Technische Universität Berlin
Kungliga Tekniska Högskolan Stockholm
Swedish Institute of Computer Science/Swedish ICT

Master Thesis

Rich window discretization techniques in distributed stream processing

TRITA-ICT-EX-2015:28

Jonas Traub

Matriculation #: 358291 (TU-Berlin) / 910522-T312 (KTH-Stockholm)

Supervisors: Volker Markl (TU-Berlin), Seif Haridi (KTH-Stockholm)
Advisors: Asterios Katsifodimos (TU-Berlin), Paris Carbone (KTH-Stockholm)

11.04.2015

Erklärung (Declaration of Academic Honesty)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

I hereby declare to have written this thesis on my own and without forbidden help of others, using only the listed resources.

____
Datum

____
Jonas Traub

Contents

List of Figures

List of Listings

List of Algorithms

List of Tables

List of Acronyms

API	Application Programming Interface	1
CEP	Complex Event Processing	22
CFA	Central Final Aggregation	53
CWD	Central Window Discretization	52
D-Stream	Discretized Stream	4
DAG	Directed Acyclic Graph	14
DFA	Distributed Final Aggregation	57
DPA	Distribruted Pre-Aggregation	53
DWD	Distributed Window Discretization	57
EE	Enterprise Edition	22
EP	Eviction Policy	34
ER	Entity Relationship	A.19
FIFO	first in first out	14
gcd	greatest common divisor	26
GFS	Google File System	13
HDFS	Hadoop Distributed File System	10
IBM	International Business Machines	4
IDC	International Data Corporation	10
ITA	Input Triggered Approach	28
JVM	Java Virtual Machine	72
MR	Map Reduce	9
NTA	Negative Tuples Approach	28
POJO	Plain Old Java Object	22
RDBMS	Relational Database Management System	13
SICS	Swedish Institute of Computer Science	56
SPL	Stream Processing Language	4
SQL	Structured Query Language	13
TP	Trigger Policy	34
UDF	User Defined Function	1
UML	Unified Modelling Language	A.12

1 English Abstract

Stream processing differs significantly from batch processing: Queries are long running, continuously consuming data from input streams and, in turn, producing output streams. A stream can be defined as a data source that infinitely produces/emits data-items. Hence, it is impossible to store all the history of emitted items in order to query them. In order to produce answers to a given query over an infinite set of items, a stream is split (or, *discretized*) into smaller subsets of data-items, called *windows*. Consecutively, an operation such as a Join, Aggregation or a Reduce operator is applied on the discretized window.

In this thesis, we designed and implemented highly expressive means of window discretisation in the Apache Flink stream processing engine. The rules for the discretization of a stream are called *windowing policies*. Our discretization operators are event-driven, and triggered by data-item arrivals. When a data-item arrives, *trigger* and *eviction policies* are notified. A trigger policy specifies, when a reduce function is executed on the current buffer content. An eviction policy specifies when data-items are removed from the buffer.

This thesis is the first to our knowledge to propose the use of windowing policies in the form of *User Defined Functions (UDFs)* in a data parallel execution engine. Expressing windowing policies as *UDFs* results in very high expressivity and flexibility. Thus, very complex queries can be defined, going much beyond the predefined count-, time-, punctuation-, and delta-based windows. This thesis is also the first to implement a streaming *Application Programming Interface (API)* that allows user-defined trigger and eviction policies in a data-parallel stream processing engine.

2 Deutscher Abstract

Die Datenstromverarbeitung (*stream processing*) unterscheidet sich signifikant von der Stapelverarbeitung (*batch processing*): Programme haben lange Laufzeiten, konsumieren Daten kontinuierlich von Eingabeströmen und produzieren im Gegenzug Ausgabeströme. Ein Datenstrom kann definiert werden als eine Datenquelle, welche kontinuerlich Datenelemente produziert/ausgibt. Daher ist es unmöglich die gesamte Historie der ausgegebenen Datenelemente für eine Abfrageausführung zu speichern. Um Ergebnisse für eine gegebene Anfrage über eine unendliche Menge von Datenelementen berechnen zu können, wird ein Datenstom in kleinere Teilmengen von Datenelementen, sogenannte *Fenster*, unterteilt (diskretisiert). Eine Operation wie ein Join, eine Aggregation oder eine Reduce-Funktion werden forlaufend auf Fenster angewendet.

Im Rahmen dieser Thesis haben wir eine äußerst ausdrucksstarke Diskretisierungsmethode entworfen und in der Apache Flink Datenstromverarbeitungsplattform implementiert. Wir nennen Diskretisierungsregeln *Fenster-Policies. Trigger- und Evictionpolicies* werden über ankommende Datenelemente banchrichtigt. Eine Triggerpolicy spezifiziert, wann eine Reduce-Funktion auf den aktuellen Pufferspeicher angewendet und ein Ergenis ausgegeben wird. Eine Evictionpolicy gibt an, wann Datenelemente aus dem Pufferspeicher entfernt werden.

In dieser Thesis werden erstmals Fenster-Policies in Form von benutzerdefinierten Funktionen in einer datenparallelen Ausführungsumgebung eingeführt. Die Möglichkeit benutzerdefinierten Funktionen als Fenster-Policies zu verwenden hat eine große Ausdrucksstärke und bietet eine hohe Flexibilität, die weit über die vordefinierten Policies, wie zum Beispiel zähler-, zeit-, interpunktions- und deltabasierte Trigger- und Evictionpolicies, hinaus geht. Zusätzlich stellt diese Thesis erstmals eine Applikationsprogrammierungsschnittstelle vor, welche die Verwendung von benutzerdefinierten Trigger- und Evictionpolicies in einer datenparallelen Plattform (namentlich, Apache Flink) ermöglicht.

3 Introduction

There are many data processing engines, which are especially designed for stream processing such as Storm[1] [54], Naiad[2] [57] and Cloud Dataflow[3]. Beside this, many big data processing engines, which were not designed for stream processing, integrate additional features to enable stream processing besides their core functionality. Popular examples are Flink[4] [5] and Spark[5] [70].

In *Fundamentals of Stream Processing* [6], Andrade et al. describe a *streaming data source* as the producer of a *data stream.* Such a stream usually is a sequence of data-items, also called tuples. *Data-items* are the smallest atomic portion of data which may be processed by a query. Data-items possibly contain multiple attributes described by a schema. A data-stream is possibly infinite and consists of tuples, which share a common schema. [6]

Stream processing differs significantly from batch processing: Queries are long running (theoretically even infinitely running), continuously consume data from inbound streams and produce an output stream. Thereby, the data rate is controlled externally, which means data is pushed to the processing engine from the outside and not pulled as needed from storage.

A stream source is a data source that possibly emits an infinite number of data-items. Thus, it is not possible to store the whole history of emitted items. Moreover, evaluating a query over an infinite stream would never emit results due to blocking operations such as joins and synopses [4], that need to have a view of the full input dataset in order to produce results. Thus, a stream is usually split into smaller finite chunks of data, which are called *windows.* Often an aggregation, such as calculating the sum or the average, is done for each *window.* As the result is only calculated over a portion of the stream it is said to be an *approximate summary* [41], also called *synopse* [7] or *digest* [71].

[1] Apache Storm: `https://storm.apache.org/`

[2] Microsoft Naiad: `http://research.microsoft.com/en-us/projects/naiad/`

[3] Google Cloud Dataflow: `https://cloud.google.com/dataflow/`

[4] Apache Flink: `http://flink.apache.org/`

[5] Apache Spark: `http://spark.apache.org/`

There are three basic window types, namely *tumbling*, *sliding* and *hopping*. Systems like Storm, Naiad, Cloud Dataflow, Spark and Flink, provide the possibility to define and compute all three kinds of windows.

Tumbling means that a data-item is contained in exactly one window.

Sliding means that at least one data-item will be contained in multiple windows.

Hopping means that at least one data-item lies between two consecutive windows and is excluded from both.

Micro-Batching vs. Streaming. A streaming query evaluation engine is not always required in order to evaluate streaming queries; stream processing has been shown to work under certain limitations even on top of batch systems. To do that, one has to treat a streaming computation as a series of deterministic micro-batch computations. *Micro-batches* are sequences of data chunks, that are made periodically from small time intervals. The streaming computation can then be computed as a series of jobs on a batch-processing system. For instance, *Discretized Streams (D-Streams)* [70], a streaming abstraction built on top of Spark, can perform micro-batch processing. However, window sizes can only be defined as multiples of the micro-batch granularity (slices to time windows). This limitation, stems from the fact that Spark implements a batch execution engine: the execution of a job graph is done in stages, and the outputs of each operator are materialized in memory (or disk) until the consuming operator is ready to consume the materialized data. Thus, *true* streaming is not possible. Apache Flink, however, implements a streaming/pipelined execution engine [14]. This entails that the whole job graph is deployed in the cluster and once an operator emits a data-item, that data-item can be immediately forwarded to the next consumer operator. In Flink's version 0.7, windows could be defined based on the data-item count (sequence number) or time.

Rich Window Discretization. The *International Business Machines (IBM) Stream Processing Language (SPL)* is a query language for defining queries over data streams that are executed on the *IBM* InfoSphere streaming platform [44]. *SPL* introduced separate trigger and eviction policies for the first time. These policies allow a user to specify different rules for the expiration (or, eviction) of data-items and the emission (or trigger) of windows. *IBM* provides a predefined set of policies containing time-, count-, punctuation-, and delta-based rules [27, 28, 44]. The expressivity in the definition of windows is crucial when a stream processing engine is selected to solve a specific problem. While in a lot of use cases, purely time based discretization is sufficient, there are also use cases where the system has to split the stream into windows with regards to query-specific data-item properties. For instance to react on concept drifts [63]. Unfortunately, *SPL* does not allow a queries with user-defined policies; as a result, examples like the detection of concept drifts cannot be expressed using *SPL*.

Performance Optimization. Another crucial point is the performance of a stream processing engine. In the *Catalog of Stream Processing Optimizations* [46], Hirzel et al. presented twelve different kinds of optimizations that apply in stream processing. The cost that one has to pay in order to have more expressive windowing semantics often hinders optimization possibilities. For instance, a very common optimization is to share computations among multiple aggregations on different windows. However, a set of user defined windows (expressed in a turing-complete language like Java, or Scala) cannot be easily analyzed and optimized. Thus, a good stream discretization model should serve a good trade off between the ability to apply performance optimizations and providing expressive windowing semantics.

3.1 Motivation

As mentioned in the previous section, current systems are designed to perform either stream (e.g. [54, 57]), or batch processing (e.g. [5, 70]). Interestingly, it turns out that batch processing can be simulated by streaming; a batch of data-items can also be seen as a finite stream of data. Thus, a streaming execution engine can easily serve as an execution engine for batch computations.

Nowadays data analytics very often requires stream and batch computations to be combined in order to perform a computation. It is common that queries use historical data stored in a filesystem along with streaming data arriving at a fast pace. For instance, a concept drift [63] can be detected by a streaming query and, once the drift is detected, the actual concept has to be recomputed from the historical data together with the newly arrived data. To implement such an query using historical and streaming data, an analyst needs two execution engines: one for the batch (historical) computations and one for the streaming computations. This entails that the analyst has to maintain two code variations and deploy multiple systems for a single query. Moreover, data cannot be easily shared between batch and stream processing computations. As a result, databases and/or filesystems are used as means of exchange (or, storage) of intermediate results between multiple execution engines.

To solve these issues, systems like Spark and Flink, implement both stream and batch processing capabilities. However, Spark processes streams only as micro-batches. Therefore, windows in Spark have to be multiples of the micro-batch granularity, so that the window result can be derived from the micro-batch computations. On the other hand, Flink implements true (non micro-batch) streaming capabilities. However, the programming model for expressing windows (at v0.7) is very limited: only count- and time-based windows are allowed.

Part of *SPL*'s success, is its ability to express very complex windows through flexible eviction and trigger policies. As it turns out, having the ability to split a stream in query-specific, user-defined windows, can enable programmers to specify more complex algorithms (such as the example of calculating a concept drift). However, no streaming computation engine so far gives the ability to define user-defined window discretization policies. Today's stream processing systems can express windows only based on time, data-item counts, punctuations and deltas between data-items. These cannot be used to express complex queries.

This thesis aims at enabling:

- (true) streaming and batch capabilities on a single system
- user-defined window discretization capabilities to allow complex algorithms that perform a mix of batch and streaming computations to be expressed in a single programming model and run on a single execution engine.

None of the present systems [3, 16, 19, 43, 44, 45, 57, 70] can be programmed to evaluate complex queries like the concept drift example mentioned above. Instead, programmers have to use very low level systems like Storm [54] that require the programmer to fully understand and hard code all details of windowing semantics, and to have very deep knowledge of distributed systems, cluster computing, and query optimization.

Thesis Objective. In this thesis, we design flexible windowing semantics that allow programmers to express very complex window-based streaming queries, without the additional programming requirements needed in Storm-like systems. We generalize rules for window discretisation, so that the logic for such rules can be provided in an user-defined fashion without introducing further implementation overhead to the programmer.

3.2 Contributions

In this thesis we design and implement highly expressive window discretization primitives in the Apache Flink stream processing engine. The contributions include the specification and implementation of the *API* as well as the architecture and development of the discretisation operators that are running underneath. We draw *SPL*'s eviction and trigger policy primitives as they provide high expressivity in window specification, while we generalize those primitives such that we allow user-defined windowing policies. Such windowing policies are currently not allowed *by any* stream processing system.

The contributions of this thesis go as follows:

1. This thesis is the first to propose expressing windowing policies as *UDFs*. The ability to express windowing policies as *UDFs*, results in very high expressivity and flexibility. As a result, very complex queries can be defined, going much beyond the predefined count-, time-, punctuation-, and delta-based windows that other systems provide [3, 16, 19, 43, 44, 45, 57, 70].

2. We are the first to implement a streaming *API* that allows user defined *eviction* and *trigger* policies in a data-parallel stream processing engine.

3. We provide the basis for streaming pre-aggregate sharing optimizations across windows as well as an architecture for parallelizing streaming window discretization and aggregation.

4. The results of this thesis were contributed and are now included in the open-source Apache Flink project (since v 0.8). The results of this thesis have been thoroughly tested and are in use by a number of Apache Flink users.

3.3 Thesis Outline

Section 4. In section 4, we present the backgrounds of parallel data processing and differentiate types of parallelization, including the Map Reduce programming model, which is the most widely used for data parallel processing. Furthermore, we provide an introduction to stream processing and window types. The section concludes with the presentation of the Apache Flink software stack for big data processing.

Section 5. Section 5 provides a survey through related publications. The section is split in two subsections. First, we provide an overview of available stream processing engines and languages. Thereby, we describe the windowing semantics of the two most closely related systems (namely *IBM SPL* and Apache Flink) in more detail. We conclude with a survey through different means of optimizations for stream processing.

Section 6. In section 6 we present our architecture for rich window discretization techniques. We describe the concept discretization operators and window discretization based on trigger and eviction policies. Moreover, we show several query examples and the *API* design for window discretization.

Section 7. We designed our solution in a way, that it serves well possibilities for parallelization. In section 7, we provide two different possible ways to parallelize our discretization operators. While the first is general applicable, the second serves an higher parallelization degree for some types of queries.

Section 8. When a stream-processing platform is selected to solve a query, the performance of the engine might be one of the most important criterias. Section 8 first points out the dimensions for optimizations, then it categorizes queries regarding the used windowing policies. Several optimizations are presented, which apply for certain categories of queries.

Section 9. In the evaluation section we point out the expressiveness of our solution in comparison to other systems. Furthermore, we depict test results for the throughput of our window operator and show that it scales linear with the amount of processed data.

Conclusion. On page 75, we provide an overview of the thesis results.

Appendix In the appendix, we present a detailed documentation of the implementation of our window discretization operators and policies. Additionally, we present pre-defined policies which can serve examples for the implementation of user-defined policies.

4 Background

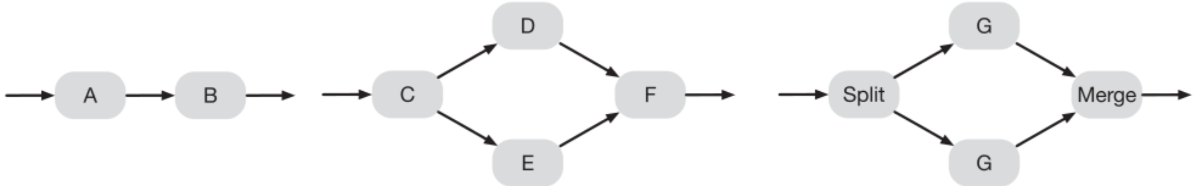
In this chapter, we present the backgrounds of this thesis. In section 4.1 we present the data parallel processing model. Data parallelism becomes a requirement when the analysis on a single machine is unfeasible due to the sheer amount of data to process. In section 4.1.1 we present *Map Reduce (MR)*, which is the most widely used programming model for data parallel applications. Unfortunately, *MR* has several limitations and disadvantages, presented in section 4.1.2. Moreover, it cannot be used to process streams. We provide an introduction to stream processing, including definitions and window types in section 4.2. Section 4.3 concludes with a presentation of the Flink big data processing platform on which we implemented the discretization techniques presented in this thesis.

4.1 Parallel Data Processing

The sheer amount of data nowadays makes data processing and analysis on single machine unfeasible. To overcome the single-machine limitations, parallel data processing becomes a requirements [30]. Multiple types of parallelism have been proposed in the past, the most important of which are depicted in Figure 1, namely pipeline-, task-, and data-parallelism.

Pipeline-parallelism. Pipeline parallelism is a parallelization technique where one operation is executed after each other concurrently; as a result each item or set of items that are produced by one task, are directly fed intor the task that follows. Since the two tasks can process data concurrently, they can be scheduled into two different cpus/machines.

Task-parallelism. Task parallelization (b) means, that there are different operation, to be executed on the data-items. This (independent) operations on the same data-item can be executed at the same time. In the depicted example, these are the operations D and E. [46] Since pipeline- and task-parallelism are limited by the number of tasks



(a) Pipeline-parallel A || B. (b) Task-parallel D || E. (c) Data-parallel G || G.

Figure 1: Figure from [46], showing pipeline, task, and data parallelism in stream graphs.

which are present in a program, they they cannot form a basis for massively parallel data processing [30].

Data-parallelism. In data-parallel processing [17], different data-items (or sets of data-items) are distributed to different cpu cores or machines that are running the same task; the same operator, is present in multiple cpus/machines which run in parallel [30].

The main idea distinguishing data parallelism from task- and pipeline-parallelism is, that there are multiple instances of the same operator; in our example, the operator G. An input stream is split, such that each instance of G processes different data-items using the same logic. Afterwards the outputs of the parallel instances of G are merged again [30, 46].

4.1.1 The MapReduce Programming Model

Map Reduce (MR) [25, 26] is a programming model for data-parallel processing. *MR* programs are inherently parallel [25]. *MR* is optimized to work with distributed file systems like the *Hadoop Distributed File System (HDFS)* [60]. Most difficulties which come up in large clusters are handled by a *MR* framework. *MR* is the most widely used programming paradigm in today's big data platforms. It influenced and is part of many parallel batch and stream processing engines, such as Spark, Flink, and Hadoop [66, 68].

We will often refer to map and reduce functions in this thesis, when it comes to parallel data transformation and aggregation. In this section we will focus on the programming model itself and the motivation behind it. Finally we will show an example and point out some limitations and disadvantages of *MR*.

During the last two decades the capacity of hard drives has increased massively. Also the amount of data generated on each day is much bigger than a few years ago. Sure, it is very difficult to measure the whole amount of data in the "digital universe" but an *International Data Corporation (IDC)* estimate [36] put the size at 281 exabytes ($2.25 \cdot 10^{21} bits$) in 2007. At this time it already was expected to grow to be ten times more in 2011. [68]

One problem is, that the reading and writing speed of hard drives has not increased that much than the storage per unit did. This results in a higher required time to read the full data from a disk. The idea for the solution is simple: If it is possible to read parts of the data from several hard drives in parallel, there will be much less time required. *HDFS* [60] in an example for a solution providing this functionality in a computing cluster. Another problem is the increased total amount of data. Massive parallel processing is required to analyse complete data sets. The *MR* programming model allows to address both issues

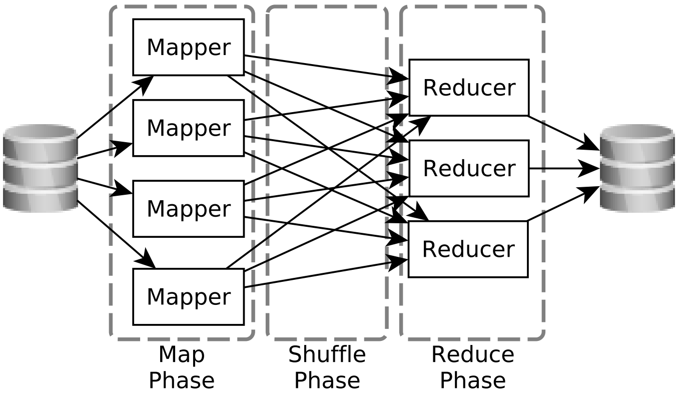


Figure 2: The execution flow of a Map Reduce job.

with its parallel nature and has evolved to the most widely used paradigm in today's big data analysis solutions.

From a programmer's perspective, in a *MR* application, the execution is split in two phases. A map and a reduce phase. In between there is a shuffling, done by the underlying *MR* framework. Figure 2 depicts the execution flow of a *MR* job. As you can see, multiple mappers and reducers can run in parallel within the respective phase, while there is a synchronization point between the phases. It is a pipelined execution, where on phase gets executed after the previous is completed. This is a contrast to continuous long standing processing of streams.

The data model of *MR* is based on key-value-pairs. We say a key value pair is element of $(K \times V)$, where K is the set of keys and V is the set of values. We use the indexes m, r, and o to differentiate between the sets of keys and values present in the input to the map phase, input to the reduce phase, and the output. The star symbol indicates either a list of values out of a set or a list of tuples, when placed after the closing parentheses [53].

Table 1 depicts an overview of the used *User Defined Functions (UDFs)* and the processing phases. The map and the reduce phase can be seen as higher order functions. They take the user-defined mapper and reducer implementations as parameter and apply them to each input tuple. The result of the phase is the concatenation of the results from the runs of the *UDFs*. The shuffling phase takes the output tuples produced by the map phase and groups them by key. For each key, one output tuple is produced, containing the key and a list of values, which is the concatenation of the values from the grouped tuples.

Functions	**Framework**			**User Defined**	
	Map	Shuffle	Reduce	Mapper	Reducer
Input	$(K_m \times V_m)^*$	$(K_r \times V_r)^*$	$(K_r \times V_r^*)^*$	$(K_m \times V_m)$[53]	$(K_r \times V_r^*)$[53]
Output	$(K_r \times V_r)^*$	$(K_r \times V_r^*)^*$	$(K_o \times V_o)^*$	$(K_r \times V_r)^*$[53]	$(K_o \times V_o)^*$[53]

Table 1: Inputs and outputs of functions and processing phases in Map Reduce.

Now, after we've presenting the theoretical backgrounds, let's have a look on a concrete example. Table 2 shows a data transformation of a word count application using exactly the input and output formats specified in Table 1. The input file is split by lines, so that an input tuple contains the line id as key and the content of the line as value. The mapper splits the input values at each space-character. For each split, one tuple is produced as output, containing the individual word as key and 1 as value. The reduce function sums up the values per key. The final output are key-value-pairs containing a word as key and the number of the occurrences of the word as value.

While the simplicity of the user defined functions and the parallelism are big advantages of *MR*, there are also downsides of the approach. Within a single job, there is always only one map and one reduce phase. As the output and the input of the job are both key-value-pairs, it is possible to run a transformation consisting of multiple jobs. Anyhow, running multiple jobs, one after each other, is not only inconvenient. It also comes with the cost of serialization of the intermediate results between jobs. Usually, *MR* frameworks also serialize the outputs of each individual mapper and reducer and write them to disk, to be able to execute following operations again in case of failures. The frequent serialization is not always necessary, but can cause a massive increase of the computation time.

	Input	**Output**
File Line 1:	Beer Beer Tea Coffee	
File Line 2:	Tea Tea Beer Tea	
Map 1:	(1, Beer Beer Tea Coffee)	[(Beer,1),(Beer,1),(Tea,1),(Coffee,1)]
Map 2:	(2, Tea Tea Beer Tea)	[(Tea,1),(Tea,1),(Beer,1),(Tea,1)]
Reduce 1	(Beer,[1,1,1])	(Beer,3)
Reduce 2	(Coffee,[1])	(Coffee,1)
Reduce 3	(Tea,[1,1,1,1])	(Tea,4)

Table 2: Word Count Example: Sample data transformation.

4.1.2 Database Systems vs. MapReduce

Since its publication in 2004, MapReduce has been used to implement various data analysis tasks, like joins and aggregations, on which database systems excelled. While MapRedce is easy to program and is very versatile, it still poses a number of limitations compared to database systems [9].

Compared to *Relational Database Management Systems (RDBMSs)*, four main disadvantages of *MR* must be pointed out:

1. *MR* is a programming model. It is not a query language like the *Structured Query Language (SQL)*. When using *MR*, programmers must specify the how to do the data transformation which leads to the desired result. In *SQL*, the programmer only describes the result and the *Relational Database Management System (RDBMS)* selects and applies the required transformations automatically.

2. Due to the massive use of *UDFs* and the possible split of tasks in multiple jobs, it's much harder to apply data flow optimizations in *MR* frameworks, as it is in *RDBMSs*. Hueske et al. [47] argue that classical *RDBMS* optimizers have, if they use *UDFs*, very strict templates for them. The main challenge is then to estimate whether it is beneficial to reorder operators in a parallel data flow. In opposite to this, in *MR* like applications already the purpose of the *UDF* is unknown and it is already a problem to verify that a reordering of operations is possible. Nevertheless, the authors show how a reordering in data flow programs that consist of arbitrary imperative *UDFs* can look like and which conditions have to be fulfilled to allow such an optimization.

3. Some operations like joins are complex to implement in *MR*. [68] *RDBMSs* often provide different kinds of implementations for the same logic operations and can select the best performing alternative depending on the situation automatically. In *MR*, the programmer has to decide which type of join to use and statically implement it in the *MR* program.

4. *MR* is heavily dependent on a distributed file systems, where a lot of the complexity is moved. Popular examples for such file systems are *HDFS* [60] and *Google File System (GFS)* [40].

Unfortunately, *MR* does also not work for the processing of data streams without changes. Due to the synchronization points between the different execution phases, *MR* would never produce a result in case the data source is a stream. Streams are possibly infinite, which means that the map phase can never tell that it is completed and the reduce phase will never start. Anyhow, a stream is usually split into smaller finite chunks of data, which are called *windows*. Such finite chunks can then be processed by a *MR* application.

4.2 Stream Processing

The stream programming model is powerful, widely used, and implemented by many systems [30]; for example in Nephele [14], which is the basis of nowadays Apache Flink runtime, Naiad [57], and Storm [54]. In addition, streaming like behaviour on top of a batch execution engine can be archived using *Discretized Streams (D-Streams)* [70], which is done by Apache Spark.

In common with *MR* is that programs are composed from sequential code blocks. Such code blocks can be executed in parallel, which means that they run in multiple instances at the same time, processing different data-items. Thus, the different instances are independent for each other [30].

The sequential code blocks are arranged in a graph, which is possibly cyclic [46], even though, many systems are limited to *Directed Acyclic Graphs (DAGs)*. The code blocks (henceforth called operators) are the vertices in this graph and data-items are forwarded along edges. This generalizes the model of *MR*. While in *MR* operators have to be either a map or a reduce function, operators can apply arbitrary data transformations in the streaming model [30].

4.2.1 Definitions

In this thesis we will use the common vocabulary introduced by Hirzel et al. [46] and inspired by Gamma et al. [35] and Fowler et al. [33].

Data Stream *A conceptual/possibly infinite sequence of data-items which comes from a stream source. Streaming systems implement streams as first in first out (FIFO) queues.*

Stream Source *A stream source is a data source that possibly emits an infinite number of data-items.*

Operator *A code block , which consumes data-items from incoming stream(s) and produces data-items on outgoing stream(s). Thereby it performs a continuous data transformation.*

Query *A streaming query consists of operators which are vertices in a graph. The edges of the graph represent the data flow between operators. A streaming query combines multiple operations to a data transformation flow. Queries are also called applications.*

Data-item *The smallest unit of data, which is processed by the streaming query, is called* data-item. *It is also the smallest unit of communication along edges in the query. Anyhow, data-items can consist of several attributes.*

Window *A finite subsequence or chunk of data-items from a stream.*

4.2.2 Window Types

Evaluating a query over an infinite stream would never emit results due to blocking operations such as joins and aggregations. Thus, a stream is usually split into smaller finite chunks of data, which are called windows. We differentiate three types of windows:

Tumbling *In case a window is tumbling it always contains all data-items which arrived already and have not been included in any previous window.*

Sliding *In case a window is sliding it can contain data-items which have been already included in a previous window again. In case three windows A, B, C follow each other in this order, a data-item X can be included in A&B&C, A&B or B&C but not only in A&C. After the element was present in A and not present in (removed before) B it cannot reappear in C again.*

Hopping *An hopping is present if there are data-items, which are not included in any emitted window at all.*

Figure 3 shows an example for a tumbling window. The left column shows a stream with arriving data-items. In the middle is the current data-item buffer and on the right are all the data-items which have been deleted from the buffer. From top to bottom, in each row arrives one of the input data-items. The setting in the example is a tumbling window by the count three, meaning that the buffer has a size of three data-items. A window is emitted (marked black), whenever the buffer is full. Every time a window is emitted, the complete buffer is deleted afterwards [28].

