Deterministic Time-Series Joins for Asynchronous High-Throughput Data Streams

Christoph Schranz
Salzburg Research
Salzburg, Austria
christoph.schranz@salzburgresearch.at

Peter Michael Jeremias
University of Salzburg
Department of Geoinformatics, Z_GIS
Salzburg, Austria
petermichael.jeremias@sbg.ac.at

***Abstract*—A variety of data stream problems that affect two or more data streams rely on joining them based on a common or similar timing attribute. With the advent of stream processing frameworks like Apache Spark and Apache Flink within the last years, processing of streamed data has become much easier. Repeated processing of relatively small data batches in so-called windows increases flexibility with respect to implementation and task distribution across multiple nodes. Using event times instead of ingestion times avoids, among other problems, incorrect joins. However, in this work we argue that batch-processing leads to a significant trade-off between increased computational complexity and latency of the resulting join pairs. A concept for time-series joins of streaming data is presented. This concept, which is built upon a resilient data stream framework, minimizes both the computational costs and latency times. It uses the guarantees associated with this underlying framework to join the data records deterministically according to event times instead of processing times. This work represents a work-in-progress paper, as detailed benchmarks are pending.**

***Index Terms*—data streaming, IoT, time-join, time-series join, high-throughput stream join, stream processing**

I. Introduction

Stream processing frameworks (SPF) as Apache Spark and Apache Flink have gained a lot of momentum within the last five years and have reached broad industrial acceptance. One of their most prominent features is their ability to distribute the processing of streaming data across multiple nodes. This is very effective for calculation-intensive tasks that can be parallelized. However, if a task requires only little computation or cannot be split, SPFs could be ineffective and lead to increased complexity and latency.

One common class of streaming tasks that fulfills these properties are *time-series joins (TSJ)*. In contrast to *temporal joins* and the better known *natural* and *equi-joins*, *TSJs* don't match two tuples based on the equivalence of their attributes, but on similarity patterns of their corresponding times. These join patterns are explained in more detail in section II-B. The term *TSJ* differs from *temporal join*, that is described by Silberschatz [1] as "a join, with the time of a tuple in the result being the intersection of the times of the tuples from which it is derived. If the times do not intersect, the tuple is removed from the result.".

This research has been funded by the Austrian Research Promotion Agency (FFG) and the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT).

The scope of this work is to show, that a class of real stream problems require a more flexible notion of join, where the pair of closest tuples of each stream is joined. To achieve our goal of optimizing the join of asynchronous data streams, (i) important pre-assumptions are defined and (ii) implications of the usage of state-of-the-art database systems frameworks on *TSJ* problems are depicted. Afterwards, (iii) a concept for a local stream buffering is presented, that is based on the discussed assumptions and a state-of-the-art data streaming technology. This approach optimizes the *TSJ* in terms of latency and computational costs with exactly-once processing guarantee. Finally, (iv) the paper is concluded and an outlook on further work is given.

It is neither the goal to solve the problem of asynchronous system times of data producers, nor to question the general performance of state-of-the-art database systems, as the considered field of application is narrowed down restrictively to *TSJs* of data streams.

II. Common stream joins

In this section an archetypal streaming problem is illustrated, that relies on *TSJs*. Although the specific problem requires the values of two data streams to be combined, *TSJs* can be applied on a broad class of problems that are based on joining two time series.

A. Data Streaming Problem

The data origins from CNC turning and milling centers and include, among other quantities, the rotational frequency n and torsional moment M of their spindles. The quantities are not streamed periodically with identical timestamps, but a record is published, whenever its relative value change exceeds a certain threshold. In order to continuously document the current effective power, it is required to compute $P_t = n_t \cdot M_t$ that is the product of the rotational frequency and torsional moment. However, as the factors are not published synchronously with identical timestamps, the resulting product has to be interpolated.

