Stream Window Aggregation Semantics and Optimization

Carbone, Paris; Katsifodimos, Asterios; Haridi, Seif

DOI
10.1007/978-3-319-63962-8_154-1

Publication date
2018

Document Version
Accepted author manuscript

Published in
Encyclopedia of Big Data Technologies

Stream Window Aggregation Semantics and Optimisation

Paris Carbone, Asterios Katsifodimos and Seif Haridi

Definition

`Sliding windows` are bounded sets which evolve together with an infinite data stream of records. Each new sliding window evicts records from the previous one while introducing newly arrived records as well. Aggregations on windows typically derive some metric such as an average or a sum of a value in each window. The main challenge of applying aggregations to sliding windows is that a naive execution can lead to a high degree of redundant computation due to a large number of common records across different windows. Special optimization techniques have been developed throughout the years to tackle redundancy and make sliding window aggregation feasible and more efficient in large data streams.

Overview

Data stream processing has evolved significantly throughout the years, both in terms of system support and in programming model primitives. Alongside adopting common data-centric operators from relational algebra and functional programming such as select, join, flatmap, reduce etc., stream processors introduced a new set of primitives that are exclusive to the evolving nature of unbounded data. Stream windows are perhaps the most common and widely studied primitive in stream processing which is used to express computation on continuously evolving subsets out of a possibly never-ending stream. In essence, stream windows grant control on the granularity and the scope of stream aggregations.

Several early stream processing systems (e.g., TelegraphCQ (Chandrasekaran et al 2003), STREAM

(Arasu et al 2004)) provided support for windowing through a predefined set of primitives to construct time- and count-based sliding windows. For example, periodic tumbling and sliding windows were already standardized as early as the SQL-99 standard and studied thoroughly in the Continuous Query Language (CQL) (Arasu et al 2006) as well as the Stream Processing Language (SPL) (Hirzel et al 2009) among others. A *tumbling* window is a simple case of a stream window type, which is defined as a sequence of periodic consecutive sets of records in a stream with a fixed length that is termed *range*. For example, if we assume a stream of car speed events the following simple query in CQL would discretize that stream into windows of every 30sec-interval and compute the maximum speed per window:

```
SELECT max(speed)
from CarEvents
[RANGE 30 Seconds]
```

In principle, in tumbling windows each record can only belong to a single window. As a result, the evaluation of each window can be performed trivially by grouping records by the window they belong to, and executing each window aggregation independently. However, sliding windows add a challenging twist to the formula, namely the 'slide'. As an example consider the following sliding window query in SQL-99:

```
SELECT AVERAGE(speed)
FROM CarEvents
[WATTR timestamp
 RANGE 7 minute
 SLIDE 3 minute]
```

The *slide* represents 'when' or 'how often' a window has to be evaluated while including all records defined in its *range* in the computation of the average speed. In this sliding window example, there is an overlap of 5 minutes between each consecutive window and thus, a naive execution would result into redundant operations in its great majority. Beyond periodic windows, today's Apache open source systems such as Flink (Carbone et al 2015, 2017), Beam and Apex, provide support for more advanced, often user-defined, sliding window definitions such as session (Akidau et al 2015) or content-driven windows (Bifet and Gavalda 2007), among others.

Sliding windows have their own set of optimization techniques that aim to reduce the redundant computations caused by the intersection of events between neighboring windows. In this paper we categorize optimization techniques into *slicing*, *pre-aggregation* and *hybrid* and analyze them throughout the rest of this chapter.