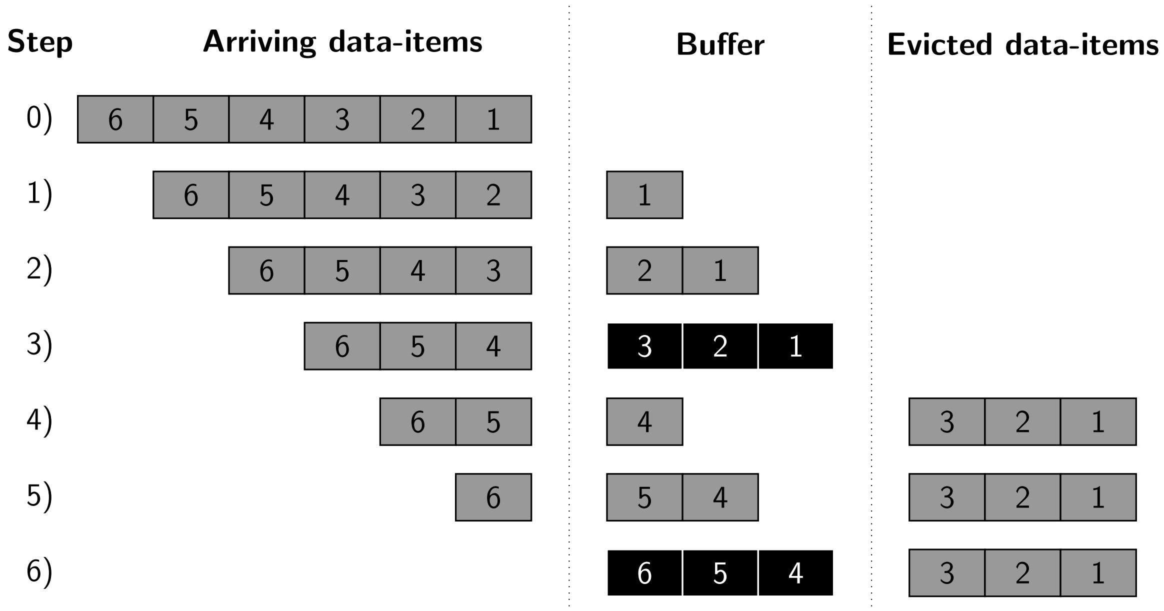


Figure 3: The data-item buffer states with count-based tumbling windows of the size three. Emitted windows are marked black.

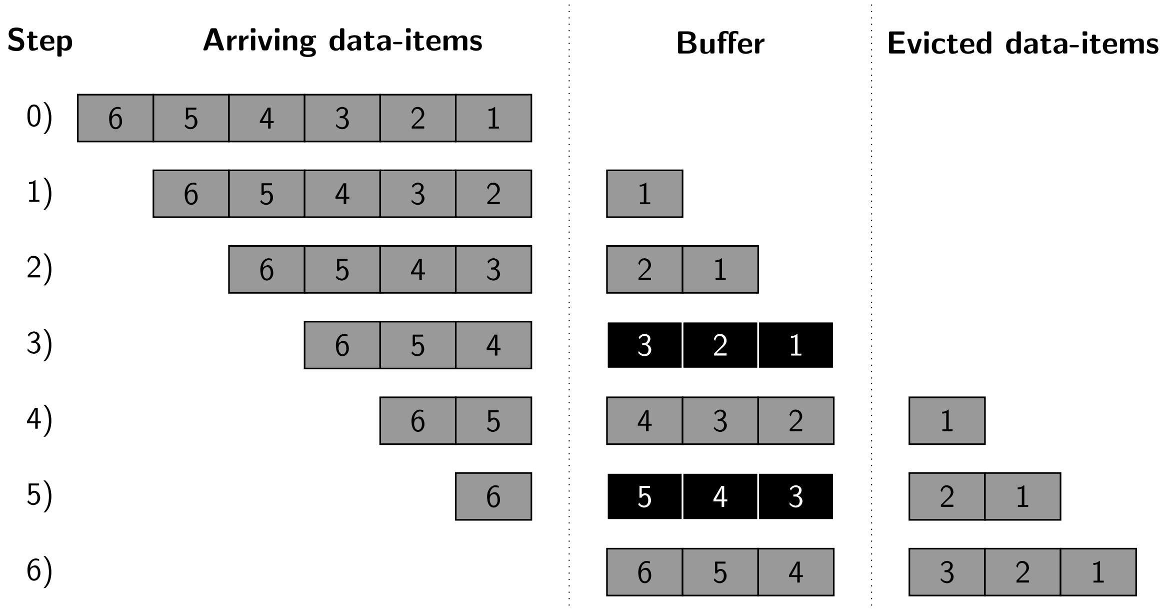


Figure 4: The data-item buffer states with count-based sliding windows having the window size three and the slide size two. Emitted windows are marked black.

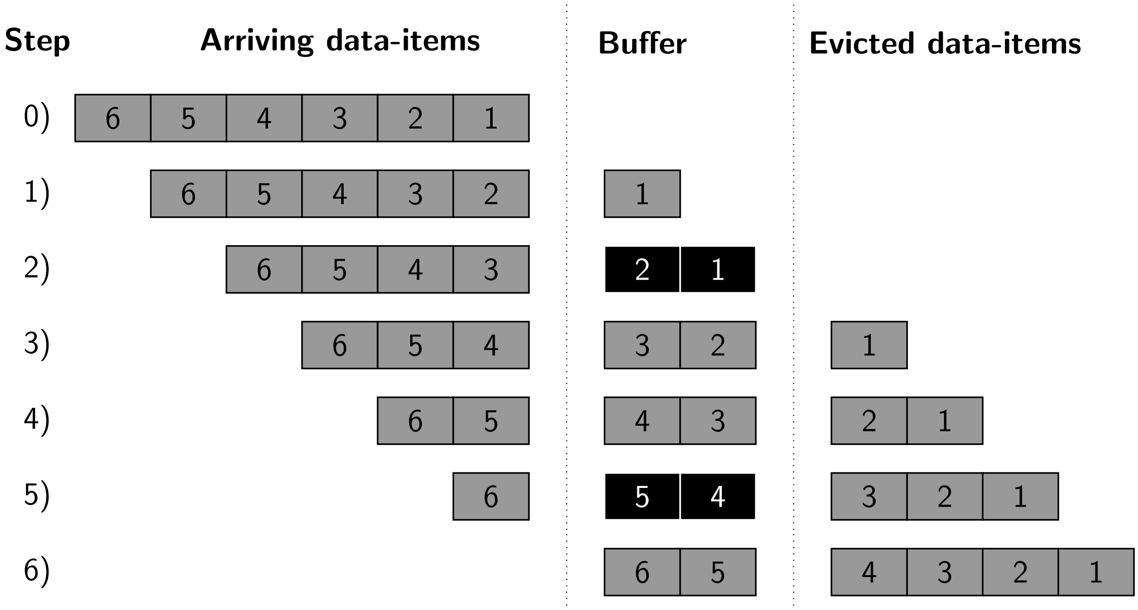


Figure 5: The data-item buffer states with count-based hopping windows having the window size two and the slide size three. Emitted windows are marked black.

Sliding windows overlap with previously emitted windows. A common way to define sliding windows, is to set a window size (s_w) and a slide size (s_s). The window size specifies the length of the window; the slide size specifies how far the window is moved on each step. Figure 4 shows the processing steps for a window discretisation based on data-item counts, where the window size is three and the slide size is two.

Hopping windows are a special case of sliding windows, where the slide size is greater than the window size. This means that data-items are skipped without being included in any window at all. Figure 5 shows the vice versa setup of the previous example; the slide size is three and the buffer size two.

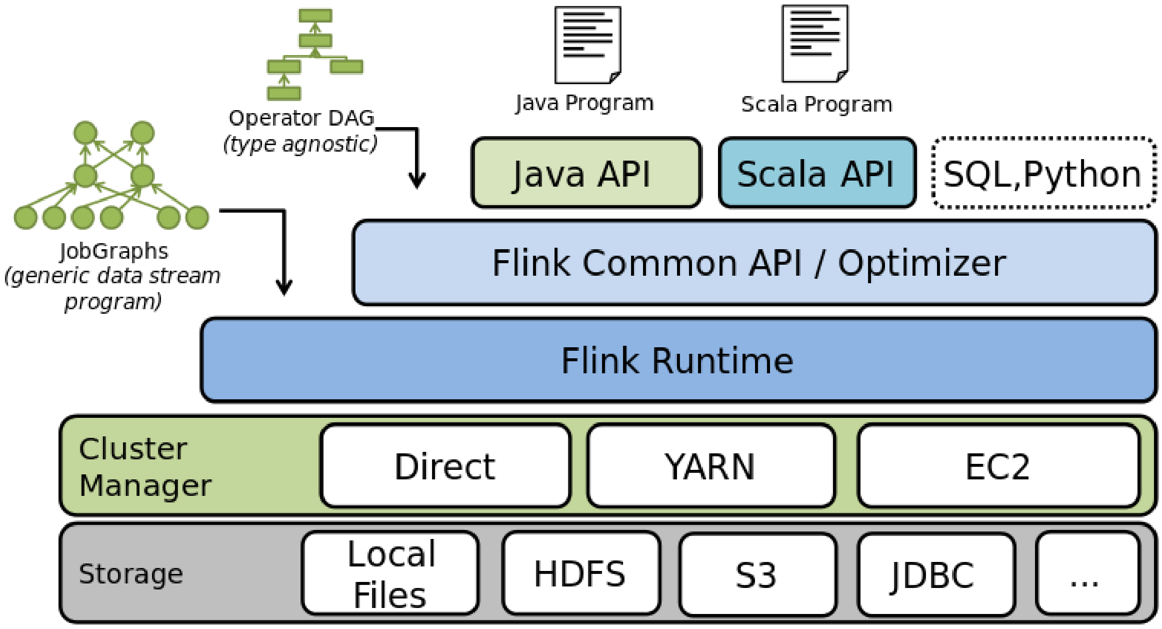


Figure 6: Figure from [64]: The Apache Flink Software Stack.

4.3 Apache Flink

Apache Flink consists of an open source software stack for parallel big data analysis. Apache Flink, also known as Stratosphere [5], offers a broad set of features and rich programming *APIs*. Flink implements both, a batch-processing and a stream-processing *API*, while all programs are executed on a unified runtime layer.

The runtime layer is based on Nephele/PACTs [14], which is a programming model and execution framework for web-scale data analysis. Nephele receives a PACT program, which consists of parallelizable operators. Hence, PACT programs fit our definition of a streaming query.

Once a PACT program is executed on the Nephele runtime, operators are spanned to multiple parallel instances, which allows data-parallel processing of streams. In opposite to Hadoop *MR*, where the map and the reduce phase run one after each other, Nephele is a real stream processing engine, where data is passed from one operator to the next, while operators run concurrently.

Figure 6 depicts the different layers of nowadays Flink versions. Flink is compatible with a variety of cluster management and storage solutions, such as *HDFS* [60], Kafka [49], YARN [65] and Apache Tez[6]. The communication with these systems is done by the Flink runtime and thereby mostly transparent to the programmer.

Programmers write an query using one of Flink's *APIs*. This program is then compiled to a program in the form of an operator *DAG*. The common *API*, which is the common layer below all *APIs* for programmers, applies flow optimizations and creates a *DAG* representing the job graph. The job graph is a generic streaming program, which can be executed on the flink runtime engine.

Flink is compatible to Hadoop *MR*. We can write a program doing the word-count example shown in Table 2 to run it on Flink using the same functions as we would do in Hadoop. In the following we will show a mapper and reducer implementation using the `FlatMapFunction` and the `GroupReduceFunction` interfaces from the Flink project. The input for our example could be the same as seen before in the data transformation example (Table 2). The file is split by lines, so that an input tuple contains the line id as key and the content of the line as value. The mapper (Listing 1) splits the input values at each space-character. For each split, one tuple is produced as output, containing the individual word as key and 1 as value. The reduce function (Listing 2) sums up the values per key. The final output are key-value-pairs containing a word as key and the number of the occurrences of the word as value.

These same functions can be applied for both, stream- and batch-processing. Map and reduce are operators. Map operations can be applied on streams on a per data-item basis without any modification, while the application of the reduce operation (aggregation) requires a finite set of data. Thus, reduce operations succeed discretisation operations and are applied on windows.

[6] Apache Tez: `http://tez.apache.org/`

```java
private class Mapper implements FlatMapFunction<String,Tuple2<String,Integer>>{
  @Override
  public void flatMap(String value, Collector<Tuple2<String,Integer>> out){
    for (String word:value.split(" ")){
      out.collect(new Tuple2<String,Integer>(word,1));
    }
  }
}
```

Listing 1: Word Count Example: A mapper implementation using Flink.

```java
private class Reducer implements GroupReduceFunction<Tuple2<String,Integer>, Tuple2<
    String,Integer>> {
  @Override
  public void reduce(Iterable<Tuple2<String,Integer>> values,Collector<Tuple2<String,
      Integer>> out){
    Iterator<Tuple2<String,Integer>> iterator = values.iterator();
    Tuple2<String,Integer> firstTuple=iterator.next();
    String key=firstTuple.f0;
    int sum=firstTuple.f1;
    while (iterator.hasNext()){
      sum+=iterator.next().f1;
    }
    out.collect(new Tuple2<String,Integer>(key,sum));
  }
}
```

Listing 2: Word Count Example: A reducer implementation using Flink.

5 Related Work

This thesis describes rich window discretization techniques in a stream processing system. Several research works on streaming systems and languages are related to this thesis. We provide an overview of systems and languages, focusing on different windowing semantics, in section 5.1.1. Two systems are the most closely related to the solution we propose.

1. *IBM SPL* serves as inspiration; our work is a generalisation of the windowing semantics provided by *SPL*. We explain the windowing semantics of *SPL* in section 5.1.2.
2. Flink is the system on which we implemented the windowing semantics introduced in this thesis. Section 5.1.3 describes the windowing semantics provided by earlier Flink versions.

When windowing is followed by aggregations, several types of optimizations can be applied. Section 5.2 provides a survey of different related optimization techniques proposed in the literature.

5.1 Streaming Systems and Languages

5.1.1 Windowing Semantics in different Systems and Languages

In the following we provide a survey through published streaming systems and languages. In section 9.1, we compare all the windowing semantics of all systems and languages presented here in order to evaluate the expressiveness of different window discretization techniques.

InfoSphere/SPL. The *IBM Stream Processing Language (SPL)* [37, 45, 44] was the first to introduce window discretisation based on trigger- and eviction policies. The work presented in this thesis is a generalisation of the windowing semantics introduces by *SPL* to enables the use of user-defined policies. We present *SPL* in more detail in section 5.1.2.

Flink v 0.7 Flink version 0.7 is the Flink version which was present before the work on this thesis started. Flink implements a streaming *API* allowing time and count based windows discretizations. Flink, former known as Strotosphere [5], executes streaming on its own runtime engine which is based on Nephele/PACTs [14]. We present Flink streaming in more detail in section 5.1.3.

Spark/DStream. Spark implements a batch execution engine: The execution of a job graph is done in stages, and the outputs of each operator are materialized in memory (or disk) until the consuming operator is ready to consume the materialized data. To allow stream processing, Spark uses *Discretized Streams (D-Streams)* [70]. Streams are interpreted as a series of deterministic batch-processing jobs with a fixed granularity. All windows defined in queries must be multiples of this smallest granularity.

Naiad. The goal of Naiad [57] is to offer an high throughput, which is typical for batch processors, together with low latency, known from stream processors, in a singly system. Additionally, Naiad has the ability to process iterative data flows. Unfortunately, Naiad can discretise windows only based on time.

StreamInsights. StreamInsights [43] is the streaming platform which is available through Microsoft's cloud. It allows three types of windows, namely count-, time- and snapshot-based. In snapshot-based windows, the discretisation is done regarding start- and end-markers provides within the stream. This makes it a punctuation-based approach.

Aurora. Aurora [18] is a landmark system in the history of distributed stream processing since it is the first design and implementation that parallelises stream computation with rich operation and windowing semantics, thus, being the first *distributed stream management system* of its kind. Aurora allows to specifies windows on any attribute of the data-item. Even though Aurora can apply windowing on different attributed than timestamps, windows are always specified as ranges on some scale. This is what we call *time based with user-defined timestamps* in case result emissions are done periodically or *delta-based* in case emissions not periodic.

NiagaraCQ/CQL. NiagaraCQ [19] focuses more on scalability than on the flexibility. The system provides various optimizations techniqes to share common computation within and across queries. Discretisation is thereby only possible based on time.

Esper. Esper [16] is an open source *Complex Event Processing (CEP)* engine which processes event-streams. It is tightly coupled to Java; made in a way, that it can be executed on Java *Enterprise Edition (EE)* application servers and describing events in *Plain Old Java Objects (POJOs)*. Windows are either time-based or count-based windows.

In the next two sections, we present the window discretisation done by *IBM SPL* and Apache Flink (v 0.7). These two systems are the most related to this thesis. We implemented and contributed the rich window discretization techniques presented in this thesis to the open source Flink project. The introduce trigger and eviction policies are a generalisation of the windowing policies provided by *SPL*.

5.1.2 The IBM Stream Processing Language

The *IBM Stream Processing Language (SPL)* [44] was the first introducing trigger and eviction policies. The trigger and eviction policies specified by *SPL* inspired the window discretization architecture presented in this thesis. In both cases, a trigger policy specifies, when a window ends and, consecutively, when further operations are applied on the discretised window. Both solutions have a data-item buffer. Once a window ends, the resulting window consists of the data-items in this buffer. The eviction policy[7] specifies, when data-items are deleted from the buffer.

Unfortunately, the runtime implementation behind the *IBM SPL* is not available to the public as open source project. Thus, we can only analyse the language itself, but not the actual logic used for the execution of the specified programs/queries.

A processing flow in *SPL* consists of a directed graph, which is similar to the Nephele execution engine on which the Flink runtime is based. The vertexes in the graph are operators. They are connected with edges. The data is passed from operator to operator along the edges.

The main three differences between the discretization solution presented in this thesis and the one present in *IBM SPL* are:

1. In *IBM SPL*, the window discretization is set as parameter to an operator, while we propose to make the discretization an operator itself. *SPL* wants to simplify the construction of window-based operators by decoupling operator logic and windowing from each other [37]. We have the same goal, but achieve is by putting the windowing operator in front of batch-processing-like succeeding operators.

2. While in *IBM SPL* only a limited set of pre-defined windowing policies can be used, our solution generalizes policies, such that users can apply their own user-defined discretization logic, implemented in user-defined policies.

3. Our solution allows to apply multiple trigger- and eviction policies at the same time. In *SPL*, there always have to be exactly one trigger policy and optionally one eviction policy.

[7] In *SPL*, tumbling windows are said to have only one policy, which is called *eviction policy*. In this work, we say that there is always both a *trigger policy* and an *eviction policy*. In case of tumbling windows, the programmer specifies a *trigger policy* and the eviction policy is set to `TumblingEvictionPolicy` automatically.

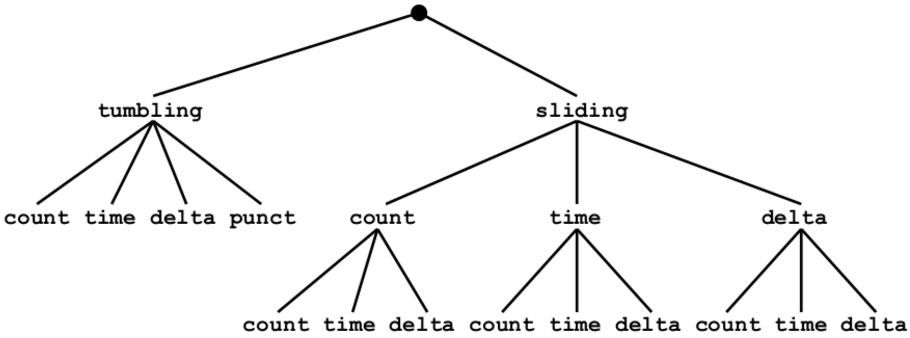


Figure 7: Figure from [45]: Window types in *SPL*.

In their paper *IBM Streams Processing Language: Analyzing Big Data in motion* [44], Hirzel at al. show different examples of *SPL* programs. Listing 3 shows an excerpt of a program based on their example.

In this example, an input stream is transformed to an output stream containing data-items of a custom type, called `MyType`. Assume this type to be defined before, having two attributes called `attribute1` and `attribute2`. The `stream`-type is comparable to the `DataStream` in Flink. It is the abstraction for a data-stream on which further data transformations can be applied.

The example discretises windows from the given `InputStream` and applies an aggregation. The discretization rules are passed as parameter following the `window`-keyword. First, the user needs to specify, that the window is supposed to be a sliding window. Then the trigger policy and the eviction policy are specified. The logic for the aggregation follows the `output` keyword. It uses the attributes `a1`, `a2`, and `a3` from the data-items in the `InputStream` and calculates the attribute values for the output from them.

There are four different kinds of policies provided by *SPL* and shown in Table 3: time-based, count-based, delta-based and punctuation-based. There is a correspondent to all of them implemented in Flink and presented in section C. A good introduction to window policies in *SPL* is provided by Dan Debrunner in two blog posts covering sliding [27] and

```
stream<MyType> outputName = Aggregate(InputStream){
  window InputStream : sliding, delta(...), count(...);
  output outputName : attribute1=Sum(a1*a2), attribute2=Sum(a3);
}
```

Listing 3: An example window discretization and aggregation in *IBM SPL*. [44]

Type	Trigger Policy	Eviction Policy
Count:	window emission on each n-th data-item arrival.	Keep the last n data-items in the buffer.
Time:	Emit a window every n time-units.	Delete all data-items which are older than n time-units.
Delta:	Emit a window if the delta between the data-item which caused the last trigger and the currently arrived data-item exceeds a threshold.	Delete all data-items for which the delta to the currently arrived exceeds a threshold.
Punctuation:	Emit a window, whenever a given punctuation is detected.	Unavailable: No punctuation based eviction is provided.

Table 3: Types of windowing policies in *SPL*.

tumbling [28] windows. The limitation to a small number of predefined policies reduces the total amount of possible different discretisation settings massively. As Figure 7 shows, there are only 13 possible combinations in *SPL*.

Delta-based policies are sensitive to the content of the arrived data-items. The delta-based eviction deletes all items where the distance to the currently arrived data-items exceeds the threshold. Using this approach, data-items can be deleted from the buffer out-of-order. This is the only feature we cannot provide with the solution presented in this thesis, as we assume the buffer to be a *FIFO* buffer.

As this thesis provides the use of windowing policies in the fashion of user-defined function, we shall have a look on *UDFs* in *SPL* as well. *SPL* provides a wide range of implementation possibilities for operators as *UDFs*. Additionally, it comes with the ability to apply optimizations even on the *UDFs*. [44] Anyhow, there is no possibility to implement windowing policies as *UDFs*. Policies are added as parameters to operators, thus, they are no operators themselves. Unfortunately, only operators can be implemented as *UDF*.

5.1.3 Window Discretization in Flink Streaming

Before the work presented in this thesis started, Flink streaming allowed only two types of window discretization: time-based and count-based. Windows were either sliding or tumbling [10].

Operators. Technically, the discretization was realized by two operators. One for count-based and one for time-based windows. This operators already coupled the discretization together with the aggregation of the discretised windows. Discretisation operators were optimized for their single case of use, which allows a very good performance for the particular use-case. The work presented in this thesis is a generalization of the discretization, allowing user-defined windowing semantics. We implemented completely new operators to allow the processing of such a user-defined discretization.

Notion of Time. In Flink's streaming API, users are allowed to implement their own user defined `Timestamp` [11] which represents an arbitrary time measure. This can be some globally maintained time from a time server or, in the simplest case, just the current system time. If the time is freely defined by the user, the user-defined `Timestamp`-object is used to determine the time, represented as `long`-value, for each data-item.

Whenever a user applied the default time-measure, the window operator started a separated thread. In this thread, a periodically check was done, telling whether a window end has been reached or not. If so, a result for the ended window got emitted. When user-defined timestamps were used, the end of a windows was detected whenever the first data-item with a timestamp beyond the window end arrived. We generalized the idea of having separated threads, which trigger window emissions, such that it can be used in a user-defined fashion. This feature is now available through active trigger policies.

Grouped Discretization. It was also possible to apply window discretization on grouped data streams: "For example a ‘`dataStream.groupBy(0).batch(100, 10)`‘ produces batches of the last 100 elements for each key value with 10 record step size." [10]
In opposite to the solution presented in this thesis, the discretization was always done on a per-group basis. This is the same what we call *distributed policies*, when we explain our solution for policy-based grouped windowing in section 6.10. Beside this, the operators presented in this thesis can apply windowing policies on the data-stream as a whole, even when the stream is grouped. This allows for example to emmit a result for all groups after a sepecific number of data-items (in total) arrived. In previous versions, this was only possible on a per-group counter basis.

Pre-Aggregation. In case sliding windows are used, the application of the aggregation on the data-item buffer can be optimized. We will explain in much more detail how this can be done in section 8. Previous versions of Flink streaming implemented an micro batch pre-aggregation based on the *greatest common divisor (gcd)* of the window size and the slide size. Pre-aggregation were made with the granularity of the *gcd*. This pre-aggregations were reused for the result emissions across multiple (overlapping) windows.

5.2 Optimizations for Streaming Systems

Several publication regard sharing of common computations for sliding windows (shared overlapping computation) or even for common processing steps across multiple queries. In this thesis we focus on the optimizations which can be made when an aggregation function is applied on a window.

Aggregation sharing. Krishnamurthy et al. [50] proposed in 2006 three different techniques for aggregation sharing across queries and across overlapping windows. The one closest related to the work in this thesis is *time sliding.*

Time slicing can be used when cross-query sliding windows exhibit differences in range or slide but still maintain periodicity. This technique does not work for arbitrary windows as we define them in this thesis (section 8.6). Also deterministic windows as defined in this thesis (section 8.4) are insufficient for the application of this technique.

At first, two types of window slicing are recognised: *paned window slices* where pre-aggregates are always computed in the lowest granularity (ie. *gcd(range, slide)*) and *paired window slices* as an optimization with lower final aggregation cost. The combination of multiple paired windows is done by extending the composed window up to the *lowest common multiple* of the participating query window slides. Thus, all aggregates are guaranteed to be derived from the composed partial pre-aggregates by using time slicing. The performance benefits of shared sliced aggregation are clearly visible when the data input rate is high. Implementation-wise, the slicing is done incrementally by a slice manager that is being updated in an ad-hoc manner with added or removed paired window definitions and then marks the points in the input stream where the pre-aggregate are to be computed.

Flink streaming also uses pre-aggregations based on the *gcd* of window size and slide size. We will explain this solution in section 5.1.3 and section 8.5. Additionally, we propose a border to border pre-aggregation, which we will present in section 8.4.

Sliding-window aggregates using panes. In 2005, Li et al. [51] proposed the concept of panes, which is very closely related to window sub-aggregates already introduced in previous work. This work is easy to read and understand. It is a good introduction to the basic idea of window pre-aggregates in a simple form.

The authors propose to use the *gcd* of the window size and the slide size to define the pane window intervals on which pre-aggregates can be computed. As we will show in section 5.1.3 and section 8.5, this optimization was already implemented in Flink streaming, before the work on this thesis began.

The proposal is followed by an analysis regarding the function types supported and optimizations for differential functions. It should be also noted that the authors argue that windows with different semantics for eviction and triggering cannot be defined by using panes. We can only partially confirm this statement. As we will show in section 8.4, the ability to calculate pre-aggregations relies more on the determinism of the policies than on their type. Still, it is not possible to combine arbitrary policies.

Incremental sliding-window evaluations. In 2007, Ghanem et al. [39] published a work focusing on the incremental calculation of sliding windows by using the previous windows. Two ways of defining the the windows are presented: The *Input Triggered Approach (ITA)* and the *Negative Tuples Approach (NTA)*. While in *ITA* the eviction of data-items from the data-item buffer is done based on the arriving items, in *NTA* data-items can set to expired by sending a so called negative item which removes or neutralizes the data-item which has expired.

As each data-item needs to expire at some time, it can be observed that the amount of data transferred doubles using *NTA* because each data-item causes a negative version of itself at some time. *ITA* does not have this problem, but can cause recognizable latency in window emissions if the triggering input elements arrives late.

Different optimizations are proposed for the *NTA* approach to reduce the described problem. Such optimizations either reduce the overhead caused by negative data-items or the amount of negative items.

A technique called *piggybacking* describes the adding of negative data-items to positive data-items. The result are combined items which allow expiration without having the requirement of processing the negative items separately. For example: If the aggregation does a sum, the value 5 would become expired by sending the negative value -5. If there arrives a positive value 2, it could be combined with the -5 to -3.

In addition to the content summarized here, an experiment is done to compare *ITA*, *NTA* and *NTA* when using the described optimizations. Therefore, the *NTA* approach is analysed in more detail providing a classification of different incremental operators, especially joins, regarding their optimization possibilities when using *NTA*.

The authors describe the expiration of data-items in *ITA* as based only on the timestamp of the newly arriving positive items, but not on other characteristics, like for example a delta-based policy. This lets *ITA* seem less powerful than it actually is. One could say, that our policy based windowing approach is an extension of the *ITA* they described, as it only reacts on input data-items, but also has fake data-items, which could be seen as negative items in their naming context. Anyhow, we neither use incremental window evaluation like they defined it nor *NTA* similar to their definition. Thus, most of the proposals are not close enough related to our ecosystem for being reused.

Processing hopping windows. In 2004, Babcock et al. [8] published a paper regarding the problem of adapting to processing demands adaptively by dropping unprocessed data-items. The main focus is on aggregation queries as a special case of minimising the degree of accuracy introduced by such a sample reduction. We also provide for solution for skipping unprocessed data-items in section 8.7. We have the definition of hopping windows, which says, that there is a hopping situation in case there are data-items which are not part of any emitted result.

6 Architecture and API-Design

In this section, we are going to present an highly expressive way of window discretization. The described discretization architecture was implemented in the Apache Flink stream processing engine. During the work for this thesis, we did the specification and implementation of the API as well as the architecture and development of the operators running underneath. A detailed documentation of the implementation can be found in appendix B.

In the following, we first describe all fundamentals of our architecture in the sections 6.1 to 6.7, including example queries in section 6.3. Afterwards, we present the *API* design in section 6.8.

Some use-cases require the use of fake data-items produced by so called active policies. We present the concept of active policies in section 6.9. Finally, in section 6.10, we show how the discretization is done in case data-items are grouped by some key.

6.1 Discretization Operators

Our solution is clearly separated from the one provided by *SPL*. While in *SPL*, the window discretization is seen as parameter of an operator, we made it an operator itself. This operator takes trigger and eviction policies as parameters and does the discretization. Following operators can work with the resulting windows without having any knowledge about the discretization. While *SPL* only provides a limited set of policies, we generalized trigger and eviction policies and provide interfaces for both of them. Hence, our discretization operators can work with the huge set of provided pre-defined policies, as well as with user-defined policy implementations.

Discretising a stream to windows makes sense, because many algorithms can not be applied to infinite streams of data, as they could never produce a result. Anyhow, such algorithms can be applied to finite chunks of data. A popular example for such algorithms are *Map Reduce (MR)* applications. While the map phase can be applied to an infinite stream, the shuffling and the reduce phase need to know, that the previous phase is completed. From a users perspective, the reduce phase consumes the output of the map-phase as a whole as input.

