Datenbank Spektrum (2022) 22:99–107
https://doi.org/10.1007/s13222-022-00417-y

SCHWERPUNKTBEITRAG

Algorithms for Windowed Aggregations and Joins on Distributed Stream Processing Systems

Juliane Verwiebe[1] · Philipp M. Grulich[1] · Jonas Traub[1] · Volker Markl[1,2]

Received: 31 January 2022 / Accepted: 13 May 2022 / Published online: 9 June 2022
© The Author(s) 2022

Abstract
Window aggregations and windowed joins are central operators of modern real-time analytic workloads and significantly impact the performance of stream processing systems.
This paper gives an overview of state-of-the-art research in this area conducted by the Berlin Institute for the Foundations of Learning and Data (BIFOLD) and the Technische Universität Berlin. To this end, we present different algorithms for efficiently processing windowed operators and discuss techniques for distributed stream processing. Recently, several approaches have leveraged modern hardware for windowed stream processing, which we will also include in this overview. Additionally, we describe the integration of windowed operators into various stream processing systems and diverse applications that use specialized window operations.

Keywords Window · Window Aggregation · Windowed Joins · Distributed · Stream Processing · Stream Processing Systems · Modern Hardware

1 Introduction

Windowing is a fundamental building block of any stream processing system. Data streams are divided into windows that capture a finite portion of tuples to which stateful operators can be applied. As a result, windowing is a prerequisite for performing aggregations or joins and enables stream processing systems to produce timely responses to long-running streaming queries.

Modern real-time analytics require complex queries, including joins, complex window types, different window measures, and diverse aggregation functions. Concurrent queries and high-velocity data streams generate increased workloads for the systems. The algorithms also have to take into account characteristics of the data streams, such as out-of-order tuples or concept drift. Consequently, stream processing systems need efficient approaches for windowed operators. Centralized computation solutions limit the scalability of applications. Thus, the efficient analysis of ever-increasing data streams requires processing with multiple nodes. However, distributed approaches need to be adapted to the characteristics of stream processing and windowed operators.

Complex query workloads in combination with data-intensive streams lead to a significant overhead in stream processing systems. Low latency and high throughput are requirements of today's real-time applications. As a result, the efficiency of window aggregation and windowed joins is critical for the performance of stream processing systems.

In this paper, we particularly provide an overview of the research conducted at TU Berlin and BIFOLD (Table 1). We present different approaches for the efficient computation of windowed aggregations and joins on stream processing systems. The rest of this paper is structured as follows: We first discuss related work in Sect. 2. Sect. 3 presents operators that utilize stream slicing, which enables efficient aggregation for overlapping windows and concurrent queries. The approaches presented in Sect. 4 exploit advantages of modern hardware, such as multi-core processors and high-

✉ Juliane Verwiebe
juliane.verwiebe@tu-berlin.de

Philipp M. Grulich
grulich@tu-berlin.de

Jonas Traub
jonas.traub@tu-berlin.de

Volker Markl
volker.markl@tu-berlin.de

[1] Technische Universität Berlin, Einsteinufer 17, 10623 Berlin, Germany

[2] DFKI GmbH, Alt-Moabit 91c, 10559 Berlin, Germany

Table 1 Overview over the presented research

Windowed Operators	Cutty [14], Scotty [65, 66, 68], AJoin [34], Eco-Join [44], Zhang et al. [76], Parallel ADWIN [26]
Modern Hardware	Zeuch et al. [72], Grizzly [28], Slash [21]
Distributed Window Processing	AStream [35], Disco [8], Rhino [6, 20]
Stream Processing Systems	Scotty Connectors to Apache Flink [13], Storm [61], Beam [3], Samza [47], Kafka Streams [38], Spark [60, 71]; NebulaStream [73, 74], Agora [69], SENSE [62, 63, 67]
Applications	Condor [48], Stream Generator [27], I2 [64], M4 [32]

speed networks, to accelerate the performance of stream processing systems. Sect. 5 discusses operators that utilize parallelism and distributed processing. In Sect. 6, we describe the integration of the techniques and algorithms in various stream processing systems. Various applications are provided in Sect. 7. We summarize the evaluation results of the presented work in Sect. 8 and point out directions for future research in Sect. 9.

2 Related Work

While this paper focuses on presenting research conducted at BIFOLD and TU Berlin, there exists a wide body of work dealing with related challenges. This section reviews related work from different research areas, which are grouped according to the topics of the remaining paper.

Window Aggregation Techniques. Li et al. [40–42] contribute to the research area of window aggregation by introducing buckets which store window aggregates that can be computed incrementally to achieve low latency. Sharing aggregates among overlapping windows is not possible with buckets, resulting in redundant computations. To overcome this issue, several techniques use partial window aggregation (e.g., Arasu and Widom [5], Theodorakis et al. [57], Zhang et al. [75]), where intermediate results of an aggregate are calculated and then combined to obtain the final result. For storing partial aggregates, Tangwongsan et al. [53–55] utilize various data structures (e.g., arrays, trees, or stacks) in different partial aggregation techniques including FlatFAT, TwoStacks, DABA, and FIBA. Slicing techniques, such as Pairs presented by Krishnamurthy et al. [39], increase throughput by assigning each tuple to exactly one non-overlapping slice, requiring only one aggregation operation. Furthermore, sharing aggregates is possible among multiple concurrent window queries, as shown, for instance, by Theodorakis et al. [58, 59]. Aggregation functions can be differentiated in commutative or non-commutative, and invertible or non-invertible. The optimization techniques of Shein et al. [51, 52] address these different types of aggregation functions. Bou et al. [10–12] present multiple techniques for out-of-order incremental sliding window aggregation. To give an overview of the different techniques, Hirzel et al. [30] survey several sliding window aggregation algorithms and summarize that the choice of the best algorithm depends on the aggregation operation, latency requirements, window type, sharing requirements, and out-of-order processing.

