



Scalable and Reliable Multi-dimensional Sensor Data Aggregation in Data Streaming Architectures

Sören Henning¹ · Wilhelm Hasselbring¹

Received: 10 February 2020 / Revised: 2 April 2020 / Accepted: 27 August 2020 / Published online: 6 October 2020
© The Author(s) 2020

Abstract

Ever-increasing amounts of data and requirements to process them in real time lead to more and more analytics platforms and software systems designed according to the concept of stream processing. A common area of application is processing continuous data streams from sensors, for example, IoT devices or performance monitoring tools. In addition to analyzing pure sensor data, analyses of data for entire groups of sensors often need to be performed. Therefore, data streams of the individual sensors have to be continuously aggregated to a data stream for a group. Motivated by a real-world application scenario of analyzing power consumption in Industry 4.0 environments, we propose that such a stream aggregation approach has to allow for aggregating sensors in hierarchical groups, support multiple such hierarchies in parallel, provide reconfiguration at runtime, and preserve the scalability and reliability qualities of stream processing techniques. We propose a stream processing architecture fulfilling these requirements, which can be integrated into existing big data architectures. As all state-of-the-art stream processing frameworks have to handle a trade-off between latency, resource-efficiency, and correctness, our proposed architecture can be configured for low latency and resource-efficient computation or for always ensuring correct results. To assist adopters in choosing appropriate configuration options, we provide an experimental comparison. We present a pilot implementation of our proposed architecture and show how it is used in industry. Furthermore, in experimental evaluations we show that our solution scales linearly with the amount of sensors and provides adequate reliability in the presence of faults.

Keywords Big data · Stream processing · Stream aggregation · IoT sensor data

Introduction

Stream processing [1, 2] has evolved as a paradigm to process and analyze continuous streams of data, for example, coming from IoT sensors. The rapid development of stream processing engines [3] over the last years has paved the way for applications that process data exclusively

online, i.e., as soon as it is recorded. Whereas a couple of years ago Lambda architectures were the de facto standard for analytics platforms, currently more and more platforms follow the Kappa architecture pattern, where data is exclusively processed online [4]. Further, entire software system architectures [5] follow patterns such as asynchronously communicating microservices [6] and event sourcing [7], which require data to be available as continuous streams instead of actively polled from databases.

When considering continuous streams of measurement data, for example, from physical IoT sensors or software performance metrics, often an aggregation of multiple such streams is required. Whereas in the traditional approach of first writing all measurements to a (relational) database and then querying this database, this is a well-known task, performing such an aggregation on continuous data streams raises challenges. This is in particular true when requirements for scalability and reliability have to be considered.

This article belongs to the Topical Collection: *Data-Enabled Discovery for Industrial Cyber-Physical Systems*
Guest Editor: Raju Gottumukkala

✉ Sören Henning
soeren.henning@email.uni-kiel.de

Wilhelm Hasselbring
hasselbring@email.uni-kiel.de

¹ Software Engineering Group, Kiel University, 24098 Kiel, Germany

In this paper, we contribute to the seminal work on stream processing by presenting and evaluating dataflow architectures to aggregate data of multiple streams in real time. This paper is an extended version of a workshop paper presented at the 2019 IEEE International Conference on Big Data [8]. We extend it by proposing two architecture variations (“[Dataflow Architecture Variations](#)”), providing now support for out-of-order record, although at the cost of increased latency and resource demand. Further, we experimentally compare our architecture variations in terms of their performance (“[Experimental Comparison of Architectures](#)”) to assist adopters in choosing an appropriate option.

The remaining paper is structured as follows: “[Motivating Example](#)” motivates the demand for our aggregation approach by an example for IoT sensor data streams. “[Requirements for Stream Aggregation](#)” derives essential requirements for an aggregation approach. “[The Dual Streaming Model](#)” provides a brief summary of the fundamental model used by our stream processing architecture for aggregating data streams, which is presented in “[Basic Dataflow Architecture](#).” “[Pilot Implementation for IoT Sensor Data](#)” shows how we implement this architecture in an industrial IoT monitoring platform. “[Experimental Scalability and Reliability Evaluation](#)” evaluates our proposed architecture in terms of scalability and reliability. “[Dataflow Architecture Variations](#)” presents two architecture variations of our basic architecture from “[Basic Dataflow Architecture](#),” supporting out-of-order records. “[Experimental Comparison of Architectures](#)” experimentally compares all three architecture variations in terms of their performance. “[Industrial Case Study](#)” describes how the proposed stream processing architecture is used in an industrial setting to aggregate power consumption data. Finally, “[Related Work](#)” discusses related work and “[Conclusions](#)” concludes this paper.

