many tuples as windows. Since *W≈PN* (see Section 9.1), we also assume *PN≤T*.

Call frequency comparison – pane-based vs. non-pane-based

Next, we compare the expected performance of the Round-Robin-Cyclic Pane-based approach with the previous approaches by comparing the call frequencies of the most frequently called parts and assuming equivalent costs for each part.

In three of the four previous approaches (RRA, RRCA, HOA), the most frequently called parts were called $\boldsymbol{max}\left(\boldsymbol{T}, \frac{\boldsymbol{T*WPT}}{\boldsymbol{P}}\right)$ times (see Tables 4, 7 and 10).
We compare this call frequency to our new approach (see Table 13, as mentioned, it holds that PPW=WPP=WPT) and consider two cases:

- In case of $T \approx W \approx PN$:
  By substituting PN for T and PPW and WPP for WPT, we can see that the most frequently called parts of the RRCPA are called $max(\frac{T}{P}, min(T, \frac{T*WPT}{P}), \frac{T*WPT}{P}) = \frac{\boldsymbol{T*WPT}}{\boldsymbol{P}}$ times.
  Compared to the previous approaches (as mentioned with a call frequency of $max(T, \frac{T*WPT}{P})$), this is better in some situations. We can reduce the call frequency below T by increasing the number of PUs working on the program, but as long as *WPT>P*, we do not have an advantage. So the RRCPA is faster by a factor of $\boldsymbol{max}(\mathbf{1}, (\frac{\boldsymbol{P}}{\boldsymbol{WPT}}))$.
- In case of $T > PN \approx W$ (which is true if $WA > ATD$):
  We can only substitute PPW and WPP for WPT. The most frequently called parts of the RRCPA are called $max(\frac{T}{P}, min(PN, \frac{PN*WPT}{P}), \frac{PN*WPT}{P}) = \boldsymbol{max}(\frac{\boldsymbol{T}}{\boldsymbol{P}}, \frac{\boldsymbol{PN*WPT}}{\boldsymbol{P}})$ times.
  We differentiate between two subcases:
  In case $T \geq PN * WPT$, the most frequently called parts are called $\frac{T}{P}$ times, which is by a factor of $\boldsymbol{max}(\boldsymbol{WPT}, \boldsymbol{P})$ better than the $max(T, \frac{T*WPT}{P}))$ calls of our previous approaches.
  In case $T < PN * WPT$, the most frequently called parts are called $\frac{PN*WPT}{P}$ times, which is by a factor of $\boldsymbol{max}(\frac{\boldsymbol{T}}{\boldsymbol{PN}}, \frac{\boldsymbol{T}}{\frac{\boldsymbol{PN*WPT}}{\boldsymbol{P}}})$ better than the $max(T, \frac{T*WPT}{P}))$ calls of our previous approaches.

So for this approach the amount of panes (or windows, since *W≈PN*) compared to the number of tuples is very important for the eventual performance. The call frequencies of six of the eight parts of the algorithm are depending on the amount of panes.

Two concrete examples
In a lot of practical cases, the number of tuples is not equal to the number of panes, but significantly larger. Let us have a short look on two examples, one with *T=PN* and one with *T>>PN*, and analyze the assumed speedup of the pane-based approach compared to the previous approaches.

- There is one machine. It sends one tuple per second (*ATD=1s* (**a**verage **t**imestamp **d**istance, see Section 5.2.1)) about its temperature into an input data stream. Every one second (*WA=1s*) we want to know the machine's average temperature during the last minute ($WS = 60s$, $WPT = \frac{WS}{WA} = 60$).
  As presented in Section 5.2.1, $W \approx \frac{T*ATD}{WA}$, so in our example $W \approx \frac{T*1s}{1s}$, so $W \approx T$ and since $W \approx PN$, it holds that $T \approx W \approx PN$.
  So our pane-based approach which calls its most frequently processed parts $\frac{T*WPT}{P} = \frac{60T}{P}$ times has by a factor of $max(1, \frac{P}{WPT}) = max(1, \frac{P}{60})$ fewer calls. Hence if we are using 60 or fewer PUs, the number of program part calls per PU does not change, but if we are using more than 60 PUs the pane-based approach exhibits by a factor of $\frac{P}{60}$ fewer calls. Thus we would for example need 120 PUs to halve the call frequency of the most commonly called part, which is not very promising at all.

- There is one machine. It sends 1000 tuples per second ($ATD = 1ms$) about its temperature into an input data stream. Every one second ($WA = 1s$) we want to know the machine's average temperature during the last minute ($WS = 60s$, $WPT = \frac{WS}{WA} = 60$).
  So in our example $W \approx \frac{T*1ms}{1s}$, *so* $W \approx 0{,}001T$ and since $W \approx PN$, it holds that $T \approx 0{,}001W \approx 0{,}001PN$.
  Hence our pane-based approach calls its most frequently processed parts $max(\frac{T}{P}, \frac{PN*WPT}{P}) = max(\frac{T}{P}, \frac{0.001T*60}{P}) = \frac{T}{P}$ times has by a factor of *max(WPT, P)=max(60, P)* fewer calls. So if we are using 60 PUs or fewer, we have 60 times fewer function calls and if we are using more than 60 PUs, we have P times fewer function calls. In either case that is a huge decrease to the number of function calls and thus the pane-based approach might accelerate the whole algorithm enormously.

Hence which part eventually bottlenecks the algorithm's performance is highly depending on the number of panes compared to the number of tuples in the data stream and the WPP parameter.

Disadvantageous settings
There are a few cases that are unfavorable for the pane-based approach compared to the previous approaches.

- If $\boldsymbol{T \approx W \approx PN}$ and $\boldsymbol{WPP \geq P}$:
  In case $T \approx W \approx PN$ (which is true if $WA \approx ATD$), so if there are as many panes as tuples, one pane contains one tuple and one tuple contributes to one pane. So instead of aggregating tuples to a window, we aggregate the same number of panes to a window, which is equally fast. Table 13 confirms that: The most frequently called parts are globally processed as often as in the previous approaches (with the exception of the Handshake approach), so the pane-based approach does not grant any advantages in the form of work reduction here. The overall costs might be even higher because of the detour through panes.
- If both holds, $\boldsymbol{PPW = \frac{WS}{WA} = 1}$ and the Aggregate Operation part (no matter if its tuple pre-processing subpart or the aggregation subpart) is so expensive that it becomes the limiting factor - according to the call frequency tables (Tables 1, 4, 7, 10 and 13) this holds for all approaches except the HSA, as long as the costs for one call of the Aggregate Operation part are by a factor $\boldsymbol{f \geq P}$ higher than the costs for one call of any other part:
  Since $PPW = \frac{WS}{WA} = 1$, one pane is basically equal to one window because each window consists of one pane and each pane contributes to one window ($PPW = WPP$). So concerning the number of aggregate operations and both of its subparts (the tuple pre-processing and the aggregation subpart) that we have to process, there is no advantage in using panes at all because the purpose of panes was to calculate them once and reuse them several times to avoid redundant calculations.
  Under our assumptions, according to the call frequency tables, the previous approaches' costliest part is the Aggregate Operation part with costs of $\frac{f*T*PPW}{P} = \frac{f*T*1}{P} \geq T$.
  The pane-based approach's costliest part is the Aggregate Operation part with costs of $\boldsymbol{\frac{f*T}{P} \geq T}$ as well.
  So the pane-based approach does not grant any advantages in the form of work reduction here. The overall costs might be even higher because of the detour through panes.

However, the pane-based approach still has the advantage that the communication costs are much lower because each tuple visits only one PU instead of all PUs. So when working with distributed systems, the pane-based approach might still be superior if the communication network is sufficiently slow.

Throughput

As discussed in this section, the speed of the pane-based approach is highly depending on the aggregation's parameters as well as the properties of the data stream.

Compared with the previous approaches, we can observe: While there are some situations in which the overall amount of work remains approximately constant or is even worse due to the detour of tuples contributing to panes which contribute to windows instead of the direct way of tuples contributing to windows, in many situations the pane-based approach is significantly faster.

As a rule of thumb, we can summarize:

- The bigger the number of tuples per pane ($\frac{WA}{ATD}$) (or with other words, the smaller the number of panes compared to the number of tuples), the higher the advantage over the non-pane-based approaches.
- The bigger the number of panes per window $PPW = \frac{WS}{WA}$, the higher the advantage over the non-pane-based approach.
- The more storage we have, the more tuples per pane each PU can store. The more tuples per pane each PU can store, the longer it can compute aggregations using its full computation capacity without receiving further tuples. The longer each PU computes using its full computation capacity, the more PUs we can effectively use (see Section 9.2.4 for a detailed discussion).

Communication traffic

The communication traffic caused by the result sets that are transported from the last PU in the ring to the first one is equal to the result set traffic of the previous approaches. The overall amount of windows that are output remains constant. The expired windows are just distributed among fewer result sets as the number of participating PUs increases (see the discussion in Section 9.1).

The communication traffic caused by tuples is reduced by a factor of P, since each tuple visits one PU instead of P number of PUs.

There is additional traffic caused by panes, as each pane is being forwarded to $min(P, WPP)$ number of PUs. The memory size of panes depends on the executed aggregation (see also Section 5.2.4 and 9.2.4).

- There are aggregations, in which panes exhibit a constant size.
  For instance, when computing the sum aggregation, each pane contains one intermediate sum. So depending on the type of the variable storing the sum (integer, long, float, double, …), the pane's size is a few bytes large.
- And there are aggregate operations, in which panes exhibit a size that is depending on the amount of tuples contributing to them.
  For example, when computing the median aggregation, each pane contains all tuples in sorted order. In this case, since each tuple contributes to exactly one pane, the sum of the memory

size of all panes is approximately equivalent to the sum of the memory size of all tuples. While in the non-pane-based approaches each tuple has to visit each PU, in the pane-based approach each pane visits $min(P, WPP)$ number of PUs. So depending on the WPP parameter, the communication costs can be as high as in the non-pane-based approaches.

In summary we can say that – compared to the non-pane-base approaches – aggregations using panes with a constant size exhibit much lower communication costs, while aggregations using panes with a size equivalent to the sum of the sizes of all contributing tuples exhibit similar communication costs.

Advantages of the structural changes

Although exhibiting a cyclic communication structure as well, the two advantages of the HOA and the RRCA (mentioned in the sections 7.2.4 and 8.2.4) do not apply to the pane-based approach.

- In many cases, the overall amount of work can be reduced drastically.
- All parts of the algorithm can scale with arbitrary numbers of participating PUs.
- Tuples are not forwarded at all. This might decrease the utilization of the communication system greatly. Only if the size of each pane is not constant, but equivalent to the size of all tuples contributing to it, there is no advantage concerning the communication costs.

Disadvantages of the structural changes

- The connections are not limited to neighbor-to-neighbor communication, since each PU has a connection to the input stream. Hence in case we want to use this approach on distributed systems, we need to maintain twice as many connections.
- Imbalances of the input data stream's tuple rate or the incoming tuples' ATD can prevent PUs from updating their safe-timestamp for longer periods of time, which can delay the output of results.
- Similar to the discussion in Section 7.1, without further effort it is not possible to dynamically compute order-sensitive aggregate operations, since in general panes are not arriving in timestamp order.
- We do not have the opportunity to use multiple input streams at the same time without merging them to order their tuples by their timestamps.
- The input data stream has to expend additional effort to determine the responsible pane for each tuple and on which PU this pane is located. So the input data stream has to maintain central coordination tasks.

In contrast to the previous approaches, the pane-based approach is basically indefinitely scalable. In many practical cases it additionally offers an enormous reduction of the computation and communication costs compared to all previous approaches. Its advantages concerning the throughput, the work distribution and the overall amount of work are highly dependent on the data stream's properties and the executed aggregation.

9.4 Suggestions for improvement

Parallelization of the input stream's coordination tasks

One weak point of the RRCPA is the input thread which has to manage the tuple distribution and may bottleneck the whole algorithm. The distribution tasks are not parallelizable within this approach yet, but an efficient parallelization of the distribution of the incoming tuples might be a good starting point for future work.

Divide panes into sub-panes

There is also another suggestion for improvement addressing two further disadvantages of the RRCPA, the high latency and – depending on the parameters – the possibly high memory consumption. We could divide panes into sub-panes to reduce the number of tuples per pane in order to distribute work more evenly and faster among the participating PUs especially in situations where the main memory is not sufficiently large. Since the time interval between a pane on a specific PU and the next pane that will be located on the same PU as well decreases as we split panes into sub-panes, the latency can be reduced with the aid of sub-panes as well. We pay for these two advantages with a higher number of pane merging steps per window, so with higher computation costs.

As a rule of thumb we can say that splitting each pane into two sub-panes approximately halves the output latency and the main memory requirements while doubling the call frequency of the Panes to Window Aggregate Operation. That means that the fraction of this part's calls among all parts' calls increases. The impact of this increase depends on the WPT and ATP parameters (see Table 13). Depending on these parameters the deceleration of the tuple processing rate can be between 0% and 50%.

