Towards General Sliding Window Stream Analysis

Simon Hanssen, Kilian Holzinger*, Henning Stubbe*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: hanssen@in.tum.de, holzinger@net.in.tum.de, stubbe@net.in.tum.de

***Abstract*—Real time stream processing becomes more and more important as data arrives continuously and information needs to be based on the latest data. To query an unbounded data stream, *sliding window queries* are utilized. Naively evaluating these queries often causes a lot of redundant computations that unnecessarily lower the performance. Over the years different ideas have been proposed that capitalize on the nature of sliding window queries to reduce redundant calculations. This paper explains the basics of sliding window aggregation and then shows different techniques that emerged. These techniques are then evaluated and compared based on the performance studies conducted by the researchers and restrictions they impose on the kind of workloads they can handle.**

The techniques investigated are paned and paired Windows and a more general version of this called stream slicing. Additionally Slider and Reactive Aggregator which utilize Trees and DABA which is based on the TwoStacks algorithm are included. As of publication of this Paper, the general version of stream slicing fits best as efficient a drop-in replacement without posing any restrictions.

***Index Terms*—sliding window, stream processing, stream aggregation**

1. Introduction

In many real time applications, data continuously arrives in a stream and needs to be processed as such since users want whatever information they query to be based on the most recent data. Because of this, batch processing is no longer an option. The amount of data that arrives is potentially infinite and older data might get irrelevant over time. To query an unbounded data stream *window queries* are utilized, specifying a section of the stream to be evaluated. This is usually the most recent part and as new data arrives, the query should be reevaluated. The amount of incoming data needed to trigger an update is often way smaller than all data currently relevant, so when computing a new output, many calculations are redundant, leaving opportunities for optimization. Solutions presented to capitalize on these opportunities often pose restrictions on the type of queries or streams they can be used for. This paper provides the basics needed to understand the challenges those solutions faced and then presents selected techniques, how they achieved performance gains, how they compare to previous ideas and what restrictions they pose.

The rest of the paper is structured as follows: Section 2 explains the basics of *sliding window analysis*, Section 3 defines aspects that are important when discussing the solutions in Section 4. Section 5 gives an outlook on how sliding window analysis might further evolve. In Section 6 related and relevant work not dealt with in detail in this paper is mentioned and Section 7 concludes.

2. Background

To query information from a theoretically unbounded amount of data, one uses *windows*. This means, giving a cutoff to that data is considered relevant for the query. A popular example that will be referred to repeatedly in this paper is a trader at the stock market who has access to a stream containing all trades for a specific stock. The stream is made of tuples that in this example would contain all the information about the trade they represent e.g. the amount of shares traded, the time of the trade, the price etc. They want to know the average price their stock was traded for recently and now give a cutoff for trades to still be considered. They might ask for the average price of the last 5000 trades or the average price of all trades that happened during the last ten minutes of them issuing the query. With a query like this they would define a window with a size of 5000 trades/tuples or ten minutes. The size of the window is also known as *range*

So windows are a way to query data streams, but the trader probably will issue their query more than once, as they want to have live data all the time. Maybe they ask for the average of the last ten minutes and want it updated every ten seconds. For this, *sliding window queries* exist. Here, additionally to the size of the window one is interested in, one also has to provide a measure indicating when to update the given window based on the newest state of the stream. This update measure is also called *slide*. So our trader's query would have a range of ten minutes and a slide of ten seconds.

Figure 1 shows visualizes the process of how a sliding window query behaves. The query in this example has a range of nine and a slide of three tuples respectively. One tuple is represented by one green circle, and the tupsles shown are the end of the stream. There might be an arbitrary number of tuples preceding them but they are not relevant for the query, since they are not even part of the first window. When the query first is issued the latest nine tuples are inside the window and aggregated. As time passes more tuples arrive and as soon as thrre new tuples arrived, a new aggregate is calculated (third row). The first window actually does not exit or matter anymore here, but it is left drawn to visualise the process. The forth

