Tutorial: The Role of Event-Time Order in Data Streaming Analysis

Vincenzo Gulisano
Chalmers University of Technology
Gothenburg, Sweden
vincenzo.gulisano@chalmers.se

Dimitris Palyvos-Giannas
Chalmers University of Technology
Gothenburg, Sweden
palyvos@chalmers.se

Bastian Havers
Chalmers University of Technology & Volvo Cars
Gothenburg, Sweden
havers@chalmers.se

Marina Papatriantafilou
Chalmers University of Technology
Gothenburg, Sweden
ptrianta@chalmers.se

ABSTRACT

The data streaming paradigm was introduced around the year 2000 to overcome the limitations of traditional store-then-process paradigms found in relational databases (DBs). Opposite to DBs' "*first-the-data-then-the-query*" approach, data streaming applications build on the "*first-the-query-then-the-data*" alternative. More concretely, data streaming applications do not rely on storage to initially persist data and later query it, but rather build on continuous single-pass analysis in which incoming streams of data are processed on the fly and result in continuous streams of outputs.

In contrast with traditional batch processing, data streaming applications require the user to reason about an additional dimension in the data: *event-time*. Numerous models have been proposed in the literature to reason about event-time, each with different guarantees and trade-offs. Since it is not always clear which of these models is appropriate for a particular application, this tutorial studies the relevant concepts and compares the available options. This study can be highly relevant for people working with data streaming applications, both researchers and industrial practitioners.

CCS CONCEPTS

• **Information systems → Stream management**; **Data streams**; **Online analytical processing engines**;

KEYWORDS

Data streaming, Stream Processing Engines, Event-time

ACM Reference Format:
Vincenzo Gulisano, Dimitris Palyvos-Giannas, Bastian Havers, and Marina Papatriantafilou. 2020. Tutorial: The Role of Event-Time Order in Data Streaming Analysis. In *The 14th ACM International Conference on Distributed and Event-based Systems (DEBS '20), July 13–17, 2020, Virtual Event, QC, Canada.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3401025.3404088

1 OVERVIEW

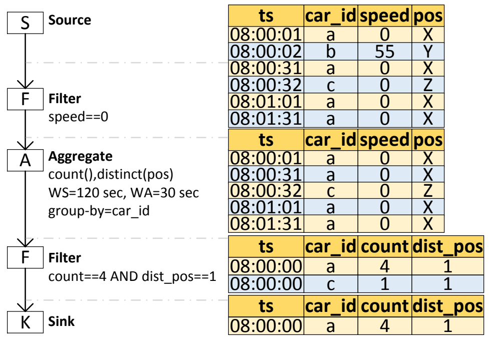
This paper overviews the topics that are covered by this tutorial. All the following sections, which are included in the tutorial and presented in the same order they appear in this publication, provide short summaries of the covered topics, with references and links to the relevant literature. In order for the tutorial to target a wide audience, we first cover basic concepts about the data streaming paradigm. All the material used for this tutorial can be found at https://github.com/vincenzo-gulisano/debs2020_tutorial_event_time.

2 BASIC CONCEPTS

Data streaming applications are used in many large and distributed systems [2, 6, 13, 18, 19, 25, 34] to process data coming in the form of unbounded *streams* of *tuples*. Tuples are comprised of a set of *attributes* according to their *schema*. Each tuple carries its *event-time* as one of its attributes, which we refer to as its *timestamp*. Each streaming application, called *continuous query* (or simply query), is a Directed Acyclic Graph (DAG) of streaming operators that transform the tuples delivered by a set of data sources, and produce new streams of tuples that are eventually delivered to end-users.

Streaming operators can be distinguished into two main classes: *stateless* and *stateful*. On the one hand, stateless operators process each input tuple individually and do not maintain a state that evolves according to the tuples being processed. *Filter* (used to discard or route tuples) and *map* (used to change the schema of tuples) are common stateless operators provided by both pioneer and modern Stream Processing Engines (SPEs) [1, 2, 8, 10, 12]. On the other hand, *stateful* operators produce results that depend on multiple input tuples. Because of the unbounded nature of data streams, stateful analysis is performed over portions of the data. Such portions, called *windows*, are usually based on the notion of event-time carried by tuples through their timestamps (in the literature, these windows are also referred to as *time-based* windows). *Aggregate* (used to combine windows of input tuples into one output tuple) and *join* (used to match tuples coming from different streams if such tuples carry event-times that are not far away more than a given window size) are common stateful operators that are also found both in pioneer and newer established SPEs [1, 2, 8, 10, 12].