Therefore, a stream processing application should multiply factors neighbored in time, triggered by new receiving records and forward the resulting product to a data streaming framework. Hence, each incoming record leads to a join of zero, one or even multiple pairs. The required relational algebra is

$\rho_{schema(t)}\pi_{r.n \cdot s.M,(r.t+s.t)/2,\ldots}(r \bowtie_t s)$, where the operator $\bowtie_t$ performs a *time-series join (TSJ)*. The characteristics of this join type are explained at a later stage.

This scenario requires a low latency and high throughput, as there are dozens of machines, multiple quantities per machine and sampling rates of up to 1000 S/s for a single stream of quantities. Furthermore, some important machine states are only transmitted occasionally, therefore exactly-once-processing must be guaranteed.

Apart from this specific problem, it can be shown that a variety of data streaming problems rely on performing efficient *TSJs* of two streams. Merging two data streams or filtering one data stream based on a value of another requires the join of their time series in advance.

B. Time-Series Join of Data Streams

In Fig. 1, we can see two examples of how records of the streams r and s can occur, ordered by their event times. On the left side, the records r_m and s_n occur alternating; the join pairs are marked by red connections between them. On the right side, the records of stream s occur in a non alternating pattern, which leads to more than two join partners for the tuple r_{i+1}.

Fig. 1. Join pairs of two example time series.

The crucial part of time joins in data streams is to join the correct pairs with minimal latency, even if records of one data stream are delayed up to a certain threshold. One kind of join in which two ordered time series are joined, is called interpolated join within Vertica's proprietary SQL derivative [12], that is based on the ANSI SQL-99 standard. The next paragraph is dedicated to this SQL derivative even though a whole section is spent on state of the art, because it helps to explain what we understand by correct time-series joins.

Vertica's SQL derivative supports an additional interpolated join clause, e.g., `SELECT * FROM t LEFT OUTER JOIN t1 ON t.x INTERPOLATE PREVIOUS VALUE t1.y;`, that joins each tuple with the previous tuple of another table [12]. This leads to a deterministic characteristic, but may exclude pairs of tuples that are of interest. Consider the stream join shown on the right side in Fig. 1: The record s_j should be joined with both r_i and r_{i+1}, because $\Delta t_{s_j,r_{i+1}}$ could be smaller than $\Delta t_{s_j,r_i}$ and therefore the tuple $u_k = (s_j, r_{i+1})$ is a more appropriate join. This tuple, however, is not yielded using Vertica's `INTERPOLATE PREVIOUS VALUE`-syntax, as indicated by the red dashed lines. Therefore, it would need a `UNION` of Vertica's `INTERPOLATE PREVIOUS VALUE-` and `INTERPOLATE NEXT VALUE-` query, to ensure joining all desired records.

To summarize, the invariance of our *TSJ* is, that for any record s_j, there are two join partners r_i and r_{i+1} with the property: $r_i.t \leq s_j.t < r_{i+1}.t$, where the suffix "$.t$" denotes the record's event time. For practical reasons, a join candidate is discarded, if a given time threshold is exceeded by $\Delta t = |r_m.t - s_n.t|$. The same holds analogously for r_i. The considerations imply, that there will be $|u| \leq 2(|r| + |s|)$ join pairs for two continuous input streams r, s and one output stream u.

This type of join yields a higher number of pairs, as joins with crossed connection lines are possible, that are illustrated on the right-hand side of Fig. 1. In this figure, the tuples have the event time order r_i, s_j, s_{j+1}, r_{i+1}, therefore (r_i, s_{j+1}) and (s_j, r_{i+1}) are crossing, independently of their processing times.

For some applications, this high number of join pairs is not required. In order to reduce this high number of pairs, all potential crossing joins could optionally be forbidden, meaning that only tuples with the lowest Δt may join. Sparse joining reduces the number of join pairs from $|u| \leq 2(|r| + |s|)$ down to $|u| \leq |r| + |s|$, as each tuple is only joined with the prior record of the other stream. The case $<$ arises in the last inequality, if some records are discarded by exceeding a given time threshold.

