Scalable Detection of Concept Drifts on Data Streams with Parallel Adaptive Windowing

Philipp M. Grulich[1] René Saitenmacher[1] Jonas Traub[1]
Sebastian Breß[1,2] Tilmann Rabl[1,2] Volker Markl[1,2]

[1]Technische Universität Berlin [2]DFKI GmbH

ABSTRACT

Machine learning techniques for data stream analysis suffer from concept drifts such as changed user preferences, varying weather conditions, or economic changes. These concept drifts cause wrong predictions and lead to incorrect business decisions. Concept drift detection methods such as adaptive windowing (ADWIN) allow for adapting to concept drifts on the fly.

In this paper, we examine ADWIN in detail and point out its throughput bottlenecks. We then introduce several parallelization alternatives to address these bottlenecks. Our optimizations lead to a speedup of two orders of magnitude over the original ADWIN implementation. Thus, we explore parallel adaptive windowing to provide scalable concept detection for high-velocity data streams with millions of tuples per second.

1 INTRODUCTION

Machine learning (ML) techniques gain more and more adoption in stream analysis. They enable various use-cases such as theft detection, event classification, and failure prediction. In a nutshell, ML techniques train a ML model and apply that model when they process stream tuples (e.g., to classify them). Concept drifts cause a discrepancy between ML models and reality, which makes concept drifts a crucial challenge for machine learning on data streams. High discrepancies lead to incorrect predictions and wrong results. As a consequence, we need to continuously monitor concept drifts and retrain ML models accordingly. Stream processing engines require scalable solutions for concept drift detection and adaptation to execute ML techniques on high-velocity data streams with millions of tuples per second.

A naive approach to cope with concept drifts is to retrain the ML model periodically on a fixed size batch of recent data. This periodic re-computation is in conflict with the real-time requirement of stream processing applications for two reasons: *(i)* Models do not yet cover the most recent data when they are applied although the most recent data might indicate a concept drift. *(ii)* The data, which is reflected in the model, indicates concept drifts leading to deviations between model and reality.

In contrast to the naive solution, a wide range of approaches detects concept drifts on the fly [8]. Some monitor the evolution of different performance indicators [10, 12], while others observe the distributions on two different time-windows [9].

We study the scalability limitations of such approaches on the example of *adaptive windowing* (ADWIN) [2]. In general, ADWIN maintains a *global window* of adaptive size which is the data basis for the model computation. It trades off the variance in the *global window* (i.e., the data variance reflected in the model) against the size of the global window (i.e., the amount of data reflected by the model). We chose ADWIN because it has proven its capabilities in a wide range of real-world applications: ADWIN was combined with Kalman Filters and demonstrated with Naìve Bayes and k-means clustering [1]. Furthermore, ADWIN is used for an online version of the Bagging method by Oza and Rusell [5] and in a parameter-free variant of the Space Saving algorithm [4]. Moreover, ADWIN is available in the open source *MOA* framework [3].

In this paper, we present the following contributions:

(1) We analyze the original ADWIN approach, point out its scalability limitations, and identify its bottlenecks.
(2) We parallelize ADWIN to overcome its bottlenecks and to provide scalable concept drift adaptation in real-time.
(3) We evaluate the latency and throughput of our solution. It achieves two orders of magnitude speedup over the original ADWIN implementation and 54 times speedup in comparison to an optimized sequential implementation.

The source code of our implementation is available on GitHub[1].

In the remainder of this paper, we discuss ADWIN and its bottlenecks in Section 2, present our parallel adaptive windowing approaches in Section 3, and evaluate them in Section 4.

2 CONCEPT DRIFT DETECTION WITH ADAPTIVE WINDOWING

In this section, we present the original ADWIN algorithm [2] (Section 2.1), discuss an optimization based on exponential histograms (Section 2.2) and analyze the performance of the open source ADWIN implementation (Section 2.3).