Figure 1 shows a sample query composed of three operators. The query (originally presented in [29]) is based on the Linear Road



ts	car_id	speed	pos
08:00:01	a	0	X
08:00:02	b	55	Y
08:00:31	a	0	X
08:00:32	c	0	Z
08:01:01	a	0	X
08:01:31	a	0	X

ts	car_id	speed	pos
08:00:01	a	0	X
08:00:31	a	0	X
08:00:32	c	0	Z
08:01:01	a	0	X
08:01:31	a	0	X

ts	car_id	count	dist_pos
08:00:00	a	4	1
08:00:00	c	1	1

ts	car_id	count	dist_pos
08:00:00	a	4	1

Figure 1: Sample continuous query (figure from [29]) spotting stopped vehicles in the Linear Road benchmark [5].

benchmark [5], which simulates vehicular traffic on linear expressways that are monitored to detect accidents and compute variable tolls. Periodic position reports are generated every 30 seconds from each vehicle and are later processed to identify stopped vehicles, that is, vehicles for which the last four consecutive reports all have zero speed and share the same position. In the example, the aggregate's window size (WS) and window advance (WA) are set to 120 and 30 seconds, respectively. Because of this, the aggregate emits, for each vehicle, a result every 30 seconds, based on the data received in the last 120 seconds.

3 EVENT-TIME ORDERING AND STATEFUL STREAMING ANALYSIS

Since streams are, by definition, unbounded and processed continuously by streaming queries, recent events usually carry more important and fresher information than past ones [11]. Because of this, many contributions in the literature work under the assumption that streams are fed to streaming queries in timestamp-order by sources that rely on protocols that guarantee ordered delivery. When each stateful operator of a streaming query is fed a single stream of data that is timestamp-sorted, the operator can trivially determine if it has received all the data that belongs to a certain window covering a portion of event-time. Thus, the operator can correctly emit the result of the analysis over that data.

For instance, an aggregate operator counting the tuples observed in a stream over a certain time interval $[t_1, t_2)$ can (i) start counting tuples as soon as it receives an input tuple with a timestamp in the interval and (ii) correctly produce the resulting count as soon as it receives an input tuple carrying a timestamp falling outside the interval. However, if the input stream of the aggregate operator is not timestamp-sorted, it is not trivial to determine when a window will stop receiving further tuples. In our example, producing the result for a specific time interval $[t_1, t_2)$ as soon as an input tuple with a timestamp greater than or equal to t_2 is received, could lead to an imprecise result if later arrivals can still carry timestamps in that interval.

Similar challenges hold for the join operator, too. Since its window size determines the maximum distance (in event-time) for pairs of tuples from different streams, it is simple to decide when to discard a certain tuple based on the event-time of other incoming tuples, under the assumption of timestamp-sorted data. Such a decision is not trivial, nonetheless, if the assumption does not hold.

4 CAUSES OF EVENT-TIME DISORDER IN DATA STREAMS

Streams that are not sorted on their event-time can be observed in streaming queries because of two main reasons. On the one hand, data sources themselves can emit unsorted streams because of the communication medium they use to forward their data. This can be the case, for instance, when sources rely on the UDP protocol [7]. On the other hand, operators can be fed out-of-timestamp-order tuples when such tuples come from multiple streams. This can happen because of two main reasons, the first being the semantics of the operators. A join operator, for instance, is defined to match tuples from different streams. Even assuming each such stream is timestamp-sorted, asynchronous or distributed execution can result in arbitrary interleaving of tuples from distinct streams [12]. The second reason is that of multi-threaded asynchronous executions of continuous queries themselves, including distributed executions (inter-operator parallelism [14]), parallel executions (intra-operator parallelism [14]), as well as fault-tolerant execution, in which operators could be fed by multiple upstream replicas for availability as well as performance reasons [19].