Basic Concepts

There are many interpretations of window semantics, from simple range and count event-time windows (Arasu et al 2006) to policy-based (Hirzel et al 2009) and composite event-time windows with retraction for out-of-order processing (Li et al 2008b). The SECRET model (Botan et al 2010) aimed to subsume most of the windowing semantics proposed in academia and commercial systems. However, for the sake of brevity, a more simplified description is used here, with a heavy focus on window aggregation, based on the recent work on the Cutty aggregator (Carbone

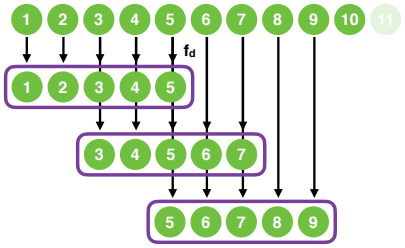

Fig. 1: Discretization of a count window of range 5 and slide 2.

et al 2016) and FlatFat (Tangwongsan et al 2015).

Stream Discretization

Data streams are unbounded sequences of records which are described by a given schema T. More formally, a stream $\bar{s} \in Seq(T)$ is a sequence out of all possible sequences $Seq(T)$ over T. Windows are finite subsequences reflecting intervals of a stream $\bar{s}$. An interval $s[a,b]$ is simply a set of records from index a to b over a stream $\bar{s}$ and the set of all possible intervals $Str(T) \subset Seq(T)$.

In their most general form, windows can be derived by *discretizing* an unbounded stream. A `Discretizer` transforms a stream $\bar{s} \in Seq(T)$ into a sequence $\overline{w} \in Seq(Str(T))$ of (possibly overlapping) windows. In the most system-agnostic manner, every possible discretization of a stream can be aided through a discretization function provided to a special *Discretize* or *Windowing* stream operator.

Definition 1. `Discretize` : $f_{disc} \quad \times Seq(T) \rightarrow Seq(Str(T))$

Figure 1 depicts a simple example of a discretization function f_d applied on a stream of elements which forms count windows with a 'range' of 5 records and a 'slide' of 2 records.

Aggregating Sliding Windows

Conceptually, in data stream programming models an `Aggregate` operator computes an aggregation on each window derived after a stream discretization, given an aggregation function f_a.

Definition 2. `Aggregate` : $(f_a \quad : Str(T) \rightarrow T') \times Seq(Str(T)) \rightarrow Seq(T')$

Examples of f_a is a `SUM` or `AVG` but also more complex aggregations can be executed in a window, such as building a machine learning model. Figure 2 shows how aggregates are formed on each consecutive window of a discretized stream. The main reason for optimizing the window aggregation process stems from two issues that can be observed in this example. First, a consecutive execution of the aggregation operation after discretization can be inefficient both in terms of space needed to log all elements of each window as windows can be very large in size. At the same time, the response time has to be minimized when iterating through all window contents in order to calculate an aggregate. Finally, and most importantly, as highlighted in Figure 2 sliding windows might involve a large amount of overlapping across consecutive windows. In this example there is an overlapping of three records between every two windows.

The first problem is solved by simply pipelining discretization with aggregation and thus, effectively providing a

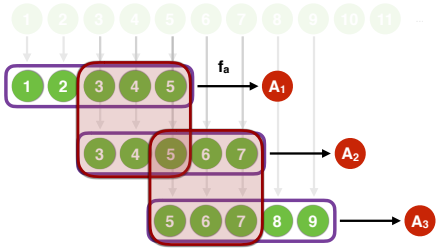

Fig. 2: Aggregation of a count window of range 5 and slide 2.

partial evaluation of window aggregates. Partial aggregation is described in section . The overlapping problem is more complex and its optimization techniques are further classified by window types into slicing, general pre-aggregation and hybrid techniques, covered thoroughly in section , and respectively.

Partial Window Aggregation

A complete partial window aggregation scheme has been proposed in both Flat-Fat (Tangwongsan et al 2015) and Cutty (Carbone et al 2016). According to that scheme, an aggregation f_a can be decomposed into partial aggregation operations in order to pipeline the process during discretization. A window aggregation function is therefore decomposed into, `lift` and `lower`, and a `combine` functions as such:
`lift` : $T \rightarrow A$ maps an element of a window to a partial aggregate of type A.
`combine` $\oplus : A \times A \rightarrow A$ combines two partial aggregates into a new partial aggregate (equivalent to a `reduce` function).
`lower` : $A \rightarrow T'$ maps a partial aggregate into an element in the type T' of output values.