Stream Join Algorithms. Kang et al. [33] examine different algorithms for stream joins over sliding windows and propose a cost model for analyzing their expected performance. Teubner and Müller [56] introduce the handshake join as a method for inter-window joins (i.e., joining overlapping windows such as sliding windows) to utilize highly parallel architectures. In a follow-up work, Roy et al. [49] demonstrate that fast-forwarding tuples between CPU cores reduces the high latency of the handshake join. Karnagel et al. [36] utilize the GPU of tightly-coupled processors with an integrated GPU for computationally intensive parts of the stream join. As another technique for inter-window joins, the SplitJoin proposed by Najafi et al. [46] introduces a top-down dataflow model that utilizes modern multicores. Elseidy et al. [22] processes intra-window joins (i.e., joining two streams over a single window) on parallel threads while adapting to data dynamics by state repartitioning and dataflow routing. The concurrent stream join of Shahvarani and Jacobsen [50] uses a shared index data structure for state materialization on multi-core processors.

Stream Processing Systems. The first stream processing systems that have been introduced are, for instance, Aurora by Abadi et al. [1, 2], TelegraphCQ by Chandrasekaran et al. [17], and NiagaraCQ by Chen et al. [18]. Systems presented more recently are Trill by Chandramouli et al. [15, 16] and Naiad by Murray et al. [45]. Koliosis et al. [37] proposed SABER which optimizes stream processing on heterogeneous processors combining CPUs and GPUs. StreamBox [43] exploits modern hardware by grouping tuples into epochs and processing them in parallel. Apache Flink (Carbone et al. [13]), Storm (Toshniwal et al. [61]), and Spark (Zaharia et al. [71]) are state-of-the-art open-source stream processing systems.

3 Stream Slicing

Window aggregation has a high impact on the performance of stream processing systems due to complex window types (e.g., tumbling, sliding, or session windows), aggregation functions (e.g., sum, avg, or median), concurrent queries,

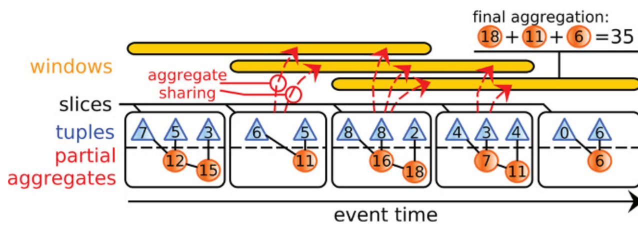

Fig. 1 Example window aggregation with stream slicing

and out-of-order events. The following work focuses on optimizing the window aggregation process for such complex workloads.

Cutty [14] combines the approaches of stream slicing and partial aggregation to support a wide range of different window types. Stream slicing (see Fig. 1) decomposes windows into non-overlapping subsets called slices. The tuples contained in the slices are used to compute partial aggregates, which can be combined to generate further intermediate results and final aggregates. Carbone et al. introduce the concept of user-defined windows (UDWs) that contain custom logic of window types defined by the user. The differentiation in deterministic and non-deterministic window types eliminates the need for knowing exact window semantics by exploiting the properties of these two classes. Deterministic window types (e.g., tumbling, sliding, session, punctuation) can decide during the processing of the tuple whether a tuple represents the beginning or the end of a window. In contrast, non-deterministic window types (e.g., delta-based [23], multi-type [14, 23, 66], adaptive windowing [9]) can not assert whether the currently processed tuple starts a window or not. The aggregator also enables sharing aggregates between concurrent windows of multiple queries.

Based on the Cutty technique, the open-source operator Scotty [65, 66, 68] introduces general and efficient window aggregation for out-of-order streams. It yields general applicability by implementing the general stream slicing technique [66] which adapts to the different workload characteristics of aggregation queries, i.e., window types, aggregation functions (e.g., invertible, associative), window measures (e.g., time-based, count-based), and stream order. Scotty extends the slicing technique in combination with out-of-order processing on complex types of windows such as session windows. To this end, the framework[1] supports multiple window types of varying complexity (e.g., tumbling, sliding, punctuation, slide-by-tuple windows). Furthermore, Scotty can be extended with user-defined window types as well as aggregation functions.

[1] Open Source Link: https://github.com/TU-Berlin-DIMA/scotty-window-processor.

4 Algorithms for Optimizations on Modern Hardware

The following approaches aim to optimize window aggregations and joins on stream processing systems by exploiting modern hardware, e.g., multi-core CPUs, GPUs, and high-speed networks.

