2018 IEEE 34th International Conference on Data Engineering

Scotty: Efficient Window Aggregation for out-of-order Stream Processing

Jonas Traub[1], Philipp M. Grulich[2], Alejandro Rodríguez Cuéllar[1], Sebastian Breß[1,2]
Asterios Katsifodimos[3], Tilmann Rabl[1,2], Volker Markl[1,2]

[1]Technische Universität Berlin [2]DFKI GmbH [3]Delft University of Technology

***Abstract*— Computing aggregates over windows is at the core of virtually every stream processing job. Typical stream processing applications involve overlapping windows and, therefore, cause redundant computations. Several techniques prevent this redundancy by sharing partial aggregates among windows. However, these techniques do not support out-of-order processing and session windows. Out-of-order processing is a key requirement to deal with delayed tuples in case of source failures such as temporary sensor outages. Session windows are widely used to separate different periods of user activity from each other.**

In this paper, we present Scotty, a high throughput operator for window discretization and aggregation. Scotty splits streams into non-overlapping slices and computes partial aggregates per slice. These partial aggregates are shared among all concurrent queries with arbitrary combinations of tumbling, sliding, and session windows. Scotty introduces the first slicing technique which (1) enables stream slicing for session windows in addition to tumbling and sliding windows and (2) processes out-of-order tuples efficiently. Our technique is generally applicable to a broad group of dataflow systems which use a unified batch and stream processing model. Our experiments show that we achieve a throughput an order of magnitude higher than alternative state-of-the-art solutions.

I. Introduction

Streaming window aggregation is a crucial building block for virtually every streaming application. Typical examples for window aggregations are: computing revenues over the past hour, monitoring the average speed of a car, or providing user statistics per browser session. Out-of-order streams [1] are now supported by many modern stream processing systems such as Millwheel [2], Apache Flink [3], and Apache Spark [4], as well as programming models such as Google Dataflow [5] and its open source implementation, Apache Beam [6].

In an out-of-order stream, tuples arrive in a different order than they are produced by sources such as sensor nodes [7]. This disorder is due to sensor failures, transmission errors, or other network issues, which cause tuples to arrive delayed. At the same time, streaming systems extend their window types beyond tumbling and sliding time windows with other types such as session windows. Sessions separate periods of user activity from each other. Typical examples of sessions are taxi trips, browser sessions, and interactions with an ATM.

Both, out-of-order streams and session windows, make window aggregation challenging. First, out-of-order processing requires to update previously computed results in case tuples arrive late. Second, unlike tumbling and sliding windows, the start and end times of session windows are not known a priori. Tuples arriving out-of-order can modify the start and end of sessions, fuse sessions, and introduce new sessions.

Aggregate sharing is a common optimization technique which prevents redundant computations for overlapping windows. Such aggregate sharing techniques have been proposed for general window aggregation [8], [9]. These techniques store a tree of partial aggregates, which they use to calculate aggregates for arbitrary time intervals. Although general aggregation techniques facilitate diverse window types they come at a high memory cost: they store aggregate trees in addition to all tuples for the duration of the longest window requested by any query.

On the other side of the spectrum, more specialized techniques propose optimizations for tumbling and sliding windows [10], [11] as well as user-defined windows [12]. These techniques compute partial aggregates for non-overlapping subsets of the data, called *slices*. They use those partial aggregates as intermediate results for the aggregation of multiple overlapping windows and queries. Since they store one partial aggregate per slice only, those techniques are considerably more memory efficient than general aggregation techniques.

However, none of the existing techniques can be used for aggregating out-of-order streams and session windows efficiently. More specifically, current techniques either suffer from a high memory footprint (FlatFAT [9]), have a heavy update complexity for out-of-order tuples (B-INT [8]), do not consider out-of-order processing (Cutty [12], SABER [13]), or do not support session windows (Panes [11], Pairs [10], Fragments [13]). As a result, current systems with support for out-of-order streams and sessions [3], [5], [6] cannot utilize any of the existing aggregate sharing techniques. Instead, they buffer incoming tuples in a separate *bucket per window* [1] and perform the aggregation when the session window has been finalized. This is suboptimal because i) it limits the throughput because it executes redundant aggregate calculations for overlapping windows, ii) it leads to a high memory consumption because tuples are stored several times in buckets of overlapping windows, and iii) it causes a high result latency because aggregates are computed lazily when windows end.