The main and only requirement for partial aggregation is to have an *associative* `combine` function so that aggregation can be used to evaluate a full window aggregate in discrete steps in discrete steps (Arasu and Widom 2004; Krishnamurthy et al 2006; Tangwongsan et al 2015).

An example of partial aggregation of a window is depicted in Figure 3. The goal in this example is to partially compute the average value out of a set of records with values 1 to 5. The invariant is that only one partial aggregate is kept in memory (initially an empty aggregate). That aggregate is incrementally updated by each record that arrives in a window. To compute an average, two values have to be maintained in the aggregate type: a *sum* and a *count*. By using the `lift` function each record is first mapped into an aggregate type of its value and a count of 1. The `combine` function updates the partial aggregate with the new sum and count until all elements of a window have arrived. Then finally, the `lower` function transforms the aggregate into the window average, in this case this is 3.

Window Slicing

The amount of overlapping across sliding windows introduces additional space and computational complexity that partial aggregation itself cannot solve.

In the case of windows with predefined periodic characteristics such as a time or count slide, a family of optimization techniques are used to further decompose windows into non-overlapping partial aggregates which can be shared and combined to calculate full window

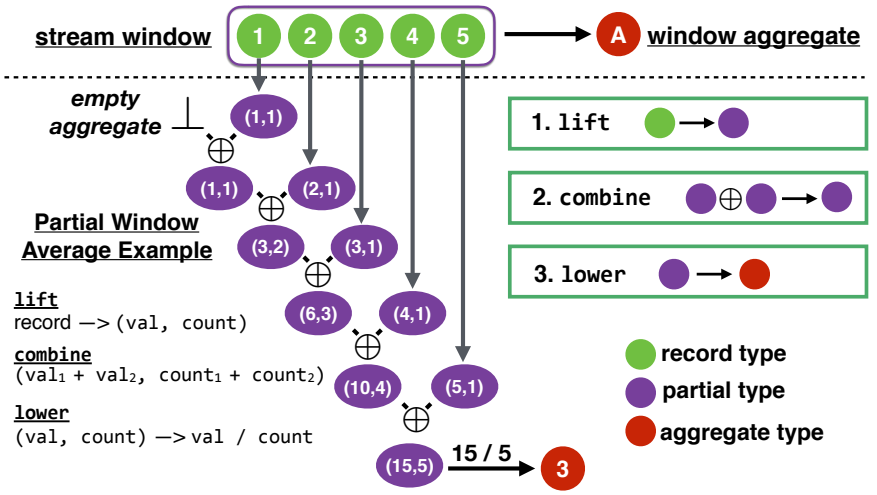

Fig. 3: Partial Aggregation Example for Window Average.

aggregates. This technique is typically name 'slicing' or 'bucketing'. The two most popular slicing techniques that are have also been deployed in production stream processing systems in the past are *panes* (Li et al 2005a) and *pairs* (Krishnamurthy et al 2006).

Panes

The main idea behind *Panes* is that if we have a periodic window query with a fixed slide and a range it is trivial to break down the aggregation process into partials with a constant size, equal to the greatest common denominator of the respective *range* and *slide*. For example if we have a sliding window with a range of 9 minutes and a slide of 6 minutes the stream would be sliced and pre-aggregated into buckets, each of which corresponds to 3 minutes of the ingested stream (greatest common denominator of 6 and 9).

Panes have been criticized (Krishnamurthy et al 2006) for their lack of general applicability, yielding unbalanced performance that depends highly on the combination of range and slide. For example, a window of range 10 seconds and a slide of 3 seconds would break down to slices of a single second, no longer exploiting the amount of non-overlapping segments in a stream (as depicted in Figure 4).