2.1 The ADWIN Algorithm

ADWIN is an algorithm which detects concept drifts on the fly and adapts ML models accordingly. The algorithm maintains an adaptive window which is the basis for computing the ML model. ADWIN grows the window (i.e., adds the most recent tuples) as long as there is no concept drift detected. As a result, the model can rely on growing training data. ADWIN shrinks the window by removing old tuples when it detects a concept drift.

The algorithm does not require users to configure minimum or maximum times between concept drifts in advance because it identifies concept drifts on a per-tuple basis. This removes an important drawback of approaches which use fixed-size windows (i.e., batches) of data to recompute models periodically. In ADWIN, users configure only the confidence value $\delta \in (0, 1)$ to adjust the sensitivity of the concept drift detection.

When processing a stream tuple, ADWIN first adds the tuple to the adaptive window. Then, the algorithm analyzes the content of the adaptive window to identify concept drifts. To that end, ADWIN iterates over all possible combinations of two *large enough* sliding subwindows, as shown in Figure 1. If the value distributions of the two subwindows are *different enough*, ADWIN detects a concept drift and removes the oldest tuple from the

[1]Source Code: https://github.com/TU-Berlin-DIMA/parallel-ADWIN

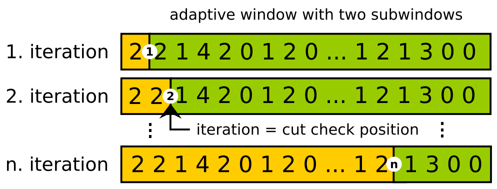


Figure 1: Iterations of the cut check procedure in ADWIN

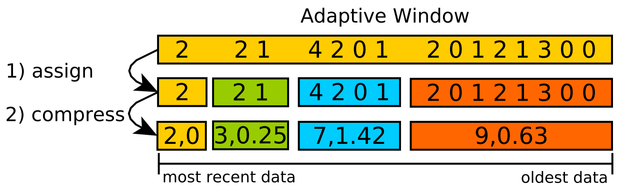


Figure 2: Exponential histogram: We first assign tuples to buckets of exponential size. We then compress buckets and store sums and variances only.

adaptive window. We call this *cut detection* because it determines when to *cut* the adaptive window in order to remove old data. The cut detection repeats removing old tuples until the content of the adaptive window does not indicate a concept drift anymore.

2.2 Exponential Histograms

A naive ADWIN implementation is computationally expensive because it performs a cut check for all possible combinations of subwindows for each tuple of the input stream. To address this problem, ADWIN uses an *exponential histogram* as underlying data structure of the adaptive window [7]. The exponential histogram assigns input tuples to buckets (Step 1 in Figure 2). Buckets of recent data contain just a few tuples. Buckets of older data contain an exponentially growing number of tuples. Each bucket stores the sum and variance of its tuples only. This reduces the memory consumption of the histogram (Step 2 in Figure 2). In summary, the number of buckets and the respective memory consumption grow logarithmically when the adaptive window grows. The cut check procedure now compares buckets instead of per-tuple subwindows which leads to an $O(log(n))$ complexity for the cut-check procedure. Thereby, n is the number of tuples in the adaptive window and $log(n)$ the number of buckets.

Figure 3 shows an overview of the ADWIN algorithm including exponential histograms. The algorithm performs two tasks:

(1) ADWIN inserts arriving tuples to the adaptive window, i.e., the exponential histogram (Step 1.1 in Figure 3). This occasionally causes bucket compression and fusion of smaller buckets to larger buckets (Step 1.2 in Figure 3).
(2) ADWIN detects concept drifts with its *cut detection* procedure and potentially removes old data from the histogram.

In the following subsection, we analyze the runtimes of the different tasks and thereby identify the bottlenecks.

2.3 Initial Performance Analysis

As a starting point of our work, we study the original ADWIN implementation, which is also part of the *MOA* framework [3]. This implementation is not parallel, i.e., it runs in a single thread. As we will show in detail in our experiments in Section 4, the single thread version has a low throughput compared to our parallel approaches which we present in Section 3.