Motivating Example

Operators of industrial production environments have high interest in getting detailed insights into the resource usage of machines and production processes. Those insights may reveal optimization potential, provide reporting for stakeholders, and can be used for predictive maintenance. Today’s industrial production environments operate a variety of network-accessible measuring instruments. Such instruments (sensors) continuously measure, inter alia, the resource usage of individual machines, for example, their electrical power consumption. They may publish these metrics via a messaging system, allowing real-time analytics systems to collect, process, store, and visualize those data.

In particular, but not exclusively in the case of electrical power consumption, it is not only producing machinery that uses resources but also other company areas such as IT infrastructure, employee offices, or building technology. This leads to the situation that the amount of resource consuming devices is often immense, which makes it difficult to assess. Therefore, metrics for groups of machines in addition to consumption metrics of the individual machines are often required. Consequently, the data streams of the individual sensors have to be continuously aggregated. Referring to Fig. 1, which exemplary shows a production environment comprising various machines and devices, operators may require to answer questions in real time such as the following: What is the resource usage of a certain machine type (e.g., the total power consumption of all turning shops)? What is the resource usage by business unit (e.g., the total power consumption of producing machinery)? What is resource usage by physically collocated machines (e.g., the total power consumption per building or shop floor)? What is the overall company-wide resource usage? Further, for machines featuring multiple independent power supplies (e.g., for redundancy), already obtaining consumption data for single machines requires to aggregate data streams of their individual power supplies.

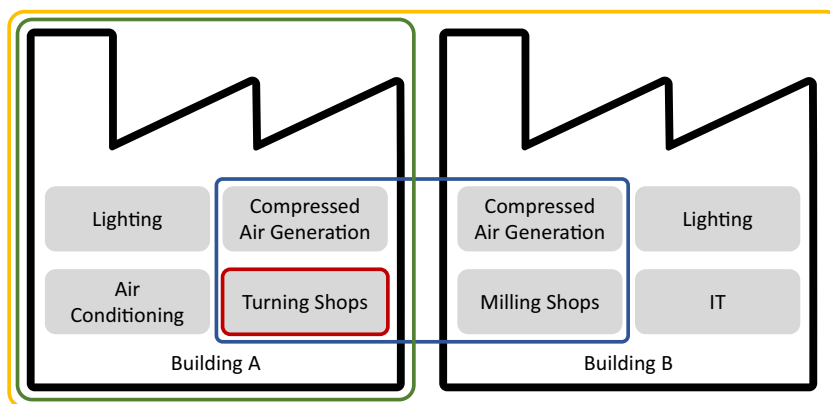
Requirements for Stream Aggregation

Even though we motivated the need for real-time aggregation of data streams by resource data recorded by IoT devices, similar kinds of data aggregations are required by several other types of continuous sensor data streams. Continuing our motivating example, we derive the following requirements for a data stream aggregation architecture:

Multi-layer Aggregation Measurements of sensors are aggregated to groups of sensors. These groups can again be aggregated to larger groups and so forth. In the previous example, these groups could first be groups of individual machines of the same type, then groups of machines fulfilling the same function, then all machines used by the same production step and, finally, all machines in the entire production environment.

Multi-hierarchy Aggregation In addition to a single hierarchy as described above, it is likely that there is a demand for supporting multiple such hierarchies. Referring to the previous example, besides a hierarchy based on the purpose of machines, one may also need a hierarchy which represents the physical location. For example, in a first step machines in the same shop floor are grouped and then all shop floors in a certain building are grouped.