Zeuch et al. [72] propose a windowing mechanism using a double-buffer and lock-free data structures that allow writing to a window buffer in parallel to minimize synchronization overhead. Multiple non-active buffers store previous window results, output the final aggregation result, and reinitialize the buffer memory. One buffer is always active to collect incoming tuples, which avoids a delay in processing the input stream and increases the throughput. To reduce the latency, tuples are incrementally aggregated in the active buffer whenever possible.

Grizzly [28] introduces adaptive query compilation to increase the efficiency of streaming queries on modern hardware. Depending on the specific window types, window measures, and window functions, Grizzly selects the physical operators and generates specialized code. This enables Grizzly to specialize the executed code with regards to the user-provided workload. Furthermore, Grizzly follows a task-based parallelization to utilize the modern multi-core CPUs fully. Similar to Zeuch et al., it leverages lock-free operations to compute window aggregates. To this end, Grizzly introduces a lightweight coordination scheme to coordinate the finalization of windows across multiple threads while avoiding coordination overhead. As streaming queries are inherently long-running, Grizzly continuously monitors the execution, collects profiling information, and applies adaptive optimizations to improve execution efficiency. For example, it specializes the data structure for keyed aggregations if it can detect a specific key distribution. In combination with stream slicing, this technique could further improve performance.

Edge devices in the Internet of Things (IoT) perform computations of intermediate results closer to the data sources to avoid network congestion and computation overhead in the cloud. Since these devices are battery-powered, they have a limited energy budget, which becomes more taxed when processing complex workloads. Michalke et al. propose ecoJoin [44], a stream join operator exploiting the modern hardware of these edge devices to reduce energy consumption. In particular, ecoJoin focuses on devices that combine embedded CPUs and GPUs on a single system. To this end, it provides a new stream join algorithm, which uses the GPU to accelerate processing. The algorithm adapts the size of tuple batches based on stream ingestion rates and latency tolerances. These batches are distributed over the cores to parallelize the join phases.

As a result, the efficiency increases even for fast input rates on large windows.

Remote Direct Memory Access (RDMA) hardware allows data transfer with high throughput and low latency. This has the potential to mitigate the bottleneck that networks pose in distributed settings while fulfilling the real-time requirements of stream processing. With a novel processing model designed for RDMA, Slash [21] accelerates distributed stream processing computations. The stateful query executor applies multiple instances of the same operator in parallel to scale the processing of streaming queries across many nodes. A special protocol enables the data exchange among nodes via RDMA channels leveraging the full speed of the RDMA network. Multiple slash executor instances store their partial state (e.g., partial aggregates of windows) in the Slash State Backend. Distributed operator states (e.g., of windows) are merged in a lazy approach. The technique also provides a windowed join based on a hash-join. Slash operates on a windowing approach that relies on general stream slicing [66]. The shared mutable state allows the technique to omit expensive re-partitioning operations which increases the throughput in contrast to scale-out stream processing systems.

5 Parallel and Distributed Stream Processing

This section presents parallel and distributed techniques that address the challenge of handling high-velocity data streams while delivering real-time results with low latency and high throughput.

Since the volume of data streams that need to be processed in modern real-time analytics increases continuously, scalability is an important factor of stream processing systems. Distributed stream processing approaches enable such scaling, but have to deal with challenges coming from the distribution of streaming queries and operators, such as windows, window aggregations, and windowed joins. Disco [8] performs complex window aggregation in a distributed manner by aggregating incoming tuples on multiple independent nodes and merging them to the final result. Merging strategies ensure correct aggregation semantics for different window types (i.e., context-free or context-aware) and aggregation functions (i.e., decomposable or holistic). In contrast to the centralized data collection that stream processing systems generally perform, streams can be processed closer to their sources.

Streaming queries have the property of being continuous and long-running, but their operators may need to be modified at some time, for instance, to adapt to varying data rates. Bartnik et al. [6] present generic protocols that allow the modification of operators and of the data flow of running queries. Many stream processing systems enable such a reconfiguration only by restarting the execution of the modified query, which leads to a redistribution of the query state and affects other systems relying on the output. The protocols enable changing the operator function or introducing new operators as well as migrating operators to different nodes for distributed processing. Running queries with very large distributed operator state can be reconfigured on the fly with the library Rhino [20]. A handover protocol migrates the processing and state of the running operator among workers, and a state migration protocol asynchronously replicates the local check-pointed state on workers. During the configuration, Rhino guarantees exactly-once processing and does not stop the query execution.

In real-world applications, it is often necessary to handle many short-term ad-hoc queries in addition to processing continuous long-running queries. The framework AStream [35] extends distributed stream processing systems to support ad-hoc query workloads while sharing computation and resources. When operators have common upstream operators and common partitioning keys, they can be shared among queries. Using a distributed pipeline-parallel architecture, AJoin [34] supports ad-hoc stream processing joins. The technique shares data and computation between multiple queries and utilizes late materialization to pass down a reduced number of intermediate results to subsequent operators. A periodic re-optimization of the query execution plan at runtime ensures efficient execution.

The IoT consists of distributed and heterogeneous sensor nodes, which brings new challenges for processing data streams. Clock offsets occur among the diverse sensor nodes due to different time synchronization techniques. Consequently, joins are affected by incoherent timestamps of tuples produced by multiple devices, resulting in incorrect predictions and false correlations. SENSE [67] provides time coherence for data acquired from distributed sensors without the need of requiring reliable clock synchronization among all nodes. Traub combines the techniques presented in SENSE with windowed joins and as well as temporal and spatial aggregation techniques [62, 63].