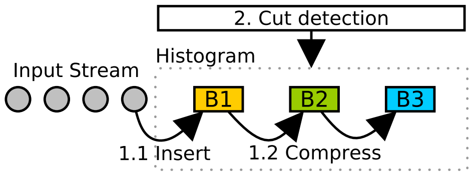


Figure 3: ADWIN overview: New tuples are added to the histogram (1.1), buckets are compressed (1.2), and the cut detection procedure identifies concept drifts (2).

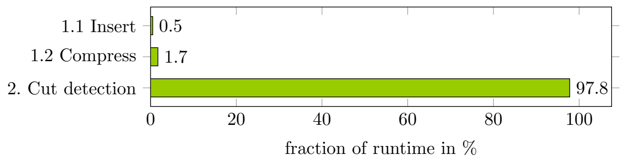


Figure 4: Runtime distribution among ADWIN tasks

In order to identify the bottlenecks of the existing implementation, we use JProfiler[2] and Java VisualVM[3] for performance profiling. In Figure 4, we show the processing time distribution among the tasks performed by ADWIN. A single thread implementation spends about 98% of its runtime with cut checks. In comparison, the time spent on maintaining the exponential histogram is almost negligible as it contributes only 2.2% to the total processing time of a tuple. Although the number of buckets grows logarithmically with the number of tuples contained in the histogram, bucket comparisons dominate the execution effort since they are performed for each tuple. Based on the observations above, we focus our optimizations on parallelizing the cut check procedure because it exhibits the largest saving potential.

In our code analysis, we identified ways to improve the original ADWIN implementation. We did a logically equivalent reimplementation to speedup the execution. The major improvement of our reimplementation is the use of circular array buffers as an underlying data structure for the histogram. This reduces memory copy operations compared to the original implementation.

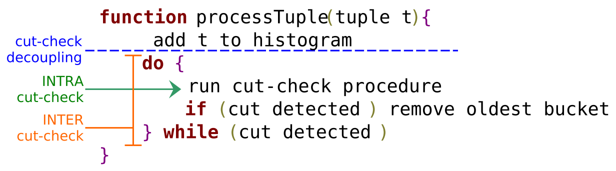
3 PARALLEL ADAPTIVE WINDOWING

In this section, we introduce several approaches to parallelize ADWIN in order to improve its throughput. We focus on parallelizing the *cut detection* because we identified it as bottleneck in Section 2.3. We discuss single-node parallelization in Section 3.1 and multi-node parallelization in Section 3.2.

3.1 Single-Node Parallelization

In Figure 5, we show in pseudo code how ADWIN processes an input tuple and point out possible single-node parallelizations. We present each parallelization in detail in the following subsections. First, we decouple histogram updates and cut-checks from each other such that cut-checks cannot delay processing input tuples (Section 3.1.1). This decoupling is generally applicable to algorithms that store stream statistics in a separate data structure. Then, we parallelize the cut-check procedure itself which we call *Intra Cut-Check Parallelization* (Section 3.1.2). Finally, we discuss how to perform multiple cut-check procedures in parallel in case ADWIN detects cuts. We call this *Inter Cut-Check Parallelization* (Section 3.1.3). Both parallelizations are generally applicable to all algorithms which use multidirectional iterable datastructures.

[2]JProfiler: https://www.ej-technologies.com/products/jprofiler/overview.html
[3]VisualVM: http://visualvm.java.net



```
function processTuple(tuple t){
    add t to histogram
    do {
        run cut-check procedure
        if (cut detected ) remove oldest bucket
    } while (cut detected )
}
```

Figure 5: Scope of Adwin parallelizations.

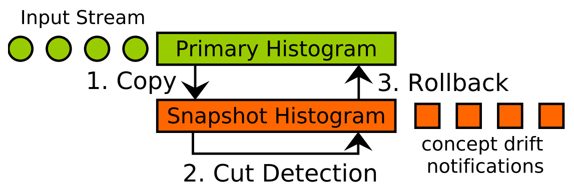


Figure 6: *Optimistic Adwin* Decoupling histogram maintenance and the cut detection procedure.