In this paper, we make the following contributions:

1) We present Scotty, an operator for efficient streaming window aggregation which enables *stream slicing*, *pre-aggregation*, and *aggregate sharing* for out-of-order data streams and session windows.
2) We show that *stream slicing*, *pre-aggregation*, and *aggregate sharing* are beneficial for combinations of different session windows as well as combination of session, sliding, and tumbling windows.
3) We design a Slice Manager which retains the minimum number of slices for tumbling, sliding, and session windows when processing tuples out-of-order.
4) We experimentally show that Scotty achieves the highest throughput compared to state-of-the-art techniques.

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00135

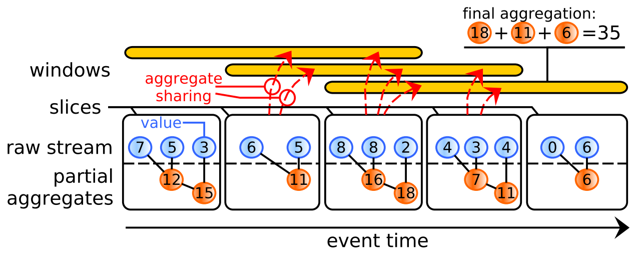


Fig. 1: Window Aggregation with Stream Slicing.

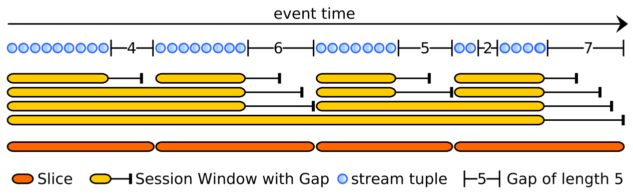


Fig. 2: Session Window Aggregate Sharing.

II. Background

Window Types. The Dataflow Model [5], defines three window types: *tumbling*, *sliding*, and *session* windows. A *tumbling* (or *fixed*) window splits the time into segments of equal length l. The end of one window marks the beginning of the next window. *Sliding* windows define a slide step of length l_s in addition to the length l to determine how often a new window starts. Consecutive windows overlap when l_s is smaller than l. A *session* window covers a period of activity followed by a period of inactivity. Thus, a session window times out (ends) if no tuple arrives for some time gap l_g.

Notion of Time. We consider two notions of time: *event-time* and *processing-time*. The *event-time* is the time when an event is captured. *Processing-time* is the time when an operator processes a tuple. We focus on event-time windows because applications typically define windows based on event-time instead of processing-time [3], [5].

Unordered Data Streams. Stream tuples may arrive at the stream processing system in non-chronologically order with with respect to their event-times [1]. In the remainder of the paper, we distinguish *in-order* tuples and *out-of-order* tuples. A tuple is *out-of-order* if at least one tuple processed before has a greater event-time. Otherwise it is considered *in-order*.

Partial Aggregation. Partial window aggregation (bottom of Figure 1) reduces the *memory consumption* and the *latency*. Instead of computing aggregates when a window ends, we update partial aggregates incrementally when tuples arrive [9]. At the end of a window, only a few final aggregation steps remain (top right of Figure 1). This reduces the output latency compared to a naive solution which buffers all tuples and aggregates upon window ends only.

Aggregate Sharing. We share partial aggregates among overlapping windows to avoid redundant computations. We compute the partial aggregate only once per slice and re-use it for all windows covering this slice. In Figure 1, dashed arrows mark multiple uses of partial aggregates.

III. In-Order vs. Out-Of-Order Stream Slicing

Stream slicing reduces the memory footprint, the processing latency, and the CPU load of streaming window aggregation. The core idea of stream slicing is to divide (i.e., *slice*) a data stream into non-overlapping and finite chunks of data (i.e., *slices*). The system computes a partial aggregate for each slice. At the end of a window, the system computes the overall aggregate for that window by combining the partial aggregates of slices. Overall, Stream slicing is beneficial for three reasons: (1) it enables partial aggregation, (2) it facilitates aggregate sharing, and (3) it allows for compressing data within slices (store just one pre-aggregated value per slice instead of buffering all tuples).