SENSE, Rhino, and Scotty are also incorporated in the the IoT platform NebulaStream (Sect. 6).

Over time, data streams are subject to changes, for instance, changing user preferences or economic changes. These so-called concept drifts lead to incorrect predictions of the trained machine learning model because it is not appropriately fitted to the current data. Adaptive windowing (ADWIN) [9] detects concept drift and dynamically adapts the model to changes. Grulich et al. [26] identify bottlenecks of the ADWIN algorithm and modify it to run in parallel on multiple threads. As a result, the scalability of adapting to concept drift increases.

Zhang et al. [76] have studied different algorithms of parallel intra-window joins (i.e., joining two streams over a single window) on multi-cores. They differentiate existing approaches in lazy execution and eager execution methods. The lazy approach buffers input tuples of windows from two streams before joining them. Eager execution immediately joins tuples as they arrive, producing partial matches. Zhang et al. conclude that the choice of an appropriate algorithm depends on workload characteristics (e.g., tuple arrival rate, window length), application requirements (e.g., latency, throughput), and hardware architectures (e.g., number of cores and vector extensions).

6 Systems Integration

As the proposed techniques provide efficient and state-of-the-art window aggregation, they have been integrated into various systems.

Scotty [65, 66, 68] can be integrated in various stream processing systems. It already provides connectors for Apache Flink [13], Apache Storm [61], Apache Beam [3], Apache Samza [47], Apache Kafka Streams [38], and Apache Spark Continous Processing [60, 71].

The NebulaStream [73, 74] platform offers an end-to-end data-management system for the IoT. It provides a unified environment for a sensor-fog-cloud infrastructure that handles heterogeneous hardware, unreliable nodes, and elastic network topology. To ensure a high performance across these heterogeneous devices, NebulaStream utilizes adaptive query compilation [28] and generates specialized code depending on the device capabilities. Furthermore, the system follows the design principle of maximized sharing. To achieve this on query level, the integration of AStream [35] enables to share data among multiple streaming queries. The general stream slicing technique [66] reuses partial aggregation results among overlapping windows and is therefore integrated in NebulaStream to share data on an operator level. Additionally, NebulaStream integrates Babelfish [29] for the acceleration of UDF-based operators, e.g., to enable the definition of user-provided windows function and aggregations.

Agora [69] provides a unified ecosystem for assets of the entire data value chain i.e. algorithms, data, and physical infrastructure components. In marketplaces, different stakeholders can offer their assets and modify and remove them. Agora enables not only to exchange assets but also to combine them to novel applications as well as the resources to execute these applications. Fair payment requires to track the asset usage. A tracking function called from asset source code, operators, or the tracking of the amount of processed data results in many function calls. The aggregation of these usage counters leverages the techniques Scotty [65, 66, 68] and Disco [8].

Darwin [7] introduces a scale-in stream processing system that attempts to maximize hardware utilization on diverse hardware setups to reduce the overall infrastructure costs. To this end, it leverages query compilation and tailors execution towards a specific hardware setup. Furthermore, it provides fault-tolerance by supporting larger-than-memory window states.

7 Applications

In this section, we present diverse applications that utilize windowed stream operations.

The framework Condor [48] allows users to write synopses-based streaming jobs. Synopses enable the approximate computation of quantities that are otherwise expensive or impossible to compute precisely. The work models synopses as stateful window aggregation functions due to their similar concept of combining several values into one total value. Condor supports parallel synopses computation and evaluation and implements all synopses types (i.e., sketches, histograms, wavelets, samplers). It uses Scotty as an underlying slicing technique for computing approximate aggregates with windowed synopses.

The open-source stream generator proposed by Grulich et al. [27] enables the evaluation of modern stream processing systems. It produces deterministic data streams from arbitrary input data sets with different data rates, distributions, and characteristics such as the fraction of out-of-order tuples and their delay. Besides providing realistic workloads by these configurations, it is able to generate the same experiment data ensuring reproducibility.

To visualize streaming data in real-time, the interactive development environment I2 [64] has been proposed. The running cluster applications dynamically adapt to changes in the visualization without restarting. The algorithm ensures that only the depicted data points are transferred, which reduces workload in the front-end and backend.

The aggregation technique M4 [32] reduces the dimensionality of time-series data by rewriting queries for RDBMS-based visualization systems. Additional operators for data reduction are incorporated in the queries that determine four values (i.e., min, max, first, last) per pixel column. This reduces the computational load for visualization while still providing loss-free plots in the form of line-charts.

8 Evaluation Summary

This section summarizes the main results of the evaluations of each of the techniques presented. More detailed descriptions of experiments and results can be found in the original publications.

8.1 Stream Slicing

The evaluations of Cutty [14] and Scotty [65, 66, 68] show that slicing techniques outperform other approaches such as tuple buffers, buckets, and aggregate trees in terms of throughput. Furthermore, they provide an increased scalability to a large number of concurrent windows. In addition, Scotty outperforms alternative techniques with regard to throughput and scales to a large amount of concurrent windows for workloads that include out-of-order tuples and context-aware windows (e.g., session windows).