In the reduce phase, an aggregation is computed per group, while the groups are defined by the keys of the output tuples of the map phase. Receiving the output of a map operation, our discretization operators can (optionally) group the input data-items they receive, create finite chunks of the input data and apply an aggregation function implemented in an user-defined reduce function. Finally, they output one aggregation result per group and per window.

We implemented two discretization operators, namely `Window` and `Grouped Window`. `Window` discretises the stream to windows by applying trigger and eviction policies. `Grouped Window` additionally does a grouping of data-items by some key. It can apply policies to the stream of data-items as a whole and on per-group basis. It can even do both at the same time.

The discretization operators we provide, can even apply multiple trigger and eviction policies at the same time, which allows to specify complex discretization rules such as *"Every 5 minutes return the sum of the last 1 million data-items, excluding data-items which are older than 10 minutes"* or *"Emit a window every 5 minutes, unless the current aggregation result differs more than 20% from the one emitted before; in that case, emit the window immediately."*. Hence, we can adjust the aggregation granularity regarding the actual data characteristics.

6.2 Trigger and Eviction Policies

The discretization operators are event-driven. An event is essentially a data-item arrival; when a data-item arrives, trigger and eviction policies are notified.

Trigger Policies (TPs) specify when the reduce function is executed on the current buffer content and, thus, define the moment that results are emitted.

Eviction Policies (EPs) specify when data-items are removed from the buffer and, thus, define the size of windows.

6.3 Query Examples

In the following, we will show two examples provided by the Flink streaming *API* guide [11] for the not grouped case. Additionally, we will show two examples for the grouped case, which we sent to the Apache Flink Developer mailing list [62] when we released the new windowing semantics.

The next example would create windows that hold elements of the last 5 seconds, and the user defined aggregation/reduce is executed on the windows every second (sliding the window by 1 second): [11] (Listing 4)

```
dataStream.window(Time.of(5, TimeUnit.SECONDS))
  .every(Time.of(1, TimeUnit.SECONDS))
```

Listing 4: *API* Example: Not grouped discretization. [11]

Different policies (count, time, etc.) can be mixed as well; for example to downsample our stream, a window that takes the latest 100 elements of the stream every minute is created as follows: [11] (Listing 5)

```
dataStream.window(Count.of(100)).every(Time.of(1, TimeUnit.MINUTES))
```

Listing 5: *API* Example: Not grouped discretization with mixed policies. [11]

The new policy based windowing can also be used for grouped streams. For example: To get the maximal value by key on the last 100 elements we use the following approach: [62] (Listing 6)

```
dataStream.window(Count.of(100)).every(...)
  .groupBy(groupingField).max(field)
```

Listing 6: *API* Example: Grouped discretization with central policies. [62]

To create fixed size windows for every key we need to reverse the order of the groupBy call. So to take the max for the last 100 elements in Each group [...] This will create separate windows for different keys and apply the trigger and eviction policies on a per group basis. [62] (Listing 7)

```
dataStream.groupBy(groupingField).window(Count.of(100))
  .every(...).max(field)
```

Listing 7: *API* Example: Grouped discretization with distributed policies. [62]

6.4 Action Order

We call the startpoint and endpoint of a window interval *borders* of the window. As specified above, trigger can only occur when a data-item arrives and data-items cannot be partially deleted from the buffer, thus, borders are always bounded to data-items. The data-item marking the startpoint of the window is included in the window interval. The data-item marking the endpoint of the window is excluded.

Whether borders are included in, or excluded from emitted windows depends on the *action order*. When a data-item arrives, three actions take place:

1. Notification of the *Trigger Policy (TP)*: The current buffer content can be emitted as a window if the *trigger* decides to.
2. Notification of the *Eviction Policy (EP)*: Data-items can be deleted from the buffer if the *eviction* decides to do so.
3. Adding to buffer: The arrived data-item is always added to the buffer (*insert*).

The order of these actions is a crucial design decision: while in *IBM SPL*, the order is specified differently for each combination of policy types [37] (action orders in *SPL* can be found in section D), we fix the order as described above.

SPL can achieve a reduced latency for some types of policy combinations by having a different action orders, while other types of policies can only work correctly in case the action order matches our fixed order. From the perspective of the *TP*, two architectural alternatives are present; either to define the endpoint of a window to be included (*insert→trigger*) in the window or to be excluded from the window (*trigger→insert*).

Defining the endpoint as included in the window can reduce the latency for some policies. A positive effect of this solution always takes place in case the end of the window can already be identified at the moment we see the last data-item of the window. One example for this are count-based triggers: If we want to trigger after ten data-items, the policy could already trigger when the tenth data-item, belonging to the current window, arrives.

Anyhow, there are cases where we cannot determine that the endpoint of a window has been reached at the moment we see its last data-item. For instance, a time-based policy: We can never know whether the next data-item will still arrive within the time interval of the current window or not. Therefore, we can only decide whether a window end has been reached (and whether the policy should trigger) after we've seen the first data-item of the next window. Hence, the event order *trigger→insert* is essentially required for some policies, such as the time-based ones.

We estimate the benefit of using the first option (included upper border) to be relatively small. Assuming a high data rate, the latency caused by triggering one data-item arrival later should not preponderate. Providing both options would have several disadvantages compared to providing only one option:

- It would cause a massively increased complexity of the discretization operators, because we would need to provide two possible execution flows at the same time including the interdependencies between them.
- It would cause a performance decrease because we would need to figure out which execution flow to choose for each individual policy at runtime.
- It would cause an elimination of optimization possibilities as the order of triggers and evictions becomes unpredictable when not only predefined policies are used, but also user-defined ones.
- The interface to implement user-defined policies would become more complex, because it needs to enable the programmer to choose between both options. Alternatively, different interfaces would be needed. One for each option.

With regard to this, we decided to always exclude the endpoint of the window from the result (action order: *trigger→insert*). This solution works for all kinds of policies and does not have preponderating disadvantages compared to the alternative solution.

The eviction happens between the *trigger* and the *insert* event. The over all event order is *trigger→eviction→insert*. This order gives the guarantee that the data-item which causes the eviction remains in the buffer. Furthermore, it guarantees that after the first data-item arrival there is always at least one data-item in the buffer.

6.5 Memory Management

Our discretization operators maintain a data-item buffer. When the operator receives an data-item as input, we say that the *data-item arrives* at the operator. The buffer at the discretization operator is a *first in first out (FIFO)* buffer containing a subsequence of the stream. Hence, data-items can only be deleted from the buffer in the same order they arrived. Especially in case user-defined trigger or eviction policies are used, this limitation has big benefits. It enables several possibilities for both, parallelization and automated optimization.

6.6 Aggregation Functions

Drawing from the Flink batch API, in Flink Streaming we also allow two types of reduce functions:

1. `Reduce.` The reduce function is applied to data-item pairs continuously until only one data-item is left. When using this function, a programmer has to specify how to combine two input data-items to one output data-item.

2. `Group Reduce`. In opposite to the reduce function, the group reduce function receives an `Iterator` for a set of grouped data-items. The computation is done having the view on the window as a whole. This function is similar to the reduce function which is known from Hadoop *MR*.

An aggregation function like median, requires the use of the the `GroupReduceFunction`, because it needs a complete view of the window. However, this requires the materialization of intermediate results in-memory and can cause problems at runtime. In contrast, the `ReduceFunction` aggregates two data-items at a time, reducing the memory consumption. Furthermore, certain reduce functions can be executed faster by performing pre-aggregations. We provide more details on this, later in this section.

6.7 Aggregation Optimization

In the evaluation of streaming queries, there are often overlapping window aggregations (sliding windows). Optimizations aim to calculate common sub-aggregates and reuse them for multiple consecutive overlapping window aggregations. Thereby, aggregation optimizations can minder the CPU- and memory utilization by eliminating recomputation.

Pre-aggregation computation has been proven to be feasible especially while using associative-decomposable functions, such as the `ReduceFunction` [69].

Definition: associative-decomposable ”We use $\overline{x}$ to denote a sequence of data-items, and use $\overline{x}_1 \oplus \overline{x}_2$ to denote the concatenation of $\overline{x}_1$ and $\overline{x}_2$. A function H is decomposable if there exist two functions I and C satisfying the following conditions:

- H is the composition of I and C:
  $\forall \overline{x}_1; \overline{x}_2 : H(\overline{x}_1 \oplus \overline{x}_2) = C(I(\overline{x}_1 \oplus \overline{x}_2)) = C(I(\overline{x}_1) \oplus I(\overline{x}_2))$

- I is commutative: $\forall \overline{x}_1; \overline{x}_2 : I(\overline{x}_1 \oplus \overline{x}_2) = I(\overline{x}_2 \oplus \overline{x}_1)$

- O is commutative: $\forall \overline{x}_1; \overline{x}_2 : O(\overline{x}_1 \oplus \overline{x}_2) = O(\overline{x}_2 \oplus \overline{x}_1)$
- C is associative: $\forall \overline{x}_1; \overline{x}_2; \overline{x}_3 : C(C(\overline{x}_1 \oplus \overline{x}_2) \oplus \overline{x}_3) = C(\overline{x}_1 \oplus C(\overline{x}_2 \oplus \overline{x}_3))$" [69]

Yu et al. assemble a group reduce function (H) with two function (I and C) mapping to an initial reduce and a combine in *MR*, where I reduces a sequence of input data-items and C reduces a sequence of outputs from I. In case of our `ReduceFunction`, the input and the output type is the same, thus, is can represent I and C at the same time.

A number of optimizations apply, which are mainly based on two facts:

- *Pre-aggregation is allowed:* The `ReduceFunction` leaves it to the operator, when a data-item pair is reduced/combined to one data-item, thus, the aggregation of two data-items can already be done, when one pair of items is available. In contrast to this, the `GroupReduceFunction` can only be invoked when all data-items contained in a window are available, increasing the latency of the operator execution.
- *Pre-aggregations are reusable:* The `ReduceFunction` allows to compute pre-aggregations over *parts* of windows, thus, pre-aggregates can be calculated for overlapping portions of (sliding) windows. Such pre-aggregates can be reused by multiple aggregate computations.

In contrast, using the `GroupReduceFunction` instead of the `ReduceFunction` can cause the following:

- *Higher memory utilization* results from the need to keep all individual data-items in the buffer instead of pre-aggregation results. Moreover, is the window does not fit in memory, the system may even crash.
- *Higher latency* results from the fact that full aggregation needs to be done once all data-items of a window have arrived, while with the `ReduceFunction`, intermediate results could already be pre-computed.
- *Higher CPU utilization* is caused by the elimination of pre-aggregation reuse across overlapping windows. Computations for overlapping portions are done multiple times.
- *Inconstant CPU utilization* is caused by executing the full aggregation on a window at the time the window ended. Using the `ReduceFunction`, the work could be spread across data-item arrivals.

6.8 API Design

The Flink stream processing *API* offers the functionalities of data stream processing to the user in a way, which is easy to learn and fast and simple to use. We integrated our window discretization techniques in the *API* with regard to this philosophy. While most users only apply classical time-based and count-based window semantics, there are some users who want to use their own user-defined policies. Both features have to be available in a way which is intuitive to use.

The changes/extensions to the *API* we made, consist of two main ideas:

1. The trigger and eviction policies are applied on a stream using two methods, called `window` and `every`, followed by an aggregation method. The `every` method is optional and allows to specify sliding and hopping windows. Tumbling windows can be defined by using only the `window` method.

2. Pre-defined policies are available through helper classes which implement the `WindowingHelper` interface. The helper classes prevent the user from a need to know the exact constructor signatures of the policies and serves factory methods to create pre-defined policies with different settings.

6.8.1 Discretisation Methods

A data stream is represented in the *API* by the class `DataStream`. This class implements the method `window` which starts the definition of a discretization (Listing 8). In case only `window`, but not `every` is used, this leads to a tumbling window setup, where the parameter of the `window` method specifies when to trigger. The eviction policy is automatically set to `TumblingEvictionPolicy`.

```
dataStream.window(...).every(...).reduce(...)
dataStream.window(...).every(...).reduceGroup(...)
dataStream.window(...).every(...).aggregate(...)
```

Listing 8: The discretization and aggregation functions in the Fling streaming *API*. [11]

After the `window` method, the user can call the `every` method to specify a sliding window discretization. The parameter of the `window` method specifies then the size of the window (represented internally by an eviction policy) and the `every` method specifies the slide size of the window (represented internally by a trigger policy).

Finally, the aggregation needs to be specified. The `reduce` method takes an instance of `ReduceFunction` as parameter and the `reduceGroup` method allows to use a `GroupReduceFunction`. Furthermore, pre-defined aggregation functions such as `sum`, `count`, `min`, and `max` can be applied.

The `window` method is overloaded. it can either take an instance of WindowHelper as parameter or two lists containing the trigger and eviction policies to be used. Thus, user-defined policies can be passed as parameter directly to the discretization method as well as helper classes which encapsulate pre-defined policies.

6.8.2 Helper Classes

We provide the helper classes `Count`, `Time` and `Delta` for the pre-defined count-based, time-based and delta-based policies. All of them implement a static method called `of`. This allows to specify policies very easily as Listing 9 shows.

In case of the `Time` helper class, the `of`-method is overloaded to allow different kinds of settings. Users can provide their own user-defined `Timestamp` implementation or use the `SystemTimestamp` as default by not providing any `Timestamp` implementation. The additional method `withDelay` allows to delay the first occurrence of a trigger. This is required to have constant window sizes, even at the stream start, in case sliding windows are used. By using Java's `TimeUnit` class, the user can decide in which time unit she wants to specify the length of the windows. Possible units are days, hours, microseconds, milliseconds, minutes, nanoseconds, and seconds, while seconds is the default.

Like the `Time` class, also the the `Count` class provides an additional method, to set a custom start value for the counter. The usage of the method is optional. The `of` method

```
Time.of(long length, Timestamp<DATA> timestamp)
  [.withDelay(long delay)]
Time.of(long length, Timestamp<DATA> timestamp, long startTime)
Time.of(long length, TimeUnit timeUnit)
  [.withDelay(long delay)]

Count.of(int count)
  [.startingAt(int startValue)]

Delta.of(double threshold, DeltaFunction<DATA> deltaFunction,
  DATA initVal)
```

Listing 9: Helper classes representing pre-defined polices in the Flink streaming *API*. [11]

of the `Delta` class expects all the required parameters to instantiate a delta-based policy. We will describe these parameters in section C.3.

6.9 Active policies

The main idea behind active policies is to allow *TPs* to create fake data-items which are processed by the systems in a special way. Our discretisation operators are event-driven by data-item arrivals. There are some cases where we need to have policy notifications between (real) data-item arrivals. Such additional notification are caused by fake data-item arrivals.

Section 6.9.1 provides an example for a query where problems occur when not active policies are used. Afterwards we present the concepts of active policies and fake data-items. In section 6.9.3 we apply the active policies to solve the problem shown in section 6.9.1. Finally, we show in section 6.9.4 how active *TPs* can be used to reduce latencies in the result emission.

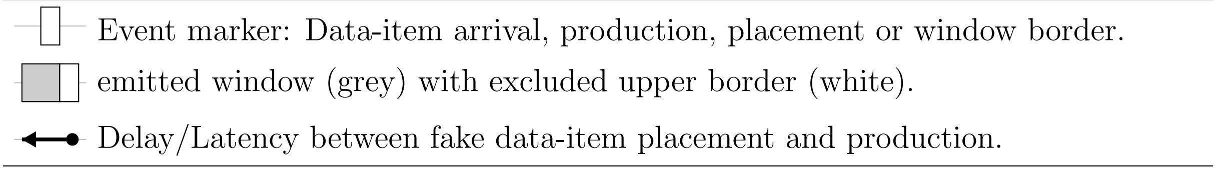
6.9.1 Problems without Active Policies

For the following examples we assume time-based policies to be used as *TP* and *EP*. Every two time units a window should be emitted containing the data-items of the last four time units. Hence, we have a sliding window query, where windows are overlapping by two time units.

Assumed time is determined by a user-defined `Timestamp`-implementation (as usually in Flink streaming; see section 5.1.3), we know that we have to emit a result for the window of the time interval `[0,4)` as soon as a data-item belonging to the time point four or greater arrives.

The example depicted in Figure 9 shows the basic case, which is covered by non-active policies already. Data-items arrive at the time points one, two, four, and fife. The arrival of the data-item belonging to time point four causes a trigger to occur, which leads to the emission of a result for the current data-item buffer, containing the data-items belonging to time point one and two. Thus, the emitted result is the one for the window of the time interval `[0,4)`.

Let's now extend Figure 9 up to time point eleven. In Figure 10, after the data-item arrival at time point fife, the next data-items arrive at time nine and eleven. According



▯ Event marker: Data-item arrival, production, placement or window border.

▨▯ emitted window (grey) with excluded upper border (white).

←● Delay/Latency between fake data-item placement and production.

Figure 8: Legend to Figure 9, 10, 11, and 12.

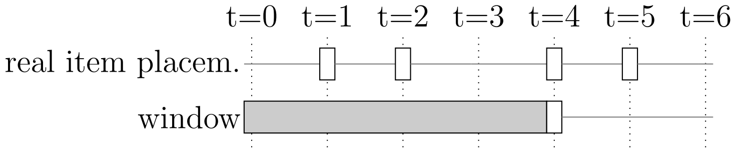


Figure 9: Time based windowing: The trivial case. (Find legend in Figure 8)

to the policies we defined, we expect results for the windows [0,4), [2,6), [4,8), and [6,10) to be emitted.

When the data-item at time point nine arrives, we expect results for the two windows [2,6) and [4,8). The last time the *EP* was notified is fife, which means it evicted only data-items up to (including) time point one. In case a not active trigger is used and just triggers when it is notified about the arrival of the data-item belonging to time point nine, the result is produced for the current buffer as usual. That is why we receive an output for the window [2,9), which is wrong. At the data-item arrival at time point eleven, we have a similar problem. At least we do not miss a result emission, but the borders of the result are still wrong. The last notification of the *EP* happened at time nine, which causes the current buffer to represent the window [6,11).

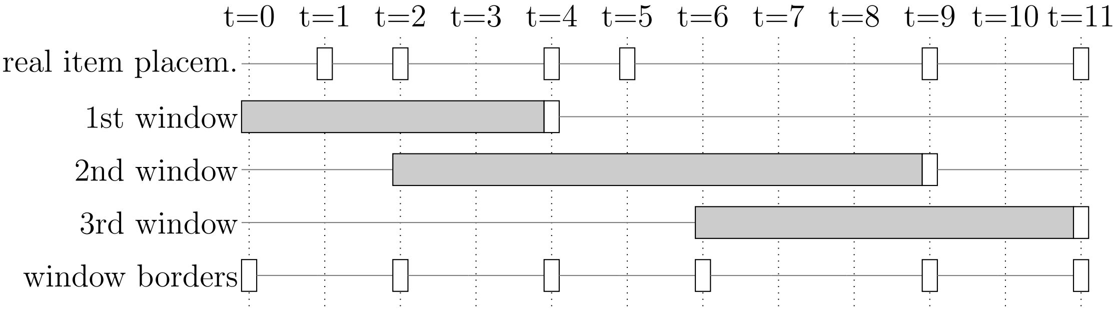


Figure 10: Time based windowing: The failure cases without active policies. (Find legend in Figure 8)

There is no way to achieve the correct result using the not active policies:

- In case a trigger occurs, the first action is the execution of the reduce function on the current buffer followed by the emission of the result. Eviction policies are notified afterwards. This can cause expired data-items to be included in the result.

- Trigger can only return `true` or `false` when they are notified. Hence, they can only cause one window emission per data-item arrival. This can cause missed result emission in case there arrives no data item between two window ends.

6.9.2 Active Policies and Fake Data-items

As already mentioned, we introduced so called active policies to address all issues pointed out in the previous section. Such policies are called *active* because they can not only react on data-item arrivals but also actively produce them. We call data-items which are produced by an active trigger *fake data-items.*

Pre-Notification. An active *TP* is pre-notified before the already presented regular notification of the trigger takes place. Like the regular notification method, also the pre-notification method receives the arrived real data-item as parameter. On each pre-notification, the policy can create fake data-items which are processed before the arrived real data-item. Thus, the *TP* has the ability to put fake data-item arrivals between two real data-item arrivals.

Data-item Placement. Where a data-item is placed, no matter whether it is a real or fake data-item, depends on its characteristics. For example, in time-based windowing data-items have a timestamp defining their position on a time-scale. In delta-based windowing, they have attributes from which a distance to the last window end can be calculated.

Notification about Fake Data-items. Fake data-items are always considered to cause the occurrence of a trigger, which means that each produced fake data-item causes the emission of a result for the current buffer. Hence, there is no need to notify *TPs* about fake data-item arrivals. *EPs* are notified about fake data-item arrivals in case they are active *EPs*.

Buffer. Fake data-items are never added to the buffer. They only serve as a kind of control elements causing triggers and evictions to occur.

Action Order. Whenever a fake data-item is processed, the action order is vice versa compared to the order used when real data-items are processed. Fake data-items are never added to the buffer, thus, the action order is *eviction→trigger*. With regard to the problems shown in the previous section, this action order is the most efficient. It allows to adjust the buffer content and to cause the emission of the result using only one fake data-item per emitted window.

6.9.3 Active Policy Example

Let's now remember the example, we presented in Figure 10 and let's utilize the possibilities provided by active *TPs* and *EPs* to produce correct results.

In Figure 11 we added three new lines for event markers to the example. The first of them shows the points in time, where fake data-items are produced, the second depicts where fake data-items are placed (In this example, this means what timestamp they have), and the third new line is the union of the real data-item placements and the fake data-item placements. The window ends of the emitted windows contain all times where fake data-items have been placed and, additionally, some of the times where real data-items arrived.

In the previous example, we got wrong results because we missed the ends of the expected windows `[2,6)`, `[4,8)`, and `[6,10)`. With the possibilities provided by active policies, we can produce fake data-items causing the triggers and evictions to occur which are necessary to gain correct results.

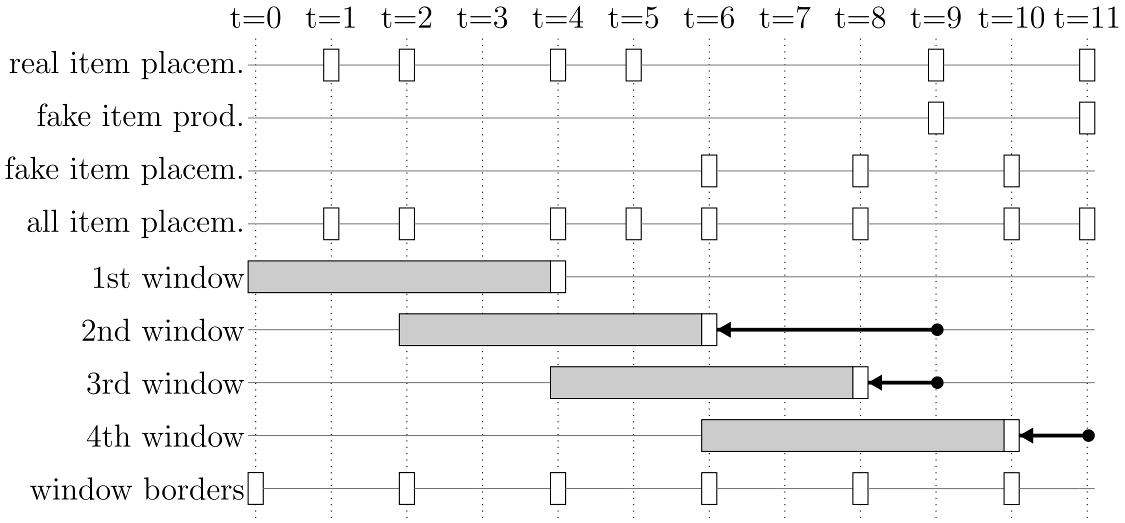


Figure 11: Time based windowing: Example for utilization of active policies. (Find legend in Figure 8)

Pre-notification. When the real data-item at time point nine arrives, there happens a pre-notification before it is processed. The active *TP* figures out the time point of the data-item, which is nine, and knows that there are two fake data-items needed. Hence, the active *TP* creates two such items, having the timestamps six and eight. Because the fake data-items are processed before the real one, trigger occur at the correct positions.

We actually simulate a situation in which real data-items would have been arrived at the time points where the fake data-items are placed. Also at time eleven, one fake data-item is produced which has the timestamp ten and causes the emission of the result for the window `[6,10)`.

Active Eviction. In all cases where fake data-items are processed, the eviction takes place before the result is calculated. Hence, the lower border of the window is always adjusted correctly.

Latency. The pre-notification of active *TPs* is always called when a real data-item arrives. This is the variant which needs the lowest computation power but is still guaranteed to produce correct results. Unfortunately, this solution comes with the price of having a latency in result emissions which is the duration between the time an window actually ends and the time the next real data-item arrives. This latency is depicted with black arrows in Figure 11.

6.9.4 Active Trigger Policy Threads

In the example shown in section 6.9.2 (Figure 11) we depicted latencies using black arrows. This latencies are the durations between the time windows actually end and the time where the next real data-item arrives. At this time, fake data-items can be produced and will cause the emission of results for the ended windows.

There are two possible cases where this latency becomes a problem:

- **When the data-rate is very low** in general or differs a lot, so that it is very low at some time: In such a case, the latency, which is the time to wait for the next real data-item, might increase to be unacceptable long.
- **When the reduce function is very computation expensive:** In such a case, having multiple emission at the same time possibly causes back-pressure. At the time where the real data-item arrives, not only one result for one windows needs to be computed, but results for several windows. One for each produced fake data-item and possibly one more after the regular notification of the triggers.

We introduced another possibility to create fake data-items to be able to reduce the latency in such cases. An active *TP* can provide a `Runnable` through a factory method. The provided `Runnable` is executed in a separated thread and can produce and submit fake data-items at any time through a callback method. This way of submitting fake data-items is completely independent from the arrival of real data-items.

Active *TP* threads can only be used in case it is possible to determine window ends without seeing a real data-item.

Applicability. This is for example not possible for arbitrary user-defined `Timestamp` implementations. They expect a data-item as parameter and return the timestamp it belongs to. This allows users to implement an extraction which reads timestamps from attributes of the arriving data-items. Unfortunately, in such cases, we can only know that a window ended when we see the first data-item which belongs to a point in time which is later than the endpoint of the window duration. Without seeing a data-item, we can not say anything about the current time defined by the `Timestamp` implementation. The only guarantee we have is, that it is monotonically increasing.

In opposite to this, for `Timestamp` implementations we know, such as our default `SystemTimestamp`, we can easily figure out the current time without the need to see any real data-item. The current time is available through the `System.currentTimeMillis()` method. Hence, we can always prove whether the current time lies beyond a window duration or not.

Also user-defined active trigger policies can utilize the factory method for runnables in case they can determine window ends without seeing a real data-item. For instance, in case one implements a `Timestamp` which retrieves the current time from some central time server. Here, the time is also independent from the actual data-items and it is always possible to retrieve the current time.

Example. Let's again take our example from the previous section into account, to show how the described solution can reduce the latency. We now assume our `Timestamp` in this example to be our default `SystemTimestamp`. We can let our active trigger policy produce a `Runnable`, proving at each fourth point in time (counting from zero on), whether a window end has been reached or not. If so, the `Runnable`, which runs in a separated thread, can submit fake data-items directly. This fake data-items will then replace some of those, who would otherwise have been produced on the pre-notification of the *TP* at the next real data-item arrival.

Note that in practice, as this is another thread, it would not run with exactly the same tact as the main thread, so the latency will be something from 0 to 1 time units more as depicted here. Additionally, one would most probably check for window ends with a much

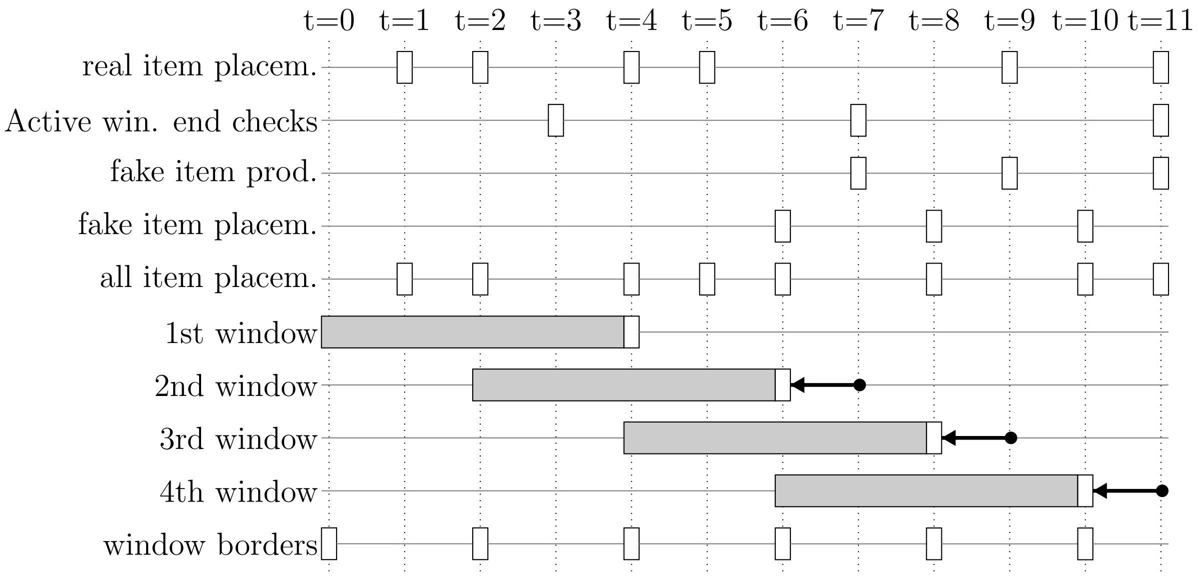


Figure 12: Time based windowing: Example for utilization of active policies including separated threads. (Find legend in Figure 8)

curse granularity to have less computation effort. It would also make sense to adjust the granularity of checks done by the runnable with regard to the specified trigger frequency. Anyhow, we use the simplified assumptions to keep the example as simple as possible.

In Figure 12, we added one more line to the diagram, showing the times where the `Runnable` does its check for window ends. Remember that, due to the submission of fake data-items threw the separated thread, there are less fake data-item creations done when the pre-notifications of the triggers happen. The total amount of created fake data-items and their positions remain the same as before.