Usage of hash functions to split tuples with different keys

As we know, the pane-based approach's input stream is supposed to send approximately $\frac{WA}{ATD}$ consecutive tuples to one PU.

In cases in which there are tuples with many different keys, the machine or thread sending tuples could use a hash function to compute an offset x, $0 \leq x \leq P-1$ for each key. Now we still use the formula explained in Section 9.1 to assign tuples to panes and PUs, but each tuple t is inserted on the PU with an additional offset of x, depending on t's key. As the previous suggestion of splitting panes into sub-panes, this could help to distribute the work more evenly among the participating PUs.

A disadvantage of this suggestion is the difficulty to maintain safe-timestamps. Depending on the key distribution of the arriving tuples, long send pauses to certain PUs are possible.

Another disadvantage of this suggestion is the even higher amount of coordination work the sending machine has to maintain continuously. However, if the sending thread is not the limiting factor or if we have an efficient parallelization for the input tasks, using hash functions could be an eligible alternative.

Usage of dummy tuples to reduce the latency

As mentioned in Section 9.1 (paragraph *Pane expiration*), PUs could send additional information messages or the input stream could send additional dummy tuples to inform PUs about arriving tuples possibly leading to the expiration of panes or windows. By doing so, we could decrease the output latency at the cost of additional computation and communication costs.

We did not use dummy tuples or additional information messages in the pane-based approach because they are only promising under certain circumstances. Especially if the ATD is not much smaller than WA, the latency benefits would be small and the additional computation and communication costs would be large. However, in other cases the usage of dummy tuples seems very reasonable if a low output latency is required.

Alternative structure

There is a conceivable alternative for the structure of the RRCPA which might be an improvement in certain cases:

In the current version (see Figure 16), we send panes along the PU ring; they meet windows to

contribute to them. We also send result sets along the PU ring from the last PU p_{n-1} to the first PU p_0 to collect windows.
Instead of panes meeting windows, we could also send windows from PU p_{n-1} to PU p_0 to meet panes (see Figure 17 below).

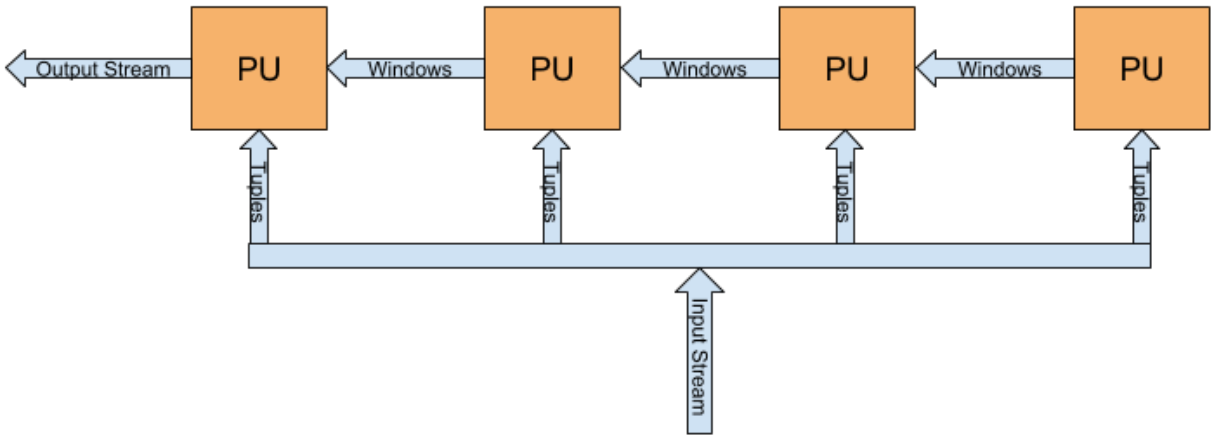


Figure 17: Alternative structure for the Round-Robin-Cyclic Pane-based approach

The most obvious advantage of this structure is that we do not need a ring topology but only a line. But there are more possible advantages that we just list up without going more into detail: The forwarding rules for panes become dispensable because panes stay at the same PU during their whole existence. Synchronization using the safe-timestamp is no longer necessary. The latency is reduced significantly compared to the current structure of the RRCPA.
An execution of order-sensitive aggregations is still not possible with the alternative structure.

The idea to this alternative version of the RRCPA came up shortly before finalizing the written report, hence it is neither analyzed in detail nor implemented and all the evaluation tests in Chapter 11 work with the original structure of the RRCPA (see Figure 16).

10 Implementation

In this chapter, we will discuss implementation related issues as the target processing system, the chosen programming language and environment, and the choice of different data structures that we use for certain parts of the algorithm.

10.1 Selection of the processing system for the implementation

One of the first questions arising when planning a new parallel algorithm refers to the system on which we want to execute it because this can be of prime importance for structural decisions.
In Chapter 3 (about the goals of this thesis), we mentioned that we want to focus on a shared-memory algorithm executed on a single machine.
We shortly want to consider possible hardware setups and their suitability for our algorithm in order to explain, why we decided to focus on multicore processors.

- implementation for one multicore computer - shared-memory version
- implementation for several computers communicating via LAN, InfiniBand or another network system - distributed-memory version
- implementation for massively data parallel graphics processing units (GPUs) - GPGPU version

There is a paper “Evaluation of streaming aggregation on parallel hardware architectures” [5], which is dealing with the suitability of different architectures for streaming aggregation problems. This paper compares three versions: One with a system using an Intel Core 2 Quad, one using an NVIDIA GeForce GTX 285 GPU and one using a Cell Broadband Engine. The first one belongs to the first mentioned group, the second one to the third group and the third one essentially to the first group as well. The results showed that GPGPUs are not suited for this sort of problem mainly due to data transfer issues between the GPU’s memory and the computer’s RAM. Even though the raw calculation speed of the GPU is 10 times higher than the Cell Broadband Engine’s and 25 times higher than the Intel Core 2 Quad’s one, the performance of the GPU is by far the slowest of the three tested systems.
The Cell Broadband Engine wins the comparison and outperforms the Intel Core 2 Quad; it possesses twice the performance. It offers efficient access to the main memory, is more data parallel and has a higher peak performance (about 100 GFLOPS instead of 40 GFLOPS).
Unfortunately, this paper does not make any statement concerning distributed memory systems and their suitability for our problem.
However, due to the poor performance of the GPU implementation and because it is more reasonable to exploit a CPU’s inherent parallelization potential (almost each modern CPU is able to execute multiple threads concurrently) before expanding an algorithm to execute it on multiple computers, we decided to focus on a shared-memory version. Whenever we see important differences when running our version on a distributed memory system, we mention that in our explanations.

We are using an Intel Xeon E5-2695 v3 (Haswell) processor with 14 cores and Hyper-Threading (up to 28 threads running concurrently).

10.2 Selection of the programming language for the implementation

There are several well-known and common programming languages that are feasible for developing multi-threaded applications; e.g. C/C++, Java and C#.
Due to the platform independency, the large amount of built-in concurrent data structures and the availability of powerful development environments (e.g. Eclipse) with useful debug tools for single-threaded as well as for multi-threaded applications, we will use Java as the first implementation’s

programming language.
C# offers similar amounts of concurrent data structures and a very advanced programming environment in the form of Visual Studio. It is also possible to execute C# programs on Linux, Windows and Mac systems. In opposition to Java, C# as well as C and C++ provide us the option to assign threads to specific cores. This feature could be used to assign neighbor PUs to neighbor cores in order to increase the communication speed between the different threads. Since the communication only takes place between neighbor PUs, a large portion of the communication will take place using fast caches (e.g. L1 cache and L2 cache). In Java, the virtual machine decides which thread (PU) is executed by which processor core. If the Java Virtual Machine would assign PUs arbitrarily among the processor cores, most likely a larger portion of the communication will be performed using slower parts of the memory hierarchy (e.g. L3 cache or main memory).
So after all, C# would be another appropriate choice.
However, due to the higher popularity and spread of Java especially on less common platforms we preferred it to C#. In addition, the creation of a C#-version on the basis of the Java one is a fast and straightforward issue because of the similarity to Java.

10.3 Classes, their interdependencies and data structures for the non-pane-based approaches

This section explains implementation details of the non-pane-based approaches, so the Handshake approach, the Round-Robin approach, the Hash-Once approach and the Round-Robin-Cyclic approach. The whole section is based upon information presented in the previous chapters, especially Chapter 2.

Figure 18 gives a rough overview over the implementation in the form of a class diagram of the different types of objects, their attributes and their relationships. We will discuss the depicted objects, attributes and interdependencies.
This figure and the discussion apply to the Handshake approach, the Round-Robin approach and with few differences also to the Hash-Once approach and the Round-Robin-Cyclic approach. These differences only concern latency measurements and the way the program terminates after the arrival of the last tuple to guarantee deterministic output behavior. Since these discrepancies are not very important for the understanding of the approaches, we will not discuss them within the scope of this thesis.

10.3.1 Tuple

General properties of tuples were already described in Section 2.2.
In the HSA and the RRA, in addition to the mentioned attributes in Section 2.2, each *Tuple* object always has the attributes *realtimeTimestamp* and *finalTuple*.
To measure the latency of the processing system we need the *realtimeTimestamp*. It has no influence on the processing of the tuple at all, is just being used for benchmark purposes. It carries accurate information about the tuple's creation time. We use the built-in Java function *System.currentTimeMillis()* whenever a tuple is created to set its *realtimeTimestamp*.
The attribute *finalTuple* is necessary to terminate the program either because the user or programmer wants to terminate the program execution or because the data stream ends. Since each window can only expire if we know that no further tuples that contribute to it will arrive, we need to flag the last tuple to guarantee the expiration of all remaining windows and hence the completeness of the results.

10.3.2 Aggregate Operation

This section is about the implementation details of the *AggregateOperation* class. General information about aggregate operations can be found in Section 2.4 and Section 17.1.

An *AggregateOperation* object defines all properties of a single aggregate operation that shall continuously and periodically be processed on the data stream.

Each aggregate operation has four different attributes: *operationId*, *windowSize*, *windowAdvance* and *groupByKeys*.

The *operationId* determines which aggregate operation is to be computed. Each aggregate operation (sum, average, count, …) has an own unique ID.

The *windowSize* and *windowAdvance* attributes store the timestamp range of each window and the time interval between two consecutive windows.

The *groupByKeys* attribute stores the information whether results shall be grouped together on the basis of the tuples' optional key.

For each aggregate operation that we want to execute simultaneously on the same data stream, we create one instance of the *AggregateOperation* class and hand each PU a reference to it, so that every PU is permanently informed about all operations currently being executed.

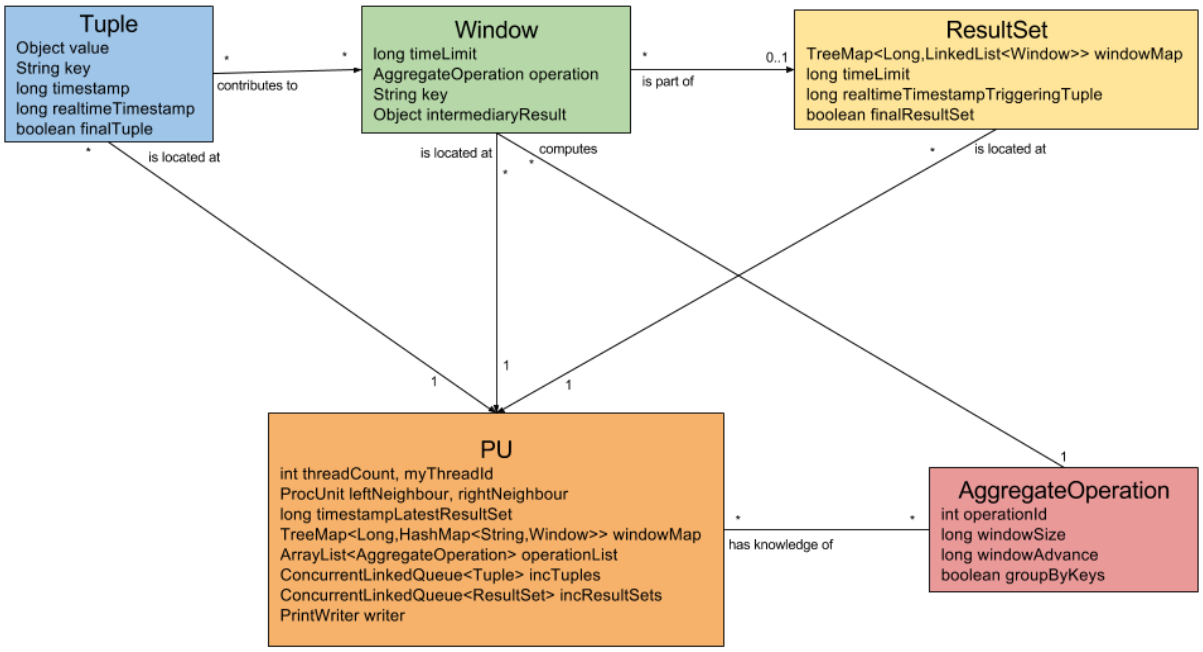


Figure 18: Class diagram of the HSA and the RRA

10.3.3 Window

To describe all properties of a *Window* object, we need 3 different attributes: *timeLimit*, *operation* and *key*.