The goal of a Stream Slicer is to produce as few slices as possible in order to save memory and to reduce the final aggregation steps when windows end (reduce latency). The key difference between in-order and out-of-order processing is in the finalization of slices: When processing in-order tuples only, slices and their aggregates cannot change once a slice ended. Thus, past slices are final. When processing out-of-order tuples, we wait for late tuples before we output window aggregates. An out-of-order tuple possibly belongs to a slice in the past which leads to updating past slices. Thus, past slices are not final. For *in-order processing*, it is sufficient to separate slices whenever a window starts as shown by Carbone et al. [12]. For *out-of-order processing*, we also need to separate slices when windows end to allow for updating slices from the past with out-of-order tuples. As a result, in-order slicing produces no more slices than there are windows because slices start when windows start. Out-of-order processing requires a maximum of twice as many slices than there are windows because window start and end separate slices. There are fewer slices when window edges coincide. According to the above observations, Scotty produces the minimum amount of slices.

IV. Stream Slicing for Session Windows

Stream slicing for session windows is more complex than for sliding or tumbling windows because we do not know the event-times of window edges up front. Instead, the start and end times of sessions depend on the processed tuples. Thus, we must monitor the differences between the event-times of consecutive tuples (i.e., the gaps between tuples) in order to detect session timeouts. We show an example for session window stream slicing in Figure 2. The example has four session window queries with the minimum gaps $l_g = 3$, 5, 6, and 7. Our example leads to five observations:
1) Multiple session window queries with different gaps benefit from aggregate sharing.
2) Session windows would also share aggregates with concurrent sliding and tumbling windows.
3) Sessions of a single query have no overlap. Thus, a single session window query cannot benefit from aggregate sharing.
4) Slices can cover the gaps between sessions because gaps do not cover any tuples by definition. Respectively, a slice which covers a session and a gap is logically equivalent to a slice which covers the session only.
5) The slicing logic solely depends on one session window - the one with the smallest gap. All session windows with larger gaps are compositions of the slices made for the session window with the smallest minimum gap (l_g).

Scotty utilizes the observations above and produces slices with respect to the session window with the smallest gap only. This guarantees a constant workload for stream slicing which is independent from the number concurrent session windows.

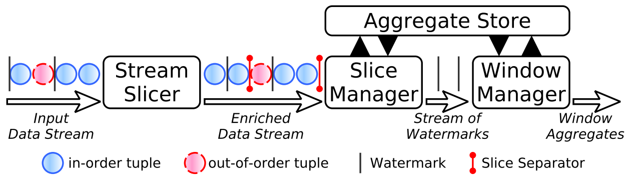


Fig. 3: Architecture Overview.

V. Architecture of Scotty

In Figure 3, we provide an overview of the architecture of Scotty, which consists of four main components: A *Stream Slicer*, a *Slice Manager*, an *Aggregate Store*, and a *Window Manager*. We now discuss the components in more detail.

Stream Slicer. The *Stream Slicer* splits the stream into non-overlapping slices for which we compute pre-aggregates. It receives the raw input stream, which consists of in-order tuples, out-of-order tuples, and watermarks. *Watermarks* are annotations embedded in the stream which propagate the progress in event time. Watermarks control how long we wait for out-of-order tuples before outputting a result [3], [5], [6]. When receiving a watermark with timestamp x, we output the aggregates of all windows which ended before x.

We determine the start of new slices based on the in-order tuples and enrich the stream with *Slice Separators* respectively. *Slice Separators* are annotations in the stream which mark the start of a new slice. The Stream Slicer forwards out-of-order tuples and watermarks without further processing, but preserves the order of its input stream. Note that we distinguish between *watermarks* and *Slice Separators*. Watermarks tell the Window Manager when to output window aggregates. Slice Separators tell the Slice Manager when to start a new slice.