When the data-item belonging to time point nine arrives, one fake data-item has been produced already from the separated thread. This fake data-item caused the emission of the result for the window with the duration `[2,6)`. Hence, only one fake data-item is created by the pre-notification method, causing the emission of the result for the window with the duration `[4,8)`.

When the data-item belonging to time point eleven arrives, we have a race condition. Either the separated thread comes first and submits a fake data-item causing the result for the window with the duration `[4,8)` to be emitted or the pre-notification is called first and does the job. We use synchronization to ensure that it never happens twice.

As you can see, we reduced the latency for the emission of the result for the window `[2,6)` by two thirds in the example. In general, we can give a guarantee, when active trigger runnables are used, saying that the latency will always be smaller or equal than the interval between two proves, for reached window ends, done by the `Runnable`.

6.10 Grouped Window Discretization

The main idea of a grouped window discretization is to combine the already described discretization functionality with a data-item grouping. There are many use-cases where a grouping of arriving data-items is done. For example, imagine a stream containing stock quote changes. It makes sense to group such a stream to gain several per-company streams. Grouping within a stream can also be seen as a split of an input stream to several output streams, while there is some rule to decide which data-item is forwarded to which output stream.

The calculation of aggregations is always executed on a per-group basis. Hence, once a trigger occurs, the reduce function is executed on the current buffer (per-group) and results are emitted per group. *TPs* and *EPs* can either work on a central buffer (*centralized policies*) or on per group buffers (*distributed policies*).

In this section, we present how data-item buffers look like in the grouped case in section 6.10.1. Afterwards, in section 6.10.2, we describe how *TPs* and *EPs* work in both cases, when they are central and when they are distributed. Furthermore, we explain how they play together in case both, central and distributed policies, are present at the same time using an example presented in section 6.10.3.

6.10.1 Data-item Buffers

The `Grouped Window` operator extracts the key from each arriving data-item. Then it forwards the data-item to the group it belongs. The groups are represented by instance of the `Window` operator (one instance per group). Similar to the not grouped case, each instance of the `Window` operator keeps a *FIFO* data-item buffer. The `Grouped Window` operator also has a buffer, but a virtual one.

Technically, the `Grouped Window` operator only remembers the order in which it forwarded data-items to the different groups, but not the forwarded data-items themselves, therefore, we call the central buffer *virtual central buffer.* From a logical point of view, we can imagine one buffer per group, containing only the data-items belonging to the respective groups, and one global buffer, containing all data-items from the stream as a whole.

The communication between the `Grouped Window` operator and the `Window` operator instances which represent the groups is unidirectional in the sense, that there is now information flow from the groups back to the central (the `Grouped Window` operator). This offers well opportunities for parallelization. The execution can be represented in a *DAG*, where the `Grouped Window` operator sends messages to succeeding `Window` operator instances, representing the groups.

6.10.2 Centralized vs. Distributed Policies

When a window discretization is applied on a grouped data stream, this can be done in two different ways.

Centralized. The first option is to monitor the stream as a whole. In this case, there is (from a logical point of view) only one central buffer. Data-items are added to this buffer, when they arrive at the discretization operator. When a trigger occurs, the data-items contained in the buffer are grouped by a key, an aggregation is calculated per group, and one result per group is returned. This means that trigger occur always for all groups together.

Data-items are still always deleted in the order they arrive (*FIFO*). In case the data-rate is not identical for all groups, the emitted windows may have different sizes.

Because the policies work with only one centralized buffer, we call this option *centralized*.

Distributed. The second option is to separate the arriving data-items first, with regard to their key. The operator keeps then one buffer for each unique key, respectively for each group. Data-items are added to the buffer of the group they belong to. Policies operate on a per-group basis and only take the data-items belonging to one group into account. This means that trigger occur always only for an individual group.

Data-items are, from a global view, possibly deleted out of order. The *FIFO* characteristic only holds when viewing the buffers for each group separately. In case the data-rate is not identical for all groups, windows may be emitted at different times for different groups to ensure constant window sizes across all result emissions.

Because the policies are replicated and work with separated buffers for each group and, because the arriving data-items are distributed to these different groups, we call this option *distributed*.

Technical Differences. From a technical point of view, there is only one difference between the implementation of a centralized and a distributed policy. The usage in a distributed manner requires the policy to implement a `clone`-method. This clone method is used to create replications of the policy object for each group.

6.10.3 Examples

The following examples clarify how centralized and distributed policies work. Imagine that we measure the speed of cars passing by in front of the office. We can group the resulting stream of speed measures by car facilitators to calculate the average speed per facilitator.

Distributed Policies. A count-based policy makes sense to use as central as well as as distributed policy. By using distributed policies, one can cause the emission of results whenever one hundred cars from one facilitator passed by. This guarantees two things:

- That the evaluation per group (per car facilitator) is as up to date as possible.
- That the computed result covers a statistically sufficient amount of individual measures (always one hundred).

Centralized Policies. One may wants to figure out for every one hundred cars passes by (in total), cars from which facilitator where the fastest. Using central policies, it is possible to emit a result for all groups whenever one hundred cars passed by, no matter how many of them belong to which group. This gives the guarantee that:

- We always receive a result for all types of cars, after we've seen one hundred cars in total.

Mixed Policies. Our grouped discretization operator even allows to use both, central and distributed policies, at the same time. In section B.7, we will explain in more detail how they work together. In general, a time-based policy is one which makes most sense as central policy. Taking our example into account, we could combine central and distributed policies as follows: We use a central time-based trigger policy and a distributed count-based eviction policy at the same time. The central time-based trigger guarantees, that we receive a result for each facilitator periodically. The distributed count-based evictor ensures, that we always keep exactly the one hundred most up to date measures per facilitator. Thus, our guarantees are a combination of the aforementioned:

- We always receive a result for all types of cars, after some time (periodically).
- The computed result covers a statistically sufficient amount of individual measures (always one hundred).

7 Parallelization

A discretization operation, in the current version of Flink streaming, always consists of a window discretization and the application of a reduce function on the discretised windows. The discretization is done by the application of *TPs* and *EPs*. The reduce function is executed on the current data-item buffer whenever a trigger occurs. The emitted results of the operator are aggregation results; one result per discretised window.

While the discretization is usually not parallelizable, there are possibilities for the execution of the reduce function in parallel. In the following, we assume that the `ReduceFunction` interface is used and not the `GroupReduceFunction` interface. As described in section 6.7, `ReduceFunction` serves better opportunities for a parallel execution and further optimizations.

We show how the execution of the discretization operators can be done in parallel. Section 7.1 serves basic observations regarding the parallelizability of the window discretization and aggregation. In section 7.2, we show how the `window` operator can be split in three operators to allow a parallel aggregation. Afterwards, we extend this concept in section 7.3 to allow further parallelization for some types of policies. In section 7.4, we point out how the `grouped window` operator can be parallelized.

7.1 Observations for Parallelizability

Parallelizability of Policies. As already mentioned, the window discretization cannot be done in a parallel fashion, because arbitrary policies may have a state which is based on the history of all previously arrived data-items. A policy may needs to see all data-items in order to be able to decide when to trigger or evict.

Anyhow, there are some pre-defined policies, where a parallel execution is imaginable. A time-based policy works with the current system time in the default case. Hence, the state depends rather on the current time and the used time measure than on the seen data-items.

History Dependent Policies. We say that a policy is ***not history dependent*** in case it does not need to see all data-items of the stream in order to work correctly. In opposite to this, delta-based policies and count-based policies are two examples from the set of pre-defined policies who definitely need to see all data-items of the stream in order to work correctly. Consecutively, such policies are called ***history dependent***.

In the following, in section 7.2, we assume a set of arbitrary, maybe user-defined policies to be used. This means, that the discretization itself is inherently not parrallelizable, because policies are history dependent.

Parallelizability of the Aggregation. What can be parallelized is the execution of the reduce function. Moving away as much as possible parts of the execution from the discretization operator makes sense. Due to the condition that it cannot be executed in parallel, the vertex doing the window discretization might become a bottleneck for the execution and should not be burdened with any additional load. In section 7.3, we will show how a further parallelization can be done, in case not history dependent policies, such as time-based policies, are used.

With a more differentiated view, two scenarios can be imagined when looking at the discretization vertex as a bottleneck:

1. **The bandwidth of the vertex doing the window discretization becomes a problem.** This will be especially the case if data-items are huge and/or the stream has a high data rate. In this case, it is *not* recommended to apply the parallelization we describe in section 7.2. The application of the reduce function within the same vertex might reduce the amount of data at the vertex output significantly.

2. **The memory or the CPU utilization at the vertex doing the window discretization becomes a problem.** This is highly probable in case the reduce function is computation intensive or the window sizes are huge, which leads to huge data-item buffers kept in memory. In this case, it makes sense to do apply the parallelization we described in section 7.2, to allow a parallel execution of the reduce function and to distribute the buffer.

7.2 Architecture for Parallel Aggregation

To allow a parallel execution of the reduce function and to distribute the buffer across multiple vertexes, we propose to split the functionality of the actually present `Window`-operator in three separated operators. Figure 13 shows the data flow between the operators and their parallel instances.

Central Window Discretization (CWD) This operator does the window discretization. It executes the policies in a centralized manner, but does not do any reduce on the data-items. Instead, meta data is added to the data-items to expose the information about where windows start and end to the next vertices. The disadvantage of this is, that it will cause the total amount of data at the output to be greater than the

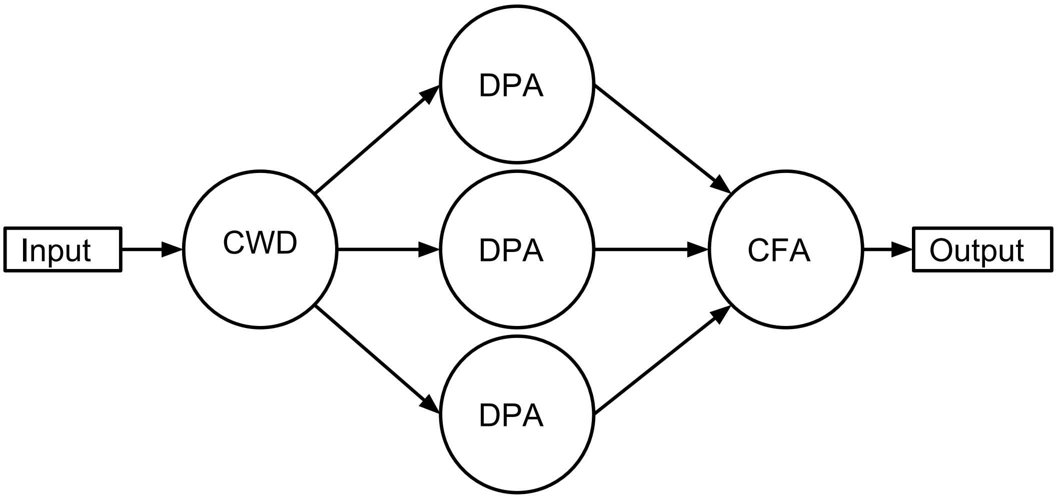


Figure 13: Operators for window discretization combined with window wise aggregation in a parallel fashion.

amount of data at the input of the operator. The advantage is, that there remains no need to keep a buffer at this operator and that the operator has no need to do any calculations except of those which are included in the policies. The state to keep in the memory here, is the state of policies. At least for all pre-defined polices, this state has a very small size, as we will show in section C. The arriving data-items are forwarded to the distributed pre-aggregation vertices. The simplest algorithm for forwarding data-items would be round robin.

Distribruted Pre-Aggregation (DPA) This operator can be executed in higher parallelism. The only thing which is important is, that the preceding and succeeding operator knows how many parallel instances of the *DPA* are present. Data-items are forwarded from the *CWD*-operator together with meta-data about window ends (respectively occurred triggers) and window start items (respectively the occurrence of evictions). Each instance of *DPA* keeps a part of the current data-item buffer. Whenever a trigger occurs, each instance executes the reduce function on its part of the buffer. The output of each *DPA* vertex is one pre-aggregation result per window.

Central Final Aggregation (CFA) The central final aggregation vertex waits until it has received all preaggregation results from the preceding *DPA* instances for one window. Then, it reduces this pre-aggregations to the final aggregation result for the window. The produced output is thereby exactly one data-item per window, which is the final aggregation result. The *CFA*-operator removes all meta-data which is added by *CWD* and *DPA*. This leads to exactly the same output as the `Window` operator (described in section 6) would produce. Thus, the distributed execution can be made fully transparent to the user.

In the following, we will describe the functionality of each of the three operators in detail. Thereby, we specify what the input and output formats of each operator are and how they behave.

7.2.1 Central Window Discretization

Input: `<DATAITEM>` Generic Type. The type of data which is delivered from the stream source

Output: `Tuple3<DATAITEM,Numeric,Numeric>` A tuple with three fields containing a data-item, which was provided as input, the id of the current window (only set if a trigger occurs) and the number of elements to evict from the buffer.

There are three possibilities how data-items can arrive:

- An active *TP* produces fake data-items on pre-notification.
- An active *TP* thread/runnable produces a fake data-item.
- A real data-item arrives.

We modified execution flows which are needed for the three mentioned cases. In general, *CWD* needs to keep a window counter. This counter counts the occurrences of triggers. When an active trigger produces fake data-items on pre-notification, each produced fake data-item is processed one after each other. First, it is send to all active *EPs*, to obtain the number of data-items which shall be deleted from the buffer. From the obtained result, the number of items to delete at each parallel instance of *DPA* can be calculated. This is fairly simple in case round robin is used to forward real data-items to *DPA* instances, but might be more complex, when another algorithms is used. Afterwards, the window counter is increased by one. For each succeeding *DPA* instance, an output tuple is emitted, containing `null` in the first field, the current window counter value in the second field, and the number of data-items to be removed from the buffer of the respective *DPA* instance in the third field. When an active trigger thread produces a fake data-item, the same steps are are executed for this fake data-item.

When we do the processing of a real data-item, we do the pre-notification of the active triggers first, allowing them to produce fake data-items. This might cause us to enter the execution flow described in the previous paragraph to process the fake data-items. After the fake data-items have been processed the regular notification of triggers is done.

In case a trigger occurred, we increase the window counter by one and emit an output tuple containing `null` in the first field, the window counter value in the second field and zero in the third field. The output tuple is forwarded to all *DPA* instances, which causes all of them to produce and emit a pre-aggregation for their current buffer content. Note that the case that a real data-item causes a trigger is rare compared to the case that it does not cause a trigger in most queries.

No matter whether a trigger occurred or not, we now notify the *EPs* and calculate how many data-items shall be deleted from each of the buffers present at the *DPA* instances. Additionally we apply our round robin algorithm (or any other suitable algorithm) to figure out which *DPA* instance shall receive the currently arrived real data-item. To this instance, we send an output tuple containing the data-item in the first field, `-1` in the second field, and the number of data-items to delete from the buffer of this instance in the third field. To all other instances, we only send an output tuple in case data-items shall be deleted from their buffers. If so, the output tuple contains `null` in its first field, `-1` in the second field, and the number of data-items to delete from the buffer in the third field. Again, note that the case that an eviction occurs is rare compared to the case that no eviction occurs in most cases.

The most frequent case when a real data-item arrives is, that it does neither cause the production of fake data-items, nor a trigger to occur, nor an eviction to occur. The proposed solution is optimized for this case. In such a case, only one output tuple is sent to one succeeding *DPA* instance.

7.2.2 Distribruted Pre-Aggregation

Input: The same as the output of *CWD*.

Output: `Tuple2<DATAITEM,Numeric>` A Tuple with two fields containing a data-item which represents a pre-aggregation result and the id of the window the pre-aggregation belongs to.

Each parallel instance of this operator keeps a data-item buffer containing the individual data-items which have been sent in the first field of the input tuples. On each arrival of an input tuple, the operator executes three processing steps:

1. The number of data-items specified in the third field of the input tuple is deleted from the data-item buffer.
2. A check is done whether the second field of the input tuple is unequal to minus one. If so, this means that a window id is provided and a trigger occurred at *CWD*. In such a case, the reduce function is executed on the current buffer and an output tuple is emitted.
3. If the third field of the input tuple is not `null`, the data-item provided in this field is added to the buffer.

7.2.3 Central Final Aggregation

Input: The same as the output of *DPA*.

Output: `<DATAITEM>` Generic Type. The type of data which is delivered from the stream source. The output format of *CFA* matches exactly the input format of *CWD*.

This operator knows how many parallel instances of *DPA* are present. It waits until it has received all the pre-aggregations for a specific window id. Thereby, it only needs to keep one data-item per window id in the buffer. As soon as a second pre-aggregation result with the same window id arrives, the reduce function can be called immediately and the two present pre-aggregations can be combined to one data-item. Once all pre-aggregations for one window have been received, the resulting data-item, representing the over-all aggregation result is emitted.

7.3 Parallelization for not History Dependent Policies

In case policies are not history dependent, even a parallelization of the window discretization is possible. An example for this is time-based windowing. Figure 14 shows how a further parallelization for not history dependent policy can look like. An implementation and the design of this execution flow, which is based on the execution flow for history dependent policies, was done by Gyula Fóra, a Apache Flink contributor and member of the Flink streaming research team at the *Swedish Institute of Computer Science (SICS)* in Stockholm.

We know from the nature of time, that it doesn't matter whether we measure the time at different places. We will, independent from the characteristics of the arriving data-items, get the same times. The impact of the relativity of time [29] is negligible here.

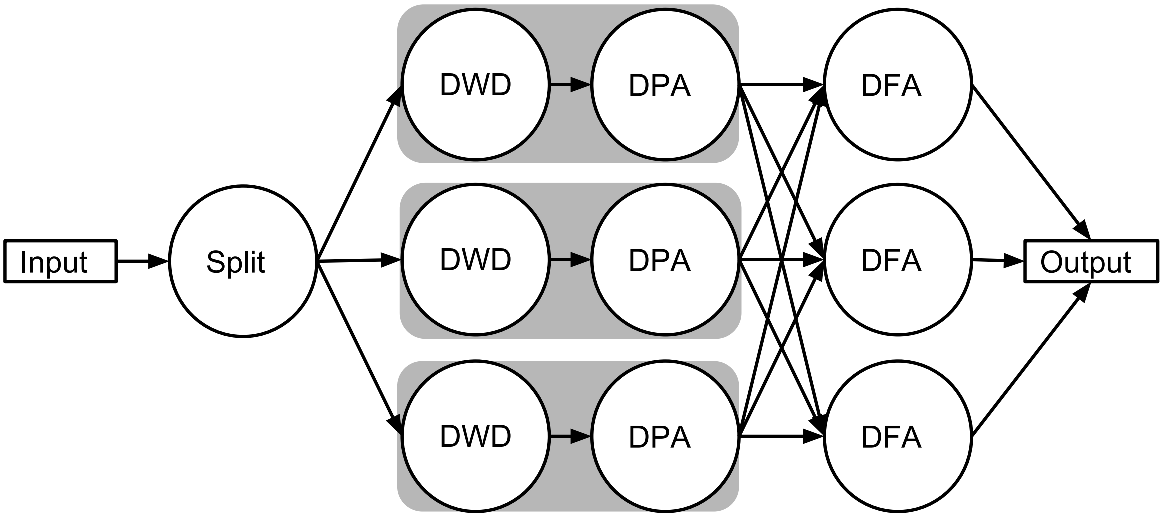


Figure 14: Fully parallelised window discretization and aggregation for not history dependent policies.

The impact of not synchronized clocks at different places in a computing cluster might be more important instead. Anyhow, in the following, we assume either a synchronized clock or an externally provided timestamps to be used. Thus, the time is somehow the same at all parallel instances.

Parallel Discretization and Pre-Aggregation. Instead of sending the input data directly to a discretization operator, which we called *CWD* before, we place a split operator. This operator sends arriving data-items to succeeding instances of a *Distributed Window Discretization (DWD)* operator. The simplest possible algorithm for the distribution of data-items to different succeeding vertecies is round robin, which is also used in the current implementation.

Each instance of *DWD* does a window discretization independent from each other, but all of them act identically, meaning that the sequence of emitted window ids is the same at all parallel instances. The aggregation can be applied in exactly in the same way as described in the previous section. In addition, one *Distributed Window Discretization (DWD)* instance can always be co-located with a *DPA* instance. This allows a chaining of the two operators and prevents us from doing any serialization within the boxes marked grey in Figure 14.

Final Aggregation. What remains is the final aggregation. We can parallelise it across multiple windows, but not within windows. In Figure 14, the *CFA* operator was replaced with a *Distributed Final Aggregation (DFA)* operator. Between the *DPA* and the *DFA* operation happens a shuffling. *DFA* instances receive pre-aggregation results grouped by window ids. As the number of expected pre-aggregations is known, the instances know when they have received all pre-aggregations for one window. Final aggregations can be done and emitted in the same way as described in the previous section. The architecture at this point is closely related to the original *MR* programming model. *DWD* can be seen as a mapper, *DPA* as a combiner, and *DFA* as a reducer. While it is not probable, that the aggregations results are emitted out of order, it is theoretically possible. An order guarantee for outputs can only be provided when either a *CFA* is used or an ordering operator is placed after *DFA*.

7.4 Parallelization of the Grouped Case

The parallelization of the grouped discretization and aggregation can be done as extension to the parallelizations described for the not grouped case. As described in section 6.10, the logic to do group-wise window discretization utilizes execution flows from the not grouped case.

Without any further adjustments, the logic of the `Grouped Window` operator could be used in a parallel manner. As the communication from the `Grouped Window` operator to the nested instances of the `Window` operator is unidirectional, it can be represented in a *DAG*. This allows a parallelization per group using exactly the logic described in section 6.10. The parallel execution described in the previous sections can then be present per group.

The alternative would be to extend the used window ids, such that they consist of a group id and a (per-group) window id. This would have the advantage that parallel instances can adaptively process data-items of separated groups. The parallel flows don't have to be group-wise separated. The disadvantages is, that the amount of added meta data would increase again and the logic of the parallel operators would become more complex, because they have to handle group-wise buffers and group-wise pre-aggregations. In opposite to this, when having separated parallel sub-graphs per group, the grouping can be made transparent for the operators in the sub-graphs.

8 Optimizations

When a stream processing platform is selected in order to solve a query, the performance of the platform is one of the crucial points influencing the decision. In this section we show how the window discretization and aggregation can be done in a more efficient manner.

Section 8.1 points out the optimization dimensions. In section 8.2, we categorise queries with regard to the used windowing policies. The succeeding sections show different means of optimization which apply for the different categories of queries.

8.1 Optimization Dimensions

Beside classical benchmarking of query executions in a real system, a theoretical evaluation of the required workload can be done, especially in case shared aggregations are used. The following categories allow the analysis of the influences of different optimization opportunities.

1. **Pre-aggregation: The number of calls to the reduce function, done before the window ends.**
   In case sliding windows are used, different windows can share some common pre-aggregations such that the overall number of calls to the reduce function should become much smaller. This is closely related to the **CPU utilization** of the discretization operators.

2. **Final aggregation: The remaining number of calls to the reduce function, at the moment the last data-item of a window arrives.**
   As soon as we know that a window ended, there will be some remaining computation which needs to be done before the aggregation result for this window can be emitted. For example, if windows in a sliding window setup share common pre-aggregations, we need to reduce pre-aggregation results to gain an overall aggregation for the window we have to emit. The amount of remaining computation at the end of a window is closely related to the **latency**. In case a `ReduceFunction` is used, we can evaluate the amount of remaining computation by counting the remaining calls to the reducer when a window end has been detected.

3. **Memory utilization: The amount of data-items stored in a buffer.**
   As we will describe in section C, the state size of policies is, at least for the predefined policies, very small. The memory utilization of the discretization operators

is instead mainly caused by the data-item buffers they keep in memory. Reducing the number of data-items in the buffer leads to lower memory utilization.

4. **IO Cost: The amount of data transferred over the network.**
   Whenever we introduce higher parallelism, the amount of data transferred over the network might increase, while it is zero in a purely central computation. Thereby, it's important to check how much more or less data needs to be transferred when using different optimizations. We can calculate the number of tuples transferred over network to evaluate the tradeoff between parallel execution and network utilization.

Like mentioned in 4, the network traffic might increase due to higher parallelization. The goal of parallelization is to remove/reduce bottlenecks by distributing work to multiple machines or processing cores. The discretization itself can indeed hardly be parallelized as shown in section 7. What can be parallelized is the aggregation. The most important evaluation to make here, is to figure out the highest workload which has to be done on a single machine.

The optimizations we present in the following sections mainly address the first three dimensions: pre-aggregation, final-aggregation and memory utilization. The IO-Costs and the parallelization of the execution are more difficult to evaluate and there is a tradeoff between them. A higher parallelism might increase the IO as data-items need to be sent more often over the network compared to a computation within a single vertex. We presented approaches for the parallelization of the discretization operators in section 7. We also stated there, in which cases a parallel execution is preferable over a not parallel execution.

The discretization operators presented in this thesis always do a window discretization and apply a reduce function on each discretised window. It is important to notice, that reduce functions in Flink are not invertible. Once a `ReduceFunction` has been applied to aggregate two data-items, the data-items given as parameter cannot be regained from the result. All optimizations presented in the following assume the use of a `ReduceFunction` implementation as aggregation function.

8.2 Categorization of Queries

For the optimization of the execution of a query, we can consider different kinds of queries. Depending on to which category a query belongs, different optimizations can be applied to address the optimization dimensions mentioned in the previous section.

In the following, we differentiate three different kinds of queries:

1. **Tumbling window queries**
   When tumbling windows are used, we know from the definition of tumbling windows, that there will not be any overlap of emitted windows or hopping. All data-items have to be included in exactly one window. We will show optimizations for such queries in section 8.3.

2. **Sliding window queries with deterministic policies**
   We say that a set of policies is deterministic, if we can tell at the time a data-item arrives, whether it is a border data-item or not. This is always the case for window ends, but only in special cases for window begins. We will explain this in more detail in section 8.4.

3. **Sliding window queries with arbitrary policies**
   Optimizations for all other sliding window queries are explained in section 8.6.

4. **Hopping window queries**
   A hopping is present in case there are data-items which are not contained in any window. In section section 8.7, we will present under which circumstances we can detect such situations and how unnecessary computations can be avoided when hopping is detected.

8.3 Pre-aggregation for Tumbling Windows

As already mentioned, tumbling windows are not overlapping and there will not be any hopping. All data-items have to be included in exactly one window.

The total number of calls to the `ReduceFunction` per window, is the number of data-items contained in the window minus one. This cannot be further optimized.

What can be optimized, is the final aggregation and the memory utilization. In a not optimized execution, the operators would keep the individual data-items in the buffer until a trigger occurs. Thus, the maximum number of data-items in the buffer would be the number of data-items contained in the emitted window. Once the trigger occurs, the `ReduceFunction` would get applied on the buffer content, meaning that the remaining number of reducer calls (final aggregation) would be the number of data-items contained in the emitted window minus one.

In an optimized execution, when the first data-item arrives, it is added to the buffer. Whenever a further data-item arrives, the `ReduceFunction` is called immediately to combine the one data-item from the buffer with the currently arrived data-item. The data-item in the buffer is then replaced with the result item returned from the `ReduceFunction`. Once a trigger occurs, the one data-item contained in the buffer can be emitted as result directly and will afterwards get replaced by the currently arrived data-item. This leads to the situation, that only one data-item at a time is kept in the buffer and whenever a window end is reached, no call to the aggregation function remains. The result can be emitted directly. Remember, that the data-item causing the trigger will be excluded from the current window (section 6) and is added to the data-item buffer after the result emission.

Summarizing, for tumbling windows:

- We can reduce the number of data-items in the buffer from the number of items contained in a window to be always one.
- We can fully eliminate final aggregation.

From a technical point of view, the detection of tumbling windows is fairly simple. Whenever all *EPs* are instance of `TumblingEvictionPolicy` (see description in section C.5), the current query is a tumbling window query.

8.4 Pre-aggregation for Deterministic Policies

This type of optimization can be done whenever a set of deterministic policies is used. We say that a set of used policies is deterministic, if we can tell at the time a data-item arrives, whether it is a border data-item or not. For *TPs*, this is obviously always the case, but for *EPs*, figuring out whether it is deterministic, is more complex. It depends on the policy itself and the used *TPs*.

Whenever a data-item arrives, a *TP* decides whether a window end has been reached and an aggregation result should be emitted. An *EP* decides on each data-item arrival, how many data-items shall be deleted from the buffer.

The questions is: *Will the currently arrived data-item be the first data-item in the buffer at the moment where any further trigger occurs?*

For arbitrary policies, this question can only be answered at the time the trigger occurs, but not at the time the lower border data-items arrives. Anyhow, some sets of policies are deterministic in a way, that the question can be answered at the time a data-item arrives. If a count-based *TP* is used together with a count-based *EP*, all window ends are known in advance. In case a trigger occurs every n-th data-item, the window begins are also known in advance, because they can be calculated by subtracting the window duration from the window ends. The same calculation can be done if both types of policies are time-based (using the same type of timestamp).

Checking for Window Borders. One could imagine methods to implement in a *TP* and a *EP*, to answer the question stated before. If this methods can be implemented, the policies are deterministic. Imagine the following two methods to be present:

- At the trigger policy:
  `value getNextTriggerPosition(value previousTriggerPosition)`
- At the eviction policy:
  `value getLowerBorder(value upperBorder)`

Remark, that the type of `value` has to be the same at the trigger and the eviction policy. In case a time-based *TP* is combined with a count-based *EP* (or the other way around), we say that the policies *operate on different scales*. One operates on a scale of sequence ids for data-item arrival and the other on a time-scale. In such a case, the set of policies is not deterministic, because it is impossible to answer the given question. Even though we can tell when the trigger occurs on one of the scales, we cannot say anything about the placement of the occurred trigger on the other scale. Thus, we cannot provide the *EP* with the information it needs to calculate the positions where windows begin.

Algorithm 1 can answer the question whether a currently arrived data-item is a border item or not. This algorithm computes the borders of windows in a loop. It starts to obtain window ends from the trigger policy, starting at the position where the currently arrived data-item is placed. For each window end, it obtains the associated window begin from the eviction policy. Whenever a border has been obtained from the *TP* or the *EP*, the position of the currently arrived data-item is compared with this border. If the positions match, the arrived data-item is a border item and the returned result is `true`. The loop continues until the window begin has reached a value beyond the position of the arrived data-item. If there was no position match before the loop stops, it means that the arrived data-item will be deleted from the buffer and not be present in any further window (*FIFO* buffer). Thus, it cannot be a border item and the returned result is `false`.