8.2 Algorithms for Optimizations on Modern Hardware

The experiments of Zeuch et al. [72] show that current stream processing systems underutilize modern hardware in terms of full computational power and memory bandwidth. The proposed streaming optimizations for modern hardware are compared to Apache Flink [13], Spark Streaming [71], Storm [61], Saber [37], and StreamBox [43] on three streaming benchmarks and enable a throughput up to two orders of magnitude higher than these systems. The evaluation shows that the lock-free windowing approach provides a high throughput for stream processing systems on modern hardware.

The optimizations of Zeuch et al. are also used in the experiments of Grulich et al. [28], where Grizzly is compared to two hand-optimized implementations [72], Apache Flink [13], StreamBox [43], and Saber [37] on the Yahoo! Streaming Benchmark [19] (Fig. 2). It's utilization of modern hardware causes Grizzly to outperform current stream processing systems by up to one order of magnitude in throughput. The query compilation approach additionally shows this performance improvement under different complex query workloads (i.e., window types, aggregation functions, concurrent queries) that have a high impact on performance.

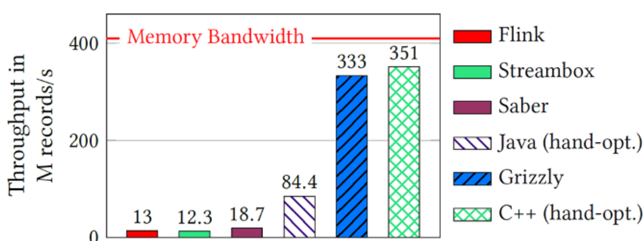

Fig. 2 Throughput of Grizzly compared to state-of-the-art systems on Yahoo! Streaming Benchmark

Michalke et al. [44] demonstrate that EcoJoin significantly enhances performance with regard to throughput and power consumption compared to state-of-the-art stream join algorithms. Adjusting batch sizes and scaling the clock frequency leads to improved energy efficiency. The experiments show that large batch sizes consume less power but result in higher latencies, leading to a trade-off between the two requirements.

In the evaluation presented by Del Monte et al. [21], Slash is compared against state-of-the-art stream processing systems such as LightSaber [58] and Apache Flink [13]. Native RDMA acceleration allows the system to scale with the number of nodes and enables higher throughput for common stream workloads than partitioning-based approaches. For window aggregations and windowed joins, Slash achieves a significantly higher throughput compared to the strongest scale-out baseline.

8.3 Parallel and Distributed Stream Processing

Benson et al. [8] compare Disco to centralized data collection and show that the distributed technique exhibits higher performance by scaling linearly with the number of nodes for decomposable and holistic aggregation functions. Disco significantly reduces network traffic by completely avoiding to send individual tuples between nodes for decomposable functions. For holistic functions, it combines tuples and sends them in slices to reduce TCP-overhead.

The protocols of Bartnik et al. [6] enable migrating operators with small state as fast as Apache Flink's [13] savepoint mechanism and outperforms it for migrating operators of jobs with large state. The advantage over Apache Flink is that the migration mechanism prevents data loss during restarting a job, which occurs when data is not consumed from a persistent source. The related work Rhino [20] reduces the latency compared to Flink [13] by up to three orders of magnitude for large state. The state migration protocol allows Rhino to reconfigure a query 50x faster than Flink and 15x faster than Megaphone [31].

As shown by Karimov et al. [35], the framework AStream supports thousand concurrent queries and a throughput of 70 million tuples per second. For a fluctuating workload of starting and deleting concurrent short-running queries, the framework creates and deleted 50 queries per 10 seconds within 1 second event-time latency. AStream deploys ad-hoc queries in the order of milliseconds which is a much lower query deployment latency than Apache Flink [13]. Along with Apache Flink and Spark, AStream is compared to AJoin [34]. The stream join processing engine outperforms all the systems for single query workloads and performs better than Flink for simultaneous submission of

all queries at compile time. With multiple joins operators in a query, AJoin performs better than these other systems.

Experiments conducted by Traub et al. show that SENSE [67] maintains a guaranteed coherence below a user-defined upper bound while optimizing the coherence estimate of tuples. Furthermore, the system is scalable to thousands of sensor nodes, robust to node failures, and able to reintegrate recovering nodes.

Grulich et al. [26] demonstrate that the parallel Optimistic ADWIN algorithm outperforms the original implementation and an optimized sequential reimplementation by an increased throughput of two orders of magnitude and a reduced latency of at least 50%.

In their experiments for evaluating stream joins, Zhang et al. [76] compare different parallel intra-window join algorithms with regard to throughput, latency, and progressiveness on several real-world and synthetic workloads. An algorithm that outperforms the other algorithms in all cases does not exist. Their results show that lazy approaches perform better than specifically designed eager algorithms on most workloads. They summarize their findings in a decision tree that should guide the choice of an appropriate algorithm based on different factors such as arrival rate, key duplication, and number of cores.

8.4 Systems Integration

Stream slicing can conceptually be supported by every dataflow system. Currently, Scotty is integrated in the various open-source systems we listed before (Sect. 6). For evaluation, Scotty was used as a window operator within those systems and compared to their built-in operators (Fig. 3). The results show that it provides higher throughput then all of the tested systems' built-in operators.