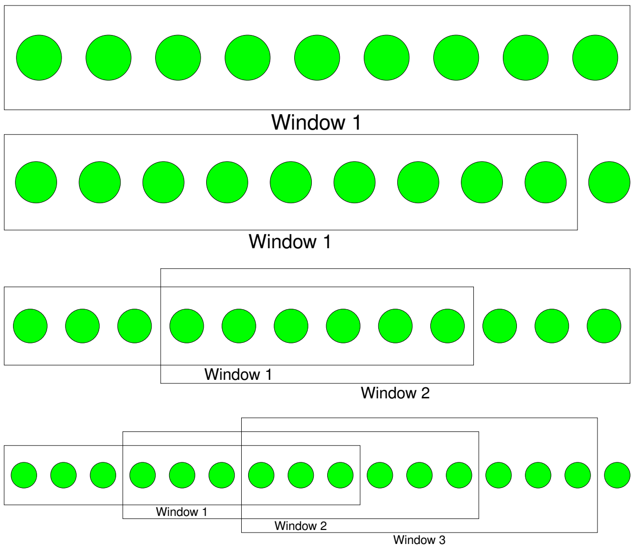

Figure 1: An example of a sliding window query with slide = 3 and range = 9 while tuples arrive over time

row shows the state after even more tuples arrived and the window is updated once more.

The naive way of answering this query would be evaluating it without any considerations about the nature of the problem: every ten seconds, find all trades in the stream that arrived at most ten minutes ago and average their price. While this might work fine for a not that frequently traded stock, it is rather inefficient in most cases. Considering that for each time the window *slides*, i.e., ten seconds pass and the new average is computed, most of the considered trades stay the same – the ones that happened in the last nine minutes and fifty seconds before the update – recomputing from scratch involves a lot of redundant calculations. Looking at Figure 1 the redundancy can be seen in the overlap of Window 1 and 2.

If we had computed the average of the last ten seconds and the other nine minutes and fifty seconds separately and then combined them, we could use that again with the new ten seconds of data that arrive. The next time ten seconds pass, we would have to recompute everything again though, since we cannot easily split off another ten second slice of the nine minute and fifty second chunk left over. This nevertheless shows the potential for optimization, later in the paper we will see how different techniques capitalize on that.

3. Workload Characteristics

Before investigating the different solutions that were developed, requirements different workloads pose to stream processing systems will be given here to allow the discussion of the solutions later.

3.1. Latency and Throughput

Naturally, memory usage and computation time are two important factors when it comes to a good query engine, but the time aspect needs to be viewed from two points here: latency and throughput. The former describes the time it takes a query to be answered once it is issued, the latter the amount of data that can be handled in a certain amount of time, which is directly dependent on how long it takes to process an incoming tuple.

3.2. Requirements to the Aggregation Functions

In general a sliding window query provides windows of tuples repeatedly and then performs some sort of operation on those tuples to gather an useful output from them. These operations can be seen as aggregation functions taking a set of tuples as input and giving the desired result as output. In the case of our stock trader that function would take the average price of all trades in the set given to it. When discussing aggregation functions in the context of stream processing, it is important to note that they are not seen as operations on sets of tuples. Instead they are split into three sub-functions. These sub-functions will be explained using *average()* as an example: (1) The first function takes one data tuple and converts it into a partial aggregate. In the case of *average()* this would take the input and convert it into a tuple *(sum,count)* with the value of the input as *sum* and 1 as *count*. (2) The second function takes two of those partial aggregates and combines them into one. In our case that would be adding up the *sum* and *count* parts of the partials respectively. (3) The last function takes one partial aggregate and converts it into a final output, here it would return *sum/count*, this would then yield the average of all tuples that were combined into the partial aggregate used as input for the third function.

When discussing aggregate functions, mainly the second function is of interest, so "*average()* is commutative" means that the combination of two partial aggregates is. This idea is crucial for many techniques presented later, since it allows partial (pre-)aggregation. This only works as long as the function is associative though, which is why all ideas presented later require this to be the case. Considering the lack of literature for cases where it is not given and authors noting that cases where it is not given are rare, associativity seems to be a reasonable assumption to make [1]–[3].

Invertibility and commutativity on the other hand are not given for every function of common interest; functions like *max* and *min* are not invertible. As an example for a not commutative aggregate Tangwongsan et al. name "collect-like" functions like *concatStrings* [2].

3.3. Requirements to the Stream

An important requirement several solutions have is that the tuples in the stream arrive in order. To show why this can be crucial, we can look at the stock trading example again. If the data about the trades does not arrive in the order that the trades took place and we also do not sort them on arrival, answering some queries can become difficult. If our stream is ordered, finding an entry that is older than ten minutes signals that up until that point, all entries about trades are within our range of interest. If we cannot guarantee that the stream is in order though, the entry might just have arrived a little late for some reason, there might be further entries of interest further behind. Maybe the internet connection in a trading hub went down, and now information about a trade that happened arrives fifteen minutes late. Under the assumption that the stream is in order, one would now stop looking further for relevant trades and get a skewed result. Assuming the stream is not in order one would struggle finding a point where it is guaranteed that no relevant trades can be found in the rest of the stream anymore.