Pairs

The *pairs* technique (Krishnamurthy et al 2006) splits a stream into two alternating slices: $p_2 = range \mod slide$ and $p_1 = slide - s_2$. This technique utilizes better non-overlapping segments in a stream and seems to work well with most combinations of range and slide. Intuitively, pairs break slicing only when a stream window starts or ends. This is visualized in Figure 4

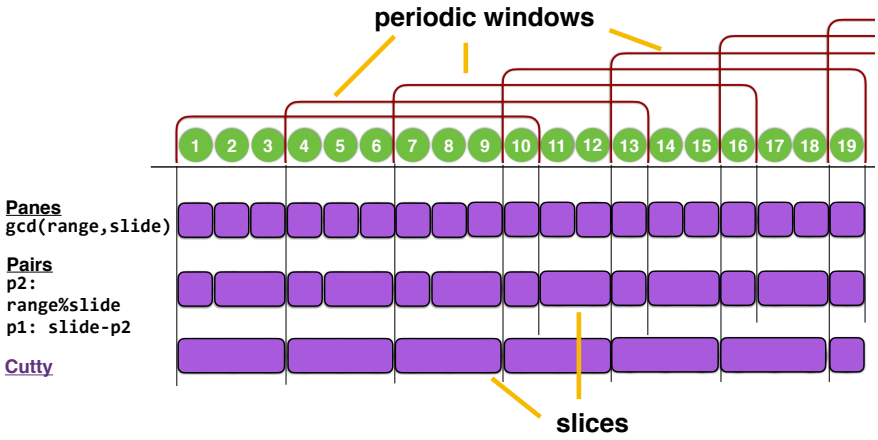

Fig. 4: Example of different slicing techniques

where it results into a lower number of slices for the same window compared to using *panes*. Contrary to *panes*, *pairs* has also been proposed for multi-query aggregation sharing where a large sequence of shared slices is decided at compilation time across shared sliding window aggregation queries in the same operator.

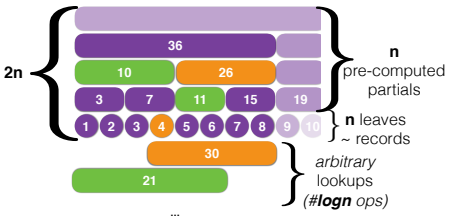

Fig. 5: Example of a general pre-aggregation tree

Slicing Limitations

While slicing offers the best known space and computational performance in sliding window aggregation its applications are limited to periodic window queries since this is the only type of windows for which their beginning and end are predefined. The *Cutty* technique (Carbone et al 2016) which is further analyzed in section avoids this strong assumption by letting user-defined functions signify during runtime when windows start or end. In the same work slicing is also combined with general pre-aggregation techniques (analyzed in section) in order to combine the strongest characteristics of both domains of window optimization.

General Pre-aggregation

The main incentive of general pre-aggregation techniques is to be able to allow for arbitrary segment lookups in a

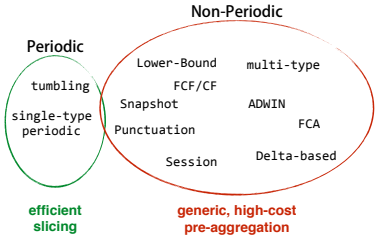

Fig. 6: Applicability of slicing and general pre-aggregation

stream (e.g., from external user queries) as depicted in Figure 5.

The earliest work on general pre-aggregation was presented in *B-int* (Arasu and Widom 2004) which pre-computes eagerly higher order partials on different segments of a stream. The application of B-int was meant to be fast aggregate retrieval for ad-hoc stream queries (i.e., using CQL), however, the applicability of general pre-aggregation makes such techniques convenient for aggregating continuous sliding windows without a known range or slide or other periodicity assumptions. The Reactive Aggregator by IBM Research (Tangwongsan et al 2015) exploits the properties of *B-Int* and introduces FlatFat: a fixed size circular heap binary tree of higher order partials that "slides" together with the records of the stream.