Periodicity. Deterministic sets of policies don't necessarily cause periodic occurrences of triggers and evictions, but the most common case is that they do. Assuming periodicity allows much better evaluations of the impacts of an optimization. In the following, we will assume periodically sliding window discretization. Anyhow, the described border to border pre-aggregation optimization can be applied to any deterministic set of policies and is not limited by any periodicity requirements.

```
Data: position of the currently arrived data-item as itemPos
Result: boolean : true if the arrived data-item is a border item, otherwise false

currentLowerBorder ← minimum_value;
currentUpperBorder ← itemPos - 1;
while itemPos > currentLowerBorder do
    currentUpperBorder ← trigger.getNextTriggerPosition(currentUpperBorder);
    if currentUpperBorder=itemPos then
        return true;
    else
        currentLowerBorder ← evictor.getLowerBorder(currentUpperBorder);
        if currentLowerBorder=itemPos then
            return true;
        end
    end
end
return false;
```

Algorithm 1: Algorithm to prove if an arrived data-item is a border item or not. This algorithm executed a maximum of s_w/s_s iteration of the contained loop.

We say, that a periodic sliding window setup has the following parameters:

- s_s : The slide size of the window. A result emission happens every s_s units.
- s_w : The size (length) of a window.
- d : The duration (length) of a stream.

In the beginning of a stream, the lower border of multiple windows might be at position 0. At the end, as we always emit a last result for the current buffer content before shutdown, there might be one last emission, having out of sequence upper and lower window borders. In the following, for the reason of simplification, we assume the first trigger to be delayed such that the size of emitted windows remains constant. We also assume the duration (d) to be selected in a way, that its end matches a upper window border. In case we would not make this assumption, we would have to special case the begin and the end of the data stream.

Due to the definition of a sliding window, $s_w > s_s$ must hold. Otherwise windows would not overlap, which would mean that the setup is not a sliding window setup.

Number of emitted Windows and Border Items. In the following descriptions and calculations, we assume count-based window discretization. Anyhow, the described optimization is not limited to policies based on data-item counts.

Due to the constant window size, the first trigger occurs after one window duration (s_w) has passed. Afterwards, trigger occur periodically with regard to the slide-size (s_s). The total number of emitted windows ($|W|$) within a duration (d) can be calculated by dividing the duration of the stream (d) minus the duration of the first window (s_w) by the slidesize of the window (s_s) and adding plus one for the first emitted window.

1. $|W| = \frac{d-s_w}{s_s} + 1$ (Number of emitted windows)

Each emitted window has two borders. In general, the set of window begins can be derived from the set of window ends by subtracting s_w from each entry in the set of window ends. In case s_w is a multiple of s_s, all window begins will match exactly the positions of window ends from previous windows. The only exception to this is the begin of the first window. Otherwise, there will be no matches and each emitted window causes the presence of two borders. With regard to this, the total number of window borders ($|B|$) can be calculated as follows:

2. $|B| = \begin{cases} |W| + 1 + \frac{s_w - s_s}{s_s}, & \exists x \in \mathbb{N}^+ : s_w = x s_s \\ 2|W|, & \text{otherwise} \end{cases}$ (Number of window borders)

Pre-aggregation Concept. To minimize the number of calls to the reduce function, we can utilize the aggregation optimization which we already mentioned for tumbling windows. We now do a pre-aggregation from border to border. When the first data-item arrives, it is added to the buffer. When further data-items arrive, they are reduced together with the last data-item in the buffer and the last data-item is replaced with the result of the `ReduceFunction`. An exception to this is the case, that an arrived data-item is a window border. In such a case, the data-item is added to the buffer as additional entry, which means a new pre-aggregation starts.

Buffer Content. Instead of containing individual data-items, the buffer contains now pre-aggregations reaching from on window border to the next. This reduces the maximal size of the buffer. Without pre-aggregation, we keep each individual data-item for one window duration in the buffer. With pre-aggregation, the maximal buffer size depends on whether s_w is a multiple of s_s. If so, as already mentioned, lower borders will match upper borders. Thus, the number of pre-aggregations covered by one windows is the factor which gives s_w when it is multiplied with s_s. If the condition does not hold, lower and upper borders are separated, which causes the number of borders to double. The number of pre-aggregations increases respectively.

3. $s_{b_i} = s_w$ (Maximum buffer size with individual data-items)

4. $s_{b_p} = \begin{cases} \frac{s_w}{s_s}, & \exists x \in \mathbb{N}^+ : s_w = x s_s \\ 2\lceil \frac{s_w}{s_s} \rceil + 1, & \text{otherwise} \end{cases}$ (Max. buffer size with pre-aggregation)

Final Aggregation. Whenever a trigger occurs, the remaining calls to the reduce function are those needed to aggregate the pre-aggregations in the buffer to a single result. Hence, a maximum of $s_{b_p} - 1$ calls is needed for the final reduce when using pre-aggregations. Compared to this, in case we would keep individual data-items in the buffer, the final aggregation would consist of $s_w - 1$ calls to the reduce function.

5. $r_{f_i} = s_w - 1$ (Final aggregation: Reducer calls with individual data-items)

6. $r_{f_p} = s_{b_p} - 1$ (Final aggregation: Max. reducer calls with pre-aggregations)

Pre-Aggregation Costs. Due to the overlap of sliding windows, pre-aggregations might be used multiple times for several window emissions. This can reduce the total amount of calls to the reduce function significantly. Having a global view on the stream for the given duration d, the made pre-aggregations reach from border to border. A new pre-aggregation is always started in case a data-item is a border item. The total number of calls to the reduce function for the calculation of all pre-aggregations in the duration d can be calculated as follows:

7. $r_p = d - |B| + 1$ (Calls to the reducer to calculate all pre-aggregations)

Total Aggregation Costs. Keeping the global view on the stream for the duration d, the total amount of calls to the reduce functions consists of all the required calls, necessary to produce the aggregations for all emitted windows. In a not optimized setup, the reduce function is called on a data-item buffer, containing individual data-items, for each emitted window separately. To reduce a single window requires $r_{f_i} = w_s - 1$ calls to the reduce function. Thus, in a not optimized execution $(w_s - 1)|W|$ calls are necessary for the duration d. With border to border pre-aggregation only r_{f_p} final calls to the reduce function are necessary per result emission. Additionally, pre-aggregations have to be calculated, but this is not done per emitted window. Pre-aggregations can be reused by several windows. With regard to this, the total number of calls to the reducer goes as follows:

8. $r_{t_i} = (w_s - 1)|W|$ (Total calls to the reduce function; not optimized)

9. $r_{t_p} = \begin{cases} |W|r_{f_p} + r_p, & \exists x \in \mathbb{N}^+ : s_w = x s_s \\ |W|r_{f_p} + r_p - r_{f_p}, & \text{otherwise} \end{cases}$ (with pre-aggregation)

Eviction of Pre-Aggregation Results. When executing discretization and aggregation with border to border pre-aggregations, the execution of the deletion of data-items from the buffer needs to be modified. Whenever an eviction occurs, we know that it either evicts data-items up to the next border or that there will be further evictions doing this, before the next trigger occurs. Anyhow, to make it applicable for not periodic but deterministic policies, we need to remember the positions of borders. As soon as an eviction occurs, we can remove the oldest pre-aggregation from the buffer. If further evictions occur, we have to check whether they evict beyond the lower border of the currently present first pre-aggregation. If so, we can delete it. Otherwise, we have to keep all pre-aggregations in the buffer.

The presented border-to-border pre-aggregation serves always equal or better performance than solutions based on the greatest common divisor (panes).

8.5 Pre-aggregation based on GCD (Panes)

As described in section 5.1.3, the previous versions of Flink streaming implemented a pre-aggregation sharing based on micro batches. Therefore, the *gcd* of the slide size (s_s) and the window size (s_w) was calculated. Pre-aggregations were calculated with the granularity of the *gcd* and reused for several result emissions for (overlapping) windows.

The presented border to border pre-aggregation goes beyond this approach. In the worst case, it has the same computation effort, but in many cases it performs much better.

By using the old *gcd*-based pre-aggregations, the computation effort can be calculated as follows (under the same assumptions and using the same parameters as in the previous section):

1. $s_{b_p} = \frac{s_w}{gcd(s_w,s_s)}$ (Max. buffer size)
2. $r_p = \frac{d}{gcd(s_w,s_s)} \cdot (gcd(s_w, s_s) - 1)$ (reducer calls; calculate all pre-aggregations)
3. $r_{f_p} = \frac{s_w}{gcd(s_w,s_s)} - 1$ (Max. reducer calls; final aggregation)
4. $r_{t_p} = |W| r_{f_p} + r_p$ (Total calls to the reduce function)

8.6 Pre-aggregation for Arbitrary Windows

Whenever a set of used policies is not deterministic, this means that it is not possible to know at the time a data-item arrives, if this data-item is a lower border of a window or not. Regarding this, the pre-aggregation from border to border cannot be applied in this case.

Nevertheless, pre-aggregations can be calculated with a fixed granularity. Such pre-aggregations than need to be stored in a separated buffer, in addition to the data-item buffer which contains the individual data-items.

Whenever a data-item causes a trigger to occur, the aggregation of the for the window consists of three parts:

1. The aggregation reaching from the end of the current window to the next smaller multiple of the aggregation granularity is represented by the intermediate result of the currently in progress pre-aggregation.

2. From there, the present pre-aggregations can be used for the aggregation until (including) the pre-aggregation having the smallest begin position grater or equal to the begin of the window to emit has been reached.

3. Due to the fact that `ReduceFunction` is not invertible, the individual data-items are now required to calculate the remaining part of the aggregation.

Whenever an eviction occurs, it must be applied on both buffers. The deletion from the buffer for individual data-items can happen like it would without pre-aggregation. The deletion from the pre-aggregation can work in exactly the same way as described for the border to border pre-aggregations.

The application of a pre-aggregation with fixed granularity comes with a trade-off between the different optimization dimensions. It has a negative impact on the buffer size, because the pre-aggregations are stored in addition to individual data-items. On the other hand it can reduce the reducer calls required for final aggregation, because of the use of pre-aggregations. It can also reduce the total amount of reducer calls, because pre-aggregations might be reused for several result emissions.

8.7 Hopping Windows

A hopping is present in case a data-item arrives at the discretization operator, which will not be included in any emitted window. When periodic windows are used, this is the case if the the slide size is greater than the window size.

When hopping is detected, the processing effort for the arrived data-item can be reduced. While the notifications of the policies must be done in the same way as before, there is no need to add the data-item to the buffer. In case pre-aggregation is used, there is also no need to calculate a pre-aggregation which covers such data-items, because the pre-aggregation would never be used.

In queries with tumbling and overlapping sliding windows, there can of course never be hopping. Thus, we can say that as soon as we see one of these window types, we can stop proving for hopping.

Detecting hopping windows

In case we assume that deterministic policies are used like we defined them in section 8.4, we can say that we know the placement of all window borders in advance. Thus, we can count window ends and begins up to a specific point (on the scale the policies operate). In case there is an equal amount of begins and ends before this specific point, a hopping situation is present at this point. Having as much window ends as window begins means that there is now window which began already but hasn't ended.

The following algorithm is suitable to prove for hopping, when a deterministic set of policies is used:

This algorithm is fairly simple. We just get the borders of the first window to emit, which has a upper border beyond the currently arrived data-item. Then we check, whether the currently arrived data-item is places before the lower border of this window. If not, it is included in the next window and we have no hopping. If it is placed before the lower border, it is evicted before the next window emission and we have a hopping. Due to the characteristic of the buffer, saying that a deleted data-item can never reappear, this means that it cannot be included in any future window. Remark that the border at a window begin is excluded from the window. This means, that we can still have a hopping even if the arrived data-item is placed at a window end.

In case non-deterministic policies are used, it is unfortunately not possible to detect hopping. We can never know whether a data-item will be evicted before or after the next occurrence of a trigger, which means we cannot know if it can be skipped for pre-

```
Data: position of the currently arrived data-item as itemPos
Result: boolean : true if the arrived data-item is included in a hopping situation.

nextUpperBorder ← trigger.getNextTriggerPosition(itemPos);
nextLowerBorder ← evictor.getLowerBorder(nextUpperBorder);
if $nextLowerBorder > itemPos$ then
    return true;
else
    return false;
end
```

Algorithm 2: Detecting hopping windows with a deterministic policy-set.

aggregation or not. This leads to the situation, that the application of the optimization described in section 8.6 has a negative impact on the total number of calls to the reduce function in case hopping is present.

Without using pre-aggregation techniques, we would just do the aggregation for the current buffer once a trigger occurs. This means skipped data-items are already excluded and are never aggregated. If we apply pre-aggregation, we always create pre-aggregations of a specific granularity and discard them in case they are not needed.

When hopping with arbitrary policies is present, there is a trade off between the application of pre-aggregation with fix-granularity and not applying any optimization. With pre-aggregation, the final calls to the reduce function can be lowered. On the other hand, the total amount of calls to the reduce function possibly increases, because data-items cannot be discarded before a not needed pre-aggregation is computed.

9 Evaluation

In this work, we introduced highly expressive means of window discretization. In section 9.1, we compare our solution with the expressiveness of the window discretization in other systems.

Furthermore, we proved in an experiment, which we present in section 9.2, that apart from higher expressiveness the throughput of our discretization operator scales linearly with the amount of processed data. Thus, the introduced higher expressiveness does not have a negative influence on the scalability.

9.1 Expressiveness

System/Language	**Time**	**Count**	**Punctuation**	**Delta**	**User Defined**
NiagaraCQ/CQL [19]	✓				
InfoSphere/SPL [37, 45, 44]	✓	✓	✓	✓	
Spark/DStream [70]	✓				
Esper [16]	✓	✓			
Naiad [57]	✓				
StreamInsights [43]	✓	✓	✓		
Aurora [3]	✓	✓		✓	
Flink v 0.7 [5, 14]	✓	✓			
This work (Flink v 0.8)	✓	✓	✓	✓	✓

Table 4: Stream Language Windowing Semantics

Many systems and languages allowing window discretization have been proposed already. Table 4 provides an overview of such such systems and languages.

We provided a short description of all the listed systems in section 5.1.1. None of the presented systems can provide user-defined window discretisation. Moreover, none of the systems is able to answer complex queries like the following:

- **Concept drift detection:** *”Detect concept drifts [63] in the data stream. If a drift is detected, recalculate the concept using the last one million data-items. Thereby, do not include data-items which are older than ten minutes.*

- **Session expiration:** *"Transform an input stream of user actions to an output stream of expired sessions, whereas a session expires in case there is no user action for 20 minutes."*

In this thesis, we presented flexible windowing semantics that allow programmers to express very complex window-based streaming queries like the two above. The discretisation techniques presented in this work go very beyond the one which are currently available in stream processing systems and languages.

9.2 Performance

Experiment Setup. We measured the throughput of three different discretization operators combined with an aggregation. For our measurements, we made an setup where windows are discretised based on time. The window size was 50 time units and the slide size was 21 time units. The applied aggregation function was the `max` function.

Time-based discretization is the most computation expensive from the set of pre-defined policies. Both, `TimeTriggerPolicy` and `TimeEvictionPolicy` are active policies, which means that pre-notifications are done for each data-item arrival. If we would use count-based discretization we would avoid pre-notifications and gain even better results for the policy-based windowing.

We made sure, that the data source is faster than the window operation, so that we actually measure the throughput of the discretization operators and not the speed of the data-source. The datasource was chained to the window-operators, which prevents serialization between the source and the discretization operation.

To make sure we only compare the discretization operators and not the different Flink runtime versions, we ported all three operator versions to the current runtime version. Thus, all of them ran on the same (the current) version of the runtime and could benefit from the latest performance improvements.

The shown processing duration also contains the start-up time of the job. We made sure that the *Java Virtual Machine (JVM)*-memory was pre-allocated before the job executions started. As the first start of a job within in a session takes much longer than the following, we always made a first run with just ten timestamps, before we run the tests with larger amounts of data. We excluded the first run with only ten processed data-items from the evaluation. All tests where executed with parallelism one on a current version mac-book.

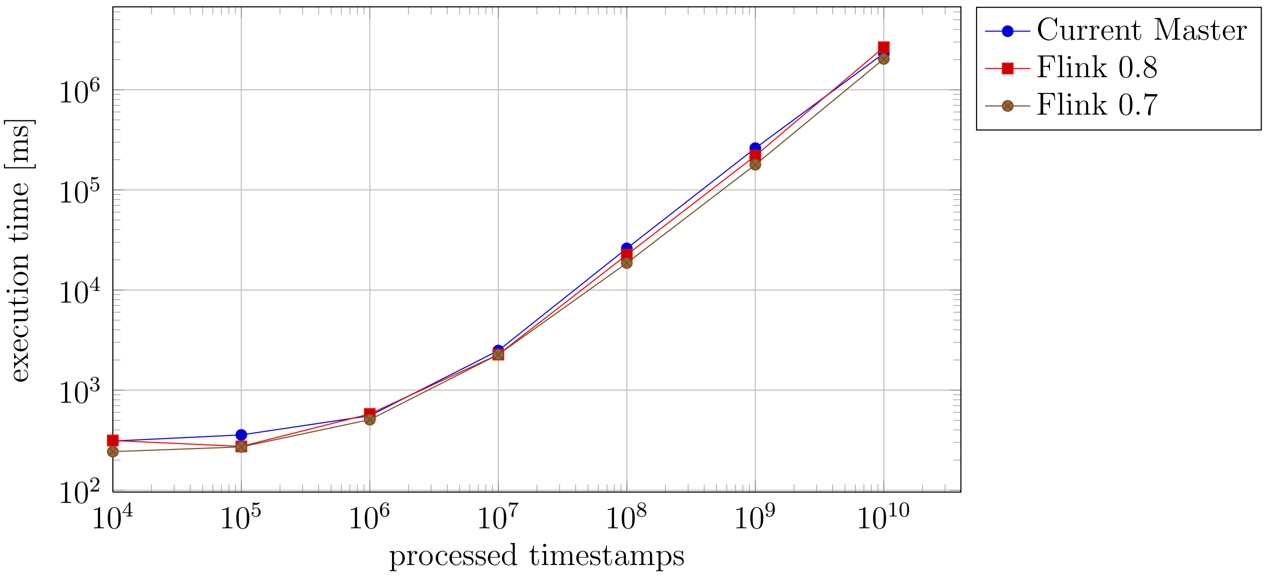


Figure 15: Performance evaluation for discretization operators with logarithmic scales.

Compared Operators. The three different discretization operators we tested are:

1. **Flink version 0.7** The version which was present before the work on this thesis started).

2. **Flink verion 0.8.1.** The released version of the operators presented in this thesis

3. **Current Master.** A further optimized implementation done by Gyula Fóra, which will be included in the next release. This version allows a parallel execution, but removed the ability of having multiple policies at the same time.

Test Data. As input, we used a sequence of `long`-values. The size shown on the x-axis of our plots is the number of values in this sequence. We used a custom `Timestamp` implementation to interpret the generated `long`-values as timestamps. Thus, the x-axis depicts the number of timestamps processed by the discretization operator and the y-axes shows the time in milliseconds which was required for the processing.

Interpretation of the Results. Figure 15 shows the measured execution times for the processing of up to 10^{10} (10 billion) timestamps. Both scales are logarithmic. Up to 10^6 processed timestamps, the startup time has a considerable effect. Afterwards the execution time increases linearly with the amount of processed data for all three versions of the discretization operators. This is the expected result. The buffer sizes are not effected by the total amount of processed data and the computation which needs to be

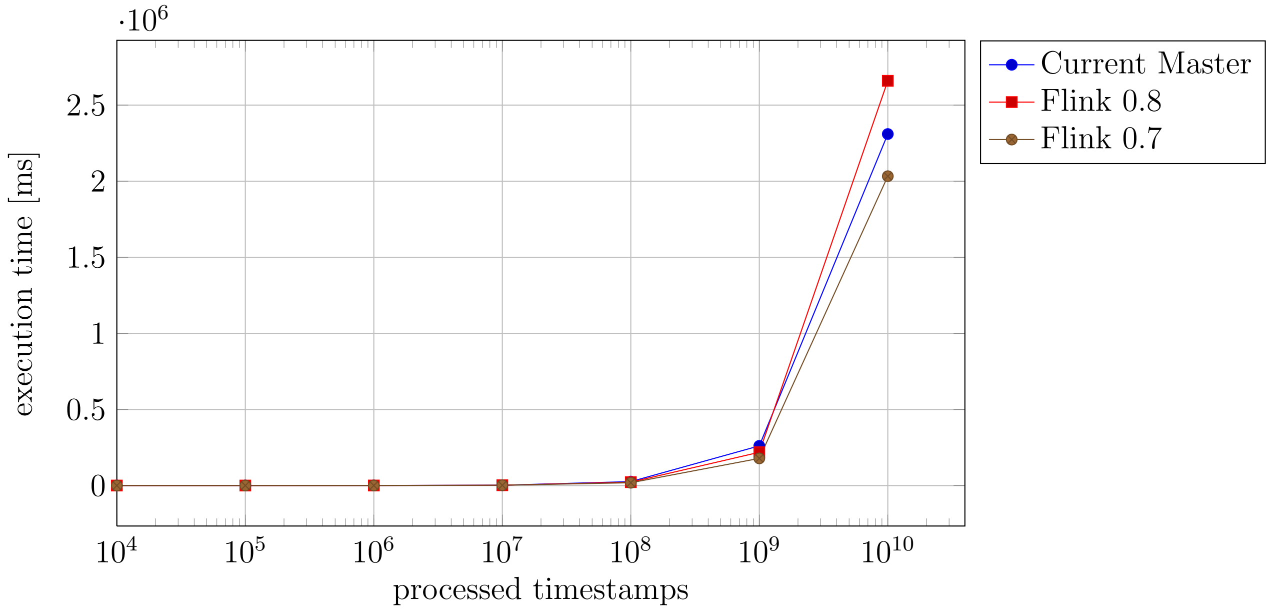


Figure 16: Performance evaluation for discretization operators with linear time scale.

done per individual data-item also remains the same. Thus, there is no reason why any of the three solution should not scale linearly.

In Figure 16 we changed the y-axis to have a linear scale. Beside this, the figure depicts the identical measurements. On the linear scale it is good to see that even though all three versions of the window discretization scale linearly, there is a performance difference between them. The 0.7 version is the fastest, which comes from the fact that it has no overhead at all. It is a single stand alone operator which can only do time-based window discretization but nothing else. This operator is well optimized for the single use case we tested. The version presented in this thesis (v 0.8) is the slowest, but it provides the best expressiveness, as it is also able to do discretization with user-defined policies and even with multiple policies at the same time.

The remaining solution (current master) shows a performance between the two mentioned before. It is faster than the solution presented in this thesis, because further optimizations were made and the ability of having multiple policies at the same time was removed. It runs still slower than the 0.7 version, because it also has an higher expressiveness causing the overhead of policy notifications. As the current master version was made to be able to run in parallel (see section 7.3), it has to start up multiple operators instead of only one. This causes an increased overall execution time for our query, which is considerable when smaller amounts of data are processed.

10 Conclusion

In this work we introduced highly expressive means of window discretization, going much beyond the one which have been provided earlier by any stream processing system or language. Windowing semantics are decomposed into trigger and eviction functions, that can be combined arbitrarily. We generalized this functions, such that they can be implemented as *User Defined Functions (UDFs)*.

We showed that there are many parallelization and optimization means that apply when using our discretization operators. The presented border-to-border pre-aggregation serves always equal or better performance than solutions based on the greatest common divisor (panes).

We proved in an experiment, that (apart from higher expressiveness) the throughput of our discretization operator scales linearly with the amount of processed data. The application of the presented parallelizations can make the processing even faster. The introduced higher expressiveness does not have preponderating effects on the operator throughput.

The presented windowing semantics have been integrated in the Apache Flink open source project and are deployed in production by several Flink users.

We plan to extend our windowing operators with more parallelization techniques. Furthermore, we believe that there are many research opportunities for single and multi query optimization using policy based window discretization.

References

[1] *Infotech. State of the art report. Software testing. Analysis and bibliography.*, volume 1. Infotech International Limited, Maidenhead, Berkshire, England, 1979.

[2] association, n. *The Oxford English Dictionary (OED)*, 2014. Web. Accessed Feb. 15.

[3] Daniel J Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal—The International Journal on Very Large Data Bases*, 12(2):120–139, 2003.

[4] Charu C Aggarwal and S Yu Philip. A survey of synopsis construction in data streams. In *Data Streams*, pages 169–207. Springer, 2007.

[5] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal—The International Journal on Very Large Data Bases*, 23(6):939–964, 2014.

[6] Henrique Andrade, Buğra Gedik, and Deepak Turaga. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics.* Cambridge University Press, 2014.

[7] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[8] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 350–361. IEEE, 2004.

[9] Shivnath Babu and Herodotos Herodotou. Massively parallel databases and mapreduce systems. *Foundations and Trends in Databases*, 5(1):1–104, 2013.

[10] Márton Balassi, Gyula Fóra, and others (Git Contributers). *Flink Stream Processing API (Version 0.7.0).* Apache Software Foundation, 2014. accessed 11.03.2015 from archives, `http://flink.apache.org/archive.html`.

[11] Márton Balassi, Gyula Fóra, Jonas Traub, and others (Git Contributers). *Flink Stream Processing API (Version 0.8.1).* Apache Software Foundation, 2015. accessed

23.03.2015, `http://ci.apache.org/projects/flink/flink-docs-release-0.8/streaming_guide.html`.

[12] Richard Barker. *Case*Method: Entity Relationship Modelling.* Addison-Wesley Professional, Jan. 1990.

[13] Graham Bath and Judy McKay. *Praxiswissen Softwaretest - Test Analyst und Technical Test Analyst: Aus- und Weiterbildung zum Certified Tester - Advanced Level nach ISTQB-Standard. The software test engineers handbook.* dpunkt, Heidelberg, 2. edition, 2011.

[14] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 119–130. ACM, 2010.

[15] Boris Beizer. *Software testing techniques.* Van Nostrand Reinhold, New York, 2. edition, 1990.

[16] Thomas Bernhardt and Alexandre Vasseur. Esper: Event stream processing and correlation. *ONJava, O'Reilly*, 2007.

[17] Luc Bouge. The data parallel programming model: A semantic perspective. In *The Data Parallel Programming Model*, pages 4–26. Springer, 1996.

[18] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 215–226. VLDB Endowment, 2002.

[19] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *ACM SIGMOD Record*, volume 29, pages 379–390. ACM, 2000.

[20] Peter Coad and Jill Nicola. *Object-oriented programming.* Yourdon Press and Prentice-Hall Internat., Englewood Cliffs, NJ, 1993.

[21] Peter Coad and Edward Yourdon. *Object oriented analysis.* Yourdon Press and Prentice-Hall Internat., Englewood Cliffs, NJ, 2. edition, 1991.

[22] Peter Coad and Edward Yourdon. *Object-oriented design.* Yourdon Press and Prentice-Hall Internat., Englewood Cliffs, NJ, 1991.

[23] Richard Connor and Robert Moss. A multivariate correlation distance for vector spaces. In *Similarity Search and Applications*, pages 209–225. Springer, 2012.

[24] Brad J. Cox. *Object-oriented programming : an evolutionary approach.* Addison-Wesley, Sandy Hook, CT, 1986.

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[26] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Communications of the ACM*, 53(1):72–77, 2010.

[27] Dan Debrunner. *SPL Sliding Windows Explained.* IBM developerWorks / Developer Centers, Aug. 2014. `https://developer.ibm.com/streamsdev/2014/08/22/spl-sliding-windows-explained/` (accessed Jan. 2015).

[28] Dan Debrunner. *SPL Tumbling Windows Explained.* IBM developerWorks / Developer Centers, May 2014. `https://developer.ibm.com/streamsdev/2014/05/06/spl-tumbling-windows-explained/` (accessed Jan. 2015).

[29] Albert Einstein. *Relativity: The special and general theory.* Penguin, 1920.

[30] Stephan Ewen. *Compiling Data-Parallel Analysis Programs for Distributed Dataflow Execution.* Technische Universität Berlin, Berlin, Germany, 2014.

[31] Michael Feathers. A set of unit testing rules. 2005. accessed 26.03.2015, `http://www.artima.com/weblogs/viewpost.jsp?thread=126923`.

[32] International Organization for Standardization (ISO), editor. *ISO 5807 - Information processing - Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts.* ISO, 1985.

[33] Martin Fowler, Kent Beck, John Brant, and William Opdyke. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[34] Deutsches Institut für Normung e.V. (DIN), editor. *DIN 66001 - Informationsverarbeitung; Sinnbilder und ihre Anwendung (Information processing; graphical symbols and their application).* Beuth Verlag, Dec. 1983.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[36] John F. Gantz, Christopher Chute, Alex Manfrediz, Stephen Minton, David Reinsel, Wolfgang Schlichting, and Anna Toncheva. *The Diverse and Exploding Digital Universe - An Updated Forecast of Worldwide Information Growth Through 2011*. IDC sponsored by EMC, March 2008.

[37] Buğra Gedik. Generic windowing support for extensible stream processing systems. *Software: Practice and Experience*, 44(9):1105–1128, 2014.

[38] E. Šilov Georgij. *Linear algebra*. Prentice-Hall, Englewood Cliffs, N.J., 1971.

[39] Thanaa M Ghanem, Moustafa A Hammad, Mohamed F Mokbel, Walid G Aref, and Ahmed K Elmagarmid. Incremental evaluation of sliding-window queries over data streams. *Knowledge and Data Engineering, IEEE Transactions on*, 19(1):57–72, 2007.

[40] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[41] Lukasz Golab and M Tamer Özsu. Issues in data stream management. *ACM Sigmod Record*, 32(2):5–14, 2003.

[42] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, (2):156–173, 1975.

[43] Torsten Grabs, Roman Schindlauer, Ramkumar Krishnan, Jonathan Goldstein, and Rafael Fernández. Introducing microsoft streaminsight. Technical report, Technical report, 2009.

[44] Martin Hirzel, Henrique Andrade, Bugra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, et al. Ibm streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7–1, 2013.

[45] Martin Hirzel, Henrique Andrade, Bugra Gedik, Vibhore Kumar, Giuliano Losa, MMH Nasgaard, R Soule, and KL Wu. Spl stream processing language specification. *IBM Research, Yorktown Heights, NY, USA, Tech. Rep. RC24*, 897, 2009.

[46] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)*, 46(4):46, 2014.