5 ENFORCING TOTAL EVENT-TIME ORDERING ACROSS STREAMING QUERIES

When the stream generated by each data source or parallel instance of an operator is timestamp-sorted, total ordering can be enforced across the whole query. This has been initially discussed in pioneer SPEs such as Borealis [8] and StreamCloud [14]. In a nutshell, the ordering is performed (1) by merge-sorting timestamp-sorted streams before any operator with more than one input stream and (2) by relying on operators that produce timestamp-sorted output streams. Several approaches have been discussed in the literature. In [12, 14], the authors rely on a dedicated operator, the *Input Merger*, which maintains dedicated queues for each input stream and forwards each input tuple if the latter is the one with the smallest event-time among the earliest tuples stored in each queue. A tuple that fulfills this requirement is defined as *ready* in [15, 16, 27, 34]. A similar operator, *SUnion*, is proposed in [8]. However, that operator does not merge-sort each input tuple as soon as it is received, but instead performs the sorting periodically over batches of input tuples. Finally, authors in [15, 16, 28] propose a streaming-tailored data structure, *ScaleGate*, that also merges input streams deterministically. While this data structure merge-sorts each input tuple upon reception and forwards it as soon as it is ready (as done by the Input Mergers in [12, 14]), it opens up for key trade-offs about the use of shared data structures in SPEs [27]. As discussed in [15], when tuples coming from n timestamp-sorted input streams are merged relying on n individual queues (one for each input stream), threads delivering tuples incur an $O(1)$ cost to add a tuple in their respective queue, while the thread in charge of merge-sorting the tuples incurs an $O(n)$ cost to check whether each tuple is ready. On the other hand, when relying on a shared data structure, as ScaleGate's modified skip-list, each thread delivering a tuple can pay a higher cost — $O(\log n)$ in expectation — to add a tuple, thus easing the cost paid by the thread delivering ready tuples (to $O(1)$ in this case) and resulting in better scalability.

6 PROS AND CONS OF TOTAL ORDERING

Total event-time ordering offers interesting trade-offs in terms of efficiency and consistency. On the one hand, merge-sorting operations incur an extra computation expense and increased latency (since whether a tuple can be immediately processed once forwarded to an operator depends on the arrival of tuples from other streams, too) [8, 14–17]. On the other hand, timestamp ordering enables the benefits discussed in the following.

The most common benefit is that of *deterministic execution* [8, 12, 22, 29]. Since tuples' processing order does not depend on the tuple transmission latency from one operator to another nor on the interleaving of tuples fed to an operator with multiple input streams, deterministic merge-sorting of the input streams of an operator allows the operator's processing step to depend exclusively on the event-time carried by the tuples themselves [29][1].

Other benefits enabled by deterministically merging the streams fed to operators and queries include:

(1) synchronization among replicas for active standby fault tolerance [8, 19],
(2) eager purging of stale state from stateful operators, which helps in keeping the size of stateful operators' states at its minimum [14–16],
(3) consistent assignment of sequence numbers to tuples fed to *tuple-based* rather than *time-based* windows [16, 19] (tuple-based windows do not advance based on tuples' timestamps, which makes the synchronization of parallel threads operating on them more challenging),
(4) synchronization of threads analyzing in parallel data maintained in shared data structures, be it for performance and load balancing purposes [15, 16] or for efficient adaptive reconfigurations [27], and
(5) designing efficient backward provenance techniques [29].

It is important to notice, nonetheless, that total ordering can be a sufficient condition for deterministic execution but is not a necessary one, as we also explain in the following section.

7 RELAXATIONS FROM TOTAL EVENT-TIME ORDERING

While a portion of the streaming literature builds on the assumption of sorted streams, there is also a large body of related works discussing relaxations that do not require nor assume that all streams are timestamp-sorted. Such relaxations are discussed for the context in which data sources themselves cannot deliver timestamp-sorted streams or when trading off result accuracy for processing costs (e.g., latency) is beneficial in a given streaming application [9].