Fig. 1 Schematic illustration of a manufacturing company operating two buildings and a wide range of power-consuming machinery and infrastructure. Operators may be, for instance, interested in the total power consumption of all turning shops (red), directly required for production processes (blue), of a certain building (green), or of the entire company (yellow)



Hierarchy Reconfiguration at Runtime Stream processing applications are often characterized by demands for high availability. Therefore, we aim for an approach that allows to modify or extend the previously described hierarchies at runtime to prevent reconfigurations causing downtimes.

Preserving Scalability and Reliability Stream processing is usually used for large volume of data where the load has to be handled by multiple CPU cores or even multiple computing nodes. Therefore, an approach that performs aggregations on these streams has to preserve scalability and reliability properties to not become the overall architecture’s bottleneck.

The Dual Streaming Model

The *dual streaming model* [9] is the foundation for the stream processing architectures described in this paper. It is a model to define the semantics of a stream processing architecture. It adopts the notion of data streams and streaming operators from other established stream processing models [10–12].

A *data stream* is an append-only sequence of immutable records, where records are key-value pairs augmented by a timestamp. Key-value pairs allow for data parallelization as records with different keys can be processed in parallel. This is the fundamental idea of building highly scalable stream processing applications.

Streaming operators are functions applied to each record of an input stream, whose results are appended to an output stream. The number of output records may be zero, one, or more than one depending on the type of operator. Usually, operators are distinguished as stateless or stateful. Stateless operators produce an output solely based on the currently processed input record, whereas stateful operators may also take previous input records and computations into account. Typical examples for stateless operations are *filter*, *map*, *flatMap*, or *merge*, whereas *aggregate* or *join* are common

examples for stateful operations. Successively connecting operator output streams with other operators allows to define complex stream processing topology architectures, for example, to build big data analytics applications.

The dual streaming model extends these models by considering the result of streaming operators as successive updates to a table. These updates may be materialized into a versioned table or represented as a stream of insert, update, and delete events, inducing a duality of streams and tables. Sax et al. [9] present a reference implementation of the dual streaming model called Kafka Streams, a stream processing framework build on top of the distributed messaging system Apache Kafka [13].

Basic Dataflow Architecture

In this section, we present a basic dataflow architecture for hierarchically aggregating streams of data. We apply the dual streaming model to model the topology of consecutive operations on the streaming data, which are required for aggregating sensor data. This model forces an unidirectional and side effect-free description of the data flow and, thus, allows the scalability and reliability facilitated by the model to be exploited. The dataflow architecture described in this way can be implemented as an encapsulated component, e.g., as a microservice, which can be integrated into existing software systems. Since in some cases the dual streaming model abstracts too many details to comprehensibly explain our dataflow architecture, we use some architectural elements which are only present in their reference implementation, but not in the model. Figure 2 visualizes our proposed architecture. The individual processing steps, realized as streaming operators, are described in the following.

1. Data Sources Our proposed architecture requires two data sources. The first one is an input stream of measurements, keyed by a sensor identifier. This is the sensor data stream

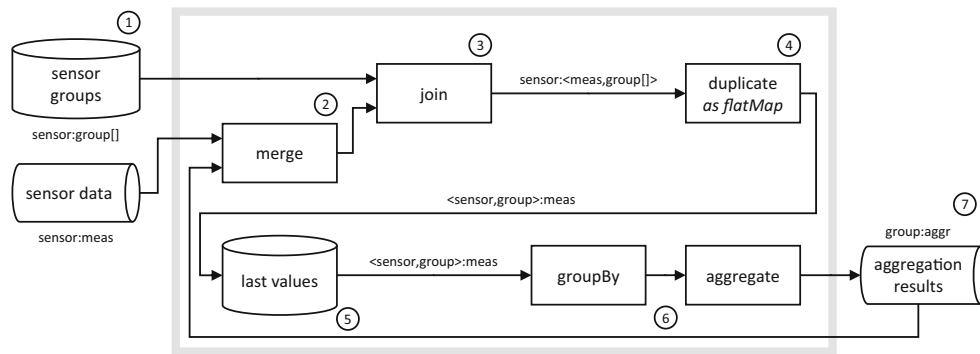


Fig. 2 Topology of our basic stream processing architecture. Vertical cylinders represent tables whereas horizontal cylinders represent data streams. Streaming operators are represented by rectangular boxes, which are connected to other operators, tables, or streams by directed arrows. Annotations at connections, tables, and streams indicate the corresponding type of contained or transmitted data, where key and value type are separated by a colon. *sensor* represents a unique identifier for a sensor and *group* such an identifier for a sensor group,