The *timeLimit* attribute saves the information when the window will expire. Having a look on Figure 2, the window objects in this example would have the *timeLimit* values 1000, 1270, 1540…

The *operation* attribute stores a reference to the aggregate operation which the window computes on its tuples. In Figure 2 all windows would point to the same aggregate operation instance, and this instance saves the information about window advance and window size as well as the aggregate function being executed on tuples of each window. So it is not necessary to save the same redundant information in each *Window* object.

Lastly the *key* is optional. It is only used if the referenced aggregate operation's *groupByKeys* attribute

is true. If *groupByKeys* is true, there is not only one window for each window time-slot (like in Figure 2). Instead, for every different *key* of each tuple falling in this window time-slot, there exists a corresponding window with the respective key.

10.3.4 ResultSet

As presented in Chapter 5, a result set – starting its way on the rightmost PU and stepping leftwards until it reaches the leftmost PU – collects expired windows up to the result set's *timeLimit* on its way. We need the expired windows to be ordered by their *timeLimits* because we want to accomplish a deterministic output behavior: We always output windows with earlier (lower) *timeLimits* before windows with later (higher) *timeLimits*.

In order to do so, PUs store the expired windows within a *TreeMap* called *windowMap* which automatically sorts all elements (expired windows) by their key (the time limit). Since it is possible that there exist multiple windows with the same time limit but different keys (or windows from different aggregate operations that we execute concurrently on the same data stream), the windows are not stored directly within the *windowMap* but within *LinkedLists* in the *windowMap*. All windows in a single *LinkedList* have the same time limit, and the *windowMap* sorts the *LinkedLists* by the time limit of their windows.

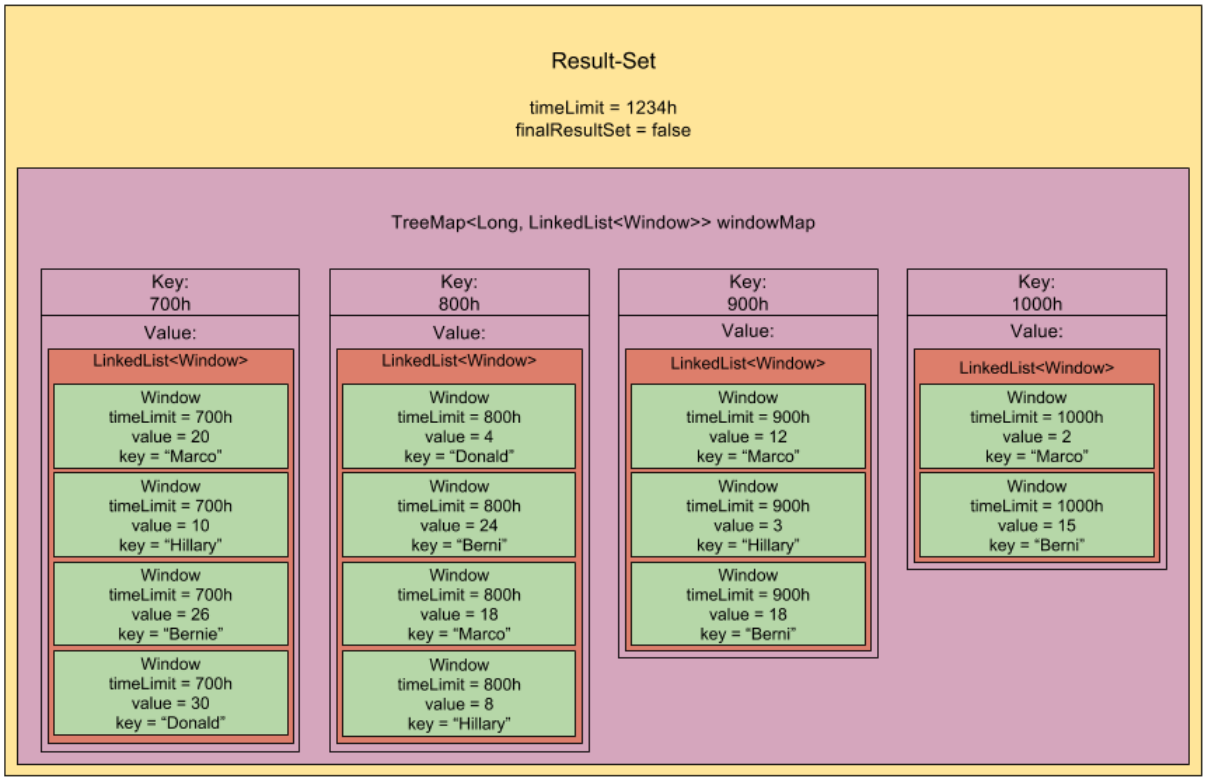


Figure 19: Example of a ResultSet's structure

Figure 19 depicts an example of an *ResultSet* object:
There are different persons who receive letters. Every *WA=100* hours we want to know how many letters each person received during the last *WS=400* hours.
In most cases each *ResultSet* only contains windows with the same time limit, since *ResultSets* are created as soon as a tuple arrives that causes a window to expire. But if there is a long send pause of

the input stream (in the example there are no tuples with timestamps between 700h and 1233h) the next tuple (in the example one with timestamp 1234h) causes several windows to expire at once.

We choose the *TreeMap* as data structure because it is efficient in sorting a possibly growing list of elements (in this case lists of windows) by their keys (the windows' time limits) and iterate through the sorted list afterwards. A *TreeMap* is sorted at each point of time, so whenever a new window is being added to the *TreeMap*, the *TreeMap* inserts it at the correct place to preserve the sorted order. Inserting a new element, searching or removing a specific element by its key and searching or removing the element with the smallest key are *O(log(n))* operations, whereat n is the number of elements in the *TreeMap*.

The attribute *realtimeTimestampTriggeringTuple* just exists for measuring the output latency and for benchmark purposes. When a *ResultSet* r is created, r sets its *realtimeTimestampTriggeringTuple* attribute to the *realtimeTimestamp* of the tuple t that caused r's creation (by leading to the expiration of one or more windows). When r is output, we compute *System.currentTimeMillis()-realtimeTimestampTriggeringTuple* to measure the elapsed time between the point of time when t entered the processing system and the output of r's windows. So in other words, this is the elapsed real time that t needed to travel from the leftmost PU to the rightmost PU plus the real time that r needed to travel back from the rightmost PU to the leftmost PU and put its windows into the output stream. This is what we call the system's *latency*.

The attribute *finalResultSet* informs the threads whether the current *ResultSet* is the last one. When the final tuple (mentioned in Section 11.2.1) arrives at the last PU, a *ResultSet* is created and marked as *finalResultSet*. This *ResultSet* will not have a time limit so that it will collect all remaining windows on all PUs, and all of them will already be expired because the final tuple led to their expiration (we choose the largest value representable by *long* variables as time limit in order to collect all remaining windows).
Then – after processing and forwarding the *ResultSet* – the threads know that they shall terminate.

10.3.5 PU

The structure of the Handshake approach and the tasks each thread has to perform were already explained in Section 5.1 and its respective figures.
This section only handles implementation decisions like the data structures used to achieve the best speed and latency results possible.

A *PU* object (see Figure 18) represents an own processing thread operating on the data stream. As such this class derives from the Java class *Thread*, and each thread executing the program has its own instance of the *PU* class.
The *PU* class is so to speak the base object, containing the majority of the program's logic and references to all the other types of objects.

Each PU has knowledge about the number of threads concurrently working on the program called *threadCount* and an own ID between 0 and *threadCount-1* named *myThreadId*.
Since we align the threads in a line, using only neighbor to neighbor communication, each PU has knowledge of its up to two neighbors, called *leftNeighbor* and *rightNeighbor*.

To store all windows that are currently being processed by a PU, we use a *TreeMap* called *windowMap* that works similar to the *windowMap* used for *ResultSets* (see Section 11.2.4). But instead of

LinkedLists, *HashMaps* store the windows within the *windowMap*: *TreeMap<Long,HashMap<String,Window>> windowMap* (see Figure 20). Since the Find or Create Window part of all approaches continuously needs to find stored windows whenever tuples (or panes respectively) arrive, we need a *HashMap* to guarantee fast access to the windows (the lookup in *HashMaps* is performed in *O(1)* time).
The reason why we do not only use a *HashMap* but also a *TreeMap* as window storage is that a *TreeMap* offers us a cheap possibility to search and remove all expired windows whenever a *ResultSet* r arrives:

(1) We read the *HashMap* h containing the oldest windows of the *windowMap.*
(2) Is the time limit of h's windows smaller than or equal to r's time limit?
(3) If that is the case, we remove h with all its windows from the *windowMap* and add them to r. Then we go back to step (1) and repeat the process.

Searching and removing the *HashMap* with the oldest windows of *windowMap* is performed in *O(log(n))* time.

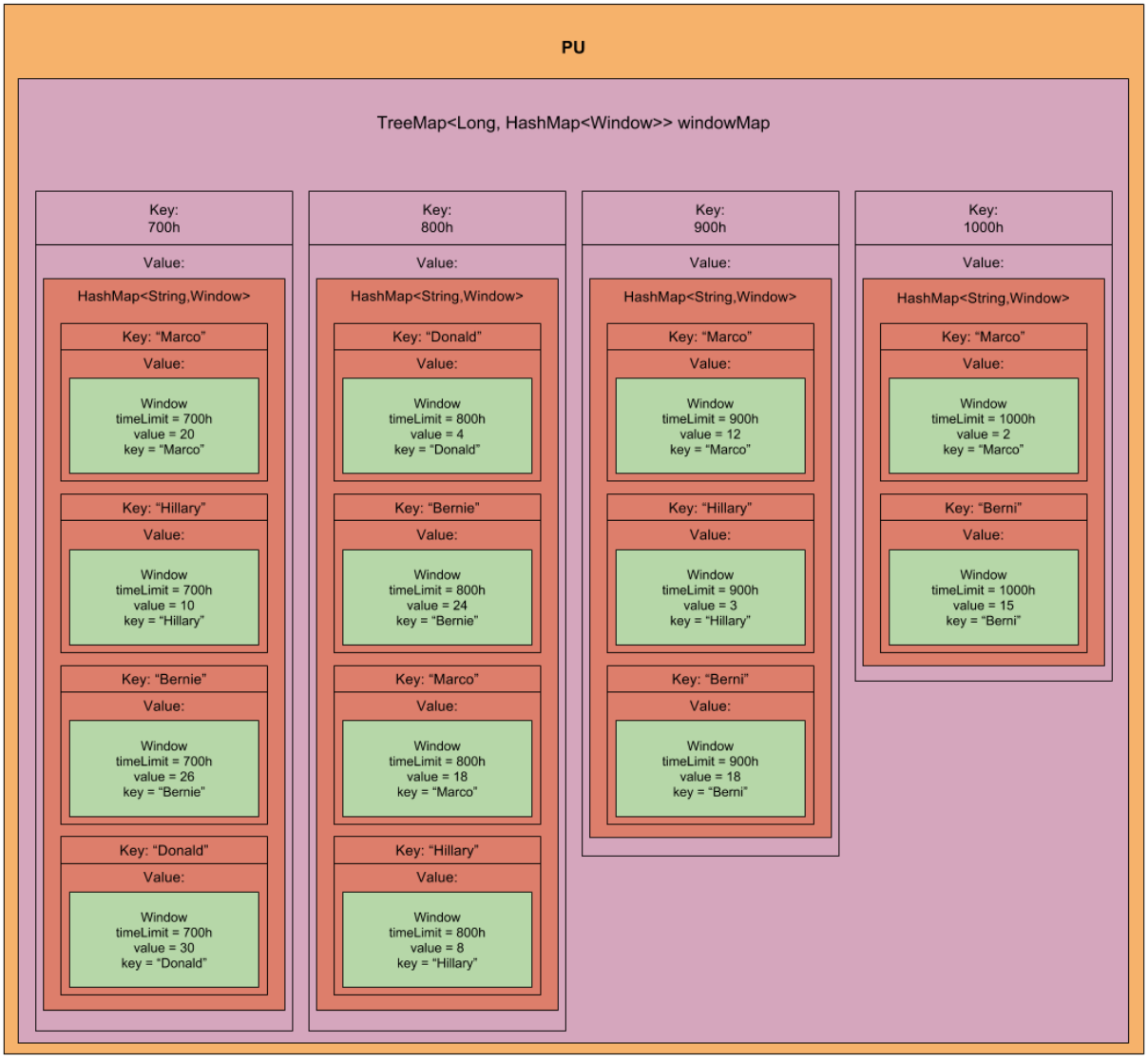


Figure 20: A *windowMap* storing all windows located on a PU before they are collected by the *ResultSet* displayed in Figure 19

Figure 20 depicts the situation of Figure 19 before the *ResultSet* was created. To simplify Figure 20, we assume that all the windows are located in the *windowMap* of one PU. As we know, they are usually distributed over the *windowMaps* of several PUs because for each window a hash function decides where it is computed.