The most common technique is for streaming queries to rely on *watermarks* (also referred to as *punctuations*, *heartbeats*, or *boundary tuples* in the literature). For example, a *watermark* can be a special timestamp, which is forwarded through a stream, and which is smaller than or equal to the timestamp of all tuples coming after it (the watermark) in the stream. With watermarks, deterministic execution can still be supported for many streaming queries. For an aggregate to correctly produce deterministic results by relying on watermarks, the output of each window can safely be produced only after a watermark falling after such a window is received from each of the aggregate's input streams. For a join to safely discard a tuple that should no longer be matched with other incoming ones, such tuple can safely be discarded once its timestamp plus the join's window size is smaller than any of the latest watermarks received from each of the join's input streams. Many variations for different watermarking techniques are discussed in the literature [3, 8, 19, 24, 26, 32, 33]. Their costs can vary, depending on the correctness guarantees they can support and also depending on whether sources themselves or rather streaming operators are expected to emit them. Watermarks are supported by many of the existing SPEs [1, 4, 10].

In the literature, some works nonetheless state that both timestamp-sorted input streams, as well as the existence of data sources that are always able to result in reliable watermarks, are assumptions that do not hold in many real-world cases. Because of this, they instead:

(1) propose the leveraging of buffering mechanisms that delay the processing of each input tuple to wait for possible later arrivals and maximize the chances of feeding operators with timestamp-sorted streams [7, 20, 21, 23, 25, 34], or instead
(2) try to estimate how long each window of a stateful operator should be kept in memory in order to wait for possible late arrivals that still contribute to it [10, 30, 31].

None of these approaches can usually guarantee that the provided results are correct and deterministic, although some can prove the error they introduce is bounded according to how users tune the parameters of the proposed techniques. Because of this, solutions like [8] have also proposed to leverage *correction tuples*, which can be created to improve the accuracy of previously produced results.

A last type of model found in the literature is that of order-independent systems [23]. The motivation behind this model is that, no matter how late a tuple is, its effect should always be reflected on historical data since the latter could be later analyzed together with new incoming data. The idea is then to maintain partial results that can be consolidated in a continuous but lazy fashion, or simply when such partial results are needed or requested.

8 CONCLUSIONS

This tutorial covers numerous models that have been discussed in the data streaming literature to reason about event-time. The sections included in this publication provide short summaries of the topics the tutorial covers, with references and links to the relevant literature. Attendees of the tutorial will learn the essential role of event-time in distributed, parallel, fault-tolerant, and elastic data streaming. We believe this knowledge can benefit both researchers as well as practitioners. The topics presented in this tutorial can also be beneficial in complementing existing research threads, and better understanding some of the pros and cons of the mechanisms provided by state-of-the-art Stream Processing Engines. All the material used for this tutorial can be found at https://github.com/vincenzo-gulisano/debs2020_tutorial_event_time.

[1] A common assumption in many related works is that streaming operators themselves define deterministic functions, with no randomness in the semantics they enforce.

9 ACKNOWLEDGEMENTS

Work supported by VINNOVA, the Swedish Government Agency for Innovation Systems, projects "Onboard/Offboard Distributed Data Analytics (OODIDA)" and "Automotive Stream Processing and Distributed Analytics (AutoSPADA)" in the funding program FFI: Strategic Vehicle Research and Innovation (DNR 2016-04260 and 2019-05884, respectively); the Swedish Foundation for Strategic Research, project "Future factories in the cloud (FiC)" (grant GMT14-0032), and the Swedish Research Council (Vetenskapsrådet), project "HARE: Self-deploying and Adaptive Data Streaming Analytics in Fog Architectures" (grant 2016-03800).

REFERENCES

[1] Apache Heron. https://heron.incubator.apache.org/. Accessed: 2020-04-15.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. In *VLDB Journal*, volume 12, pages 120–139, aug 2003.

[3] T. Akidau, S. Chernyak, and R. Lax. Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing. Technical Report 4, 2018.

[4] Apache. Beam, 2020.

[5] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 480–491, Toronto, Canada, 2004. VLDB Endowment.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–16, 2002.

[7] S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems*, 29(3):545–590, sep 2004.

[8] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems*, 33(1):1–44, mar 2008.

[9] R. Cai, W. Wu, N. Huang, and L. Wu. Processing partially ordered requests in distributed stream processing systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10048 LNCS, pages 211–219. Springer Verlag, dec 2016.

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

[11] G. Cormode, F. Korn, and S. Tirthapura. Time-decaying aggregates in out-of-order streams. In *Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 89–98, 2008.

[12] V. Gulisano. *StreamCloud: an elastic parallel-distributed stream processing engine.* PhD thesis, 2012.