C. Requirements

In summary, our deterministic *TSJ* framework must take the following features into account.

- Guarantee **correct joins of records based on their event times**, i.e, the timestamps the data is sensed, even if some are delayed up to a given threshold. This also includes neither to omit any join pair (unless a time threshold is exceeded), nor to accidentally join an incorrect pair.
- **Determinism**, i.e., replay join candidates in case the process crashes and restarts.
- **Low-latency**, emit resulting join tuples as soon as their correctness is approved, e.g., into another stream or pipeline.
- Minimize the required computational costs, allowing **high sample rates** and therefore **high throughput**.

To meet these requirements, the stream processing has to be built on a robust data streaming framework that supports these guarantees.

III. State of the Art

To begin with, the difference between data streaming and stream processing has to be clarified. As Wampler [4] pointed out, data streaming refers to the task of delivering data from a single producer to an arbitrary number of consumers. Some of its quality criteria are ordering and delivery guarantees, replay capabilities, data throughput, fault-tolerance, clustering capabilities, encryption and access control. In contrast to that, stream processing focuses on the computation of streamed data. In this section, multiple existing approaches to implement *TSJ* are presented.

A. Traditional SQL database systems

For reasons of completeness, it should be depicted how this problem can be solved with a traditional approach [1]. The state of the database for a given point in time is called snapshot. Assuming that tuples within two relations R and S contain their event time and a temporal interval in which the tuples are valid (e.g. one second). Then, all pairs of tuples $(r, s) : r \in R, s \in S$ should be joined, if their respective time intervals overlap. Not only are these joins not designed for streaming, but since theta joins are used with a temporal condition, they are computationally expensive. Vertica's ANSI SQL-99 derivative described in section II-B belongs to this class of traditional SQL database systems, as it is implemented for snapshots of relations.

B. Stream processing frameworks (SPF)

In contrast to traditional database systems, SPFs are specialized to process continuous streams of time-series data. The batch-processing approach is to connect to stateful data streaming frameworks and to process data using windowing, i.e., iterative computation of data sequences.

The length of these data frames can be based on time or on the number of received records. Moreover, there can be several window-triggering mechanisms. Therefore, windows are defined problem specific, e.g., some windows are created each five minutes with the same duration; others are count-based and created after each ten records and their window span involve the latest 100 records, meaning that these windows are overlapping.

For the given class of problem described in Section II, the following considerations have to be made: (i) A window for both streams is processed in order to find join pairs. (ii) The fix duration of the window has to be as high as the maximal accepted time delay of a record. (iii) The achieved latency is influenced by a window-triggering interval. The lower the interval, the higher the number of created windows and therefore the number of potential join candidates. These considerations reveal a weakness of window-based stream joins, as they lead to a trade-off between join latency and computational cost associated with increased storage utilization.

Assuming, there are two streams of data r and s, both with a sample rate of 1000 records per second. The highest accepted delay of a record, and consequently the window duration, is one second and the maximal additional latency should not exceed 50 ms. Therefore, a single join of two windows leads to $1000 \cdot 1000 = 10^6$ pairs that have to be filtered. To ensure a low latency, this join is performed each 50 ms, leading to $1s/50ms \cdot 10^6 = 2 \cdot 10^7$ pairs per second, although the resulting join rate can be upper bounded by $|\dot{u}| \leq 2(|\dot{r}| + |\dot{s}|) = 4000/s$.

This heuristic demonstrates the necessity to optimize *TSJs* for asynchronous data streams. However, leading SPFs such as Apache Spark, Apache Flink or ksqlDB do not take this into account. [3], [5], [6], [8] As SPFs strongly optimize their process before execution and therefore reduce the number of join candidates by filtering in advance, a direct comparison of multiple frameworks for the same task is of interest. Unfortunately, data throughput tests of *TSJ* were not found so far, although there are benchmarks for regular equi-joins between Apache Flink and Apache Spark. [6].