3.1.1 Cut-Check Decoupling. We introduce an optimistic parallelization of Adwin which assumes that most input tuples do not indicate a concept drift (i.e., a cut). This assumption regularly holds for big data streams, because real-world data follows laws of nature and causes few cuts compared to the number of tuples.

As explained in Section 2, Adwin performs two tasks: updating the histogram with new tuples and detecting concept drifts with a cut detection procedure. Originally, Adwin performs these tasks in succession for each tuple. This limits the throughput because the cut detection blocks the processing of the input stream.

In *Optimistic Adwin*, we decouple histogram updates and cut checks from each other to overcome throughput limitations. The algorithm performs cut checks on a separated *snapshot histogram* and adds new tuples to a *primary histogram* concurrently. We synchronize histograms after each run of the cut check procedure.

We illustrate *Optimistic Adwin* in Figure 6. One thread adds new input tuples to the primary histogram and a redo log. Another thread creates a deep copy of the primary histogram (step 1) and performs the cut check procedure on this copy (step 2). In case of a concept drift, we initiate a rollback which replaces the primary histogram with the snapshot manipulated by the cut detection procedure (step 3). Finally, we use the redo log to add missing input tuples to the primary histogram again. The overall process repeats continuously. Thereby, the majority of runs does not detect a cut and requires no rollback.

The main benefit of *Optimistic Adwin* is that the cut detection procedure cannot block the input stream anymore. During one execution of the cut check procedure, new tuples are processed already. It is important to notice that *Optimistic Adwin* introduces a latency between the insertion of a new tuple in the primary histogram and the notification about a cut. We discuss this effect in detail in Section 4. Our experiments show that we achieve detection latencies below 15μs with *Optimistic Adwin*.

3.1.2 Intra-Cut-Check Parallelization. A single thread execution of the cut-check procedure starts to check for cuts at the ending of the histogram and moves towards the beginning (top of Figure 7). Thereby, the algorithm compares sliding subwindows as described in Section 2.1. This comparison requires sums and variances of subwindows, which are aggregates of sums and variances of buckets covered by the subwindows. The algorithm updates the aggregates of subwindow (i.e., the aggregation of buckets) incrementally when moving from one iteration to the next. As initialization for the cut-check iteration, the histogram stores and maintains its overall aggregate.

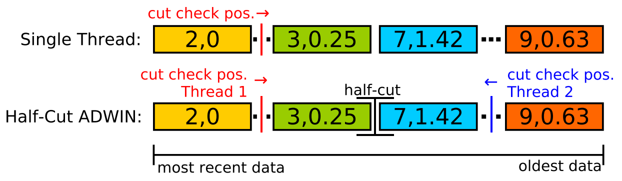


Figure 7: *Half-Cut Adwin*: Parallelisation of the cut check procedure with two threads. Both threads iterate till the middle of the histogram.

We change the cut-check iteration such that two cut checks run in parallel. To that end, we introduce *Half-Cut Adwin* in Figure 7 (bottom). Since the cut check in each iteration step is independent of cut checks of previous iteration steps, two threads can iterate over the histogram concurrently. Thereby, each thread covers half of the cut-check positions. One thread starts at the beginning and moves towards the end of the histogram and the other thread moves in the opposite direction. Both threads still update subwindow aggregates incrementally in each iteration step. *Half-Cut Adwin* terminates the cut check procedure when both threads reach the middle of the histogram or when it finds a cut. This leads to a maximal speedup factor of two and halves the latency of the cut detection. It is also easy to add on top of existing histogram implementations.

In general, the concept of *Half-Cut Adwin* extends to more than two threads. Therefore, the histogram needs to maintain additional aggregates of subwindows as initialization for additional threads. This overhead pays off for large histograms only. Since the number of buckets grows logarithmically, a high degree of Intra-Cut-Check Parallelism pays off seldom considering the overhead for aggregate maintenance in the histogram. *Half-Cut Adwin* does not have this overhead because it uses the same aggregate as initialization for both threads.