General Pre-Aggregation Limitations

General pre-aggregation offers fast retrievals of arbitrary windows on a stream at the cost of additional space and incremental update computation requirements. That is due to the fact that every time a new aggregate is added to the binary tree a sum of $log(N)$ additional partial aggregations need to be employed in order to update all higher order aggregates of the tree (given N: the number of active records/leaves in the tree). In summary, the runtime costs of employing eager-aggregation, which are also visualized in Figure 5 are the following:
space: $2N$ partials need to always be kept on heap to hold the full aggregation binary tree.
update/lookup: both update and full window lookup have $O(logN)$ computational complexity. In the case of updates the complexity is fixed ($logN$) against slicing which typically involves a single aggregation per record.

All these costs pose an interesting trade-off when eager aggregation is employed per-record in a data stream, which often results in more operations than a naive execution of each redundant window aggregation separately. As a result, it is likely that if general pre-aggregation is de-facto applied, its runtime cost would never be amortized across a full run of a continuous application.

Nevertheless, the power of general pre-aggregation lies at the observation that it can be generally applied to any type of windows, thus, covering a large space of non-periodic window types used within research and industrial applications, as depicted in Figure 6.

Cutty: A Hybrid Approach

Slicing and general pre-aggregation are orthogonal techniques that can be potentially combined to support a wider

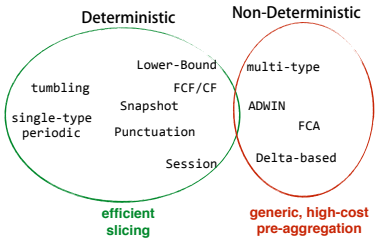

Fig. 7: Visualizing the expressive power of Deterministic Windows for Efficient Aggregation

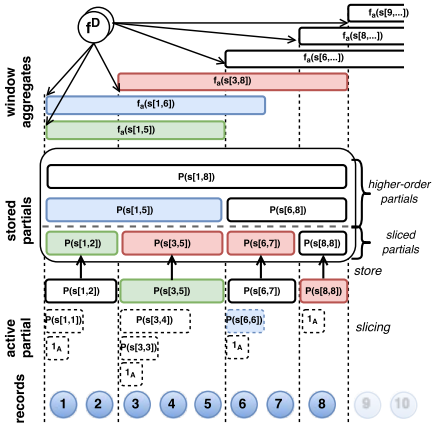

Fig. 8: An Overview Example of *Cutty*

variety of stream windows for aggregation. The *Cutty* aggregator (Carbone et al 2016) employs such a hybrid approach that can lead to efficient aggregation of a broader number of window types than simply periodic. The main observation behind its design is that there is an implicit class of windows (a superclass of periodic ones), termed *deterministic* which can be obtained by using the right core primitives in the programming model. Deterministic windows can be used to enable efficient shared aggregation without limiting window expressivity. Figure 7 shows the expressive power of deterministic windows, being able to provide optimal pre-aggregation to more than the limited periodic windows.

Deterministic Windows

The concept of deterministic windows stems from the observation that all it takes to achieve optimal slicing is not the *apriori* knowledge of the periodicity of windows (if any) but the *runtime* knowledge of where a new window starts. While partial aggregation is employed, as described in section a single partial result can be kept in active memory until we receive a record that marks the beginning of a new window. Conceptually, a new window start means that we will later need a partial aggregate (or slice) starting at that point to evaluate the window that started there.

Cutty proposes user-defined windowing through the use of a *discretization function* f_{disk}, defined as follows:

$$f_{disc} : T \rightarrow \langle W_{begin} : \mathbb{N}, W_{end} : \mathbb{N} \rangle$$

where for each record $r \in T$: *i)* W_{begin} is the number of windows beginning with r and *ii)* W_{end} the number of windows ending with r.