C. State stores

Some joins of streams are based on storing the latest states of streams in a key-value store which uses an index-key to look up the respective value [7]. E.g., Kafka Streams is able to perform stream-stream joins from version 0.10.2.0 on [9] [8]. Apache Samza and ksqlDB are using the RocksDB key-value store underneath [10], [11], but implement windowing as well.

A disadvantage of this approach is, that it increases the latency, because each new record in the stream leads to one or multiple look-ups in the key-value store [8]. Because both streams have a high sample rate, leading to a high number of OLTP (Online Transactional Processing) to update the respective states, the usage of key-value stores leads to serious performance lacks. Moreover, key-value stores are based on hashing [2] and are therefore not optimized for the efficient processing of range queries, that are required for *TSJs*.

IV. Local Stream Buffering Approach

As discussed in the previous section, the generality of state-of-the-art SPFs may solve a wide range of problems, but it goes hand in hand with a trade-off between increased latency and computational costs for *time-series join (TSJ)*. In this section, we present a concept for *TSJ* based on event time, that optimizes both data throughput and latency.

A. Assumptions

We assume the prerequisites, that (i) the records in each input data stream distinguish in their event times and (ii) are received in the correct chronological order. There are multiple data streaming frameworks, such as Apache Kafka and Amazon's Kinesis, that guarantee the correct order of records using unique keys. Furthermore, (iii) statefulness of a stream has to be ensured. From the consumer's side, this means to distinguish between the consumption of a record and the commitment that a record was received and there is no necessity to receive it again. Both named frameworks guarantee at-least-once delivery (Apache Kafka even exactly-once delivery) and provide replay capability [4], [11], [13].

B. Concept

The main data structure is a composition of two FIFO (first in, first out) queues, implemented as linked lists. These queues buffer all records of r and s, that could find a join partner in future records. While new records are respectively enqueued and possible join pairs are identified, tuples that cannot find any further join partners are dequeued.

In Fig. 2, six examples of records from the streams r and s are illustrated, whereby the records r_i, s_j are ordered by event time. An important insight is, that all possible states of the buffers that imply any *action*, involving those described in Fig. 1, can be reduced to one of these six cases (each three of them are symmetrical). An *action* is either the join of two tuples or the deletion of the least recent record from the buffer.

The red lines connecting r_i and s_j represent a join-pair. In case A, the record s_j is joined with r_i and r_{i+1}, as they are the previous respectively subsequent record in r based on their event time. The correctness of the joins in case B is approved by the guarantee that all records within the same stream are in order. This means, that the ingestion of r_{i+1} in B guarantees, that there won't be any record with a timestamp between those of r_i and r_{i+1}, formally: $\exists! r' : r_{i.t} < r'_t < r_{i+1.t}$. This holds analogously for D and E. Therefore, all pairs connected by a red line must be join partners.

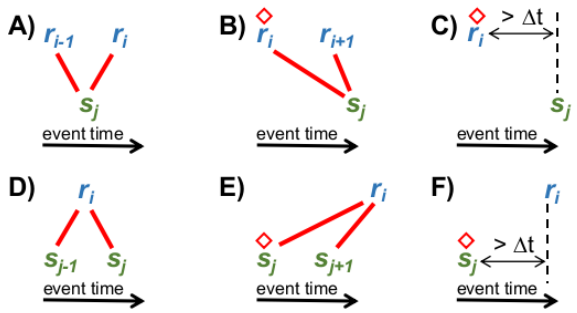

Fig. 2. Six cases that trigger a join of r_i and s_j.

The second type of *action* is a commit, represented by red diamonds. A commit is needed in case B (and E), where a record can't find any further join partner, as the more recent record r_{i+1} prevents the record r_i from joining with future s_k : $k > j$. This marked record is then committed as obsolete to the data streaming framework and is simultaneously removed from the buffer. This mechanism is required to ensure at-least-once processing, in case the join processor fails and restarts. Then, each record lost in the buffer is received again.