3.1.3 Inter-Cut-Check Parallelization. If Adwin finds a cut, it removes the oldest bucket from the histogram and repeats the cut check procedure till no further cuts are detected. This enables a *pessimistic parallelization* which assumes that we detect further cuts after removing old buckets from the histogram. While thread 1 performs cut checks on all buckets $1..n$, thread 2 could already check if there will be another cut after removing an old bucket and perform cut detection on buckets $2..n$. This extends to $n-1$ parallel cut check procedures each of which could also apply *Half-Cut Adwin*. However, the detection of cuts is usually rare compared to the total number of cut check procedures. Inter-Cut-Check Parallelization is not beneficial when we detect no cut. Respectively, we expect a minimal speedup from this approach. Still, it can reduce the latency of the cut detection, which is valuable for situations with frequent concept drifts.

3.2 Multi-Node parallelization

It is desirable to distribute stream processing applications over multiple nodes in a cluster in order to achieve linear speedup. Common distributed streaming engines such as Apache Flink [6] and Storm [11] achieve data parallelism on multiple nodes with data partitioning. Thus, each node is responsible for processing tuples of certain keys (e.g., user ids, regions, or event classes). *Half-Cut Adwin* and *Optimistic Adwin* are complementary to this approach and increase the throughput per partition.

It is also possible to distribute the cut detection procedure on multiple nodes. However, this requires a central shared histogram or histogram copies with multi-version concurrency control. The coordination and network overhead for these histograms would dominate the processing in our network-bounded cluster and prevent a speedup. However, multi-node parallelization of cut checks can be beneficial if we overcome the network bottleneck with technologies such as InfiniBand.

4 EVALUATION

In this section, we provide an experimental evaluation of *Half-Cut Adwin*, *Optimistic Adwin*, and the state-of-the-art. We discuss our setup in Section 4.1 and present our results in Section 4.2.

4.1 Experiment Design

Throughput. The throughput of Adwin depends on the size of the histogram. Large histograms contain more buckets and, thus, require more cut checks. More concept drifts (i.e., cuts) increase the throughput by reducing the histogram size. Our experiment measures the throughput for different histogram sizes.
Latency. We analyze the latency between adding a tuple to the histogram and the completion of the cut check procedure. This corresponds to the detection latency of cuts. We show the worst-case execution time of the cut check procedure which corresponds to finding a cut at the last cut check position. In addition, the latency of *Optimistic Adwin* includes the waiting time of a tuple in the primary histogram before the decoupled cut check procedure starts. We show a range of latencies which reflects the shortest and longest possible waiting time.
Data. Since the overhead of *Optimistic Adwin* is negligible, the performance depends on the histogram size only. Therefore, we generate batches of constant values and measure their insertion times. The insertion times include performing cut detections.
Execution Environment. We measure runtimes and latencies with *JMH* [4] which enables reliable and reproducible microbenchmarks on a Java Virtual Machine. We run all experiments on a machine with 8GB RAM and an Intel Core i5-4210U processor.
Algorithms. We compare four versions of Adwin. The open source version[5], our optimized sequential version (Section 2.3), *Optimistic Adwin* (Section 3.1.1), *Half-Cut Adwin* (Section 3.1.2).

4.2 Results

Throughput. The left plot in Figure 8 shows that our sequential reimplementation is on average 5 times faster than the original implementation. *Half-Cut Adwin* is 10 times faster than the original implementation. As expected, *Half-Cut Adwin* is almost 2 times faster than our optimized sequential reimplementation because it reduces the execution time of cut checks by almost 50%. *Optimistic Adwin* improves upon *Half-Cut Adwin* by a factor of 27 and is 54 times faster than our optimized reimplementation. This leads to a 274 times speedup compared to the original Adwin implementation. Moreover, *Optimistic Adwin* decouples the insertion of tuples into the histogram from the cut check procedure. Therefore, its throughput is not directly correlated to the histogram size, which leads to a better scalability.
Latency. In the right plot in Figure 8, we show the latencies of different Adwin versions depending on the histogram size. Our new Adwin versions reduce latencies compared to the original implementation by up to 90% and at least by 50%. *Half-Cut Adwin* has the lowest latency because it distributes the cut detection procedure on two threads without any snapshot and concurrency control overhead. *Optimistic Adwin* exhibits the largest variance (10µs) among the latencies of different tuples. This is because of the additional waiting time between adding a tuple to the primary histogram and the start of the next cut check procedure. In the worst case, a snapshot of the histogram was taken directly before inserting the tuple. This roughly doubles the latency because we first finish the cut check procedure on the recent snapshot before we do a cut check which includes the new tuple. While *Half-Cut Adwin* decreases the latency compared to *Optimistic Adwin* by 70% on average, *Optimistic Adwin* show a 27 times higher throughput. In general, all latencies are in the range of microseconds which enables fast reactions on concept drifts.