Slice Manager. The Slice Manager performs three tasks: *(i)* It notifies the Aggregate Store about the start of a new slice when it receives a *Slice Separator* from the Stream Slicer. *(ii)* It appends in-order tuples to the latest slice in the Aggregate Store. *(iii)* As its most complex task, it updates past slices when out-of-order tuples arrive. This includes adding slices, fusing slices, changing the start and end timestamps of slices, and updating partial aggregates. We will discuss managing out-of-order tuples in more detail in the next section.

Aggregate Store. The Aggregate Store computes aggregates, stores partial aggregates for slices, and buffers tuples. We implement and evaluate different Aggregate Stores. Each store keeps at least one pre-aggregate per slice (lazy aggregation). In addition, stores can keep aggregates for combinations of multiple slices in an aggregate tree (eager aggregation).

Window Manager. The *Window Manager* combines pre-aggregates to final aggregates (results for windows).

VI. The Slice Manager

In this section, we provide a high-level architecture overview of the Slice Manager of Scotty.

A. Adding Tuples to Slices

In-order Tuples. Adding in-order tuples to slices has low computational costs because it does not require a lookup operation for finding the correct slice. In-order tuples always belong to the most recent slice which also makes processing costs independent from the number of windows and queries.

Out-of-order Tuples. The computation effort when processing a tuple out-of-order depends on the delay of the tuple. If a tuple has a small delay but still belongs to the most recent slice, we can add the tuple just like an in-order tuple. If the tuple has a larger delay, we lookup the slice which covers the event-time of the tuple and add the tuple to that slice. When processing session windows, an out-of-order tuple might fuse sessions or add a new session in the past. Thus, the Slice Manager possibly adds or changes slices in the past.

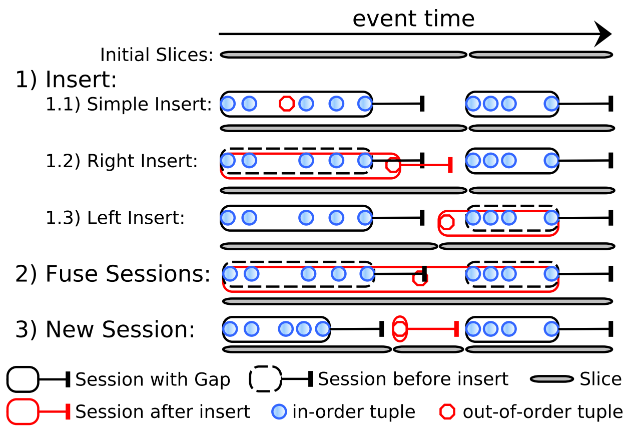


Fig. 4: Out-of-order Processing with Session Windows.

B. Changing Slices for Out-of-order Tuples

Tumbling and Sliding Windows. For tumbling and sliding windows, we know the times of all window edges a priori. Thus, the Stream Slicer always initiates the correct creation of all slices for these windows. The Slice Manager ensures to always retain correct slices for tumbling and sliding windows.

Session Windows. The Slice Manager operates based on the session window query with the smallest gap (cf. Section IV). An out-of-order tuple either belongs to an existing session, fuses sessions, or forms a new session. We show all cases in Figure 4. If an out-of-order tuple belongs to an existing session (Case 1.1) or extends a session at the session end (Case 1.2), we insert the tuple into the respective slice. Thereby, the start and end times of slices remain unchanged. If an out-of-order tuple extends a session at the session start (Case 1.3), we change session edges respectively and add the tuple afterwards.

An out-of-order tuple can also fuse two sessions. This is the case whenever the gap between sessions shrinks below the minimum session gap (Case 2). Fusing sessions also combines the slices of the sessions. Finally, an out-of-order tuple can form a new session on its own if its gap on both sides is larger than the minimum session gap (Case 3). In this case, we split a slice between sessions (i.e., within the gap). This is possible because gaps contain no data by definition.

VII. Evaluation

Techniques. We compare the throughput of four techniques which support out-of-order processing and session windows on Apache Flink (v1.3; Commit b0cd48d): *(i)* a lazy version of Scotty which stores slices in an ordered list, *(ii)* an eager version of Scotty which stores an aggregate tree on top of slices, *(iii)* Buckets as implemented in Flink [3], and *(iv)* eager aggregation without stream slicing (FlatFAT [9]).