3.4. Requirements to the Window

Different solutions have varying restrictions on the windows they can handle. A good way to classify those is presented by Traub et al. [4]. They divide windows into three categories:
(1) **Context free windows**: a window is context free, if its boundaries are already defined before it begins. The first stock trading example (average of the last ten minutes every ten seconds) is context free, all windows start ten seconds apart and span ten minutes, independent of the trades that happen.
(2) **Forward context free windows**: a window is forward context free, if we can tell whether a tuple marks the beginning or end of a window as soon as it arrives. An example: our stock trader wants to know the amount of trades that happened, separated each time the price of the stock passes the $100 mark. We cannot tell when the window will end until a trade happens that crosses the mark, but for every trade we already know of, it is clear whether or not it closed the preceding window.
(3) **Forward context aware windows**: for a forward context aware window, we cannot tell if a tuple marks the beginning of a window in the moment that it is processed. The second version of the stock trading example (average of the last 5000 trades every ten seconds) falls into this category. Until ten seconds have passed and the window slides, it is unclear which tuples are within the cutoff of the 5000 tuple range we specified. This is because each arriving tuple pushes the the cutoff further, and only after the ten seconds are over we can be sure that no tuples relevant for that update will arrive.

Another occurring aspect are concurrent windows. The same or different users might issue more than one query over the same data stream with different slide and range parameters, forcing parallel evaluation of each of those on their own if not taken into consideration. While this does not impose any restrictions if the system is aware of concurrent queries existing it can capitalize on that for further improvements.

4. Techniques

In this section different techniques for speeding up stream processing in comparison to from scratch recomputation will be presented.

4.1. Paned Windows

A first step to prevent having to recompute the whole aggregate every time the window slides was made with the introduction of *paned windows* by Jin Li et al. [5] They split the arriving data stream up into smaller sections - *panes* - of the same size, choosing the size as the greatest common divisor of the *slide* and *range* of the window. Using the first stock trading example this would mean creating panes with the size of ten seconds. For each of these panes the partial aggregate is computed, and when the next full aggregate is needed, only these partial aggregates need to be combined. This is beneficial in two ways: the partial aggregate for one pane only needs to be computed once and can be used again (remember: nine minutes and fifty seconds of the window stay the same each time it slides) and when the final aggregation is due, most of the work has already been done by computing the partial aggregates. For calculating and combining the partial aggregates the associativity of the function is crucial.

Once a pane has been aggregated, the tuples making it up are not required to stay in memory anymore, allowing for savings here, too. Taking our example again, instead of having to save the thousands of trades that happened in the last ten minutes, only 60 partial aggregates are needed.

This is the first instance of a technique that will later become known as *stream slicing* [4], [6]. Several other ideas presented later pick up on the idea and improve it, overcoming the restrictions that paned windows still have: (1) windows need to be *context free* so the size of the panes can be determined. (2) The stream needs to be in order as arriving tuples are inserted into the currently active pane.

While the performance evaluation conducted was not very thorough, in the cases that are relevant for actual applications, i.e., more than a few tuples per pane and panes per window, they find a speedup of 5 to 10 times when processing the query compared to recalculation. The greater the number of tuples per pane and panes per window, the better and as Krishnamurthy et al. noted it is reasonable to expect those numbers to be large enough for significant efficiency gains in real life scenarios [7].

4.2. Sharing Paired Windows and Fragments

Krishnamurthy et al. pick up the idea of paned windows and improve and extend it in the following ways: (1) They introduce *paired windows*, a way to slice a stream into fewer slices than when using paned windows, allowing for faster final aggregation. (2) They present a way of handling multiple sliding window queries over the same stream efficiently. They do this by slicing the stream so that the partial aggregates can be *shared*, i. e., used by all queries. While this leads to smaller and therefore more slices, it prevents redundant computations because in a non-sharing case every query would calculate the aggregates leading to the slices independently. (3) They introduce *shared Data Fragments*, a way to allow different selection predicates in concurrent queries while still taking advantage of *sharing* partial aggregates [7].