[4]JHM - http://openjdk.java.net/projects/code-tools/jmh/
[5]Adwin open source repository: https://github.com/abifet/adwin

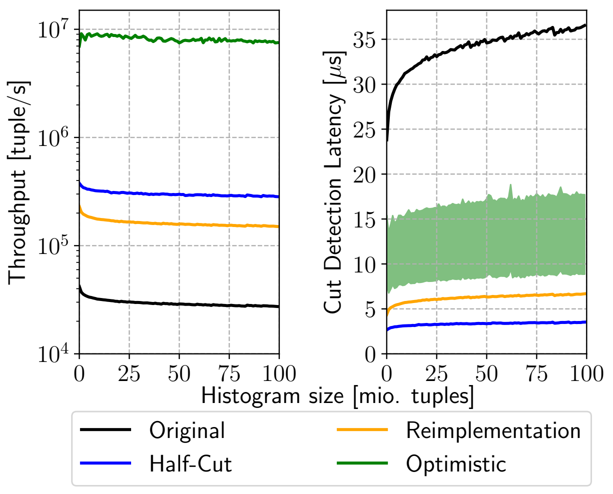


Figure 8: Throughput (left), Latency (right)

5 CONCLUSIONS

Concept drift detection, with algorithms such as Adwin, is a crucial component of stream analysis. We analyzed the bottlenecks of the Adwin algorithm and discussed several approaches for its parallelization. Our *Optimistic Adwin* algorithm decouples the concept drift detection and the window maintenance. Its evaluation shows that it has two orders of magnitude higher throughput and an at least 50% lower latency than state-of-the-art solutions.

Acknowledgements: This work was supported by the EU project Streamline (688191), DFG Stratosphere (606902), and the German Ministry for Education and Research as BBDC (01IS14013A) and Software Campus (01IS12056).

REFERENCES

[1] A. Bifet and R. Gavalda. Kalman filters and adaptive windows for learning in data streams. In *Discovery science*, volume 4265, pages 29–40. Springer, 2006.
[2] A. Bifet and R. Gavalda. Learning from time-changing data with adaptive windowing. In *SDM*, pages 443–448. SIAM, 2007.
[3] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. MOA: Massive online analysis. *Journal of Machine Learning Research*, 11(May):1601–1604, 2010.
[4] A. Bifet, G. Holmes, and B. Pfahringer. Moa-tweetreader: real-time analysis in twitter streaming data. In *Discovery Science*, pages 46–60. Springer, 2011.
[5] A. Bifet, G. Holmes, B. Pfahringer, R. Kirkby, and R. Gavaldà. New ensemble methods for evolving data streams. In *SIGKDD*, pages 139–148. ACM, 2009.
[6] P. Carbone, A. Katsifodimos, S. Ewen, et al. Apache Flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 36(4), 2015.
[7] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.
[8] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM Computing Surveys (CSUR)*, 46(4):44, 2014.
[9] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *VLDB*, pages 180–191. VLDB Endowment, 2004.
[10] R. Klinkenberg and I. Renz. Adaptive information filtering: Learning in the presence of concept drifts. *Learning for Text Categorization*, pages 33–40, 1998.
[11] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, et al. Storm@twitter. In *SIGMOD*, pages 147–156. ACM, 2014.
[12] G. Widmer and M. Kubat. Learning in the presence of concept drift and hidden contexts. *Machine learning*, 23(1):69–101, 1996.