Setup and Workload. We run experiments with 8 GB main memory and an Intel Core i5 processor with 2.4 GHz. We measure throughput exactly like the Yahoo Streaming Benchmark implementation for Apache Flink [14], [15]. We replay real-world sensor data recorded at a football match [16] and generate additional tuples based on the original data to simulate high ingestion rates. We add 5 gaps per minute which separate sessions. This is representative for the ball possession moving from one player to another. We base our queries on the workload of a live-visualisation dashboard [17].

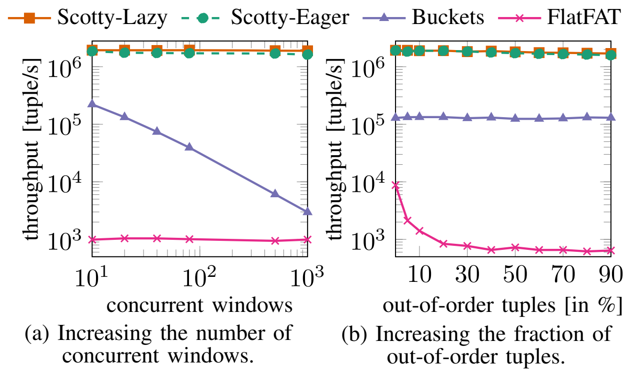
Concurrent Windows. In Figure 5a, we increase the number of concurrent windows. Window lengths are equally distributed from 1 to 20 seconds. This is representative for window aggregations which facilitate plotting line charts at different zoom levels in a dashboard [17], [18]. In addition, we run a session window query to separate ball possessions (l_g=1s). Note that the performance depends on the number of concurrent windows only. This makes tumbling and sliding windows exchangeable: 20 concurrent tumbling windows are equivalent to a single sliding window with l=20s and l_s=1s (again 20 concurrent windows). We simulate 20% out-of-order tuples with equally distributed delays between 0 and 2 seconds.

Scotty achieves an order of magnitude higher throughput than alternative techniques which do not use stream slicing. Moreover, Scotty scales to large numbers of concurrent windows with almost constant throughput. Scotty-Lazy has the highest throughput (1.9 Million tuples/s) because it uses stream slicing and does not compute an aggregate tree. The throughput remains constant when increasing the number of concurrent windows, because the per-tuple complexity remains constant: we assign each tuple to exactly one slice. Scotty-Eager achieves an 8% lower throughput than Scotty-Lazy, because out-of-order tuples cause updates in the aggregate tree. Buckets achieve orders of magnitude less throughput than Scotty and do not scale to large numbers of concurrent windows. With Buckets, we must assign each tuple to all buckets (i.e., windows) which cover the timestamp of the tuple. Thus, tuples belong to up to 1000 buckets causing 1000 redundant aggregation steps per tuple. In contrast, Scotty assigns tuples to exactly one slice. FlatFAT processes less than 2000 tuples/s, because out-of-order tuples require expensive leave inserts in the aggregate tree (aggregate updates & rebalancing).

Out-Of-Order Processing. In Figure 5b, we increase the fraction of out-of-order tuples and fix the number of concurrent windows to 20. All other settings remain as before. Scotty and Buckets process out-of-order tuples nearly as fast as in-order tuples. FlatFAT exhibits a throughput decay when we increase the fraction of out-of-order tuples.

Scotty processes out-of-order tuples efficiently because there are just a few hundred slices. This makes it fast to find the correct slices for tuples. The aggregate tree in Scotty-Eager stores slices only. It has few levels which leads to fast updates for past slices. Buckets have a constant throughput which is independent from out-of-order tuples. Our implementation stores buckets in a hash map which allows for assigning out-of-order tuples to buckets as fast as in-order tuples. FlatFAT exhibits a throughput decay when processing out-of-order tuples because it requires inserting past leave nodes in the aggregate tree. This causes a rebalancing of the tree and the recomputation of partial aggregates. Scotty-Eager seldom faces this issue because it stores slices instead of tuples in the aggregate tree. The majority of out-of-order tuples falls in an existing slice which prevents rebalancing.