So if one trader only wants to consider trades where more than 100 shares were traded, and another one is only interested in those where the price was at least $4000 in total, their queries do not need to be handled separately anymore. While this idea is not discussed in further literature, it should be easy to include it in other stream slicing techniques like the one described in Section 4.6. This is because the splitting in shards happens after and independently of the slicing, so no matter what slicing technique is used one can split up the resulting slices afterwards.

While their analysis shows that paired windows only offer a minor improvement to paned ones, when sharing partial aggregates between concurrent windows with a regular workload, their implementation only needed 10% of the time to calculate aggregates.

4.3. Slider

Bhatotia et al. present an idea based on the idea of *incremental computing* [1]. They introduce *self-adjusting contraction trees*, a set of data structures that is able to deal with the requirements of inserting and removing tuples constantly as sliding windows demand. The leaf nodes of these trees are the output of applying the first aggregation sub-function to the tuples arriving from the stream and each inner node contains the partial aggregate resulting from combining its children. The value in the root node is then used to compute the final aggregate. They implemented a system they named *Slider* that uses these trees to efficiently evaluate sliding window queries.

The resulting speedup is only mediocre though; compared against recomputation from scratch their tests showed a speedup of up to four times. While they did not include a reference implementation of the solutions presented earlier in their tests and they use other metrics to evaluate, results presented in other papers indicate a better performance of those compared to Slider [1], [4], [5], [7], [8]. Furthermore Slider does not allow concurrent windows and one of their optimizations requires the aggregation function to be commutative, which is, as explained, not always the case. Slider accepts queries stated in *Pig-Latin* [9] allowing all types of windows to be utilized [1]. Like with the previous ideas, the stream needs to be ordered here, too.

4.4. Reactive Aggregator

Tangwongsan et al. present a very similar approach to Slider [3]. They create a binary tree on top of all tuples currently in the window. For this they created the *FlatFAT* data structure which stands for *flat fixed-size aggregator* and the *Reactive Aggregator (RA)* framework which uses FlatFAT to efficiently evaluate window queries. They show that their implementation needs at most $\mathcal{O}(m + m \log(n/m))$ partial aggregations for an update of size *m* to a window of size *n*. They also only compare their implementation against one that recomputes everything from scratch every time and use a *slide* of 1 for those comparisons, showing that, in that case, their solution becomes more than 10x faster than recomputation. They reason that this slide granularity is the worst for their implementation but the same goes for recomputation. Because of the similarity of the ideas, the real performance gains probably are comparable to Slider.

Traub et al. included an implementation of RA into performance studies they made in [4], [6]. It showed excellent latency but because each new arriving tuple forced the binary tree to be updated, the throughput suffered compared to slicing techniques. In comparison to Slider they do not require commutativity of aggregate functions in any way. They also allow tuples to be *evicted* out of order. While this can be useful in some cases, the way more important case of *inserting* tuples out of order was not addressed. Since RA only allows aggregation on all tuples currently handled, it does not support concurrent queries, but at the same time allows all types of windows to be used.

4.5. DABA

The *De-Amortized Banker's Aggregator* or *DABA* is an algorithm developed by Tangwongsan et al. [8] It guarantees worst case constant time for each window operation, without requiring the aggregation function to be invertible and by this allowing for consistently low latency. For an invertible aggregation function this is easy to achieve: adding a tuple just means aggregating it with what already has been accumulated and removing one uses the inverted function with the tuple to be removed. DABA uses a system of pointers and partial aggregates to allow this for non-invertible functions as well. DABA is based on an algorithm called *Two-Stacks* that only had amortized constant cost, causing occasional peaks in latency. The modifications made allow the work to be spread across all operations made.

In their performance study these effects show: while Two-Stacks has a lower average latency, the standard deviation of latencies of single operations was about 20 times higher compared to DABA. The overhead for spreading out the work between all operations results in slightly lower throughput for DABA. It still outperforms Reactive Aggregator regarding throughput while the only restriction compared to RA is that windows must be FIFO, but since it is unusual for tuples to be evicted out of order, this is usually the case.

4.6. General Stream Slicing

Traub et al. developed a generalization of stream slicing techniques that removes all restrictions given except the associativity of the aggregation function [4]. The basic idea stays the same: the stream is split up into smaller slices that do not need to be divided further because there are no window borders within them. Because of that they can be partially aggregated allowing working on slice basis instead of tuple basis.