[47] Fabian Hueske, Mathias Peters, Matthias J Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the black boxes in data flow opti-

mization. *Proceedings of the VLDB Endowment*, 5(11):1256–1267, 2012.

[48] Valerie Illingworth, editor. *Dictionary of computing.* Oxford University Press, Oxford, 2. edition, 1986.

[49] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*, 2011.

[50] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. On-the-fly sharing for streamed aggregation. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 623–634. ACM, 2006.

[51] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *ACM SIGMOD Record*, 34(1):39–44, 2005.

[52] Zohar Manna and Richard Waldinger. The logic of computer programming. *IEEE transactions on Software Engineering*, 4(3):199–229, 1978.

[53] Volker Markl and Max Heimel. Internals of database systems: Web-scale data management - analytical processing. Lecture Slides. Fachgebiet Datenbanksysteme und Informationsmanagement. Technische Universität Berlin., 2014.

[54] Nathan Marz. *A Storm is coming: more details and plans for release.* Twitter Engineering Blog, Aug. 2011. `https://blog.twitter.com/2011/storm-coming-more-details-and-plans-release` (accessed Jan. 2015).

[55] Klaus Meffert. *JUnit Profi-Tipps.* entwickler.press, 2005.

[56] E. F. Miller. An annotated bibliography of software testing. *Infotech. State of the art report. Software testing. Analysis and bibliography.*, 1:275–305, 1979.

[57] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[58] Cedric Otaku. Why unit tests are disappearing. 2005. accessed 26.03.2015, `http://beust.com/weblog2/archives/000319.html`.

[59] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets.* Cambridge University Press, 2011.

[60] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[61] Kathy Sierra and Bert Bates. *SCJP Sun Certified Programmer for Java 6 Study Guide : Exam (310-065).* McGraw-Hill Osborne Media, Sandy Hook, CT, 2008.

[62] Jonas Traub, Gyula Fóra, and Paris Carbone. *Flink Streaming windowing rework.* 2015. Contribution to mailing list dev@flink.incubator.apache.org (06.12.2014).

[63] Alexey Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106, 2004.

[64] unknown. *Flink: General Architecture and Process Model.* Apache Software Foundation, 2015. accessed 30.03.2015, `http://ci.apache.org/projects/flink/flink-docs-master/internal_general_arch.html`.

[65] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[66] Jason Venner. *Pro Hadoop.* Apress, 2009.

[67] Richard Warburton. *Java 8 Lambdas : functional programming for the masses.* O'Reilly, Sebastopol, CA, 1. edition, 2014.

[68] Tom White. *Hadoop - The Definitive Guide.* O'Reilly, Sebastopol, CA, 3. edition, 2012.

[69] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: interfaces and implementations. In *ACM SIGOPS*, 2009.

[70] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Ccomputing*, pages 10–10. USENIX Association, 2012.

[71] Yunyue Zhu and Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th international conference on Very Large Data Bases*, pages 358–369. VLDB Endowment, 2002.

Appendix

A Implementation Backgrounds

A.1 Object Oriented Programming

Object oriented programming is a widely used approach in software development. The implementations presented in this thesis are done in Java, one of the most popular object orientated programming languages. Object orientation provides many advantages especially for managing the complexity of huge software projects. With its principles it provides uniformity, understandability, flexibility, stability and reusability of implementations. [20]

To understand the architectural concepts presented in this thesis it is highly important to understand the principles of object oriented programming. Object oriented programming became famous in the beginning of the 90th and several authors stated it to be an revolutionary change in programming [20, 24]. While in the beginning *Smalltalk* and *C++* where the most used object oriented programming languages, nowadays *Java* evolved to be most widely used, especially in big data processing.

Moreover, the development of the solutions presented in this thesis was done using Java, hence presented architectural concepts and the explanations in this section are related to the Java language and do not include any features, which are unavailable there, such as multiple inheritance or overloading with different return types. Anyhow, we summarize the principles of the object oriented programming model and clearly define the terminology used in the following chapters. We do not provide a full tutorial threw the Java language, nevertheless, we state specialities of Java wherever we consider them to be relevant for the following sections.

Before we come to more details, let's have a look on the most elementary definitions:

Object In general "an object is a person, place, or thing." [20] In object oriented design it is "an *abstraction* of something in the domain of a problem or its implementation, reflecting the capabilities of a system to keep information about it, interact with it, or both; an *encapsulation* of attribute values and their exclusive services." [22]

Class In general "a class is a grouping of objects together, based on common characteristics." [20] In object-oriented design, it is a "description of one ore more objects, describable with a uniform set of attributes and services; in addition, it may describe how to create new objects in the class." [22] In Java, when a new object of a class is created, a special method of the class, called *constructor*, is executed.

Abstract Class "In OOP a class that has no objects is called an abstract class" [20] "An abstract class can never be instantiated. Its sole purpose, mission in life, raison d'être, is to be extended (subclassed)." [61] This allows the reuse of code by inheritance and to write code working with instances of (we define *instance of* below) the abstract class, even if not all functionality can be provided on the abstraction level of the abstract class.

Interface Interfaces are special versions of abstract classes having only abstract methods.[8] "When you create an interface, you're defining a contract for what a class can do, without saying anything about how the class will do it. An interface is a contract." [61] All classes implementing the interface must implement all the methods specified by the interface (except the class is abstract). [61]

Package "Java organizes classes into packages, and uses import statements to give programmers a consistent way to manage naming of, and access to, classes they need." [61] The package structure is same as the folder structure in the file system, where the code is located.

The main idea behind object orientation is an *encapsulation* of data by code. In principle all data in an object should (and can[9]) only be accessed threw procedures mediating access to the data. [24] Thereby, the data and the logic working with the data is encapsulated within an object. This clearly separates object oriented programming from earlier approaches, which tried to separate the data and the program state from the logic. The second main approach is *inheritance*. It allows to share common behaviour among multiple classes. Programmers don't start each class from scratch, but make their class extend an already present one from the library. Thereby an hierarchy of classes is build where code is broadcasted automatically from top to bottom. The top classes are the most general ones, while each inheriting class is a specialisation of the extended class. The statements written by the programmer define how the specialised class differs from the general one. [24]

So far we already mentioned three techniques used for managing complexity: Abstraction, Encapsulation and Inheritance. One more is Association. [21] All of these techniques make Object orientation a powerful tool to handle complexity. In the following we provide definitions of the four mentioned techniques. Further techniques and detailed descriptions of each technique can be found in the book *Object-oriented analysis* [21] by Coad and Yourdon.

[8] With Java 8, Oracle introduced default implementations of methods in interfaces. Therefore it is nowadays possible to put logic inside an interface. [67] In this thesis we only use Java up to version 7 and do not use default implementations in interfaces.

[9] In Java the technical possibility of accessing a variable directly rather than threw a procedure call, depends on the selected access modifier.

Abstraction ”The principle of ignoring those aspects of a subject that are not relevant to the current purpose in order to concentrate solely on those that are” [48]

Encapsulation ”A principle, used when developing an overall program structure, that each component of a program should encapsulate or hide a single design decision. [...] The interface to each module is defined in such a way as to reveal as little as possible about its inner workings.” [48] (Synonym: Information hiding) [21]

Inheritance ”A mechanism for expressing similarity among classes, simplifying definition of classes similar to one(s) previously defined. It portrays generalisation and specialisation, making common attributes and services explicit within a class hierarchy or lattice.” [21]

Association ”The action of combining together for a common purpose.” [2] ”People use association to tie together certain things that happen at some point in time or under similar circumstances.” [21]

Different classes and objects can be associated with each other using different kinds of relationships which are defined in the following.

is-a (also referred to as *gen-spec* [20]) ”In OO, the concept of IS-A is based on class inheritance or interface implementation. IS-A is a way of saying ’This thing is a type of that thing.’ For example, a Mustang is a type of horse, so in OO terms we can say, ’Mustang IS-A Horse.’ [...] You express the IS-A relationship in Java through the keywords `extends` (for class inheritance) and `implements` (for interface implementation).” [61]

has-a ”HAS-A relationships are based on usage, rather than inheritance. In other words, class A HAS-A B if code in class A has a reference to an instance of class B.” [61]

instance of An object of class A is an instance of class B if A IS-A B. An object of A is an instance of A, all classes A inherits from, and all interfaces it implements. Additionally, when a class extends another, it inherits also the IS-A relationships of the extended class. [61] ”In Java the `instanceof` operator is used for object reference variables only, and you can use it to check whether an object is of a particular type.” [61]

Each class consists of procedures, which are called methods in Java, and variables (and constants), which are called attributes. The combination of the values of the attributes of an object is said to be the objects state. In Java, programmers have the possibility to use overloading and overriding when implementing methods in a class.

Overloading "Overloaded methods let you reuse the same method name in a class, but with different arguments. [...] Overloaded methods MUST change the argument list." [61] They are allowed to change the return type and the access modifier and they can throw broader or new checked exceptions. [61]

Overriding "Any time you have a class that inherits a method from a superclass, you have the opportunity to override the method.[10] [...] The key benefit of overriding is the ability to define behavior that's specific to a particular subclass type." [61] Overriding a method means, that a method with the same signature[11] is implemented in a inheriting class, replacing the functionality inherited from the extended class.

When developing an object oriented program, the goal is to archive tight encapsulation, loose coupling, and high cohesion in classes. We already described encapsulation. Let's have a look at coupling and cohesion as well now to complete the set of architectural goals.

Coupling "Coupling is the degree to which one class knows about another class. If the only knowledge that class A has about class B, is what class B has exposed through its interface, then class A and class B are said to be loosely coupled [...] If, on the other hand, class A relies on parts of class B that are not part of class B's interface, then the coupling between the classes is tighter" [61]

Cohesion "Cohesion is all about how a single class is designed. The term cohesion is used to indicate the degree to which a class has a single, well-focused purpose. [...] The more focused a class is, the higher its cohesiveness [...] The key benefit of high cohesion is that such classes are typically much easier to maintain [...] than classes with low cohesion. Another benefit of high cohesion is that classes with a well-focused purpose tend to be more reusable than other classes." [61]

[10] In Java, "A method marked with the final keyword cannot be overridden." [61] A class marked with the final keyword cannot get extended. "You cannot override a method marked static." [61]

[11] In Java, the signature don't have to match exactly. The access modifier can be same or less restrictive, the return type can be any subtype of one declared in the overridden method, and thrown catched-exception can be any subtype of the one declared by the overridden method. [61]

A.2 Software Testing

In software engineering, software testing is an essential part of quality assurance. Testing allows to detect symptoms caused by bugs rather then preventing bugs in the system. With a clear diagnoses of such symptoms bugs can be corrected easily. Anyhow, preventing bugs is the main goal of software testing. Already the test design requires thinking about possible corner cases and program states and thereby lowers the risk of introducing bugs before the software is coded. [15]

More formalized, ”Testing is a collection of activities that provides a practical demonstration of conformity between the program and the specification, based upon systematic selection of test cases and execution of program paths and segments.” (Wasserman [1]). According to this, testing is clearly separated from debugging, which ”is to find a known error, or to find out if there are any obvious errors”(Miller [1]).

Software testing is one of the oldest fields in computer science. Already in the end of the 70th there was a state of the art report on this topic [1] and even in this report authors refer to a long history of software testing. Miller, who also published an early annotated bibliography [56] remarks: ”In fact, there is even a citation to Turing indicating that ’testing is the empirical way of software quality assurance, while proving the theoretical way’” [1].

Already at this early time the field was split in the sub-fields ”verification techniques”, ”static testing”, ”test data selection” and the ”automation of testing techniques”. [1] Verification techniques try to logically proof the correctness of an application. Thereby it is closely related to many proofing techniques known from today’s field of theoretical computer science. Unfortunately, a formal proof is in most cases as error-prone as testing while the effort to spent is much higher. [42] Static testing (also called static analysis [13, 15]) means testing without running the code. [1] According to this, the simplest method of static testing is a manual code review. Automated testing (also called dynamic analysis [13, 15]) is the opposite of static testing where at least parts of the code are executed during the test.

Whenever a test is automated, the selection of sufficient test data is crucial to ensure the tested software reaches all states one wants to test. Until today automated software testing has evolved to be practically applicable and many test automation frameworks are available on the market. Nevertheless, the three general limitations first stated by Manna 1978 [52] remained unchanged:

1. ”We can never be sure that the specifications are correct.”

2. ”No verification system can verify every correct program.”

3. "We can never be certain that a verification system is correct."

Having this limitations in mind one has to define a reasonable trade-off between effort to spent on test design and the complexity of the execution of the test on one side and the fraction of the situations/states which are validated by the test on the other side.

During the last decade, test standardisation continuously won in importance and job roles such as *test analyst* and *technical test analyst* have been established, including standardized certification programs for professionals. [13] Thus, kinds of systems, the steps of test processes, testing approaches, and test purposes have been pointed out and categorized in literature. *The software test engineers handbook* [13] provides a complete list and description.

A.3 Unit testing

All our test cases for automated testing are implemented using the JUnit test framework. We designed and implemented a wide range of tests to ensure a correct implementation as far as possible (Remember the three genera limitations of testing [52] stated in section A.2).

JUnit is a unit test framework for programs implemented in the Java programming language. In general, as the name tells, automated tests per unit *unit* are executed. A *unit* is an isolated and understandable (small) piece of code. In object oriented programming, units are usually represented by classes or methods. [55]

Different definitions of unit tests are present in literature. In 2005, Feathers [31] stated that different types of tests are no unit tests. Especially, if they talk to a database, communicates across the network or touch the file system. Otaku [58] criticises this limiting definition. [55] As we are implementing unit tests for a big data processing engine, it is important to run tests accessing files and data-bases. Even though it might not fit to all unit-test definitions, we use the term unit-test for test-cases which can be implemented in JUnit.

As classes and methods can be seen as units, it makes sense to have a one to one mapping between *test classes* (implementing the tests) and *real classes* (implementing the logic to be tested) and an one to one mapping between accessible methods in the real class and methods in the corresponding test class. The test classes are then named *test cases*. The methods in the test classes are called *test methods*. [55]

When a test case is implemented in JUnit, the programmer marks the test method with an `@Test` annotation. Inside the test method implementation, the programmer can use different assert-methods to compare actual with expected results of the program execution. [55]

Listing 10 shows an example test case implementation. The test class called `MyClassTest` tests a method in the class `MyClass`. It is a method which calculates sum of two numbers. For the reason of simplicity, the test is not complete in the sense, that is is sufficient to prove the correct implementation. It just verifies that a the single addition $2 + 3 = 5$ is done correctly. If not, the test will fail and an error message will be printed.

```java
public class MyClassTest {
  @Test
  public void sumTest() {
    assertEquals("2 + 3 must be 5", 5, MyClass.sum(2,3));
  }
}
```

Listing 10: An example JUnit test case implementation

In the Apache Flink project, JUnit is the standard tool for the implementation of automated tests. Hundreds of test cases are present in the project to ensure that all implemented features still work correctly after a change was made.

B Implementation

B.1 Overview

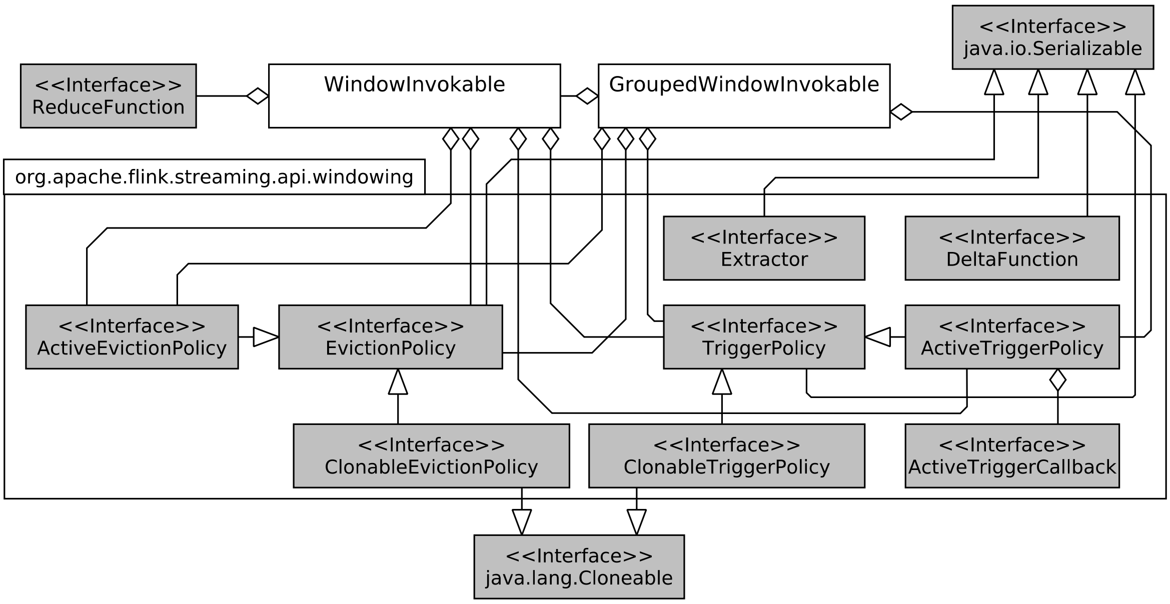


Figure 17: Implementation overview: Discretization operator implementations and interfaces in the windowing package.

Figure 17 shows the two newly introduced discretization operators and their dependencies. More generally spoken, it depicts all the *is-a* and *has-a* relationships between the shown classes and interfaces, which are part of the solution we propose.

In the following sections, we will describe the different interfaces and the operators in more detail.

B.2 The Window Operator

In this section we will describe the execution flow at the `WindowInvokable`. In section 6.9, we introduce active policies to cover some special cases. In section B.2.1 we present the basic execution flow without active polices. In section B.2.2 we extend it to also deal with active *TPs* and *EPs*.

B.2.1 Basic Execution Flow

One of the main ideas behind the concept we present is, that the window discretization is done by an own operator. The `WindowInvokable` implements the logic for this operator. The *TPs* and *EPs* are parameters of the discretization operator. This clearly separates the approach from *SPL* where the window discretization is set as a parameter to an operator instead of being an operator itself.

Another main point is that the discretation operator is event driven. All actions which happen at the `WindowInvokable` are always caused by the arrival of a data-item. A data-item can either be *real* or *fake*. Real data-items are all data-items which come from a data source[12]. Fake data-items are all data-items which are produced by `ActiveTriggerPolicy` implementations. For the moment, we only take real data-items into consideration.

For any real data-item arrival the following actions are executed:

1. The *TPs* are notified with the arrived data-item as parameter. Thus, a check is executed whether the arrived data-item is a window end border or not. In case multiple *TPs* are present, all of them are notified. A result is emitted in case at least one of them triggers.

2. In case a trigger occurred, the user-defined reduce function is executed on the current data-item buffer and the result is emitted. This automatically makes the first data-item in the buffer the begin border of the window.

3. The *EPs* are notified with the arrived data-item as parameter. In case multiple *EPs* are present, all of them are notified. For the further actions, the greatest returned value is considered as the number of data-items to delete from the buffer.

4. If the number of data-items to evict from the buffer is greater than zero, the respective number of data-items is deleted from the buffer.

5. The arrived data-item is added to the data-item buffer. Because this is done as last step, there will be always at least one data-item in the buffer after the first data-item arrival. This also leads to the behaviour that in case a data-item causes a trigger, it will not be contained in the result emission it causes. In section 6 we explained why this behaviour is desirable.

[12] Note, that *data source* means a data source of the `WindowInvokable` here. Any vertex/operator can be connected to the `WindowInvokable` and will then be its data source. It doesn't have to be a `DataStreamSource`.

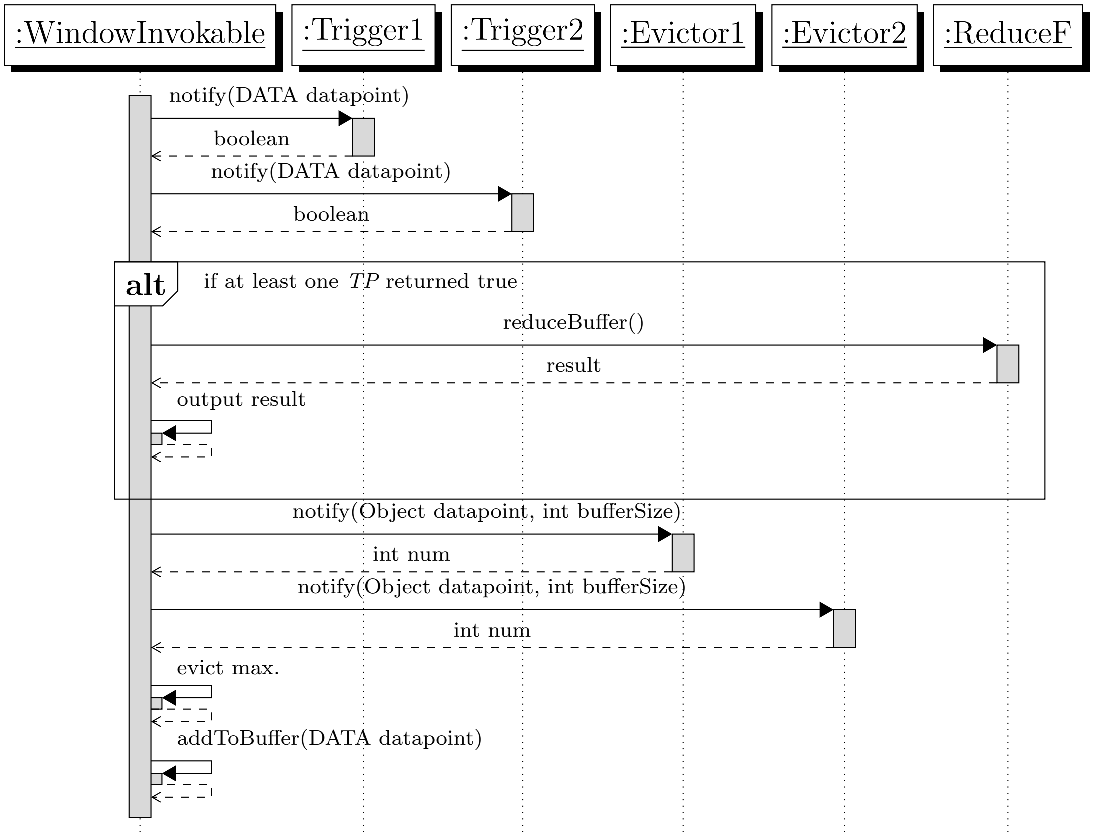


Figure 18: Sequence diagram for the basic execution flow at the window invokable.

Figure 18 visualizes the explained execution flow for an example query as *Unified Modelling Language (UML)* sequence diagram. The diagram shows all processing steps which are done when a data-item arrives. Two *TPs* and two *EPs* are used in the example. In general, any number of *TPs* and *EPs* can be used at the same time, but there has to be at least one of each kind. The two *TPs* and the two *EPs* are always notified directly after each other. In case there would be more than two of them, the third would follow directly after the second, the fourth after the third and so on and so forth.

B.2.2 Extended Execution Flow with Active Policies

The extended execution flow at the `WindowInvokable` covers the processing of fake data-items in addition to real data-items. As soon as one active trigger policy is present, fake data-items can appear in addition to real data-items in two ways:

1. They can be returned by the pre-notification method of an active *TP* when a real data-item arrives.
2. They can be created by a `Runnable` which is returned by the factory method of an active *TP* and is executed in a separated thread.

Due to the first point, the execution flow needs to be extended such that, it first calls the pre-notification methods of active *TPs* and afterwards processes the fake data-items they returned. Both needs to happen before the regular notification methods of the *TPs* are called. If multiple fake data-items are returned from pre-notification methods of the active *TPs*, they are processed one after each other in the order they are returned. For each fake data-item, the eviction happens first and the calculation and the emission of the result for the current buffer is done afterwards (Action order *eviction*→*trigger*).

Figure 19 visualises the extended execution flow for an example query as *UML* sequence diagram. The diagram shows all processing steps which are done when a real data-item arrives. Two active *TPs* and two active *EPs* are used in the example. In general, any number of *TPs* and *EPs* can be used at the same time, but there has to be at least one of each kind. The two *TPs* and the two *EP* are always notified directly after each other. In case there would be more than two of them, the third would follow directly after the second, the fourth after the third and so on and so forth. There is also no limitation in the combination of active and not active *TPs* and *EPs*. In case a *TP* is not active it is not pre-notified and cannot produce fake data-items. In case an *EP* is not active, it is not notified for fake data-items.

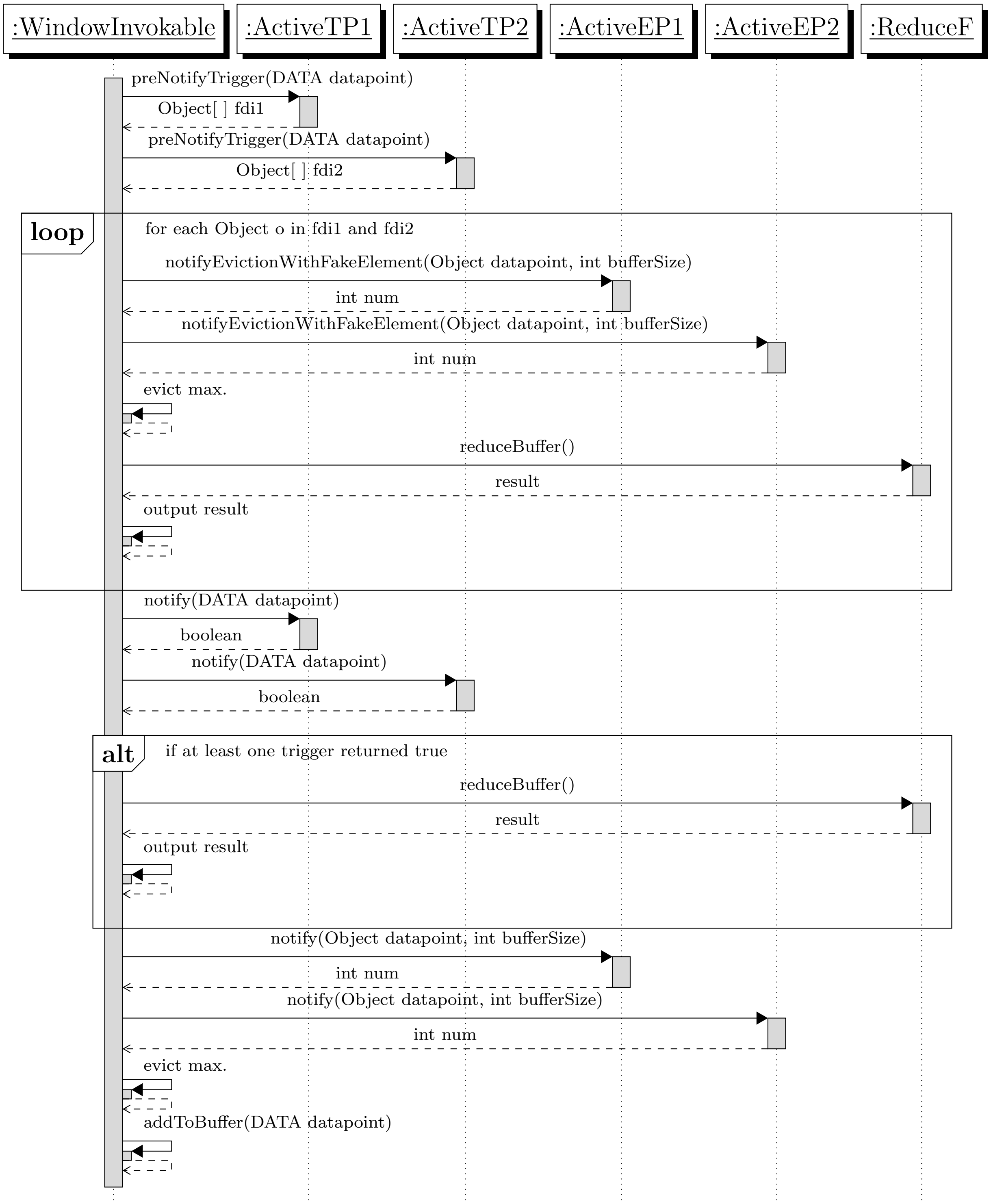

:WindowInvokable
:ActiveTP1
:ActiveTP2
:ActiveEP1
:ActiveEP2
:ReduceF
preNotifyTrigger(DATA datapoint)
Object[ ] fdi1
preNotifyTrigger(DATA datapoint)
Object[ ] fdi2
loop
for each Object o in fdi1 and fdi2
notifyEvictionWithFakeElement(Object datapoint, int bufferSize)
int num
notifyEvictionWithFakeElement(Object datapoint, int bufferSize)
int num
evict max.
reduceBuffer()
result
output result
notify(DATA datapoint)
boolean
notify(DATA datapoint)
boolean
alt
if at least one trigger returned true
reduceBuffer()
result
output result
notify(Object datapoint, int bufferSize)
int num
notify(Object datapoint, int bufferSize)
int num
evict max.
addToBuffer(DATA datapoint)

Figure 19: Sequence diagram for the extended execution flow at the window invokable.

The calls to the factory methods (for `Runnable` instances) in the active *TPs* are done when the `WindowInvokable` is instantiated. Also the separated threads, which execute the returned runnables, are created and started at the time of instantiation. The logic for both is implemented in the constructor of the `WindowInvokable`. Hence, this is not part of the execution flow depicted in Figure 19, because it is completely independent from the arrival of real data-items.

Due to the creation of fake data-items in separated threads, the `WindowInvokable` cannot be seen as single threaded any more and we have to think about problems coming up, when the processing flows for multiple data-items run concurrently. While a real data-item is processed, no fake data-items produced by other threads can be processed and vice versa. Otherwise, the processing flows possibly interleave each other which would cause unexpected results because of unexpected data-item deletions from the buffer.

It's important to understand, that fake data-items, which are created by the pre-notification method of an active trigger policy, are not created by another thread, but within the same thread. Accordingly, they are processed before the real data-item as usual.

In Figure 19, we depicted the execution for real data-item arrivals. This automatically includes the processing of fake data-items which are produced when the pre-notification methods of active *TPs* are called. The execution flow for fake data-item arrivals, which are caused by the creation of fake data-items through separated threads, should never be executed at the same time. Both flows are accessed through a method in the `WindowInvokable`. We prevent the interleaving of the two execution flows by adding Java's `synchronized` keyword to these methods. This provides us the guarantee, that never both execution flows are executed at the same time and that no interleaving is possible.