In an experiment performed by Zeuch et al. [73], the throughput of the Yahoo! Streaming Benchmark [70] was evaluated on on a RaspberryPi 3B+ using NebulaStream, Python, Flink, and a hand-optimized Java program. NebulaStream achieves a throughput of more than 10 million tuples per second and a higher performance than the others. Additionally, the system reduces energy requirements while achieving the same performance.

In the evaluations of Benson et al. [7], Darwin is compared to the scale-up engine Grizzly [28] and scale-out engine Apache Flink [13]. The results show the same performance as Grizzly for in-memory processing. Grizzly's additional optimizations could be utilized orthogonal to Darwin. In contrast, Darwin performs better than Flink by over an order of magnitude.

8.5 Applications

Poepsel et al. [48] evaluate Condor on several representative jobs one real and four synthetic datasets and compared to one-off implementations of the count-min sketch and Yahoo! DataSketches [70]. In the case of one-off custom implementations, Condor performs better and even maintains high performance for a large number of concurrent windows. Secondly, Condor's sketch libraries are designed to support high parallelism applications and results prove they scale linearly with the number of cores in the system which is not the case for Yahoo! DataSketches. This also holds for the other provided synopses and evaluation operators. In summary, the framework allows high-throughput for parallel synopsis maintenance while keeping the same accuracy as centralized techniques.

Traub et al. [64] show that the environment I2 significantly reduces the number of processed and transferred data points. It supports visualizing high-bandwidth data streams without a reduction of the quality.

In experiments of Jugel et al. [32], the M4 aggregation technique is compared with line simplification techniques and common naive approaches (e.g., averaging, sampling, rounding) on real-world data sets. Measuring the visualization quality shows that M4 achieves error-free visualizations. Furthermore, it provides a reduction of data volume by two orders of magnitude and a decrease of latency by one order of magnitude.

9 Future Work

There are several directions for future research on efficient algorithms for window aggregations and windowed stream

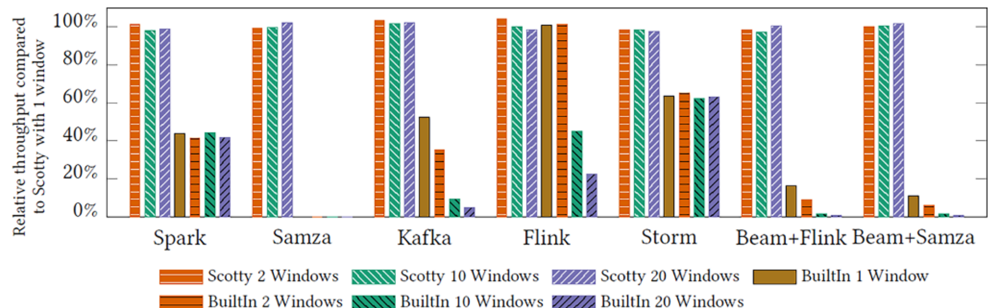

Fig. 3 Throughput of Scotty compared to built-in operators on different systems

joins on distributed stream processing systems. Window concepts become more complex since new window measures have been introduced (e.g., delta-based [23], multi-measure [14, 23, 66]) and novel window types have been proposed (e.g., Frames [25], window policies [23], snapshot windows [4, 24]). Window aggregation and windowed joins algorithms have to adapt to such sophisticated window schemes. Intra-window join algorithms have to consider and dynamically adjust to various factors such as workload, metrics, and hardware [76].

The presented optimization techniques are orthogonal and can be combined to enhance multiple aspects in a system. Future stream processing systems have to include multiple optimization techniques on operator, query, hardware, and application level to meet the high-throughput and low-latency requirements of modern streaming applications. We are working towards this goal at BIFOLD by developing the NebulaStream System [73, 74].