Within an *ArrayList* called *operationList*, each PU stores all aggregate operations that shall be executed.

ConcurrentLinkedQueues are being used to store incoming tuples and incoming *ResultSets*. Queues ensure that the tuple's input order remains unchanged and Java's *ConcurrentLinkedQueue* is a non-blocking data structure on which several threads (in our case two, one sender, one receiver) can work concurrently.
To prevent steadily increasing latency and memory utilization caused by growing queues in case of a suboptimal work distribution among the PUs, we define a maximum size for the *incTuples* queues. If the number of tuples in one PU p's *incTuples* queue exceeds this limit, p forces the sender (that is the left or counterclockwise PU depending on the approach) to stop sending further tuples. In case the sender is the input stream, p forces the input stream to stop sending further tuples as long as p's queue is full.
The selection of the maximum size for queues is a trade-off between throughput and latency.
If we accept very large queues, it happens less often that PUs need to stop sending tuples. Then only long-term discrepancies in the work distribution lead to full queues and hence idle PUs. We pay for this advantage of higher PU utilization and accordingly higher throughput with an increased latency and higher memory requirements.
Very small queues lead to very low latencies, since incoming tuples will be handled very quickly by the receiving PU, but already short-term discrepancies in the work distribution lead to full queues and correspondingly idle PUs.
The queues in the implementation have a customizable maximum size. The default size (if not changed by the user) is 1024. In many cases this seemed to be a good compromise between throughput and latency. However, also maximum sizes of 10 or 100000 can be justified if there are exceptionally high latency requirements or very loose ones.
Another possible approach to attain a very low latency is to reduce the rate tuples are being sent by the input stream; e.g. to 90% of the maximum rate the processing system can handle. This eventually leads to very few or no elements at all in the PUs' queues and consequently is a second way of gaining low latency by slightly reducing the throughput.

Lastly, the results have to be outputted by the leftmost PU. For test purposes, we use a *PrintWriter* to output the results into a file.

10.4 The pane-based approach

There are not many noteworthy implementation details compared to the other approaches. Like for the other cyclic approaches, most of the differences evolve from the different ways to measure the system's latency and from the way of handling the final *ResultSet* and final tuple.

We shortly want to discuss the way PUs store panes.

10.4.1 Panes

At each point of time, if we do not group tuples by their key, there is only one active pane in the whole processing system, so a PU does not need an advanced data structure at all to store the single pane. Else, if we group tuples, there exists one active pane for each distinct key of tuples having a timestamp

in the current pane time-slot. All non-expired existing panes are located on the same PU and in order to store panes – which is necessary whenever a tuple with a new distinct key arrives – and find panes – which is necessary whenever a tuples arrives that contributes to an already stored pane – in *O(1)* time, we use a *HashMap*. Each element of the *HashMap* uses a pane's key as key and the pane itself as value.

11 Experimental Evaluation

In this chapter, we investigate the different approaches' performance in terms of throughput and latency.

In the corresponding sections of the different approaches, we already discussed the circumstances under which we expect the respective approach to perform well or bad and compared it to the previous approaches. Now we will determine the suitability of the different approaches for different settings in practice.

11.1 Structure of the performance tests

As examples we will construct and analyze three different settings. Each setting will exhibit specific values of the aggregation's parameters that we expect to be advantageous for one group of our approaches: The hash-based approaches (HSA and HOA), the round-robin-based approaches (RRA, RRCA) or the Pane-based approach (RRCPA).

So we expect the hash-based approaches – the Handshake approach and the Hash-Once approach – to be superior on the Hash-Advantage settings. We expect the round-robin-based approaches – the Round-Robin approach and the Round-Robin-Cyclic approach – to be superior on the RR-Advantage settings. And we expect the Round-Robin-Cyclic-Paned-Based approach to be superior on the RRCP-Advantage settings.

We will discuss why we chose the given settings and which performance we expect from the approaches in the respective sections of Section 11.3. We will compare the evaluation's results with the predictions and analyze whether they match.

The input data stream

Our input data stream sending the input tuples is being simulated by an extra thread.

The ATD of the tuples is equal to one. So it holds that $t_i.timestamp = t_{i-1}.timestamp+1$ (each tuple has the timestamp of its predecessor tuple increased by one).

It sends tuples as fast as possible as long as the receiving PU's incoming tuple queue does not exceed a certain number of elements (1024 in our tests, see Section 10.3 for more information).

It is important to know that in this test the tuples' timestamps have nothing to do with the real point of their creation time. We could imagine the situation as follows: There is a machine that produces information about its temperature every millisecond. It started one year ago and saved all measured temperature values since the beginning. We want to aggregate all of the stored data and continue aggregating the machine's data in real time afterwards. So we begin with the already stored data and process it as fast as we can to catch up on the computations of all the stored data in order to continue monitoring it with the current data in real time.

If we group tuples by their keys (whenever K is greater than 1), each tuple gets a random key$\in\{0,1,\ldots,K-1\}$.

In Section 11.3, we explain for each of the different settings where this input data stream could occur in practice (so which real-world environment would produce a similar data stream).

At this point it should be mentioned again that windows are both time-based and tuple-based as long as the TD between consecutive tuples is constant (see Section 2.3 for more information). If we group tuples by their keys, this statement is only true if each key's tuples have a constant TD among themselves. Since in this evaluation each tuple's key is chosen randomly among a certain set of possible keys, all settings in which we group tuples by their keys are only time-based and not tuple-based.

11.2 Evaluation Setup

The processing system used for the evaluation works with a 14-core Intel Xeon E5-2695 v3 (Haswell) processor. Since it features Hyper-Threading, we have access to 28 logical cores altogether. The system's main memory has a size of 64GB (DDR3 memory). As mentioned, all the approaches have been written in Java.

11.3 The Evaluation's parameters

Table 16 shows the aggregations' parameters of the different settings: name N, window size WS, window advance WA, amount of distinct keys DK and aggregate operation(s) O.

N	WS	WA	DK	O
Hash-Advantage	10000	10000	1000	$4 * 1k$ Arithmetic Ops
RRCP-Advantage	10000	100	10	$4 * 1k$ Arithmetic Ops
RR-Advantage	1000	1	1	Count (1 Arithmetic Op)

Table 16: Settings for the different benchmarks

The three base cases

First, we shortly want to discuss why we chose the three base cases, so why we expect the chosen settings to be advantageous for the respective approaches.
Thereafter, we imagine a practical use case where those settings would be used.
Next, we examine the plot and discuss if it matches our expectations.

Notes regarding the following plots

The x-axes of our plots always depict the number of threads working simultaneously on the problem, while the y-axes show the throughput in $\frac{t}{s}$ (**t**uples per **s**econd) or latency in ms (**m**illi**s**econds) when the respective amount of PUs are participating.

As discussed in Section 5.2.5, the (output) latency of a window w is defined as the real-time difference between the point of time when a window w left the processing system to be put into the output streaming and the point of time when the tuple t that led to the expiration of w entered the processing system.

We average the evaluation's results over two runs to reduce random fluctuations.

11.3.1.1 The Hash-Advantage setting

The analysis of the HSA's and HOA's runtime properties and a discussion about their advantages and disadvantages can be found in the sections 5.2, 5.3, 7.2 and 7.3.
The Hash-Advantage settings exhibits the smallest possible ratio of WS to WA. Each tuple is just contributing to one single window (*WPT=1*). There are many distinct keys that we want to group. The hash-based approaches, HSA and HOA, still distribute the windows evenly among the participating PUs, while the Round-Robin approach, the Round-Robin-Cyclic approach and the Round-Robin-Cyclic-Pane-based approach do not (see sections 6.3, 8.3 and 9.3). The costs of the aggregate operation are high due to the expensive tuple pre-processing, and the effort the hash-based approaches have to invest in evaluating the hash functions is very low, since *WPT=1* (see sections 5.4 and 7.4), so RRA, RRCA and RRCPA are not able to compensate their very bad window distribution behavior.
So we expect the HSA and the HOA to be up to P-times faster because they distribute the work among P PUs while the others do not.

Expectations

We expect an exception for the RRCPA if we allow the PUs' incoming tuple queues to store at least $\frac{WA}{ATD}$ number of tuples (see Section 9.3); this is the amount of tuples that contribute to each pane. Under this circumstance, the input stream can send all tuples that contribute to a pane to the responsible PU, and then the stream can immediately continue sending tuples to the next pane on the consequent PU. If the PUs' tuple queues would be smaller, the input stream would repeatedly have to wait for PUs. Whether a PU can store all tuples contributing to one pane or not is a question of the main memory size. In our evaluation, we will consider both cases: In one case there is enough memory to store all tuples contributing to one pane, in the other case we limit the tuple queues' size to 1024 (as in all other approaches).

Concerning the latency, we expect advantages for the line-based approaches (HSA and RRA) compared to the cycle-based approaches (HOA, RRCA, RRCPA) because of the communication structure and advantages for the hash-based approaches (HSA and HOA) compared to the round-robin-based approaches (RRA, RRCA, RRCPA) due to the unfavorable throughput expectations of the latter ones. It shall be noted that the latency for the pane-based approach mainly depends on WA and P (see Section 9.2.5). Since the tuples' timestamps are not linked to the real time in this evaluation (see Section 11.1, paragraph *The input data stream*), the time indicated by the timestamps passes more quickly as an approach works faster. Since we expect the RRCPA to scale well if it has sufficient main memory, there should be two opposing trends when we increase P: The higher number of PUs increases the latency, but the faster throughput decreases the amount of real time that is necessary to compute tuples and windows and thus – expressed in real-time units – WA shrinks.

Possible use case of this setting in practice

We have 1000 sensors. Once every ms, one arbitrary sensor among them sends a tuple. We use a very expensive aggregate function. Every ten seconds for each sensor, we want to get the average value of all tuples sent by this sensor during the last ten seconds.

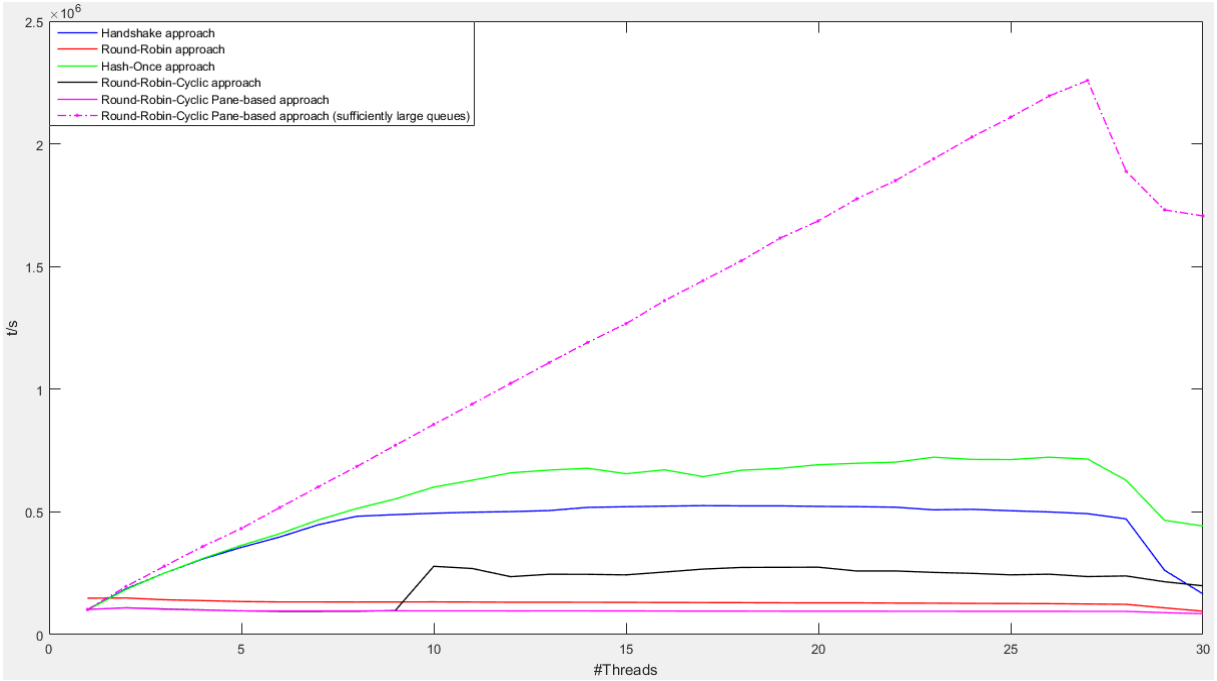


Figure 21: Processing speed in tuples per second depending on the number of participating threads. Hash-Advantage setting: WS=10k, WA=10k, DK=1k, aggregate operation: sum (Pre-process tuple part: 1k additions, 1k multiplications, 1k subtractions, 1k divisions)