They present a set of decision trees that based on properties of the windows, stream and aggregation function allow to decide, the best way to handle a query for a specific case. One example: for an in-order stream with forward context free windows, it is sufficient to only keep the partial aggregates of slices in memory, allowing tuples to be discarded after they were handled and by this reducing memory usage.

They implemented an eager and a lazy version; the eager one computes a tree based on Reactive Aggregator, but with the slices as leaves instead of tuples. By doing this it provides great latency while not suffering from as massive drawbacks in throughput as RA compared to the lazy version since the tree is way smaller.

The results from their experiments show great potential, general stream slicing matches or outperforms other techniques they compared it to even if the conditions for those were optimal.

5. The Future

While work in the past mainly was focused on finding new ideas that allow for more efficient stream processing than naively recomputing everything when needed and lowering restrictions those ideas posed, we now arrived at a state where a general approach has been found. It already incorporated different concepts from before but there is still room for improvement. As Tangwongsan et al. lately suggested, DABA could be used in general stream slicing if the stream is in-order [10]. So instead of improving each technique on its own or coming up with new ones, combining the different strengths of different approaches seems promising.

6. Related Work

Cutty [6] was another step towards general stream slicing that picked up on some aspects introduced by RA. Scotty [11] is an open-source implementation of general stream slicing. FiBA [2] uses finger trees to allow efficient handling of out-of-order tuples, showing excellent results on its own and being another attractive candidate to include in general stream slicing to improve the out-of-order case [11]. Zhang et al. analyzed different techniques that focus on utilizing hardware as well as possible for fast stream processing [12].

7. Conclusion

Optimizing sliding window aggregation poses different challenges that can prevent optimizations to reduce often occurring redundant computations from being used in general stream processing systems. With the introduction of paned windows came the idea of *stream slicing*, i.e., the realization that one can take partial aggregates of several tuples as a new smallest unit and still compute everything needed. This came with a rather strict set of restrictions though. Later this concept was generalized lifting those restrictions. This general version already performs well and on top of that offers opportunities to include other specialized optimizations like DABA, RA or FiBA to further increase the speed of the different cases that need to be handled. After a lot of ideas that rather were a proof of concept than really applicable, with this there now exists an efficient alternative to conventional, recomputing-based solutions that can be used as a drop-in replacement with potential to increase performance even further with more research done.

References

[1] P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar, "Slider: Incremental sliding-window computations for large-scale data analysis," *MPI-SWS, CITI/Universidade Nova de Lisboa, CMUTechnical Report: MPI-SWS-2012-004 September*, 2012.

[2] K. Tangwongsan, M. Hirzel, and S. Schneider, "Optimal and general out-of-order sliding-window aggregation," *Proc. VLDB Endow.*, vol. 12, no. 10, p. 1167–1180, Jun. 2019. [Online]. Available: https://doi.org/10.14778/3339490.3339499

[3] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 702–713, 2015.

[4] J. Traub, P. M. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl, "Efficient window aggregation with general stream slicing." in *EDBT*, 2019, pp. 97–108.

[5] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *Acm Sigmod Record*, vol. 34, no. 1, pp. 39–44, 2005.

[6] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl, "Cutty: Aggregate sharing for user-defined windows," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016, pp. 1201–1210.

[7] S. Krishnamurthy, C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, 2006, pp. 623–634.

[8] K. Tangwongsan, M. Hirzel, and S. Schneider, "Low-latency sliding-window aggregation in worst-case constant time," in *Proceedings of the 11th ACM international conference on distributed and event-based systems*, 2017, pp. 66–77.

[9] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," ser. SIGMOD '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 1099–1110. [Online]. Available: https://doi.org/10.1145/1376616.1376726

[10] K. Tangwongsan, M. Hirzel, and S. Schneider, "In-order sliding-window aggregation in worst-case constant time," *CoRR*, vol. abs/2009.13768, 2020. [Online]. Available: https://arxiv.org/abs/2009.13768

[11] J. Traub, P. M. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl, "Scotty: General and efficient open-source window aggregation for stream processing systems," *ACM Trans. Database Syst.*, vol. 46, no. 1, Mar. 2021. [Online]. Available: https://doi.org/10.1145/3433675

[12] S. Zhang, F. Zhang, Y. Wu, B. He, and P. Johns, "Hardware-conscious stream processing: A survey," *SIGMOD Rec.*, vol. 48, no. 4, p. 18–29, Feb. 2020. [Online]. Available: https://doi.org/10.1145/3385658.3385662