10 Conclusion

Researchers at TU Berlin and BIFOLD have conducted various research related to windowed operations on streaming systems. We surveyed these works covering efficient window aggregations, modern hardware optimizations, distributed stream processing approaches, and applications.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abadi DJ, Carney D, Cetintemel U, Cherniack M, Convey C, Lee S, Stonebraker M, Tatbul N, Zdonik S (2003) Aurora: a new model and architecture for data stream management. VLDB J 12(2):120–139
2. Abadi DJ, Ahmad Y, Balazinska M, Cetintemel U, Cherniack M, Hwang J-H, Lindner W, Maskey A, Rasin A, Ryvkina E et al (2005) The design of the borealis stream processing engine. CIDR 5(2005):277–289
3. Akidau T, Bradshaw R, Chambers C et al (2015) The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. PVLDB 8(12):1792–1803
4. Ali M, Chandramouli B, Goldstein J et al (2011) The extensibility framework in microsoft streaminsight. In: ICDE (IEEE)
5. Arasu A, Widom J (2004) Resource sharing in continuous sliding-window aggregates. VLDB 4:336–347
6. Bartnik A, Del Monte B, Rabl T et al (2019) On-the-fly reconfiguration of query plans for stateful stream processing engines. In: BTW 2019
7. Benson L, Rabl T (2022) Darwin: Scale-in stream processing. In: CIDR
8. Benson L, Grulich PM, Zeuch S et al (2020) Disco: Efficient distributed window aggregation. In: EDBT
9. Bifet A, Gavaldà R (2007) Learning from time-changing data with adaptive windowing. In: SIAM SDM
10. Bou S, Kitagawa H, Amagasa T (2018) Cbix: Incremental sliding-window aggregation for real-time analytics over out-of-order data streams
11. Bou S, Kitagawa H, Amagasa T (2020) L-bix: incremental sliding-window aggregation over data streams using linear bidirectional aggregating indexes. Knowl Inf Syst 62(8):3107–3131
12. Bou S, Kitagawa H, Amagasa T (2021) Cpix: Real-time analytics over out-of-order data streams by incremental sliding-window aggregation. In: IEEE TKDE
13. Carbone P, Katsifodimos A, Ewen S et al (2015) Apache flink™: Stream and batch processing in a single engine. In: IEEE CS
14. Carbone P, Traub J, Katsifodimos A et al (2016) Cutty: Aggregate sharing for user-defined windows. In: CIKM
15. Chandramouli B, Goldstein J, Barnett M, DeLine R, Fisher D, Platt JC, Terwilliger JF, Wernsing J (2014) Trill: A high-performance incremental query processor for diverse analytics. PVLDB 8(4):401–412
16. Chandramouli B, Goldstein J, Barnett M et al (2014) The trill incremental analytics engine. Tech. rep., Microsoft Research
17. Chandrasekaran S, Cooper O, Deshpande A et al (2003) Telegraphcq: Continuous dataflow processing. In: SIGMOD
18. Chen J, DeWitt DJ, Tian F et al (2000) Niagaracq: A scalable continuous query system for internet databases. In: SIGMOD
19. Chintapalli S, Dagit D, Evans B et al (2016) Benchmarking streaming computation engines: Storm, flink and spark streaming. In: IPDPSW (IEEE)
20. Del Monte B, Zeuch S, Rabl T et al (2020) Rhino: Efficient management of very large distributed state for stream processing engines. In: SIGMOD
21. Del Monte B, Zeuch S, Rabl T et al (2022) Rethinking stateful stream processing with rdma. In: SIGMOD (to appear)
22. Elseidy M, Elguindy A, Vitorovic A et al (2014) Scalable and adaptive online joins. VLDB 7(6):441–452
23. Gedik B (2014) Generic windowing support for extensible stream processing systems. Softw Pract Exp 44(9):1105–1128
24. Grabs T, Schindlauer R, Krishnan R et al (2009) Introducing microsoft streaminsight (Tech. rep.)
25. Grossniklaus M, Maier D, Miller J et al (2016) Frames: Data-driven windows. In: DEBS (ACM)
26. Grulich PM, Saitenmacher R, Traub J et al (2018) Scalable detection of concept drifts on data streams with parallel adaptive windowing. In: EDBT
27. Grulich PM, Traub J, Breß S et al (2019) Generating reproducible out-of-order data streams. In: DEBS
28. Grulich PM, Sebastian B, Zeuch S et al (2020) Grizzly: Efficient stream processing through adaptive query compilation. In: SIGMOD
29. Grulich PM, Zeuch S, Markl V (2021) Babelfish: Efficient execution of polyglot queries. PVLDB 15(2):196–210
30. Hirzel M, Schneider S, Tangwongsan K (2017) Tutorial: Sliding-window aggregation algorithms. In: DEBS