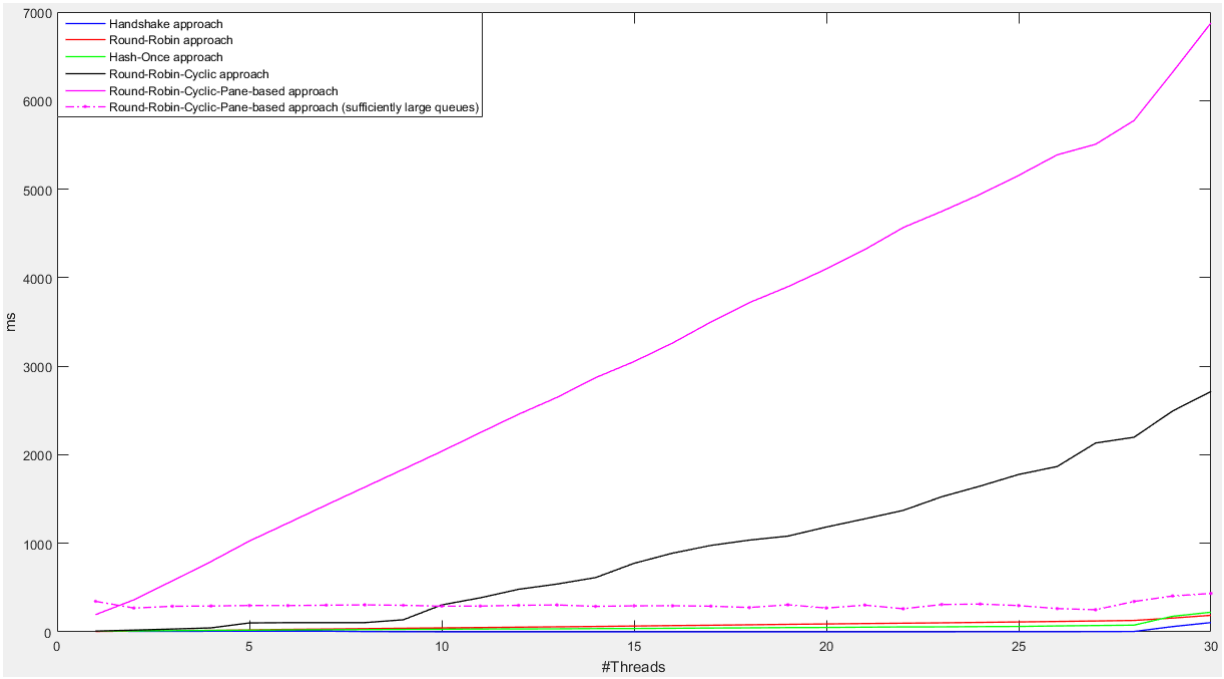


Figure 22: Hash-Advantage setting: Latency in ms depending on the number of participating threads.

Assessment of the evaluation results

As expected, the hash-based approaches are superior to the other approaches, since they can distribute the work among all PUs. Until up to 7 PUs, they scale almost linearly, while the other approaches do not scale at all. If we assume sufficient main memory space (see Section 9.3), the RRCPA outperforms all other approaches and scales linearly.

The poor scaling behavior of the hash-based approaches for higher numbers of threads is caused by the different non-scaling parts (4 for the HOA, 5 for the HSA).

The latency behaves as expected as well. A line-based structure is superior to a cycle-based one and a hash-based window distribution is superior to a round-robin-based one. In case of sufficiently large queues to store at least $\frac{WA}{ATD}$ elements, due to the linear increase of the RRCPA's throughput when increasing the number of participating threads, WA – expressed in real-time units – shrinks inversely proportional to P and hence the product of both remains constant. In case the queues are too small, the throughput remains constant and hence the latency increases as we raise P.

11.3.1.2 The Round-Robin-Pane-Based-Advantage setting

The RRCPA is basically in all situations superior to the RRA and the RRCA, where $WA > 1$ and $WS > WA$ (see sections 9.3, 6.3 and 8.3). Its maximum speed-up compared to the RRA and the RRCA for a certain WA is reached as long as $WS \geq WA^2$. So the RRCP-Advantage settings exhibit a big WA and a $WS = WA^2$. We expect neither the number of distinct keys nor the aggregation's costs to have a significant influence on the superiority over the RRA and the RRCA, so we can choose arbitrary values for them.

Expectations

We expect the RRCPA to be up to $\frac{WS}{WA}$-times faster than the other non-pane-based approaches because it has to invest $\frac{WS}{WA}$-times less work into the aggregation operations. The other approaches should scale as well but on a much lower level. The round-robin-based approaches should be superior to the hash-based approaches due to the additional work the latter ones have to invest.

Concerning the latency, we expect the RRCPA to exhibit a very low one compared to the others due to the very high throughput expectations and accordingly the short time the processing of $3 * WA$ tuples consumes, which is decisive for the latency (see Section 9.2.5 and Section 11.3.1.1). Since we expect the round-robin-based approaches to be faster than the hash-based, they might get advantages at this point due to the shorter computation time per tuple and PU. As usual, the line-based approaches should exhibit lower latencies than the cycle-based ones.

Possible use case of this setting in practice

We have ten sensors. Once every ms, one arbitrary sensor among them sends a tuple. We use a very expensive aggregate function. Every 0.1 seconds for each sensor we want to get the average value of all tuples sent by this sensor during the last ten seconds.

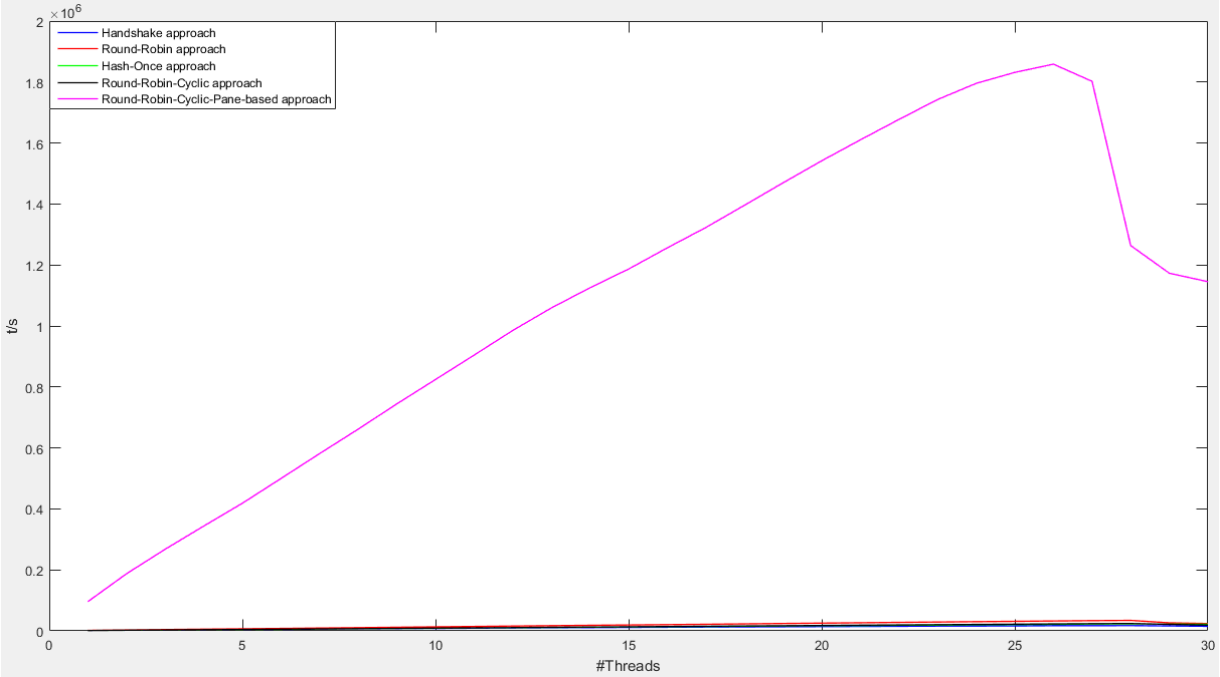


Figure 23: Processing speed in tuples per second depending on the number of participating threads. RRCP-Advantage setting: WS=10k, WA=100, DK=10, aggregate operation: sum (Pre-process tuple part: 1k additions, 1k multiplications, 1k subtractions, 1k divisions)

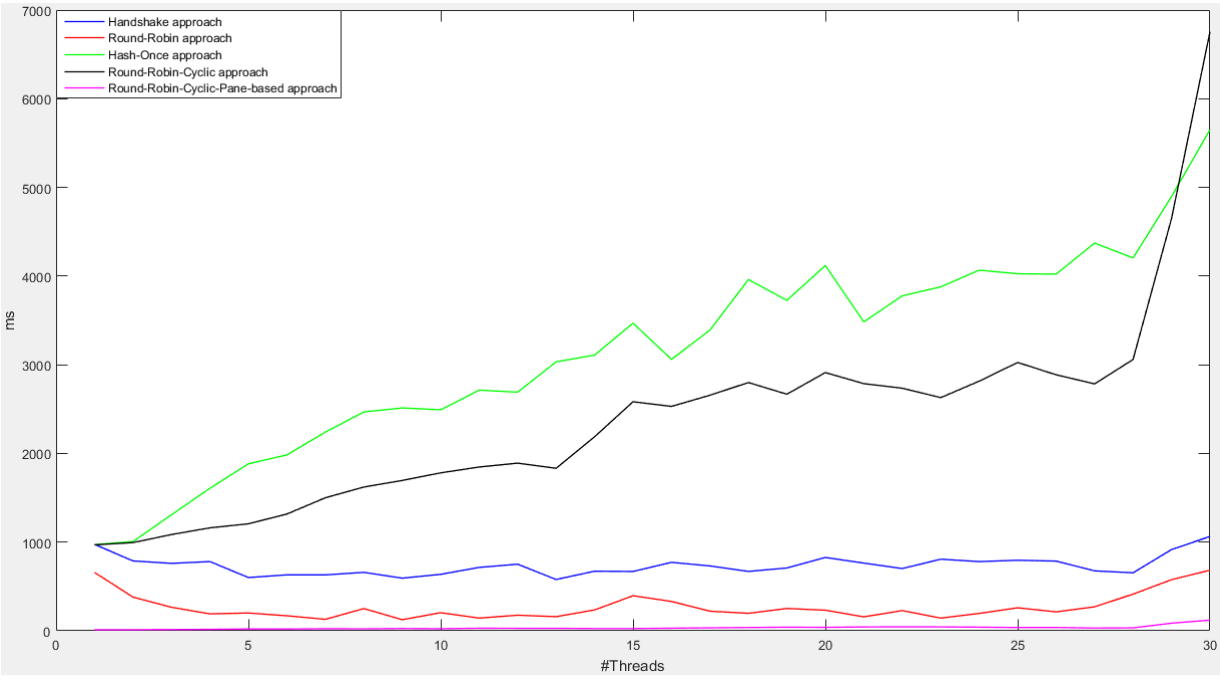


Figure 24: RRCP-Advantage setting: Latency in ms depending on the number of participating threads.

Assessment of the evaluation results

As anticipated, the pane-based approach is 70-120 times faster than the other approaches (a zoomed version of this plot can be found in the appendix in Section 17.3). It scales linearly up to 28 threads. The other approaches scale linearly as well but on a much lower level, since the non-pane-based approaches have to do significantly more work in settings like this one.

The latency behaves as expected. The line-based approaches are superior to the cycle-based ones, and the round-robin-based approaches have advantages over the hash-based ones. The RRCPA exhibits the lowest latency due to its extremely rapid throughput rate and accordingly the faster elapsing of the up to $3 * WA * (P - 1)$ time units (see Section 9.2.5).

11.3.1.3 The Round-Robin-Advantage setting

The RRA and RRCA are in most cases significantly faster than the HSA and also slightly faster than the HOA. However, in theory they are only approximately as fast as the RRCPA approach if either (case 1) $WA \approx ATD$ or (case two) $\frac{WS}{WA} = 1$ and might in practice even be slightly superior due to the calculation and communication costs of the RRCPA's pane management (see sections 6.3, 8.3 and 9.3).

There is a downside in the latter case (2): None of the approaches are able to distribute the work at all (except the hash-based approaches if $DK >> P$) and hence there is no potential for parallelization. So the RRA and the RRCA might be slightly faster for this setting, but they as well as the other approaches are not scaling with more than one PU at all.
So we consider the other case (1) and choose a setting in which $WA = ATD = 1ms$. Now, in the pane-based approach, all panes will contain only one tuple, so there is no advantage in using panes at all (see Sections 9.3). Additionally, we choose a large value for WS to get a big WPT parameter which is very disadvantageous for the hash-based approaches (see sections 5.3 and 7.3).

Expectations

We expect the pane-based approach to scale with increasing numbers of threads but on a lower level than the other round-robin-based approaches, since the pane management consumes more computing time. We expect the hash-based approaches to suffer from the very high WPT parameter. The Prepare & Execute Hash Operation part of the HSA will entirely bottleneck its performance while the HOA will store huge amounts of responsibility information within each tuple, slowing down the communication between the processing cores (even if we transport data via reference within shared-memory machines, the data is still transferred between the main memory and different cache levels, which takes some time).