In figure Figure 20, we depict the execution flow for fake data-item arrivals which are caused by the creation of fake data-items through separated threads. You can assume the same setup as in the previous example. Because fake data-items are not send to any *TP* again, we removed the lines for the two active *TPs*. Instead, we added two new lines for the `Runnable` which creates the fake data-item and the callback object (`ActiveTriggerCallback`) it uses to submit the fake data-item. `ActiveTriggerCallback` is implemented as inner class in the `WindowInvokable`. Hence, it has access to its methods and can forward the produced fake data-item. The `Runnable` can be implemented as inner class in an active *TP*. Hence, it can get access to the state of the *TP*.

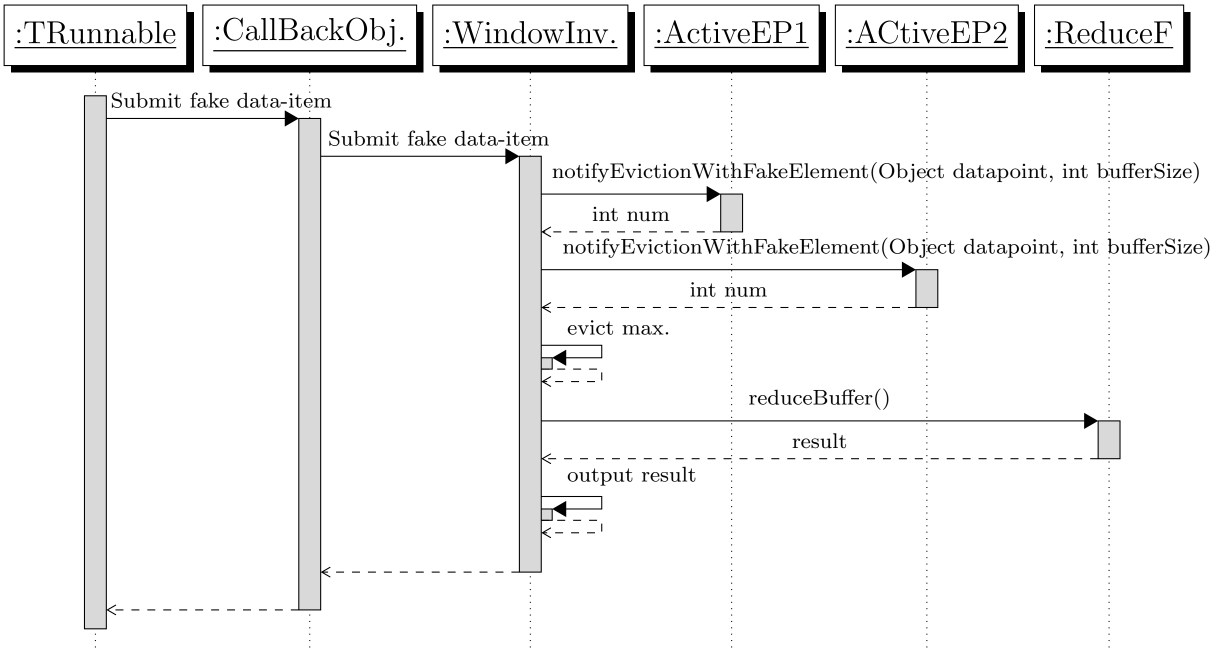


Figure 20: Sequence diagram for the extended execution flow at the window invokable for fake data-item arrivals which are caused by the creation of fake data-items through separated threads.

B.3 The Trigger Policy Interface

On each data-item arrival, a *TP* proves and returns whether a window end has been reached or not. If so, a result for the current buffer, which represents the window, is emitted. We say *'a trigger occurs'* meaning that the notification method of some trigger returns `true`.

```
public boolean notifyTrigger(DATA datapoint);
```

Listing 11: The interface for trigger policies.

In section 6.4, we described our fixed action order. By deciding for only one fixed action order, we are able to keep the interface for *TPs* as simple as possible, defining only one method shown in Listing 11. To make it applicable for arbitrary kinds of data streams, the `TriggerPolicy`-interface uses the generic type `DATA`, representing the type of the data-items it expects as parameter of the notification method.

B.4 The Eviction Policy Interface

An *Eviction Policy (EP)* specifies under which condition data-items should be deleted from the buffer. Like a *TP*, an *EP* is notified threw a notification method, whenever a data-item arrives. On data-item arrival, the policy proves if and how many data-items should be deleted from the data-item buffer.

The buffer is a *FIFO* buffer. Data-items can only be deleted in the order they arrived. Therefore, the notification method only returns the number of data-items to be removed from the buffer instead of references to specific objects.

```
public int notifyEviction(DATA datapoint, boolean triggered, int bufferSize);
```

Listing 12: The interface for eviction policies.

The `EvictionPolicy`-interface is shown in Listing 12 and defines only one method. The generic type `DATA` is again the place-holder for the data-item type handled by the policy. Respectively, the `datapoint` parameter is used to give the arrived data-item to the policy. Beside the arrived data-item, the notification method receives two further parameters called `triggered` and `bufferSize`.

The parameter `triggered` of the notification method is not desperately needed, but it allows to achieve a better performance. Hence, we decided to lower the encapsulation of `EvictionPolicy` in order to gain performance benefits. The parameter will be set to `true` in case the given data-item also caused a trigger to occur. In some cases the *EP* can be prevented from doing computations again, which were already done by the previously executed *TP*. The most obvious case is the usage of tumbling windows. Here, the *EP* can just delete the complete buffer in case a trigger occurred.

The remaining parameter contains the current buffer size as data-item count. This parameter is required for some policies in case multiple *EPs* are used at the same time. *EPs* possibly need to know, whether another *EP* caused the occurrence of an eviction. For instance if one wants to keep 1k data-items in the buffer, but never a data-item which is older than 10 minutes. One would use two *EPs* at the same time in this case: A count-based and a time-based. Both need to know, whether the other caused an eviction. The time-based *EP* always needs to check whether the oldest data-item in the buffer is expired. Hence, it needs an information in case the oldest data-item has changed. The count-based *EP* keeps a data-item count. Hence, it needs to update its counter in case the time-based policy caused an eviction.

B.5 The Active Trigger Policy Interface

Active trigger policies are defined with an interface called `ActiveTriggerPolicy`. This interface extends `TriggerPolicy` and thereby inherits its regular notification method. Additionally, two new methods are defined called `preNotifyTrigger` and `createActiveTriggerRunnable`. Listing 13 shows the complete interface for active trigger policies.

```java
public interface ActiveTriggerPolicy<DATA> extends TriggerPolicy<DATA> {
  public Object[] preNotifyTrigger(DATA datapoint);
  public Runnable createActiveTriggerRunnable(ActiveTriggerCallback callback);
}
```

Listing 13: The interface for active trigger policies.

For each real data-item arrival, the `preNotifyTrigger` methods of all present active trigger policies are called first and can produce fake data-items. The produced fake data-items are then processed before the real data-item. Anyhow, after all fake data-items have been processed, the regular notification methods are called as usual. Also with the real data-item as parameter. The policy can still trigger as usual and thereby mark the real data-item as window end (border) which causes the emission of a result.

A speciality of the pre-notification method is its return type. It returns an array of objects instead of an array of the generic type `DATA`. We allowed the method to return instances of the most general possible class, which is `Object` in Java. All classes inherit from `Object`. This prevents active policies from being enforced to produce fake data-items which are instance of exactly the same class as the real data-items. Even though, we expect them to be of the same type from a logical point of view.

There are cases where it makes sense to return instances of a different class than `DATA` as fake data-items. An active time-based trigger can be prevented from having a need to know how to create instances of `DATA` and from having the overhead of doing it. This reduces the complexity of the policy implementations, increases the performance and makes the predefined time-based trigger policy generally applicable for arbitrary types of real data-items.

Instead of fake data-items which are instance of `DATA`, the `TimeTriggerPolicy` returns instances of `Long` as fake data-items which represent the timestamps where the fake data-items shall be placed. Thus, all required information is contained in the `long`-value. Active *EPs* can work with the directly provided timestamp in the same way as they would do it with one they extracted from a real data-item. Anyhow, the *EP* needs to be aware of possibly receiving data-items which are instances of different classes than `DATA`.

Hence, the parameter types in `ActiveEvictionPolicy` are adjusted respectively. We will have a closer look on this interface in section B.6.

The mentioned factory method for runnables is called `createActiveTriggerRunnable`. This method is called at the start-up of the operator. The method receives an instance of `ActiveTriggerCallback` as parameter. Listing 14 shows this interface. The purpose of the callback interface is, to provide the ability to send fake data-items from the produced `Runnable` back to the operator instance.

```
public interface ActiveTriggerCallback {
  public void sendFakeElement(Object datapoint);
}
```

Listing 14: The callback interface for active trigger policies.

We encapsulated the callback in a separated interface to prevent the policies from a need to be changed in case the discretization operators get changed. From the perspective of the `WindowInvokable`, `ActiveTriggerCallback` can be implemented in an inner class. Thus, access rights to methods and variables of the `WindowInvokable` can be easily provided to the `ActiveTriggerCallback` without hurting its cohesion and encapsulation.

B.6 The Active Eviction Policy Interface

There are not only active *TPs* but also active *EPs*. In comparison to the already known *EPs*, an `ActiveEvictionPolicy` is notified even when the processed data-item is a fake data-item. The `ActiveEvictionPolicy` interface is shown in Listing 15. It extends `EvictionPolicy` and adds one further method called `notifyEvictionWithFakeElement`. For each fake data-item, the policy is notified threw this method with the fake data-item and the current buffer size as parameter.

```
public interface ActiveEvictionPolicy<DATA> extends EvictionPolicy<DATA> {
    public int notifyEvictionWithFakeElement(Object datapoint, int bufferSize);
}
```

Listing 15: The interface for active eviction policies.

As already mentioned in the previous section, fake data-items can (from a technical perspective) be instance of any type. Hence, the interface for active eviction policies declares the type of the fake data-item, passed as parameter to the pre-notification method, to be an `Object`. This does not effect the regular notification method which is inherited from `EvictionPolicy`. The regular notification method still expects to receive data-items of the generic type `DATA` as parameter. In most cases, fake data-items will either always be instance of `DATA`, like real data-items, or always be instance of one specific other class. Therefor, it is in most cases not required to do an `instance of` check in the `notifyEvictionWithFakeElement`-method. Anyhow, if one wants to be completely type save, there is no way around it.

Active *EPs* are notified through different methods depending on whether the data-item is real or fake, thus, an *EP* can act differently in the two cases. This provides a high flexibility when implementing user-defined active *EPs*.

In the current implementation of the pre-defined time-based eviction policy, we already utilize this flexibility although there is no logical difference between the processing of real and fake data-items. While in case of real data-items, we extract the time from the data-item using a `Timestamp` instance, we directly send `Long` objects as fake data-items, which represent the timestamps where the fake data-items shall be placed. Hence, the technical realizations of the pre-notification and the notification method are different. One receives timestamps directly but has to perform a type cast from `Object` to `Long` and the other needs to figure out the timestamp by passing the data-item to a `Timestamp`-implementation.

B.7 The Grouped Window Operator

In this section we describe the technical backgrounds of Grouped Discretization Operator presented in section 6.10.

Grouping by Key. In case of Flink streaming, a grouping is always done by key. Keys are extracted from data-items using a `KeySelector` instance. The `KeySelector` interface allows the user to implement the extraction of a key from arbitrary data-item types. Hence, similar to policies, a `KeySelector` receives an arrived data-item as parameter. The return value is then the key of the data-item. Our grouped discretization operator creates one group per unique key.

Overview of Classes and Objects. Figure 21 depicts the different classes, which are used in the grouped case, in an *Entity Relationship (ER)* diagram. Classes are therefore treated as entities. The diagram shows the relations between objects of each type at runtime, including the cardinalities.

The grouped discretization operator is implemented in the class `GroupedWindowInvokable`. At the runtime, the operator has multiple instances of

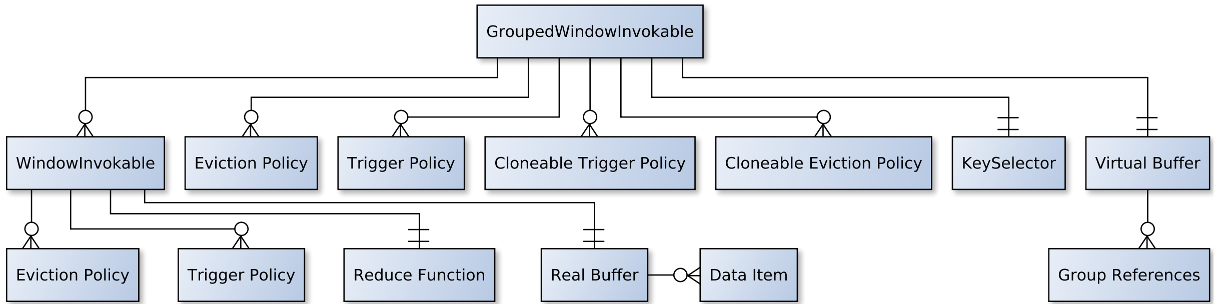


Figure 21: Grouped windowing: Used classes in an *ER* diagram, showing the relations between objects of each type, including the cardinalities, at runtime

`WindowInvokable`. One for each group. Both, the `GroupedWindowInvokable` and the `WindowInvokable` can have any number of active and/or not active trigger and eviction policies. Additionally, the figure contains different buffers, cloneable policies, and the reduce function to be used.

Cloneable Interfaces. The interfaces `CloneableTriggerPolicy`, which extends `TriggerPolicy`, and `CloneableEvictionPolicy`, which extends `EvictionPolicy`, specify the `clone`-method needed for the usage of policies in a distributed manner. All predefined policies are cloneable. Using the `clone`-method, copies of a policy can be created, such that each group has its own set of policy objects which are independent from the policy objects used by other groups.

Nested Window Operators. The execution at the grouped discretization operator is driven by two invokables. One of them, the `GroupedWindowInvokable` is actually executed by the system as an operator. It has one instance of `WindowInvokable` per group. Thus, the second invokable is not executed in an own vertex, but just kept as instance by the first one. This allows to reuse parts of the logic implemented and described before and prevents duplicated code.

Execution Flow at the Grouped Window Operator. In the following we will describe the execution at the `GroupedWindowInvokable`. Thereby, we refer to the logic which is implemented in other classes as sub-programs. Finally, we will show a summary of the communication between the `GroupedWindowInvokable` and the nested instances of the `WindowInvokable`. Figure 22 shows the execution flow for a real data-item arrival in the grouped case.

Whenever a real data-item arrives, the first action which takes place is the extraction of its key. Afterwards, the invokable proves whether a group for this key is already present. If not, a new group with the respective key is created.

The creation of a new group covers to create a clone of all distributed policies. Therefore, the distributed policies passed to the constructor are kept as they are, with their initial state. For each group, a new set of clones is created based on the policies given at construction time. This, assumed that the `clone`-method was implemented by the user correctly, leads to the guarantee, that all groups behave similar, because their processing always starts with policy instances having the same initial state. After the creation of the clones, the instance of the `WindowInvokable` to represent the new group is created, receiving the distributed policies as parameter. The present groups are stored in a `HashMap`, mapping the group keys to the `WindowInvokable` instances which are used to process data-items belonging to the respective group.

After it is ensured, that the group where the just arrived data-item belongs to is present, the pre-notification methods of all central active *TPs* are called and can produce fake data-items. For each produced fake data-item, the central active *EPs* are called first and can request the deletion of data-items from the buffer. Such deletions are executed through the virtual central buffer. Afterwards, the produced fake data-item is broadcasted to all groups and is there processed as usual (see description in section B.2.2).

As next step, all central triggers are notified with the arrived data-item as parameter threw their regular notification methods. Depending on whether a trigger occurs or not, different actions take place.

- **If no central trigger occurs:** This is the simpler of the two cases. The data-item is just forwarded to the group it belongs to. Within the group, it is processed exactly the same way as described in section B.2.2.

- **If a central trigger occurs:** In case there occurs a central trigger, the processing is slightly more complicated. We expect the emission of a result for all present groups now. Additionally, as this was an occurrence of a trigger on the regular notification call, the eviction needs to be executed after the emission of the result.

  We introduced a new method in the `WindowInvokable`, which allows us to forward a real data-item to the group, saying that it shall be considered as causing the occurrence of a trigger in any case, even if no trigger in the `WindowInvokable` occurs. We give the arrived data-item to the group it belongs using this method. Beside the remembered trigger, the processing within the group follows the same flow as in the not grouped case.

  Now, we have to realize the emission of an result for all other groups. We cannot give the arrived data-item to the groups as real data-item, because it would be added to the buffers there, even though it doesn't belong to the group. Hence, we have to forward it as fake data-item. Remember that fake data-items are always considered to cause the occurrence of a trigger. Again, as this was an occurrence of a trigger on the regular notification call, the eviction needs to be executed after the emission of the result. This is not the usual behaviour of the `WindowInvokable` when it processes fake data-items. It was required to add another method to the class which emits a result first and notifies the active distributed *EPs* afterwards.

By now, we did all the processing steps which are needed per group. What remains is the notification of the central *EPs* and, if requested, the deletion of data-items through the virtual central buffer. Finally, we add the group reference of the arrived data-item to the virtual buffer.

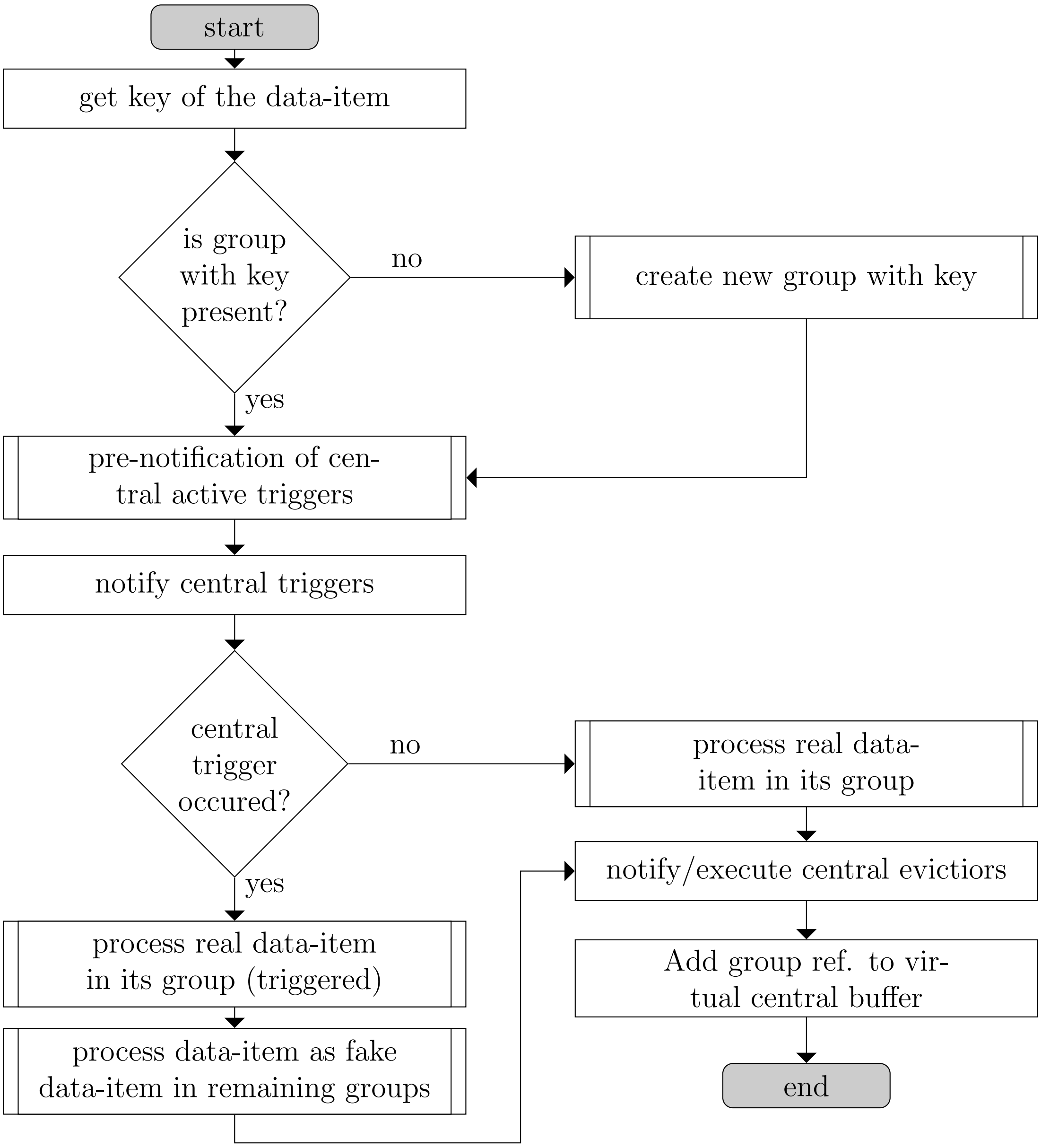

start
get key of the data-item
is group with key present?
no
create new group with key
yes
pre-notification of central active triggers
notify central triggers
central trigger occured?
no
process real data-item in its group
yes
notify/execute central evictions
process real data-item in its group (triggered)
Add group ref. to virtual central buffer
process data-item as fake data-item in remaining groups
end

Figure 22: Execution flow for a real data-item arrivals at the grouped window discretization operator.

Communication Summary. Summarizing the communication between the `GroupedWindowInvokable` and the nested `WindowInvokable` instances, the `WindowInvokable` provides four methods to send data-items to it and one for the eviction of data-items:

1. **Real data-item processing:** A method receiving a real data-item as parameter. The execution flow in the group is then exactly the one shown in section B.2.2.
2. **Triggered real data-item processing:** The same as the first, but with the guarantee that the data-item is considered to cause a trigger. This is used to send real data-items which caused the occurrence of a central trigger.
3. **Processing of fake data-items:** The regular fake data-item processing flow. This is used to send fake data-items produced either by separated threads or on pre-notification of central *TPs*. The execution flow for fake data-item processing in the group is exactly the one shown in section B.2.2.
4. **Processing of real data-items as fake data-items:** This is the only newly added execution flow. It is used to send real data-items which caused a central trigger to occur to the groups they don't belong to. Here the emission of the result happens first, then the active distributed *EPs* are notified, treating the passed data-item as fake data-item. The data-item is not added to the groups buffer.
5. **Eviction of data-items:** When a central *TP* requests the deletion of data-items from the buffer, it does it through the virtual buffer. The virtual buffer stores not the actual data-items, but the references to the groups where data-items have been forwarded to. Thus, deletion requests for data-items are forwarded to the group to which the data-items where sent.

Remark, that the communication is unidirectional in the sense, that `WindowInvokable` never returns a value to the `GroupedWindowInvokable`. This offers well opportunities for parallelization. The execution can be represented in a *DAG*, where the `GroupedWindowInvokable` sends messages to the `WindowInvokable` instances.

B.8 Open source contribution and Design principles

Apache Flink, including the streaming *API*, is implemented in the object oriented programming language Java. Additionally, there are *APIs* provided for other languages. In case of streaming, there is a Scala-*API* as alternative to the Java-*API*.

The architecture we presented in this thesis, was implemented and contributed to the Apache Flink project. Most of the presented features have been released with Flink version 0.8.1. There is a one to one mapping between program classes and JUnit test classes validating the correctness of the implementation. The overall contribution, made to the Apache Flink open source project in the context of this thesis, consists of more than 7800 lines of code.

We presented the object oriented programming model in section A.1. When a object oriented application is developed, there are three general design goals one want to archive in and between classes: tight encapsulation, loose coupling, and high cohesion. [61] The architecture presented in this thesis was made in a way that covers all three design goals.

Cohesion All classes we presented have a single and well defined purpose, which serves a high cohesion. The purpose of almost every used class or interface, can be summarized in one sentence:

- `WindowInvokable` : Applying windowing policies and an aggregation function on a stream of data-items.
- `GroupedWindowInvokable` : Applying windowing policies to a stream of data-items and forward data-items to succeeding operators/groups regarding the data-item keys.
- `TriggerPolicy` : Specify when a window ends.
- `EvictionPolicy` : Specify when data-items in the buffer expire.
- `Timestamp` : Map timestamps to data-items.
- `ReduceFunction` : Combine two data-items two one by applying an aggregation.

Coupling All implemented classes expose their functionality only through their interfaces. In case of policies, timestamps and aggregation functions, methods are even defined in separated interface-classes. Furthermore, there are no global variables used. All attributes are marked with the private keyword. Summarizing, the presented architecture provides a very loose coupling.

Encapsulation The details about the implementations are hidden from the view of other classes. Like mentioned before, we made intensive use of interfaces, especially for the definition of policies. This is the basis for user-defined windowing policies. Thus, the information about the design decisions, made in (user-defined) policies, is indeed hidden from the operators. To be able to work with user-defined policies, aggregations and timestamps might be the best prove for a tight encapsulation.

C Examples for Policy Implementations (Pre-defined Policies)

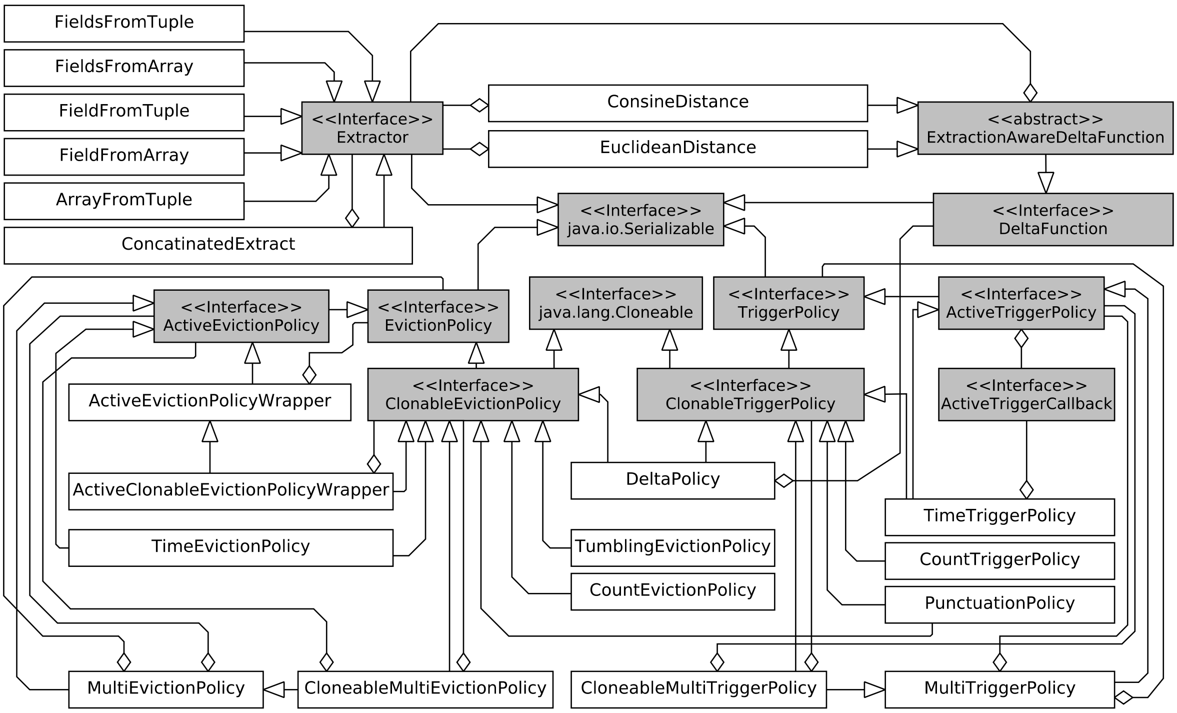


Figure 23: UML class diagram of the windowing package.

Beside the possibility of having user-defined policies, we implemented a powerful set of pre-defined policies using the same interfaces, which can also be used for user-defined once. Thus, the pre-defined policies are on one hand sufficient for most use cases and serve on the other hand as examples for user-defined policies. Additionally, the pre-defined policies can be seen as a prove of the expressiveness which is provided by the presented architecture. Figure 23 shows the interfaces and classes present in the windowing package. We presented most of the depicted interfaces already in section 6 and section B. The missing ones are all related to delta-based policies and we will explain them in section C.3.

In the following sections, we will describe the different implementations of the interfaces, serving various kinds of policies, delta functions, and data-extraction functionalities. Table 5 provides an overview of all policy implementations and also depicts which of them are active, cloneable and stateful and what the size of the state is for each of them.

Policy Type		Cloneable	Active	Stateful	Statesize
Count	Trigger	✓	✗	✓	4 B
	Eviction	✓	✗	(✓)	4 B
Time	Trigger	✓	✓	(✓)	8 B
	Eviction	✓	✓	✓	$8y$ B
Delta	Trigger	✓	✗	✓	x B
	Eviction	✓	✗	✓	$x \cdot y$ B
Punctuation	Trigger	✓	✗	✗	0 B
	Eviction	✓	✗	(✓)	4 B
Tumbling	Eviction	✓	✗	(✓)	4 B
Activation Wrapper	Eviction	like nested	✓	like nested	like nested
Multi	Trigger	if all nested are cloneable	(✓)	if at least one nested has state	$\sum nested$
	Eviction	if all nested are cloneable	(✓)	if at least one nested has state	$\sum nested$

Table 5: Characteristics and state-sizes of predefined policies. A data-item has a size of x B and the data-item buffer contains y data-items.)

Although, Java objects always have a state, we say that a policy has a state only if the policy acts differently depending on the data-items it has seen so far. When calculating the size of the state, we count only the class attributes which are required to keep the necessary information about the seen data-items. For example, in case of a count-based *TP*, this is only one `int`-value keeping the current data-item count.

For the reason of simplification, we assume collections and arrays to have a size of zero and take only the items they contain into account. We assume an `int`-value to have a size of 4 B and a `long`-value to have a size of 8 B. Additionally, we say that a data-item has a size of x B and that the data-item buffer contains y data-items.

All pre-defined policies are cloneable, which means they can be used as distributed policies by the `GroupedWindowInvokable`.
The three classes `ActiveEvictionPolicyWrapper`, `MultiTriggerPolicy`, and `MultiEvictionPolicy` are special policies which wrap around other policies and add further functionality to them. Whether they are cloneable or not depends on the cloneability of the nested policies. In case all nested policies are cloneable, the classes `ActiveCloneableEvictionPolicyWrapper`, `CloneableMultiTriggerPolicy`, and `CloneableMultiEvictionPolicy` can be used, which allows to use the wrapping policies as distributed policies in grouped windowing.