31. Hoffmann M, Lattuada A, McSherry F, Kalavri V, Liagouris J, Roscoe T (2019) Megaphone: Latency-conscious state migration for distributed streaming dataflows. PVLDB 12(9):1002–1015
32. Jugel U, Jerzak Z, Hackenbroich G, Markl V (2014) M4: a visualization-oriented time series data aggregation. PVLDB 7(10):797–808
33. Kang J, Naughton JF, Viglas SD (2003) Evaluating window joins over unbounded streams. In: ICDE (IEEE)
34. Karimov J, Rabl T, Markl V (2019) Ajoin: ad-hoc stream joins at scale. PVLDB 13(4):435–448
35. Karimov J, Rabl T, Markl V (2019) Astream: Ad-hoc shared stream processing. In: SIGMOD
36. Karnagel T, Habich D, Schlegel B et al (2013) The hells-join: a heterogeneous stream join for extremely large windows. In: DaMoN
37. Koliousis A, Weidlich M, Castro Fernandez R, Wolf AL, Costa P, Pietzuch P (2016) Saber: Window-based hybrid stream processing for heterogeneous architectures. In: SIGMOD. p 555–569
38. Kreps J (2016) Introducing kafka streams: Stream processing made simple (Confluent Blog, March)
39. Krishnamurthy S, Wu C, Franklin MJ (2006) On-the-fly sharing for streamed aggregation. In: SIGMOD
40. Li J, Maier D, Tufte K, Papadimos V, Tucker PA (2005) No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. SIGMOD Rec 34(1):39–44
41. Li J, Maier D, Tufte K et al (2005) Semantics and evaluation techniques for window aggregates in data streams. In: SIGMOD
42. Li J, Tufte K, Shkapenyuk V, Papadimos V, Johnson T, Maier D (2008) Out-of-order processing: a new architecture for high-performance stream systems. PVLDB 1(1):274–288
43. Miao H, Park H, Jeon M et al (2017) StreamBox: Modern stream processing on a multicore machine. In: USENIX
44. Michalke A, Grulich PM, Lutz C et al (2021) An energy-efficient stream join for the internet of things. In: Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)
45. Murray DG, McSherry F, Isaacs R et al (2013) Naiad: a timely dataflow system. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles
46. Najafi M, Sadoghi M, Jacobsen HA (2016) Splitjoin: A scalable, low-latency stream join architecture with adjustable ordering precision. In: ATC
47. Noghabi SA, Paramasivam K, Pan Y, Ramesh N, Bringhurst J, Gupta I, Campbell RH (2017) Samza: stateful scalable stream processing at linkedin. PVLDB 10(12):1634–1645
48. Poepsel-Lemaitre R, Kiefer M, von Hein J, Quiané-Ruiz J-A, Markl V (2021) In the land of data streams where synopses are missing, one framework to bring them all. PVLDB 14(10):1818–1831
49. Roy P, Teubner J, Gemulla R (2014) Low-latency handshake join. PVLDB 7(9):709–720
50. Shahvarani A, Jacobsen HA (2020) Parallel index-based stream join on a multicore cpu. In: SIGMOD
51. Shein AU, Chrysanthis PK, Labrinidis A (2017) Flatfit: Accelerated incremental sliding-window aggregation for real-time analytics. In: SSDBM
52. Shein AU, Chrysanthis PK, Labrinidis A (2018) Slickdeque: High throughput and low latency incremental sliding-window aggregation. In: EDBT
53. Tangwongsan K, Hirzel M, Schneider S, Wu K-L (2015) General incremental sliding-window aggregation. PVLDB 8(7):702–713
54. Tangwongsan K, Hirzel M, Schneider S (2017) Low-latency sliding-window aggregation in worst-case constant time. In: DEBS, pp 66–77
55. Tangwongsan K, Hirzel M, Schneider S (2019) Optimal and general out-of-order sliding-window aggregation. PVLDB 12(10):1167–1180
56. Teubner J, Mueller R (2011) How soccer players would do stream joins. In: SIGMOD
57. Theodorakis G, Koliousis A, Pietzuch P et al (2018) Hammer slide: work-and cpu-efficient streaming window aggregation
58. Theodorakis G, Koliousis A, Pietzuch P et al (2020) Lightsaber: Efficient window aggregation on multi-core processors. In: SIGMOD
59. Theodorakis G, Pietzuch PR, Pirk H (2020) Slideside: A fast incremental stream processing algorithm for multiple queries. In: EDBT
60. Torres J, Armbrust M, Das T et al (2018) Introducing low-latency continuous processing mode in structured streaming in apache spark 2.3. Databricks Blog
61. Toshniwal A, Taneja S, Shukla A et al (2014) Storm@ twitter. In: SIGMOD
62. Traub J (2019) Demand-based data stream gathering, processing, and transmission. PhD thesis, Technische Universität Berlin. https://www.depositonce.tu-berlin.de/handle/11303/10519. Accessed 25.01.2022
63. Traub J (2021) Demand-based data stream gathering, processing, and transmission: efficient solutions for real-time data analytics in the Internet of things. Books on Demand, Norderstedt
64. Traub J, Steenbergen N, Grulich PM et al (2017) I2: Interactive real-time visualization for streaming data. In: EDBT
65. Traub J, Grulich P, Cuéllar AR et al (2018) Scotty: Efficient window aggregation for out-of-order stream processing. In: ICDE
66. Traub J, Grulich P, Cuéllar AR et al (2019) Efficient window aggregation with general stream slicing. In: EDBT
67. Traub J, Hülsmann J, Breß S et al (2019) SENSE: Scalable data acquisition from distributed sensors with guaranteed time coherence. arXiv preprint arXiv, vol 191204648
68. Traub J, Grulich PM, Cuéllar AR et al (2021) Scotty: General and efficient open-source window aggregation for stream processing systems. In: TODS
69. Traub J, Kaoudi Z, Quiané-Ruiz J-A, Markl V (2021) Agora: Bringing together datasets, algorithms, models and more in a unified ecosystem [vision]. SIGMOD Rec 49(4):6–11
70. Yahoo! (2020) Sketches library from Yahoo! https://datasketches.apache.org/. Accessed 12.04.2022
71. Zaharia M, Xin RS, Wendell P, Das T, Armbrust M, Dave A, Meng X, Rosen J, Venkataraman S, Franklin MJ et al (2016) Apache spark: a unified engine for big data processing. Commun ACM 59(11):56–65
72. Zeuch S, Monte BD, Karimov J, Lutz C, Renz M, Traub J, Breß S, Rabl T, Markl V (2019) Analyzing efficient stream processing on modern hardware. PVLDB 12(5):516–530
73. Zeuch S, Chaudhary A, Monte B et al (2020) The NebulaStream Platform: Data and application management for the internet of things. In: CIDR
74. Zeuch S, Zacharatou ET, Zhang S, Chatziliadis X, Chaudhary A, Del Monte B, Giouroukis D, Grulich PM, Ziehn A, Mark V (2020) NebulaStream: Complex analytics beyond the cloud. Open J Internet Things 6(1):66–81
75. Zhang C, Akbarinia R, Toumani F (2021) Efficient incremental computation of aggregations over sliding windows. In: ACM SIGKDD
76. Zhang S, Mao Y, He J et al (2021) Parallelizing intra-window join on multicores: An experimental study. In: SIGMOD