Overview of Cutty

Optimal slicing with deterministic windows minimizes but does not eliminate redundancy. As an example, consider the slices produced by Cutty during the example execution of Figure 4. A de-

tailed observation of the slices used per window reveals a level of redundancy that cannot be handled by slicing. For example, slices $s[4,6]$ and $s[7,9]$ would have to be combined together twice: once for computing $f_a(s[1,10])$ itself, and once for computing $f_a(s[4,13])$. Instead, if somehow the evaluation of $f_a(s[4,9])$ was stored, it would not be necessary to repeat that aggregation.

Cutty utilizes general pre-aggregation (FlatFat) in order to further reduce the cost of full window aggregate evaluation and compute higher order combinations of aggregates only once. This idea gives a new purpose to general pre-aggregation techniques and grants a low memory footprint since the space complexity of the aggregation tree is bounded by the number of active slices (which is equivalent to the minimum number of non-overlapping segments in a stream). A full example run of *Cutty* is visualized in Figure 8, showing both slicing of deterministic windows and general pre-aggregation and evaluation on stored partials. In the same example notice that the partial $P(s[6,7])$ is computed once and reused for both $f_a(s[1,5])$ and $f_a(s[1,6])$.

For non-deterministic windows, *Cutty* falls back to general pre-aggregation (simply using FlatFat) since slicing cannot be applied. Nevertheless, the generality and flexible (runtime specific) nature of this aggregation technique also enables the prospect of using it for applying operator sharing (Hirzel et al 2014) on data streams. A full complexity and performance analysis and comparison are provided in the original *Cutty* paper (Carbone et al 2016).

Further Works

Window aggregation is an interesting research topic and there are many relevant proposed ideas to the ones presented here. For example, Li et al. (Li et al 2005b, 2008a) classified window types by their evaluation context requirements, leaving the characterization of each class as an open research question. Performing certain types of aggregates in constant-time was recently proposed (Tangwongsan et al 2017). Deterministic functions in *Cutty* subsume all forward-context-free windows (no future records are required to know when a window starts), while non-deterministic discretization functions are forward-context-aware. Heuristic-based plan optimizers have also been proposed (e.g., *TriWeave*(Guirguis et al 2012)) to fine-tune the execution of periodic time queries dynamically using runtime metrics (i.e. input rate and shared aggregate rate).

Future Directions

Windowing semantics are becoming increasingly more complex and sophisticated as data stream processing is widely adopted. Aggregation techniques will have to follow the trends in windowing semantics and adapt to more dynamic, data-centric window types. One of the most prominent future directions in stream windowing is its standardization and encapsulation in stream SQL standards that are undergoing in open-source communities (e.g., the Calcite project and Google Dataflow (Akidau et al 2015)). However, no significant efforts have been made to

apply relational optimizations in stream windowing. Another future direction is to extend slicing capabilities beyond deterministic windows (if possible) and cover cases of fully data-driven windows without FIFO guarantees such as ADWIN (Bifet and Gavalda 2007). Finally, general pre-aggregation data structures have to employ the notion of out-of-orderness (Traub et al 2018). Currently, with existing out-of-the-box solution such as FlatFat it is not possible to retract already evaluated window aggregates, thus, making it impossible to use for systems like Beam and Flink with out-of-order logic.

Conclusions

Windows over streaming data continue to be the most central abstraction in data stream processing. Aggregation techniques aim to reduce the computational redundancy to the maximum extent possible for sliding windows. Most often, approaches to efficient aggregation are entangled with actual windowing semantics, such as assuming periodic queries to provide efficient pre-aggregation. Slicing techniques provide low memory footprint and generally good performance at the cost of limited applicability while general pre-aggregation techniques can be employed for any window lookup at the cost of high computational and memory footpring. Recent approaches aim for a hybrid solution by generalizing slicing further while combining data structures from general pre-aggregation. Sliding window aggregation remains a challenging topic today, and new challenges will arise with the adoption of richer and more complex windowing semantics and out-of-order streams.