Only time-based policies and wrapper policies are active.
For the `ActiveEvictionPolicyWrapper`, being active is its whole purpose, as we will

explain in section C.6. The multi policy wrappers are always active, but in case they only wrap around not active policies, they simply do nothing on pre-notification (in case of *TPs*) and on fake data-item arrivals (in case of *EPs*). Accordingly, one could also say that they are not active in such cases even though they are technically always instance of `ActiveTriggerPolicy` or `ActiveEvictionPolicy`.

Finally, let's have a look on the states of policies. It's important to evaluate what the actual size of states is. When it comes to fault tolerance it might be crucial for the performance to have as small as possible state sizes. To provide fault tolerance in stream processing the state of a policy needs to be serialized and stored periodically (check-pointing). In case of failure, the execution can be replayed starting from the latest checkpoint.

For a count-based *TP*, as already mentioned, the state has a size of 4 B an contains one `int`-value to keep a count of data-items. In the current implementation the same holds for the count-based *EP*, the punctuation-based *EP* and the tumbling *EP*. Nevertheless, there are possibilities to make them stateless. The tumbling *EP* could simply return `Integer.MAX_VALUE`. All three mentioned policies could also act with regard to the `buffersize`-parameter of the notification method and thereby be avoided from having a state. The reason for implementing them using an own counter is to make them as independent from external circumstances as possible to have a better encapsulation.

The implementation is already planned to be used with pre-aggregation optimizations and in highly distributed fashions. In such cases the buffer-size might be unknown at the vertex where the policy is placed. Additionally, when multiple policies are used, users consider a *TP* and a *EP* often to work like a pair, which is wrong. Anyhow, keeping own counts in the policies allows to implement them in a way which let them behave like those users expect them to behave.

A time is represented by a `long`-value. Therefore, the time-based *TP* keeps the time it triggered last as state. This is required to know which fake data-items need to be emitted on pre-notification or whether this was already done by a separated thread. The time-based *EP* needs to keep the timestamps of all elements which are in the buffer to check whether they are expired or not.

For delta-based *TPs*, we need to keep one data-item as old data-item from which the delta to the current data-item can be calculated. Hence, one data-item is required the be kept as state. For the delta based *EP* we need to be able to compare each individual data-item in the buffer to check whether it is expired or not by calculating its delta to the newest data-item. As long as the buffer and the policy are located at the same vertex, this does not cause a huge memory utilization because the policy will not clone the data-items but just keep a reference to them. Anyhow, when it comes to serialization, the references will cause the data-items in the buffer to be serialized together with the policy.

What remains are the punctuation-based *TP* and wrapping policies. The punctuation-based *TP* is stateless. When it is notified, it checks for the punctuation and in case the punctuation is present it returns `true`. There is nothing to keep as state. The wrapping policies does not have an own state, but inherit the state characteristics of their nested policies. Hence, whether they are statefull or not depends on whether at least one nested policy has a state and the state-size is the sum of the state-sizes of the nested policies.

C.1 Count-based Trigger and Eviction

The `CountTriggerPolicy` counts the arriving data-items. It is not active, so only real data-items are counted. The threshold of the counter is set in the constructor. The constructor also allows to set a custom start value for the data-item counter. This can be used to delay the first trigger by setting a negative start value.

Often the first trigger should be delayed in case sliding windows are used. For example, if the size of a window should be four data-items and a trigger should happen every two data-items, a start value of minus two would allow to also have the first window of size four. Without the negative start value the first window would contain only two data-items.

For count-based *EPs* the constructor of `CountEvictionPolicy` also allows to set up the number of data-items to be deleted from the buffer in case of an eviction. Eviction only takes place if the counter of arriving data-items would be higher than the set threshold otherwise. In such a case, not the whole buffer, but the specified number of data-items is deleted. As Listing 16 shows, the counter of arriving data-items is adjusted respectively, but never set below zero.

```
counter=(counter-deleteOnEviction<0)?0:counter-deleteOnEviction
```

Listing 16: Calculation of the counter value in the count-based eviction policy.

The manual setting of the number of data-items to delete on eviction is useful in case the user wants to specify a custom overlap of windows. For example, we can trigger every three data-items and set the start value of the count-based *TP* to minus two. Additionally, we set the number of elements to delete on eviction to three and the maximum number of data-items in the buffer to fife in the count-based *EP*. Using this setup, we will receive windows containing fife elements with an overlap of two elements.

C.2 Time-based Trigger and Eviction

The TimeTriggerPolicy works based on time measures. To determine the time for a data-item, this policy uses a Timestamp-implementation. Thus, the time measure can either be a pre-defined (default) one or any user-defined time measure (*UDF*-Timestamp). In the current version, Flink streaming uses the system time provided by the *JVM* (System.currentTimeMillis()) as default time measure. A point in time is always represented as long-value. Hence, the window granularity can be set as long value as well. If the granularity attribute is set to 100 for example, the policy will trigger at every 100th point in time. In addition to the granularity and the Timestamp-implementation, a delay can be specified for the first trigger. If the start time given by the Timestamp-implementation is x, the delay is y, and the granularity is z, the first trigger will happen at $x + y + z$.

Time policies are active policies, which means that time-based *TP* might have to produce fake data-items. We provided a detailed description of the concept behind active policies in section 6.9. If the time-based *TP* is used together with SystemTimestamp (default case), the trigger will also provide a runnable, which actively creates fake data-items without the need of waiting for a real data-item arrival.

The TimeEvictionPolicy evicts all data-items which are older than a specified time. Listing 17 shows a pseudo code for the removal of elements from the buffer. The check for expiration will start with the first data-item in the current buffer and end as soon as the buffer is either empty or its first data-item is not expired.

```
while (time(firstInBuffer)<current_time-granularity){
    evict firstInBuffer;
}
```

Listing 17: Eviction algorithm for the pre-defined time-based eviction policy

In general, the time-based policies expect data-items to arrive in order. An arriving data-item should always have a greater or equal timestamp than the maximum timestamp seen at any data-item arrived previously. The only exception are fake data-items. As they are never added to the buffer, they could theoretically occur out of order to enforce special eviction or trigger behavior. Anyhow, the currently provided implementations always keep the order, even for fake data-items.

C.3 Delta-based Trigger and Eviction

The `DeltaPolicy` implements both, `TriggerPolicy` and `EvictionPolicy`. If it is used as *TP*, it calculates a delta between the data-point which triggered last and the currently arrived data-point. It triggers if the delta is higher than a specified threshold. In case it is used for eviction, the policy starts from the first element of the buffer and removes all elements from the buffer which have a higher delta to the currently arrived data-item then the threshold. As soon as the first data-item in the buffer has a lower delta, the eviction stops. By default, this policy is not active and does not react on fake data-items. It can be made an active *EP* using a wrapper class as described in section C.6.

The `DeltaFunction` to be used has to be provided as parameter to the constructor. As `DeltaFunction` is an interface, any user defined distance measure can be used. Additionally, a sample data-item needs to be provided as parameter. This is used to calculate the deltas before the first trigger or eviction occurred.

C.3.1 Delta Functions

Listing 18 shows the methods which needs to be implemented in order to implement the `DeltaFunction` interface. The method uses the generic type `DATA` and thereby just expects to get two inputs of the same type. The calculated delta between the two data-items is returned as `double`-value.

```java
public double getDelta(DATA oldDataPoint, DATA newDataPoint)
```

Listing 18: The interface for delta functions used by the delta-based policies.

In many cases the delta should be calculated using only some attributes included in the data-item, instead of the data-item as a whole.
The abstract class `ExtractionAwareDeltaFunction` is provided to allow the application of an `Extractor` before the delta is calculated. All predefined delta functions extend this class. Listing 19 shows an excerpt of the class containing all important methods and the constructor. The `Extractor` interface and predefined extractors are presented in section C.3.2. `ExtractionAwareDeltaFunction` implements `DeltaFunction`. In the `getDelta`-method it either converts directly from `FROM` to `TO` in case no extractor is present or it applies the extractor to do the converting if one is present. The actual delta is then calculated by the abstract method `getNestedDelta` which has the same signature as `getDelta` but with parameters of the extracted type (`TO`). All classes inheriting from `ExtractionAwareDeltaFunction`, which are all pre-defined delta functions, implement this abstract method instead of the `getDelta`-method.

```
public ExtractionAwareDeltaFunction(Extractor<DATA, TO> converter) {
  this.converter = converter;
}
@Override
public double getDelta(DATA oldDataPoint, DATA newDataPoint) {
  if (converter == null) {
    return getNestedDelta((TO) oldDataPoint, (TO) newDataPoint);
  } else {
    return getNestedDelta(converter.extract(oldDataPoint), converter.extract(
        newDataPoint));
  }
}
public abstract double getNestedDelta(TO oldDataPoint, TO newDataPoint);
```

Listing 19: The extraction aware delta function (abstract class).

Two different delta functions are implemented so far as examples. One is the euclidean distance ($d_e(p,q)$), implemented in `EuclideanDistance`, and the other is the cosine distance ($d_c(p,q)$), which is defined as one minus the cosine similarity ($s_c(p,q)$) and has been implemented in `CosineDistance`.

The euclidean distance is the most intuitive distance measure. p and q are said to be points in a n-dimensional euclidean space. First the distance is calculated for each dimension. The results are squared and summed up. Finally we take the positive square root of the calculated sum [59].

$$d_e(p,q) = d_e(q,p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + ... + (q_n - p_n)^2} = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \quad [59]$$

In case of the cosine distance, p and q are said to be vectors with n dimensions. Two vectors are said to be equal if they point in the the same direction. The resulting distance will be 0 in this case. The highest possible distance would be 1 in case the vectors are independent[13] [23].

$$d_c(p,q) = 1 - s_c(p,q) = 1 - \frac{p \cdot q}{|p||q|} = 1 - \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2} \cdot \sqrt{\sum_{i=1}^{n} q_i^2}} \quad [23]$$

[13] ”Let $x_1, x_2, ..., x_k$ be vectors of the linear space K over a field K, and let $\alpha_1, \alpha_2, ..., \alpha_k$ be numbers from K. Then the vector $y = \alpha_1 x_1 + \alpha_2 x_1 + ... + \alpha_k x_k$ is called a linear combination of the vectors $x_1, x_2, ..., x_k$.” [38] If $y = 0$ ”is only possible in the case where $\alpha_1 = \alpha_2 = ... = \alpha_k = 0$, the vectors $x_1, x_2, ..., x_k$ are said to be linear independent (over K).” [38]

It's important to notice, that there are different definitions of the cosine distance present in literature [23, 59]. The one we implemented (and which is shown above) is the algebraic one which is most commonly defined [23].

C.3.2 Extractors

```
public interface Extractor<FROM, TO> extends Serializable {
  public TO extract(FROM in);
}
```

Listing 20: The Extractor interface to isolate partial data from data-items.

The `Extractor` interface (Listing 20) was build to provide a good abstraction of an arbitrary data transformation from one data type to another. Extractors can be used to isolate parts of the incoming data point. Later on, a delta between the extracted parts from two data-items can be calculated. Extractors are also used in punctuation-based policies to isolate the punctuation from the data-item. Currently, six different pre-defined `Extractor`-implementations are available, providing a huge variety of standard extraction procedures. Anyhow, user-defined extractors can easily be implemented and used. Extractions can also consist of a concatenation of extraction procedures provided by multiple `Extractor`-implementations. The class `ConcatinatedExtract`, which itself is an `Extractor`, can be used to encapsulate such an extraction flow. Table 6 list all pre-defined extractors with their input and output types and a short description.

Name	**FROM**	**TO**
`ArrayFromTuple`	`Tuple`	`Object[ ]`
Short description:	This extractor converts a Tuple to an array. If the default constructor is used, the fields of the `Tuple` will be contained as elements in the output array in same order as they are in the `Tuple`. By using the constructor with a var-args parameter, one can select any fields from the `Tuple` by putting the respective field ids. The order of fields in the output array will then be the same as the order of the ids passed to the constructor.	
`FieldFromArray`	`Object[ ]`	OUT (Generic)
Short description:	This extractor returns a single field of the generic type OUT from an array. The id of the field needs to be set as parameter in the constructor.	
`FieldFromTuple`	`Tuple`	OUT (Generic)
Short description:	This extractor returns a single field of the generic type OUT from a `Tuple`-type. The id of the field needs to be set as parameter in the constructor.	
`FieldsFromArray`	`Object[ ]`	OUT[ ] (Generic Array)
Short description:	This extractor allows to isolate fields of the same type (instance of OUT) from an array. To be able to create an array of the generic type OUT, a `Class` of this type needs to be passed to the constructor as first parameter. The second parameter is a var-arg parameter to set the ids of the fields to be extracted. The order of fields in the output will be the same as specified in this constructor parameter.	
`FieldsFromTuple`	`Tuple`	double[ ]
Short description:	This extractor isolates `double`-values from a `Tuple`-type and returns them as array. This can be used produce a `double` vector from some fields of a `Tuple`. The indexes of the fields to be extracted can be specified in the constructor using a var-args parameter. The order of fields in the output will be the same as specified in the constructor parameter.	
`ConcatinatedExtract`	Input type of first	Output type of second
Short description:	The constructor of this extractor expects two extractors as parameter and will concatenate their extractions. Hence, the input type is the input type of the first nested extractor, the output type of the first nested extractor have to be the same as the input type of the second nested extractor and the output type at the end is the one of the second nested extractor.	

Table 6: An overview of all pre-defined extractors.

C.4 Punctuation-based Trigger and Eviction

The `PunctuationPolicy` implements both, `TriggerPolicy` and `EvictionPolicy`. Hence, it can be used to trigger and evict based on a punctuation which is present within the arriving data. Using this policy, one can react on an externally defined arbitrary windowing semantics. The constructor expects a sample of the punctuation as parameter. The policy then uses the `equals`-method to compare the sample punctuation with the one which is present in any arrived data-item.

In case this policy is used for eviction, the complete buffer is deleted in case the punctuation is detected. By default, this policy is not active and does not react on fake data-items. It can be made an active *EP* using a wrapper class as described in section C.6. When the policy is used as *TP*, it triggers if the punctuation is detected.

Optionally, an `Extractor` implementation can be provided as second parameter of the constructor. The extractor is then uses to isolate the punctuation from the incoming data-item. Hence, it is simply possible to extract a punctuation which is present in a field of an array, a `Tuple`-type or any other arbitrary complex data structure. The available predefined extractors are presented in section C.3.2. If no extractor is provided, the arrived data-item itself (as a whole) is compared with the sample punctuation.

C.5 Tumbling Eviction Policy

The `TumblingEvictionPolicy` was made to prevent the *EP* from doing computations, which where already done by the *TP*, again. In case tumbling windows are used, we need to clear the whole buffer whenever a trigger occurred. If we want to achieve this behaviour having a count-based *TP*, we could use a count-based *EP*, but this is inconvenient. The *EP* would do the same counting task again. Additionally, it is complex when multiple *TPs* are used at the same time. The `TumblingEvictionPolicy` serves better performance and simplicity. It is sensitive to the `triggered` parameter of the `notifyEviction` method and deletes the whole buffer whenever this parameter is set to `true` which means that a trigger occurred.

By default, `TumblingEvictionPolicy` is not instance of `ActiveEvictionPolicy`, because there are some use-cases where the buffer should not be deleted when fake data-items are produced. Anyhow, one can make `TumblingEvictionPolicy` active by wrapping it in `ActiveEvictionPolicyWrapper` as described in section C.6.

C.6 Active Eviction Policy Wrapper

The `ActiveEvictionPolicyWrapper` takes any `EvictionPolicy` instance as parameter in its constructor. It wraps around this given policy and thereby makes it an `ActiveEvictionPolicy`. Technically, this is done by forwarding all notifications caused by fake data-item arrivals to the regular notification method of the nested *EP*.

In case an already active policy is wrapped into `ActiveEvictionPolicyWrapper`, the wrapper will hide the nested policy's method for fake data-item notifications and forward fake data-items to the regular notification method, like it is usually only done with real data-items.

C.7 Multi Policy Wrapper

As described previously already, it is possible to use multiple *EPs* and *TPs* at the same time. This can be done by adding these policies directly to the `WindowInvokable` or `GroupedWindowInvokable`. In addition to this way of having multiple policies, there is another way of doing it. The class `MultiTriggerPolicy` implements `ActiveTriggerPolicy` and can encapsulate multiple *TPs*. `MultiEvictionPolicy` implements `ActiveEvictionPolicy` and can do the same for *EPs*.

Both approaches have advantages and disadvantages. Having the policies directly in the invokable makes it much easier to identify logical equivalences across queries or within huge queries. Hence, this approach might serve better optimization opportunities. Having policies separated can also allow better parallelization opportunities, because different steps in the window discretization can possibly be split across multiple execution vertexes. The biggest disadvantage is the increased complexity of the invokables. Having multiple policies requires to have an extension made to the execution flow and enforces the developer of the invokable to handle a bunch of corner cases and dependencies between the data-item buffer and multiple policies. Preventing this increased complexity is on the other hand the most important advantage of using `MultiTriggerPolicy` and `MultiEvictionPolicy` instead.

Another advantage is the possibility of having different strategies for multiple triggers and evictions. For triggers, the invokables always says that a trigger occurred in case at least one policy triggered. For evictions the invokables always take the greatest returned value into account. The multi policy wrappers can provide different strategies without increasing the complexity at the invokable. The application of the strategy is encapsulated by the wrapper and thereby completely transparent from the invokable's perspective. Different strategies for *TP* could for example be that all nested *TPs* need to trigger or that

```
case MAX:
  result = 0;
  for (Integer item : items) {
    if (result < item) {
      result = item;
    }
  }
  return result;
```

Listing 21: The MAX eviction strategy

```
case MIN:
  result = Integer.MAX_VALUE;
  for (Integer item : items) {
    if (result > item) {
      result = item;
    }
  }
  return result;
```

Listing 22: The MIN eviction strategy

```
case SUM:
  result = 0;
  for (Integer item : items) {
    System.out.print(item+":");
    result += item;
  }
  System.out.println(result);
  return result;
```

Listing 23: The SUM eviction strategy

```
case PRIORITY:
  for (Integer item : items) {
    //first value != 0
    if (item > 0) {
      return item;
    }
  }
  return 0;
```

Listing 24: The PRIORITY eviction strategy

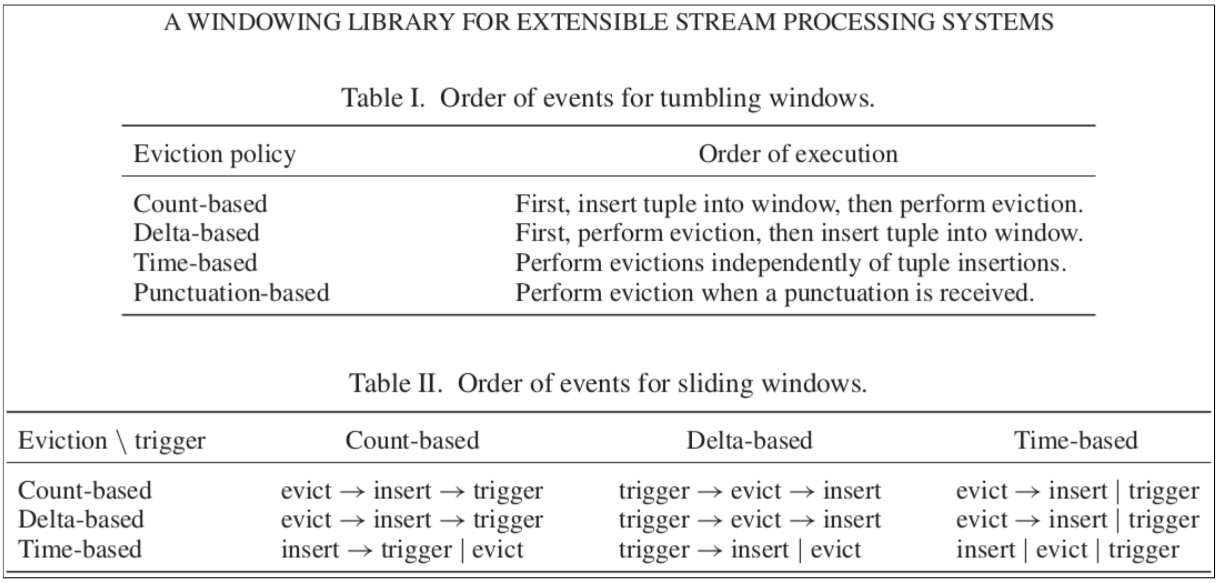
exactly one needs to trigger. For *EPs*, four different strategies have been implemented, shown in listings 21 to 24. All available strategies are listed in an `enum`. Within a `switch-case`-block, only the selected strategy is applied.

Combining both approaches even allows to have different strategies for different *groups* or *groups of groups* of policies within the same setup. This can be a powerful tool and enables many possibilities to make policies reusable for different queries.

In general, when a multi policy wrapper is notified, it will notify all nested policies through their respective methods and collect the return values. The selected strategy is then applied on the set of collected return values to calculate a result which can be returned to the invokable. A special case is the factory method for runnables in active triggers. When this factory method is called in `MultiTriggerPolicy`, it collects the runnables from all nested active *TPs*, but can only return one runnable to the invokable. `MultiTriggerPolicy` has its own `Runnable`-implementation as inner class, which is returned to the invokable. This inner class takes all the collected runnables as parameter in the constructor. At the moment the `Runnable` is executed by the invokable, it will start one separated thread for each of the runnables it got as parameter in its constructor.

D Action Order in SPL

Figure 24 is an excerpt from *Generic windowing support for extensible stream processing systems* [37] by Gedic. It shows the action orders in *SPL* for different combinations of policies.



A WINDOWING LIBRARY FOR EXTENSIBLE STREAM PROCESSING SYSTEMS

Table I. Order of events for tumbling windows.

Eviction policy	Order of execution
Count-based	First, insert tuple into window, then perform eviction.
Delta-based	First, perform eviction, then insert tuple into window.
Time-based	Perform evictions independently of tuple insertions.
Punctuation-based	Perform eviction when a punctuation is received.

Table II. Order of events for sliding windows.

Eviction \ trigger	Count-based	Delta-based	Time-based
Count-based	evict → insert → trigger	trigger → evict → insert	evict → insert | trigger
Delta-based	evict → insert → trigger	trigger → evict → insert	evict → insert | trigger
Time-based	insert → trigger | evict	trigger → insert | evict	insert | evict | trigger

Figure 24: Figure from [37]: Order of actions for tumbling and sliding windows in *SPL*.

E Entity Relationship Diagrams and Flow Charts

This thesis contains entity relationship diagrams with cardinality symbols according to the Richard Barker annotation[12].

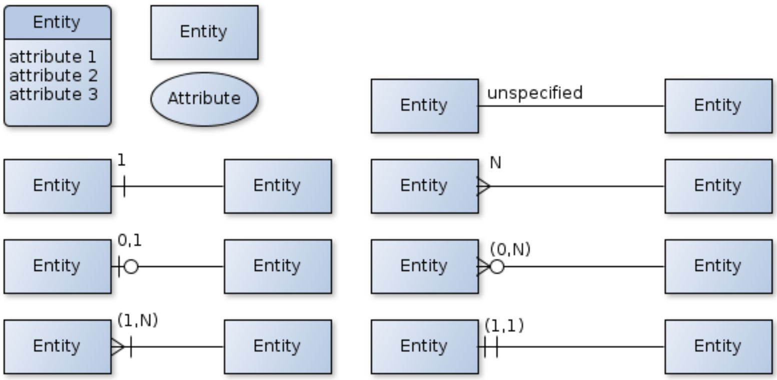


Figure 25: Elements of entity relationship diagrams according to the Richard Barker annotation [12]

Furthermore this thesis contains flow charts according to DIN 66001[34] and ISO 5807[32].

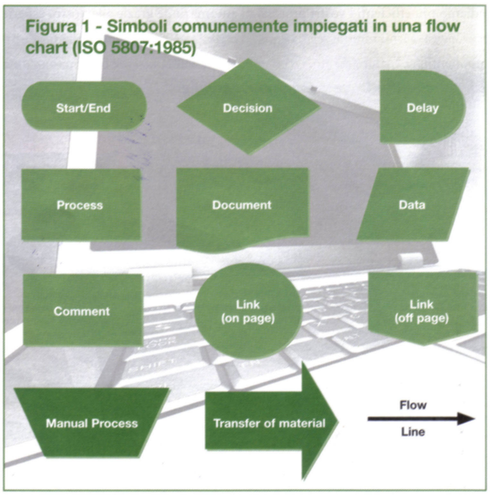


Figure 26: Flow chart elements according to DIN 66001[34] and ISO 5807[32] [Image: internationalpbi.com]

F Referenced Java Classes

ActiveCloneableEvictionPolicyWrapper org.apache.flink.streaming.api.windowing.policy.ActiveCloneableEvictionPolicyWrapper
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/ActiveCloneableEvictionPolicyWrapper.html

ActiveEvictionPolicy org.apache.flink.streaming.api.windowing.policy.ActiveEvictionPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/ActiveEvictionPolicy.html

ActiveEvictionPolicyWrapper org.apache.flink.streaming.api.windowing.policy.ActiveEvictionPolicyWrapper
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/ActiveEvictionPolicyWrapper.html

ActiveTriggerCallback org.apache.flink.streaming.api.windowing.policy.ActiveTriggerCallback
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/ActiveTriggerCallback.html

ActiveTriggerPolicy org.apache.flink.streaming.api.windowing.policy.ActiveTriggerPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/ActiveTriggerPolicy.html

CloneableEvictionPolicy org.apache.flink.streaming.api.windowing.policy.CloneableEvictionPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/CloneableEvictionPolicy.html

CloneableMultiEvictionPolicy org.apache.flink.streaming.api.windowing.policy.CloneableMultiEvictionPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/CloneableMultiEvictionPolicy.html

CloneableMultiTriggerPolicy org.apache.flink.streaming.api.windowing.policy.CloneableMultiTriggerPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/CloneableMultiTriggerPolicy.html

CloneableTriggerPolicy org.apache.flink.streaming.api.windowing.policy.CloneableTriggerPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/CloneableTriggerPolicy.html

ConcatinatedExtract org.apache.flink.streaming.api.windowing.extractor.ConcatinatedExtract
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/extractor/ConcatinatedExtract.html

CosineDistance org.apache.flink.streaming.api.windowing.deltafunction.CosineDistance
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/deltafunction/CosineDistance.html

Count org.apache.flink.streaming.api.windowing.helper.Count
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/helper/Count.html

CountEvictionPolicy org.apache.flink.streaming.api.windowing.policy.CountEvictionPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/CountEvictionPolicy.html

CountTriggerPolicy org.apache.flink.streaming.api.windowing.policy.CountTriggerPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/CountTriggerPolicy.html

DataStream org.apache.flink.streaming.api.datastream.DataStream
DOCHOME/api/java/org/apache/flink/streaming/api/datastream/DataStream.html

Delta org.apache.flink.streaming.api.windowing.helper.Delta
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/helper/Delta.html

DeltaFunction org.apache.flink.streaming.api.windowing.deltafunction.DeltaFunction
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/deltafunction/DeltaFunction.html

DeltaPolicy org.apache.flink.streaming.api.windowing.policy.DeltaPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/DeltaPolicy.html

EuclideanDistance org.apache.flink.streaming.api.windowing.deltafunction.EuclideanDistance
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/deltafunction/EuclideanDistance.html

EvictionPolicy org.apache.flink.streaming.api.windowing.policy.EvictionPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/EvictionPolicy.html

ExtractionAwareDeltaFunction org.apache.flink.streaming.api.windowing.deltafunction.ExtractionAwareDeltaFunction
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/deltafunction/ExtractionAwareDeltaFunction.html

Extractor org.apache.flink.streaming.api.windowing.extractor.Extractor
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/extractor/Extractor.html

FlatMapFunction org.apache.flink.api.common.functions.FlatMapFunction
DOCHOME/api/java/org/apache/flink/api/common/functions/FlatMapFunction.html

GroupedWindowInvokable org.apache.flink.streaming.api.invokable.operator.GroupedWindowInvokable
DOCHOME/api/java/org/apache/flink/streaming/api/invokable/operator/GroupedWindowInvokable.html

GroupReduceFunction org.apache.flink.api.common.functions.GroupReduceFunction
DOCHOME/api/java/org/apache/flink/api/common/functions/GroupReduceFunction.html

HashMap java.util.HashMap
http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

KeySelector org.apache.flink.api.java.functions.KeySelector
DOCHOME/org/apache/flink/api/java/functions/KeySelector.html

Long java.lang.Long
http://docs.oracle.com/javase/7/docs/api/java/lang/Long.html

MultiEvictionPolicy org.apache.flink.streaming.api.windowing.policy.MultiEvictionPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/MultiEvictionPolicy.html

MultiTriggerPolicy org.apache.flink.streaming.api.windowing.policy.MultiTriggerPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/MultiTriggerPolicy.html

Object java.lang.Object
http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html

PunctuationPolicy org.apache.flink.streaming.api.windowing.policy.PunctuationPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/PunctuationPolicy.html

ReduceFunction org.apache.flink.api.common.functions.ReduceFunction
DOCHOME/api/java/org/apache/flink/api/common/functions/ReduceFunction.html

Runnable java.lang.Runnable
http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html

SystemTimestamp org.apache.flink.streaming.api.windowing.helper.SystemTimestamp
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/helper/SystemTimestamp.html

Time org.apache.flink.streaming.api.windowing.helper.Time
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/helper/Time.html

TimeEvictionPolicy org.apache.flink.streaming.api.windowing.policy.TimeEvictionPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/TimeEvictionPolicy.html

Timestamp org.apache.flink.streaming.api.windowing.helper.Timestamp
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/helper/Timestamp.html

TimeTriggerPolicy org.apache.flink.streaming.api.windowing.policy.TimeTriggerPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/TimeTriggerPolicy.html

TimeUnit java.util.concurrent.TimeUnit
http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html

TriggerPolicy org.apache.flink.streaming.api.windowing.policy.TriggerPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/TriggerPolicy.html

TumblingEvictionPolicy org.apache.flink.streaming.api.windowing.policy.TumblingEvictionPolicy
DOCHOME/api/java/org/apache/flink/streaming/api/windowing/policy/TumblingEvictionPolicy.html

Tuple org.apache.flink.api.java.tuple.Tuple
DOCHOME/org/apache/flink/api/java/tuple/class-use/Tuple.html

WindowInvokable org.apache.flink.streaming.api.invokable.operator.WindowInvokable
DOCHOME/api/java/org/apache/flink/streaming/api/invokable/operator/WindowInvokable.html