Memory. Out-of-order processing with Scotty requires roughly twice as many slices than in-order processing with Cutty (cf. Section III). We refer to the Cutty paper [12] for a discussion of space complexities of all other techniques.



(a) Increasing the number of concurrent windows.
(b) Increasing the fraction of out-of-order tuples.

Fig. 5: Throughput Experiments - Scotty outperforms state-of-the art when processing out-of-order tuples and sessions.

VIII. Conclusion

Stream slicing is a key technique for efficient window discretization and aggregation. We present Scotty, an operator that applies stream slicing for arbitrary combinations of concurrent tumbling, sliding, and session windows. Scotty shares partial aggregates among all queries and window types and incorporates efficient processing of out-of-order tuples.

Scotty increases the throughput of window discretization and aggregation by an order of magnitude. Moreover, Scotty retains a high throughput for large numbers of queries and high fractions of out-of-order tuples.

Acknowledgements: This work was supported by the EU project Proteus (687691), DFG Stratosphere (606902), and the German Ministry for Education and Research as BBDC (01IS14013A) and Software Campus (01IS12056).

References

[1] J. Li *et al.*, "Out-of-order processing: a new architecture for high-performance stream systems," *PVLDB*, vol. 1, no. 1, pp. 274–288, 2008.
[2] T. Akidau, A. Balikov *et al.*, "Millwheel: fault-tolerant stream processing at internet scale," *PVLDB*, vol. 6, no. 11, pp. 1033–1044, 2013.
[3] P. Carbone *et al.*, "Apache Flink: Stream and batch processing in a single engine," *IEEE Data Engineering Bulletin*, vol. 36, no. 4, 2015.
[4] M. Zaharia *et al.*, "Apache Spark: A unified engine for big data processing," *Communications of the ACM*, vol. 59, no. 11, 2016.
[5] T. Akidau *et al.*, "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *PVLDB*, vol. 8, no. 12, pp. 1792–1803, 2015.
[6] Apache Beam, "An advanced unified programming model," https://beam.apache.org/ (project website).
[7] J. Traub, S. Breß, T. Rabl, A. Katsifodimos, and V. Markl, "Optimized on-demand data streaming from sensor nodes," *SoCC*, 2017.
[8] A. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates," in *VLDB*, 2004, pp. 336–347.
[9] K. Tangwongsan, M. Hirzel, S. Schneider *et al.*, "General incremental sliding-window aggregation," *PVLDB*, vol. 8, no. 7, pp. 702–713, 2015.
[10] S. Krishnamurthy, C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," in *SIGMOD*, 2006, pp. 623–634.
[11] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," in *ACM SIGMOD Record*, vol. 34, no. 1, 2005, pp. 39–44.
[12] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi *et al.*, "Cutty: Aggregate sharing for user-defined windows," in *CIKM*, 2016, pp. 1201–1210.
[13] A. Koliousis *et al.*, "SABER: Window-based hybrid stream processing for heterogeneous architectures," in *SIGMOD*, 2016, pp. 555–569.
[14] S. Chintapalli *et al.*, "Benchmarking streaming computation engines: Storm, flink and spark streaming," in *IPDPS*, 2016, pp. 1789–1792.
[15] K. Tzoumas, S. Ewen, and R. Metzger, "High-throughput, low-latency, and exactly-once stream processing with Apache Flink," 2015, https://data-artisans.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink.
[16] C. Mutschler, H. Ziekow, and Z. Jerzak, "The debs 2013 grand challenge," in *DEBS*, 2013, pp. 289–294.
[17] J. Traub, N. Steenbergen, P. Grulich, T. Rabl *et al.*, "I2: Interactive real-time visualization for streaming data." in *EDBT*, 2017, pp. 526–529.
[18] U. Jugel, Z. Jerzak, G. Hackenbroich *et al.*, "M4: a visualization-oriented time series data aggregation," *PVLDB*, vol. 7, no. 10, pp. 797–808, 2014.