[13] V. Gulisano, M. Almgren, and M. Papatriantafilou. When smart cities meet big data. *Smart Cities*, 1(98):40, 2014.

[14] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente, and P. Valduriez. StreamCloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[15] V. Gulisano, Y. Nikolakopoulos, D. Cederman, M. Papatriantafilou, and P. Tsigas. Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types. *ACM Transactions on Parallel Computing*, 4(2):1–28, oct 2017.

[16] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. ScaleJoin: a Deterministic, Disjoint-Parallel and Skew-Resilient Stream Join. *IEEE Transactions on Big Data*, pages 1–1, nov 2016.

[17] V. Gulisano, A. V. Papadopoulos, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. Performance modeling of stream joins. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, pages 191–202, 2017.

[18] B. Havers, R. Duvignau, H. Najdataei, V. Gulisano, A. C. Koppisetty, and M. Papatriantafilou. Driven: a framework for efficient data retrieval and clustering in vehicular networks. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1850–1861. IEEE, 2019.

[19] J. H. Hwang, U. Çetintemel, and S. Zdonik. Fast and reliable stream processing over wide area networks. In *Proceedings - International Conference on Data Engineering*, pages 604–613, 2007.

[20] Y. Ji, J. Sun, A. Nica, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Quality-driven disorder handling for m-way sliding window stream joins. In *2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016*, pages 493–504. Institute of Electrical and Electronics Engineers Inc., jun 2016.

[21] Y. Ji, H. Zhou, Z. Jerzak, A. Nica, G. Hackenbroich, and C. Fetzer. Quality-driven processing of sliding window aggregates over out-of-order data streams. In *DEBS 2015 - Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 68–79, New York, New York, USA, jun 2015. Association for Computing Machinery, Inc.

[22] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch. Themis: Fairness in federated stream processing under overload. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 541–553, New York, NY, USA, 2016. ACM.

[23] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1081–1092, 2010.

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

[25] C. Mutschler and M. Philippsen. Distributed low-latency out-of-order event processing for high data rate sensor streams. In *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*, pages 1133–1144, 2013.

[26] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. SplitJoin: A Scalable, Low-Latency Stream Join Architecture with Adjustable Ordering Precision. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 493–505, USA, 2016. USENIX Association.

[27] H. Najdataei, Y. Nikolakopoulos, M. Papatriantafilou, P. Tsigas, and V. Gulisano. STRETCH: Scalable and elastic deterministic streaming analysis with virtual shared-nothing parallelism. In *DEBS 2019 - Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, pages 7–18, New York, NY, USA, jun 2019. Association for Computing Machinery, Inc.

[28] Y. Nikolakopoulos, M. Papatriantafilou, P. Brauer, M. Lundqvist, V. Gulisano, and P. Tsigas. Highly concurrent stream synchronization in many-core embedded systems. In *Proceedings of the Third ACM International Workshop on Many-core Embedded Systems*, pages 2–9, 2016.

[29] D. Palyvos-Giannas, V. Gulisano, and M. Papatriantafilou. GeneaLog: Fine-grained data streaming provenance in cyber-physical systems. *Parallel Computing*, 89:102552, nov 2019.

[30] S. Peros, S. Delbruel, S. Michiels, W. Joosen, and D. Hughes. Khronos: Middleware for simplified time management in CPS. In *DEBS 2019 - Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems*, pages 127–138, New York, NY, USA, jun 2019. Association for Computing Machinery, Inc.

[31] N. Rivetti, A. Gal, N. Zacheilas, and V. Kalogeraki. Probabilistic management of late arrival of events. In *DEBS 2018 - Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, pages 52–63, New York, New York, USA, jun 2018. Association for Computing Machinery, Inc.

[32] U. Srivastava and J. Widom. Flexible Time Management in Data Stream Systems. In *Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '04, pages 263–274, New York, NY, USA, 2004. Association for Computing Machinery.

[33] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, 2003.

[34] N. Zacheilas, V. Kalogeraki, Y. Nikolakopoulos, V. Gulisano, M. Papatriantafilou, and P. Tsigas. Maximizing determinism in stream processing under latency constraints. In *DEBS 2017 - Proceedings of the 11th ACM International Conference on Distributed Event-Based Systems*, pages 112–123, New York, New York, USA, jun 2017. Association for Computing Machinery, Inc.