Concerning the latency, we expect similar proportions between the non-pane-based approaches as in the last setting. The assumed speed advantage of the RRA's and RRCA's throughput should lead to lower latencies compared to the hash-based approaches. As usual, the line-based approaches should exhibit lower latencies as the cycle-based ones. The RRCPA's latency should be much higher in this setting compared to the previous one, since we expect its throughput to be significantly lower which decelerates the speed at which the latency time (up to $3 * WA * (P - 1)$ time units, see Section 9.2.5) elapses.

Possible use case of this setting in practice

We have one sensor. Once every ms, it sends a tuple. We use a cheap aggregate function (count). Every 1 ms we want to get the average value of all tuples sent by the sensor during the last second.

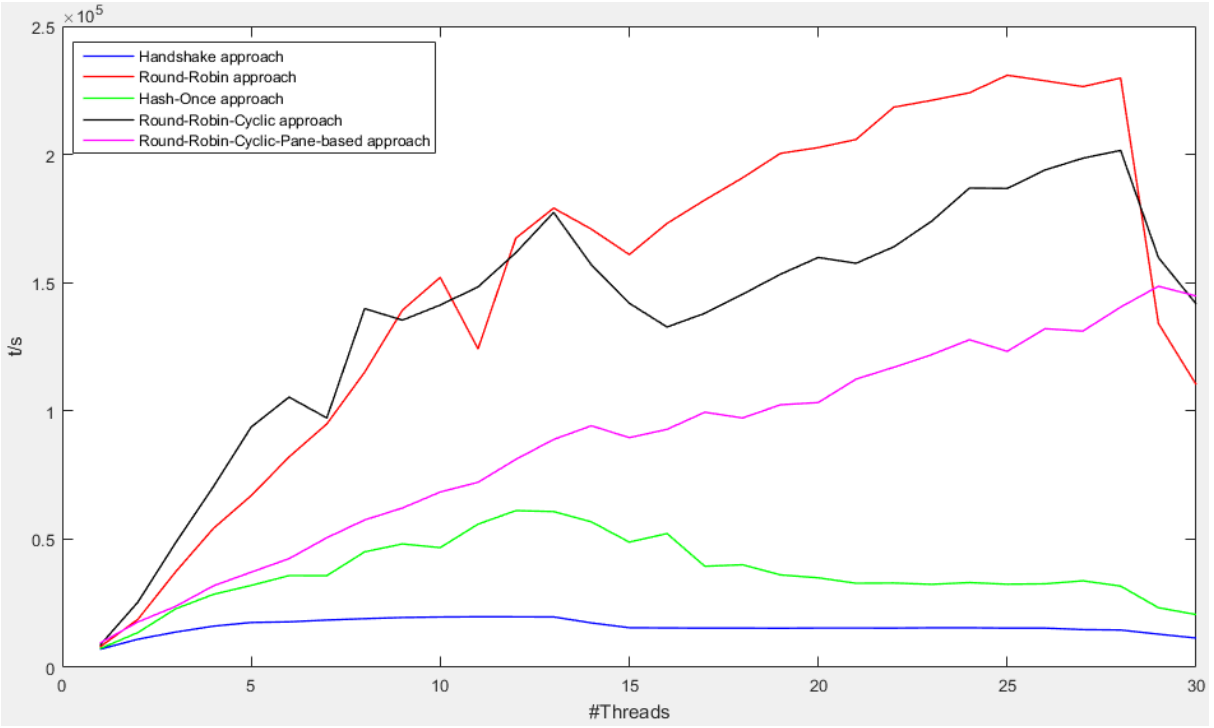


Figure 25: Processing speed in tuples per second depending on the number of participating threads. RR-Advantage setting: WS=1k, WA=1, DK=1, aggregate operation: count (No pre-process tuple part)

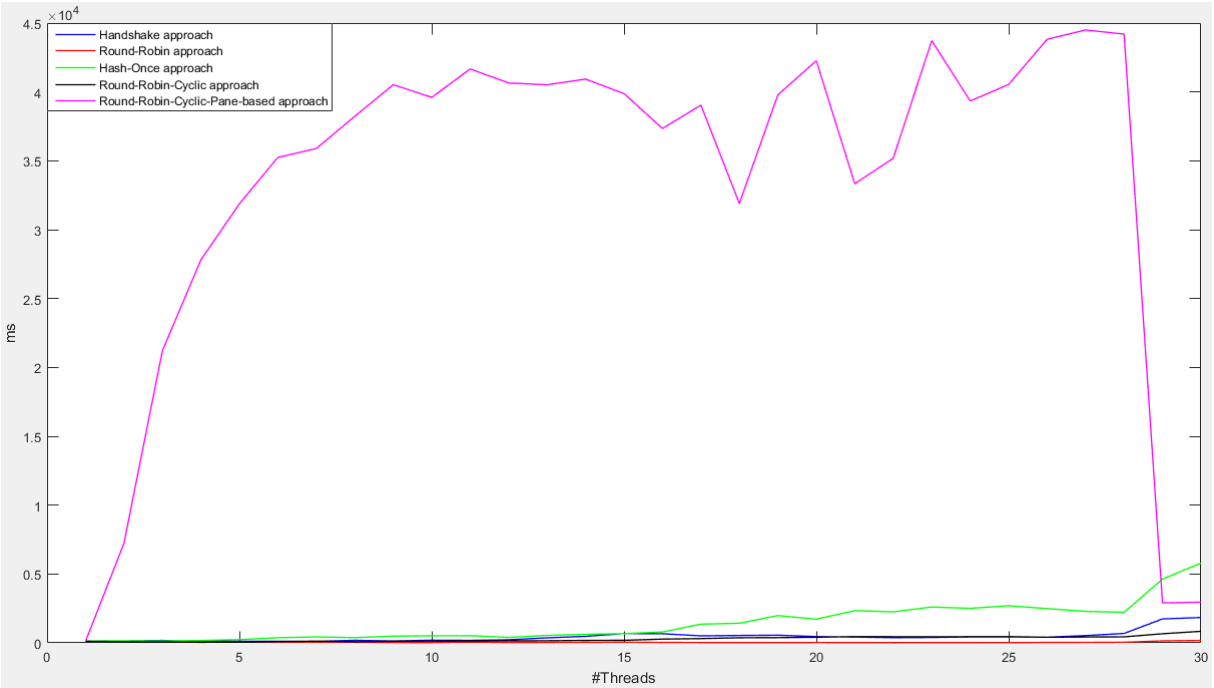


Figure 26: RR-Advantage setting: Latency in ms depending on the number of participating threads.

Assessment of the evaluation results

The results are close to the expectations. The Round-Robin and Round-Robin-Cyclic approaches scale linearly up to 28 threads (after 14 cores with a lower gradient, probably because the used processor has 14 cores with Hyper-Threading). Exactly as expected, the pane-based approach scales linearly as well but with a lower gradient as the other round-robin-based approaches. As predicted, the hash-based approaches scale poorly. The HSA almost does not scale at all. The HOA scales more or less

linearly up to 14 cores and deteriorates afterwards. So it seems that the Hyper-Threading cannot compensate for the increasing amount of communication traffic arising due to the large tuple size in the HOA.

Concerning the latency, we can observe the already familiar pattern: The line-based approaches are better than the respective cycle-based approaches.
The round-robin-based approaches have advantages over the hash-based approaches.
As predicted, the latency of the RRCPA is much higher than the latency of the other approaches. Although the measurements are averaged results of several benchmarks, the RRCPA's latency consistently decreases to a great extent when using more than 28 threads without a significant impact on the throughput. We do not have an explanation for this particular behavior yet.

12 Related Work

There are many papers introducing techniques or data structures that focus on improving specific aspects of streaming aggregation problems; among them several could be combined with the approaches presented in this thesis to enhance certain components of it.

Mouratidis et al. [6] introduce two algorithms that monitor top-k queries over sliding windows. Using a new data structure – the skyband – they avoid re-computing all top-k results from scratch for each of the sliding windows. In case we want to compute top-k results on a data stream as well, this data structure could be used to avoid constantly executing redundant computations, while keeping the advantages of disjoint parallelizations.

Balkesen et al. [4] present two techniques for partitioning windows and tuples to reduce the communication transfer between the participating threads and to decrease the amount of redundant calculations.
One technique is the basis of the pane-based approach presented in Chapter 9.
In contrast to the presented approach:

- Our approach does not need a central processing unit that merges results.
- Our approach does not only run on a ring topology, where each PU is connected to one input thread, but also on a line topology with each PU having a connection to one input thread.
- We analyze the approach and its advantages and disadvantages more in detail.

The second introduced technique based on batching windows has advantages and disadvantages compared to the pane-based technique, depending on the aggregation settings. It could also be combined with our way of gathering expired windows to avoid a central coordinator that needs to merge the results in chronological order.

Depending on the aggregation's settings, especially in large distributed systems it could be a reasonable option to make use of advanced data structures, network topologies or novel techniques. Skip graphs [7] or Chord [8] could be used to locate tuples, windows or panes, reduce the network traffic and increase the communication speed, especially if we are using distributed systems utilizing a large number of machines.
ScaleGate objects, introduced and used in the articles [9] and [10], could help to distribute data stream merging tasks among all participating processing units.
The article [11] also deals with data structures for multiple input streams that could be of use in the cycle-based approaches (HOA, RRA, RRCPA) which can receive and process data from up to P independent data streams.
Also the splitting of queries into subqueries [12] that are being executed in a parallel and distributed fashion might be an option to distribute the work more evenly among a large number of participating processing units as it allows a more fine-grained parallelization.

13 Conclusions

Referring to Chapter 3, the existing approaches for sliding-window streaming aggregations are working with central coordinators that manage the distribution of the incoming data and the merging of the results processed on the participating processing units. Central coordination of the tasks might cause bottlenecks once the managing costs exceed the coordinator's computing capacity, which is the reason why this thesis dealt with the development of disjoint parallelizations of sliding-window streaming aggregation, hence parallelizations without centralized coordination.
Desired properties of these parallelizations include the correctness of the aggregation results as well as a deterministic output behavior, high tuple processing rates and low latencies.
We decided to focus on shared-memory systems, but preferably the parallelizations should be re-usable for distributed setups as well.

We successfully developed five different approaches for the disjoint parallelization of sliding-window streaming aggregation: The Handshake approach presented in Chapter 5, the Round-Robin approach presented in Chapter 6, the Hash-Once approach presented in Chapter 7, the Round-Robin-Cyclic approach presented in Chapter 8 and the Round-Robin-Cyclic Pane-based approach presented in Chapter 9.

The first two, the Handshake approach and the Round-Robin approach, stick to all three original principles adopted by the Handshake join algorithm presented in Chapter 4.
The Hash-Once approach and the Round-Robin-Cyclic approach break with one principle by using a more complex communication structure in order to gain advantages over the previous approaches under certain circumstances.
The Round-Robin-Cyclic Pane-based approach even partially breaks with the principle of avoiding central coordination by committing tuple distribution tasks to the input stream. So as long as this approach does not additionally feature a disjoint parallelization of the tuple distribution tasks, we cannot construe it as a disjoint parallelization of sliding window streaming aggregation.

The most suitable among our five approaches for a given problem depends on different parameters as discussed in the analysis and discussion sections of the approaches (Sections 2 and 3 in the Chapters 5-9).
It depends on the properties of the input stream. Do we have constant or fluctuating timestamp differences of consecutive tuples? Do we have zero, few or many distinct keys? Are the distinct keys distributed evenly or arbitrarily among the data stream's tuples?
It depends on the aggregation parameters. What is the window size and the window advance? Which aggregate function do we want to compute? Do we group tuples by their keys?
It depends on the main memory size. Do we have sufficient main memory for the RRCPA to scale in unfavorable cases?
It depends on the latency demands. Is a very low latency necessary?
It depends on the processing system. Do we use a single machine with multiple cores or a distributed system, a cluster of machines?
Depending on the given circumstances, we have to choose the best approach individually.

There still remain many options to improve the approaches as discussed in the fourth section of the approaches' respective chapters.
In the cyclic approaches (Hash-Once approach, Round-Robin-Cyclic approach and Round-Robin-Cyclic Pane-based approach) dummy tuples could be used to decrease the approaches' latency under certain

conditions (see Section 9.1 and Section 9.4).
For the pane-based approach we considered an alternative structure, which we expect to be superior to the current structure of the RRCPA in most aspects. Additionally, we suggested to split panes into several sub-panes in order to improve the work distribution among the participating PUs. We also proposed to combine the round-robin window distribution with hash functions to enhance the work distribution in cases in which the incoming tuples have many different keys (see Section 9.4). In order to offer a disjoint parallelization for all parts of the pane-based approach, the most promising improvement would be a parallelization of the input stream's tuple distribution tasks (see Section 9.4 and Chapter 12).
As discussed in the *Related Work* chapter, there are also other options to improve certain aspects of our approaches. For example, ScaleGate objects could help to distribute data stream merging tasks among all participating processing units. Parallel data structures as Skip graphs or Chord could be used to locate objects when working with distributed systems containing large numbers of machines. Certain aggregate operations or other stream operations could be accelerated by using specific data structures as the skyband. Different suggestions are presented in Chapter 12.
Since most of the proposed suggestions in the *Related Work* chapter as well as in the *Suggestions for improvement* sections only aim to improve specific aspects of an approach, it always depends on the particular case, whether those suggestions are eventually promising and worthwhile.