References

Akidau T, Bradshaw R, Chambers C, Chernyak S, Fernández-Moctezuma RJ, Lax R, McVeety S, Mills D, Perry F, Schmidt E, et al (2015) The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. In: VLDB

Arasu A, Widom J (2004) Resource sharing in continuous sliding-window aggregates. In: VLDB

Arasu A, Babcock B, Babu S, Cieslewicz J, Datar M, Ito K, Motwani R, Srivastava U, Widom J (2004) Stream: The stanford data stream management system. Book chapter

Arasu A, Babu S, Widom J (2006) The CQL continuous query language: semantic foundations and query execution. VLDBJ

Bifet A, Gavalda R (2007) Learning from time-changing data with adaptive windowing. In: SDM, SIAM

Botan I, Derakhshan R, Dindar N, Haas L, Miller RJ, Tatbul N (2010) Secret: A model for analysis of the execution semantics of stream processing systems. In: VLDB

Carbone P, Katsifodimos A, Ewen S, Markl V, Haridi S, Tzoumas K (2015) Apache flink: Stream and batch processing in a single engine. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 36(4)

Carbone P, Traub J, Katsifodimos A, Haridi S, Markl V (2016) Cutty: Aggregate sharing for user-defined windows. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, ACM

Carbone P, Ewen S, Fóra G, Haridi S, Richter S, Tzoumas K (2017) State management in apache flink®: consistent stateful distributed stream processing. Proceedings of the VLDB Endowment 10(12):1718–1729

Chandrasekaran S, Cooper O, Deshpande A, Franklin MJ, Hellerstein JM, Hong W, Krishnamurthy S, Madden SR, Reiss F, Shah MA (2003) TelegraphCQ: continuous dataflow processing. In: Proceedings of the

2003 ACM SIGMOD international conference on Management of data, ACM, pp 668–668

Guirguis S, Sharaf MA, Chrysanthis PK, Labrinidis A (2012) Three-level processing of multiple aggregate continuous queries. In: IEEE ICDE

Hirzel M, Andrade H, Gedik B, Kumar V, Losa G, Nasgaard M, Soule R, Wu K (2009) SPL stream processing language specification. NewYork: IBMResearchDivisionTJ WatsonResearchCenter, IBM ResearchReport: RC24897 (W0911 044)

Hirzel M, Soulé R, Schneider S, Gedik B, Grimm R (2014) A catalog of stream processing optimizations. ACM Computing Surveys (CSUR) 46(4):46

Krishnamurthy S, Wu C, Franklin M (2006) On-the-fly sharing for streamed aggregation. In: AMC SIGMOD

Li J, Maier D, Tufte K, Papadimos V, Tucker PA (2005a) No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. ACM SIGMOD Record

Li J, Maier D, Tufte K, Papadimos V, Tucker PA (2005b) Semantics and evaluation techniques for window aggregates in data streams. In: ACM SIGMOD

Li J, Tufte K, Maier D, Papadimos V (2008a) Adaptwid: An adaptive, memory-efficient window aggregation implementation. IEEE Internet Computing

Li J, Tufte K, Shkapenyuk V, Papadimos V, Johnson T, Maier D (2008b) Out-of-order processing: a new architecture for high-performance stream systems. Proceedings of the VLDB Endowment 1(1):274–288

Tangwongsan K, Hirzel M, Schneider S, Wu KL (2015) General incremental sliding-window aggregation. In: VLDB

Tangwongsan K, Hirzel M, Schneider S (2017) Low-latency sliding-window aggregation in worst-case constant time. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, ACM, pp 66–77

Traub J, Grulich P, Rodriguez Cuellar A, Bress S, Katsifodimos A, Rable T, Markl V (2018) Scotty: Efficient window aggregation for out-of-order stream processing. In: Data Engineering (ICDE), 2012 IEEE 34th International Conference on, IEEE