Lastly, in cases C and F an exceedance of a time threshold Δt leads to an omission. This timeout has two advantages: First, the additional latency of yielded joins is upper bounded by this timeout. Second, the stream buffer is protected from a memory overflow in the event, that values in one stream are stored for an extended period of time and no join partners are found in the other stream. Analog to inner, left, right and full outer joins, a preferred join behaviour for records that can't find any join partner can be considered.

V. Conclusion & Further Work

In this work, it was shown that a class of data streaming problems can benefit from efficient *time-series joins (TSJs)* with regard to low-latency, minimal computational costs and deterministic behavior even if some records may delay. We investigated traditional relational databases, stream processing frameworks and state stores and found that they are not optimized in terms of low-latency and high-throughput of time-series joins. Therefore, a concept was proposed, that is specialized in performing efficient *TSJs* of data streams and fits the identified functional requirements.

As SPFs clearly optimize queries to a high degree, benchmarks are required for a detailed comparison. Intensive tests are planned within the next months, that will not only cover the introduced problem, but also some applications for that SPFs can play their strengths when it comes to task distribution capabilities.

The current implementation of the proposed local stream buffer can be found in this Github repository [1]. In a follow-up paper the benchmarks of this solution are presented. In addition, a larger number of state-of-the-art approaches is considered and the field of application of the proposed work is outlined in detail.

Acknowledgment

This research has been funded by the Austrian Research Promotion Agency (FFG) and the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT), within the project IoT4CPS (Trustworthy IoT for CPS) (12/2017-11/2020).

References

[1] Abraham Silberschatz, Henry F. Korth, S. Sudarshan, "Database System Concepts," McGraw-Hill, 2010, 6th Edition, p. 1064.
[2] Abraham Silberschatz, Henry F. Korth, S. Sudarshan, "Database System Concepts," McGraw-Hill, 2010, 6th Edition, p. 799.
[3] Spark Blog post, "Stream-stream joins," https://databricks.com/blog/2018/03/13/introducing-stream-stream-joins-in-apache-spark-2-3.html, 2018.
[4] D. Wampler, "Fast Data Architectures," O'Reilly Media, 2 edition, 2016.
[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," IEEE Data Eng. Bull., 38:28–38, 2015.
[6] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, V. Markl, "Benchmarking distributed stream data processing systems," IEEE 34th International Conference on Data Engineering, 24-06-2019.
[7] R. Cattell, "Scalable sql and nosql data stores," ACM SIGMOD Record, p. 14-15, 2011.
[8] Confluent, "Distributed, Real-time Joins and Aggregations on User Activity Events using Kafka Streams," blog post, https://www.confluent.io/blog/distributed-real-time-joins-and-aggregations-on-user-activity-events-using-kafka-streams/, 2016.
[9] Confluence Wiki for Kafka, "Kafka Streams Join Semantics," https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Join+Semantics, 2018.
[10] Confluent, "Introducing ksqlDB," blog post, https://www.confluent.io/blog/intro-to-ksqldb-sql-database-streaming/, 2019, last updated: 13.03.2020.
[11] M. Kleppmann, J. Kreps, "Kafka, Samza and the Unix Philosophy of Distributed Data," IBulletin of the Technical Committee on Data Engineering, p. 10, 2015.
[12] HP Vertica Analytic Database, "SQL Reference Manual,", online resource https://usermanual.wiki/Pdf/HPVertica70xSQLReferenceManual.195394116/view, p. 39, 95-99, 24-02-2014.
[13] AWS Kinesis developer guide, "Handling Duplicate Records,", online resource https://docs.aws.amazon.com/streams/latest/dev/kinesis-record-processor-duplicates.html, 6.5.2019.

[1] Github, StatefulStreamProcessor: https://github.com/ChristophSchranz/StatefulStreamProcessor/blob/master/05_LocalStreamBuffer/local_stream_buffer.py