14 Bibliography

[1] Stonebraker, Michael, Uğur Çetintemel, and Stan Zdonik. "The 8 requirements of real-time stream processing." ACM SIGMOD Record 34.4 (2005): 42-47.
[2] Teubner, Jens, and Rene Mueller. "How soccer players would do stream joins." Proceedings of the 2011 ACM SIGMOD International Conference on Management of data. ACM, 2011.
[3] Roy, Pratanu, Jens Teubner, and Rainer Gemulla. "Low-latency handshake join." Proceedings of the VLDB Endowment 7.9 (2014): 709-720.
[4] Balkesen, Cagri, and Nesime Tatbul. "Scalable data partitioning techniques for parallel sliding window processing over data streams." International Workshop on Data Management for Sensor Networks (DMSN). 2011.
[5] Schneidert, Scott, et al. "Evaluation of streaming aggregation on parallel hardware architectures." Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems. ACM, 2010.
[6] Mouratidis, Kyriakos, Spiridon Bakiras, and Dimitris Papadias. "Continuous monitoring of top-k queries over sliding windows." Proceedings of the 2006 ACM SIGMOD international conference on Management of data. ACM, 2006.
[7] Aspnes, James, and Gauri Shah. "Skip graphs." Acm transactions on algorithms (talg) 3.4 (2007): 37.
[8] Stoica, Ion, et al. "Chord: a scalable peer-to-peer lookup protocol for internet applications." IEEE/ACM Transactions on Networking (TON) 11.1 (2003): 17-32.
[9] Gulisano, Vincenzo, et al. "Deterministic real-time analytics of geospatial data streams through ScaleGate objects." Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems. ACM, 2015.
[10] Gulisano, Vincenzo, et al. "Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join." Big Data (Big Data), 2015 IEEE International Conference on. IEEE, 2015.
[11] Cederman, Daniel, et al. "Concurrent data structures for efficient streaming aggregation." *Report, Chalmers University of Technology* (2013).
[12] Gulisano, Vincenzo. StreamCloud: An Elastic Parallel-Distributed Stream Processing Engine. Diss. Universidad Politécnica de Madrid, 2012.

All figures and plots have been created using:
Google Drive – Google Drawings
Paint.NET
MATLAB

All different approaches have been implemented in Java and tested using:
Eclipse IDE – MARS
MobaXterm

All tables and texts have been written using:
Microsoft Word
Microsoft Excel

Used hash function:
Murmur Hash (http://murmurhash.googlepages.com/)

The implementations can be found at https://github.com/Zukatah/StreamAggregation.

15 List of abbreviations

ATD ..... *average timestamp difference*
DK ..... *number of distinct keys*
HOA ..... *Hash-Once approach*
HSA ..... *Handshake approach*
LWA ..... *local window advance*
P ..... *number of participating PUs*
PN ..... *overall number of panes*
PNA ..... *pane advance*
PNS ..... *pane size*
PPW ..... *panes per window*
PU ..... *processing unit*
RRA ..... *Round-Robin approach*
RRCA ..... *Round-Robin-Cyclic approach*
RRCPA ..... *Round-Robin-Cyclic Pane-based approach*
T ..... *number of tuples we consider*
TD ..... *timestamp difference*
W ..... *overall number of windows*
WA ..... *window advance*
WPP ..... *windows per pane*
WPT ..... *windows per tuple*
WS ..... *window size*

16 List of figures and list of tables

17 Appendix

17.1 Computing aggregate operations dynamically

This section is based on the short introduction about aggregate operations given in Section 2.4 and the figures of Chapter 2, but gives more detailed information. Since it is not necessary to understand the thesis' content, but offers useful further information that might be of importance under certain circumstances, we decided to put this section into the thesis' appendix.

17.1.1 Out-of-order tuples and order-sensitive aggregate functions

For some order-sensitive aggregate functions, it is only possible to start processing a window w before its expiration if out-of-order tuples are not possible.

One example for such an order-sensitive aggregate operation is a continued fraction. If one window contains tuples (in correct order) with the values 1, 2 and 3, the result of a continued fraction over these values would be $\frac{1}{2} = \frac{1}{6}$. If the tuples with their values are arriving in a different order, let us say 2, 1 and 3, then the continued fraction's result would be $\frac{2}{1} = \frac{2}{3}$. There might still be cases, were we can start the calculation of the aggregate operation before all tuples have arrived, but only as soon as the communication system guarantees us that the order of the tuples we want to use in our calculation is final and possible further incoming extremely belated out-of-order tuples will not be taken into account.

Since the HOA, the RRCA and the RRCPA do not guarantee non-decreasing timestamps for arriving tuples at all PUs, only the HSA and the RRA can be used to compute these aggregate functions dynamically.

In case of certain other aggregate operations, we have to wait and store the tuples until all tuples have arrived, no matter if out-of-order tuples can occur or not.

If we for example calculate the sum of the first, the last and the middle tuple of each window, we do not know which tuple will be the last and the middle one finally.

However, we could already remove some tuples from the window that will definitely not have an influence on the result: If we have already received ten tuples and know that no out-of-order tuple with a timestamp in between the first ten will arrive, we know for sure that none of the second, third and fourth tuple will be the middle or the final tuple.

17.1.2 Trade-offs between throughput and latency

For some aggregate operations there are two different viable approaches for the implementation: One focuses on continuously creating intermediate results as new tuples arrive within the corresponding window. Another one just stores all tuples contributing to a window within it and starts calculating the aggregate operation after all tuples have arrived.

Both approaches may have certain advantages and disadvantages.

As an example we take the median aggregate operation. By applying this operation, we want to get the value, which is separating the higher half of all tuple's values from the lower half.

Basically there are two different approaches:

- We can store all incoming tuples of a window within a list (or a similar data structure) and after the expiration of the window we sort the tuples' values and take the one in the middle of the list.

- We can continuously add incoming tuples into a *TreeMap* – a data structure, which automatically sorts all contained tuples by their values – and in the end we just have to retrieve the value of the tuple in the middle of the *TreeMap*.

Both approaches' runtime is in $O(n * log(n))$ – where n is the number of tuples.
The first approach has the disadvantage that it has to do almost all the work in the end after the window's expiration, which may increase the latency after which we finally have the result.
The second approach has the disadvantage that, although the asymptotic runtime is the same, there is more overall work, since the continuous updating process of the *TreeMap* costs more time (higher constants of the asymptotically equally expensive operations) than the one sorting phase of the list. A few tests showed that approximately three times as much computation time is needed.

17.2 Example figure for the Handshake approach

Further below, there is an example execution of the Handshake approach. Three PUs are participating and the hash function used to determine the window responsibilities is:
$ResponsiblePU(window\ w) = w.timeLimit\%P.$
So the responsible PU for the window with time limit 5 is $5\%3 = PU2$, the PU responsible for the window with time limit 10 is $10\%3 = PU1$ and so on.

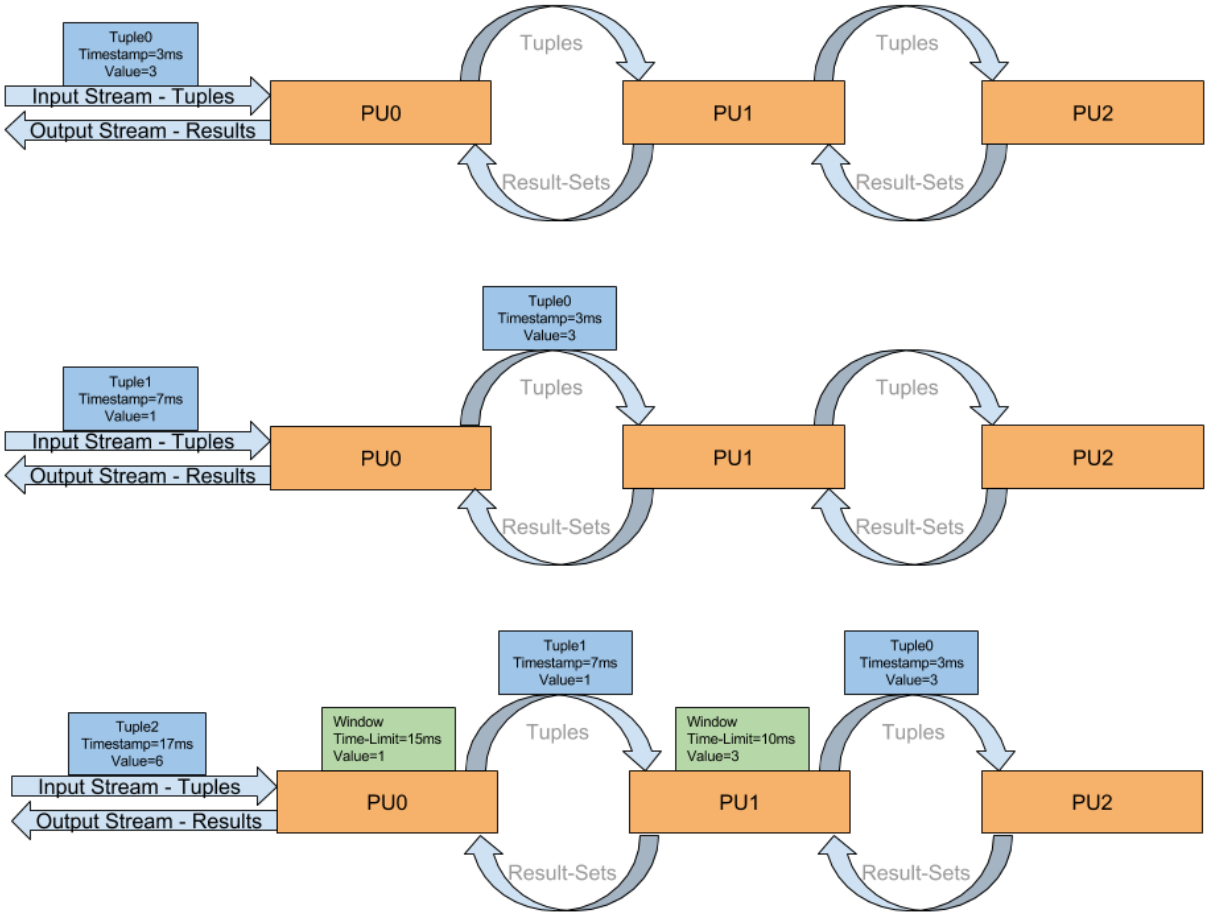


Figure 27: Example execution of the HSA, part 1. Aggregate function: Sum; WS: 10ms; WA: 5ms; group by keys: false.

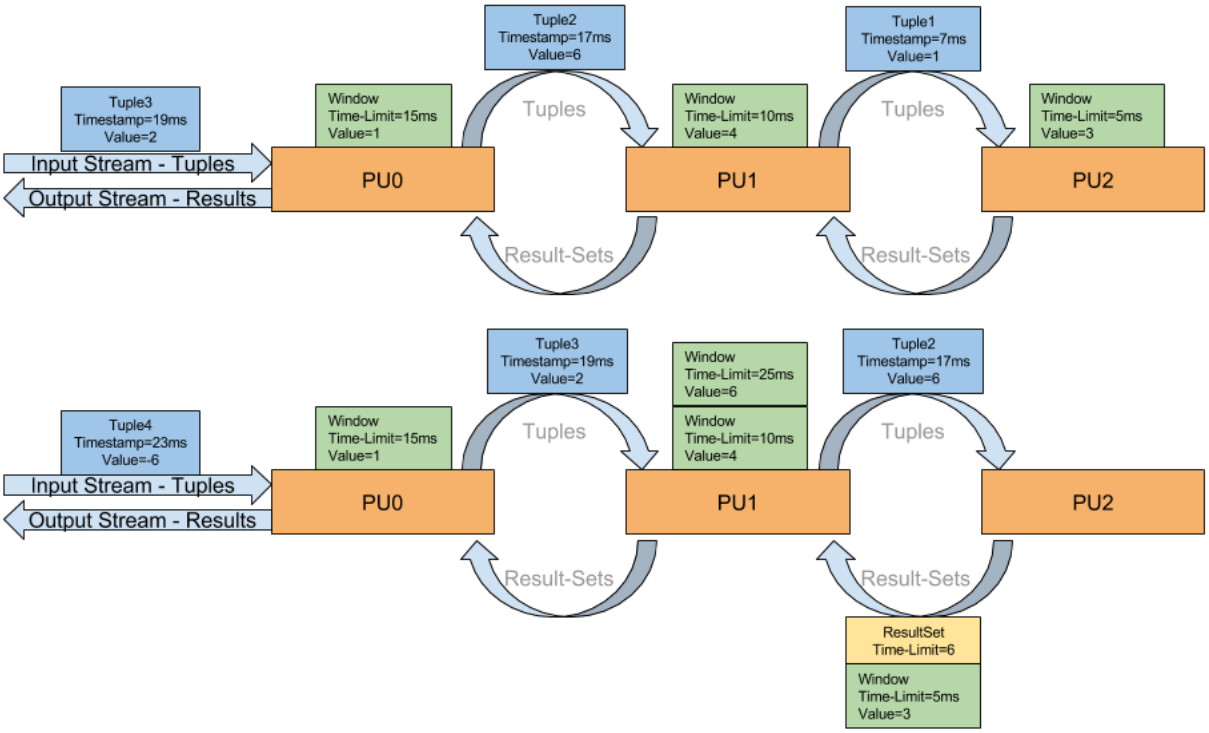


Figure 28: Example execution of the HSA, part 2.

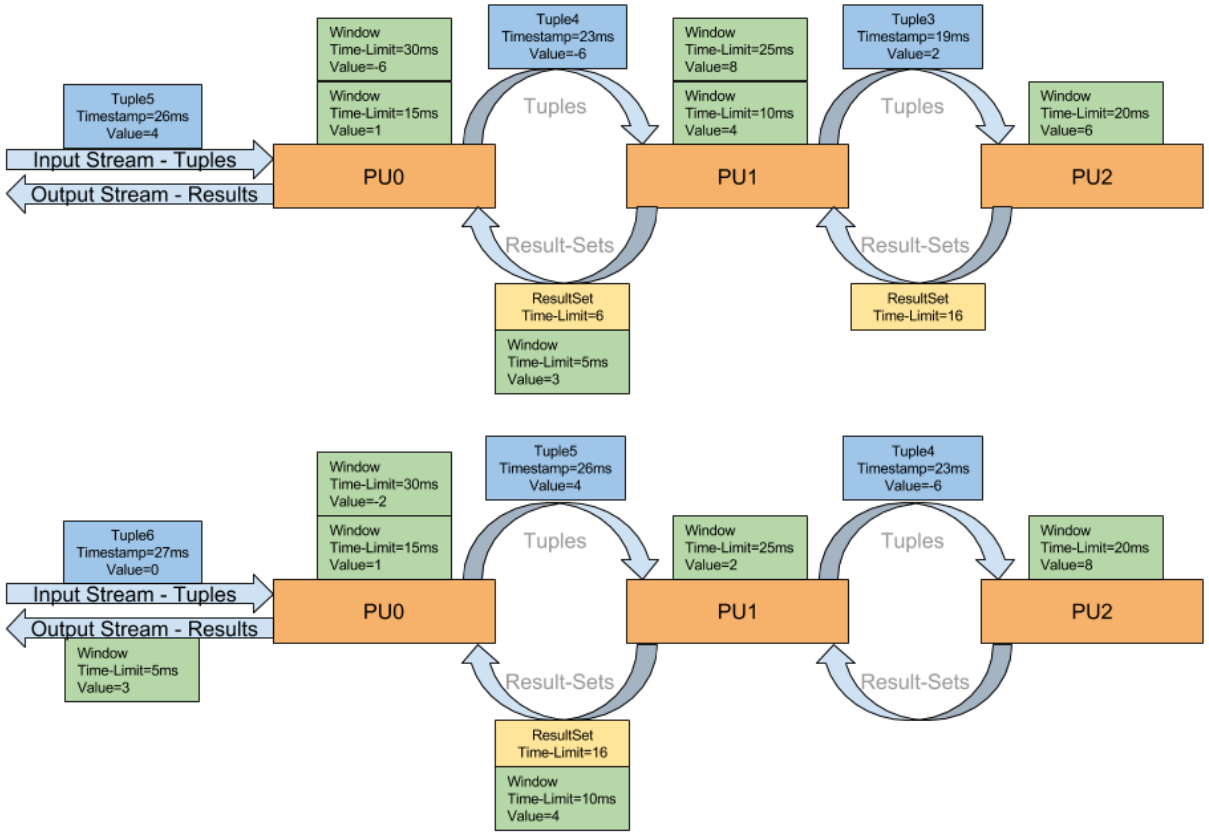


Figure 29: Example execution of the HSA, part 3.

17.3 Zoomed plot of the RRCPA-Advantage setting

The next figure is equivalent to the plot depicting the RRCPA-Advantage setting in the evaluation chapter, but it is vertically zoomed to make the 4 non-pane-based approaches more readable.

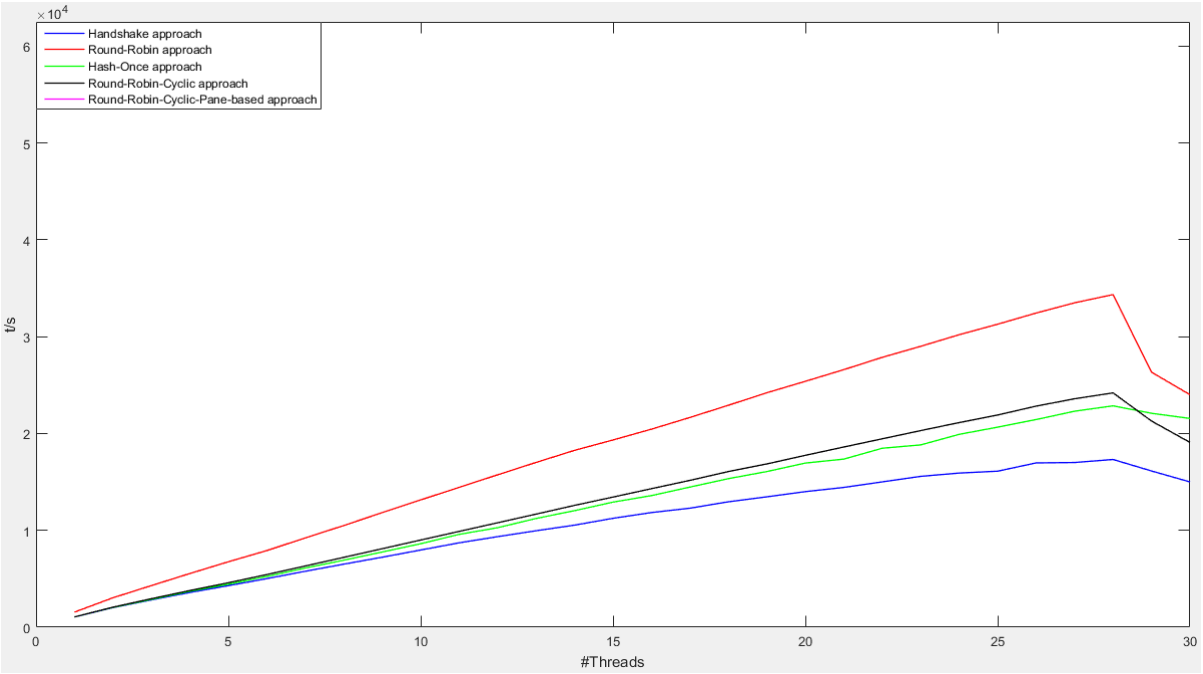
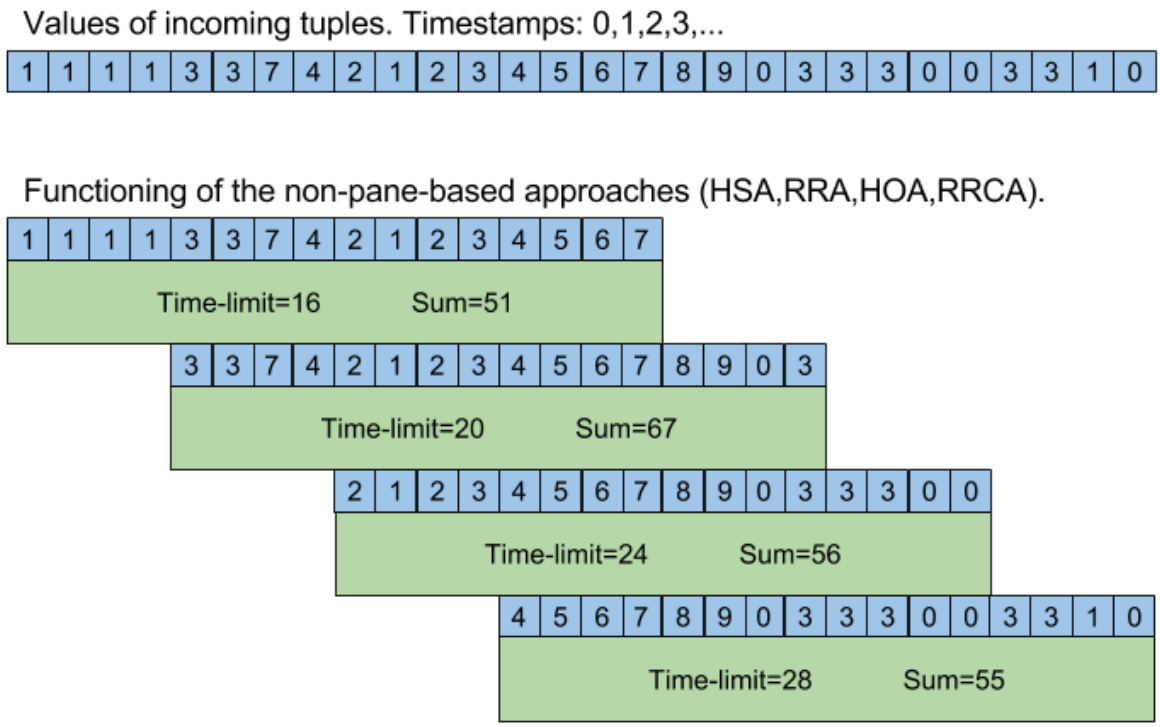


Figure 30: Processing speed of the RRCP-Advantage setting as in Figure 23 (zoomed)

17.4 Example execution – non-pane-based vs. pane-based

The next figure depicts the differences between a pane-based and a non-pane-based streaming aggregation. Tuples are colored blue, windows are colored green and panes are colored red. It demonstrates that in the pane-based approach, redundant computations are avoided.



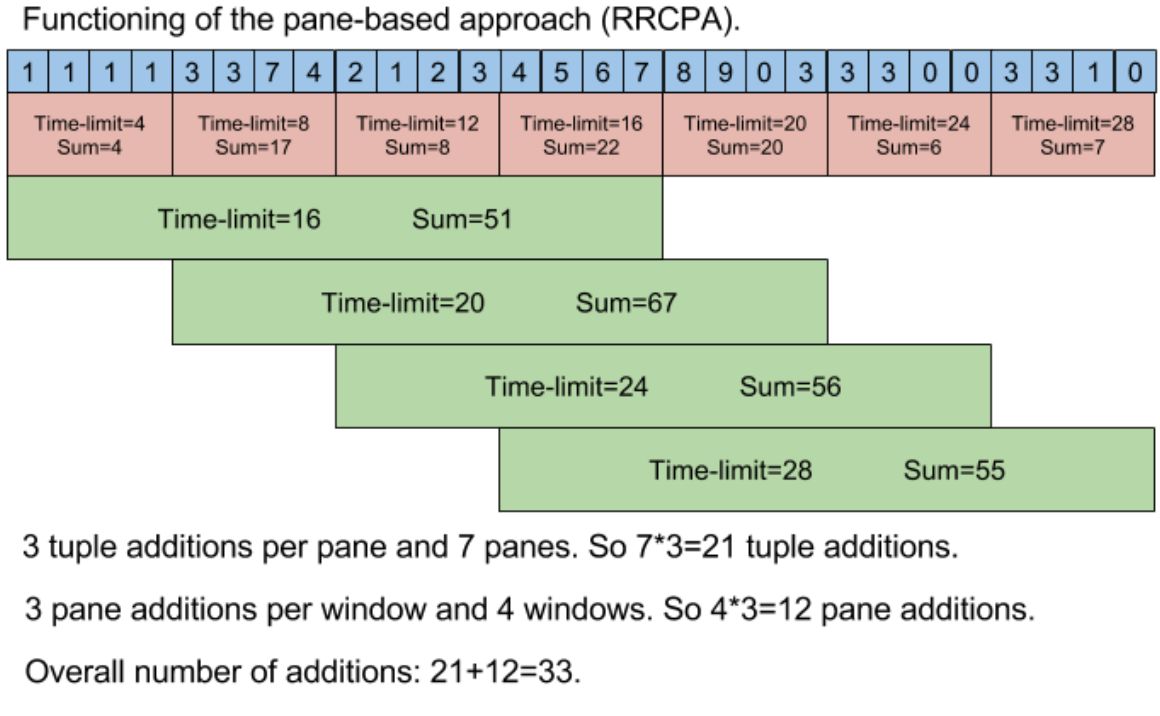


Figure 31: Example execution: non-pane-based vs. pane-based; aggregate function: Sum, WA=4, WS=16, DK=1