as it comes, for example, from IoT devices or performance monitoring tools. Measurements can be numeric values or more complex data structures. The second data source is a table, mapping sensors or sensor groups to the sensor groups containing them. A table entry consists of a key, which is the identifier of a sensor or group and a value, which is a set of all groups, this sensor (group) is part of. This table can, for example, be created from a stream, which captures changes in the hierarchy. It is not important which hierarchy a group belongs to, the only requirement is that identifiers are unique among multiple hierarchies.

2. Merging Measurement Streams The first operator merges the input stream with a stream of already calculated aggregation results. For the creation of this aggregated stream, see step 7. Note that the aggregated stream may require an additional converting step. In the following, we make no distinction between sensor measurements and aggregation results and call these values simply measurements.

3. Joining Measurement Stream and Group Table The measurement stream and the groups table are joined using an inner join operation. This leads to a new update of the resulting table whenever either a measurement arrives or the groups a sensor belongs to changes. The result of this join operation is a tuple consisting of the measurement and the set of all groups this measurement has effect on.

4. Duplicating Join Results In the next step, the measurements are duplicated in a way that for each group a new record is forwarded. This record has the following form: The key is a pair of sensor identifier and the corresponding

group[] represents a set of group identifiers, *meas* represent a measurement, and *aggr* represent an aggregation result. Everything located inside the gray box corresponds to the part of our approach, which can be deployed as an individual microservice. The tables and streams placed outside the gray box can be considered as interfaces to other components

group this record is created for. The value is the measurement. This operation is stateful as it always stores the last set of sensor groups and compares them with the new one. If a sensor was part of a group in a previous record but not in the currently processed one, a special record is forwarded with a “tombstone” value. This value serves for informing the following topology operators that the corresponding sensor is not longer part of the corresponding group.

5. Immediate Result: Last Value Table The duplicated records are materialized to a table, which lists the last measured value per sensor and group. An arriving tombstone record for a sensor-group-pair deletes the corresponding entry in the table. This table of last values is the entry point for the following aggregation.

6. Grouping and Aggregating Similar to an SQL “group-by” operation, table entries are grouped by their group name (second part of the key). The result is a grouped table containing one entry per group identifier. This table is then aggregated using appropriate adding and subtracting functions resulting in one aggregation result per group identifier. As defined by the dual streaming model, whenever an entry in the last values table is updated, a corresponding update record updates the grouped table and triggers the computation of a new aggregation result. This is done by first calling the subtract function for the previous record (e.g., arithmetically subtracting the sensors previous measurement from the total group’s value), followed by calling the add function for the new value (e.g., arithmetically adding the sensors new measurement to the total group’s value). Deleting an entry in the last value table (via a tombstone record) solely causes calling the subtract function.

7. Output: Aggregation Results As a last step, the aggregated values per sensor group are published to a data stream. On the one hand, this stream serves as an interface so that other applications or services can use these data as they were real measurements. On the other hand, the stream is fed back to the beginning of the topology, where it can be used to compute aggregated values for sensor groups containing this group.

Pilot Implementation for IoT Sensor Data

In this section, we return to our motivating example from “Motivating Example” and show in a pilot implementation how our proposed architecture can be used to aggregate IoT sensor data. The Titan Control Center [14] is a microservice-based application for analyzing the energy consumption in manufacturing enterprises. It integrates energy consumption

data of different data sources (e.g., machine-level data, building technology, or external software systems) and aggregates, analyzes, and visualizes them in near real time (see Fig. 3).

We extended the Control Center’s architecture by an additional microservice, which implements the topology described above and replaces a former, less scalable and reliable data aggregation. This microservice subscribes to a stream providing energy consumption data of individual machines and to a stream forwarding changes to the sensor hierarchy (provided by the Configuration microservice of the Control Center). It aggregates the data of all sensors in a group by summing them up and publishes every result to a dedicated topic allowing other services to subscribe to this aggregated data. Other microservices use these data, for example, to calculate power consumption statistics, to produce forecasts, or to visualize them.

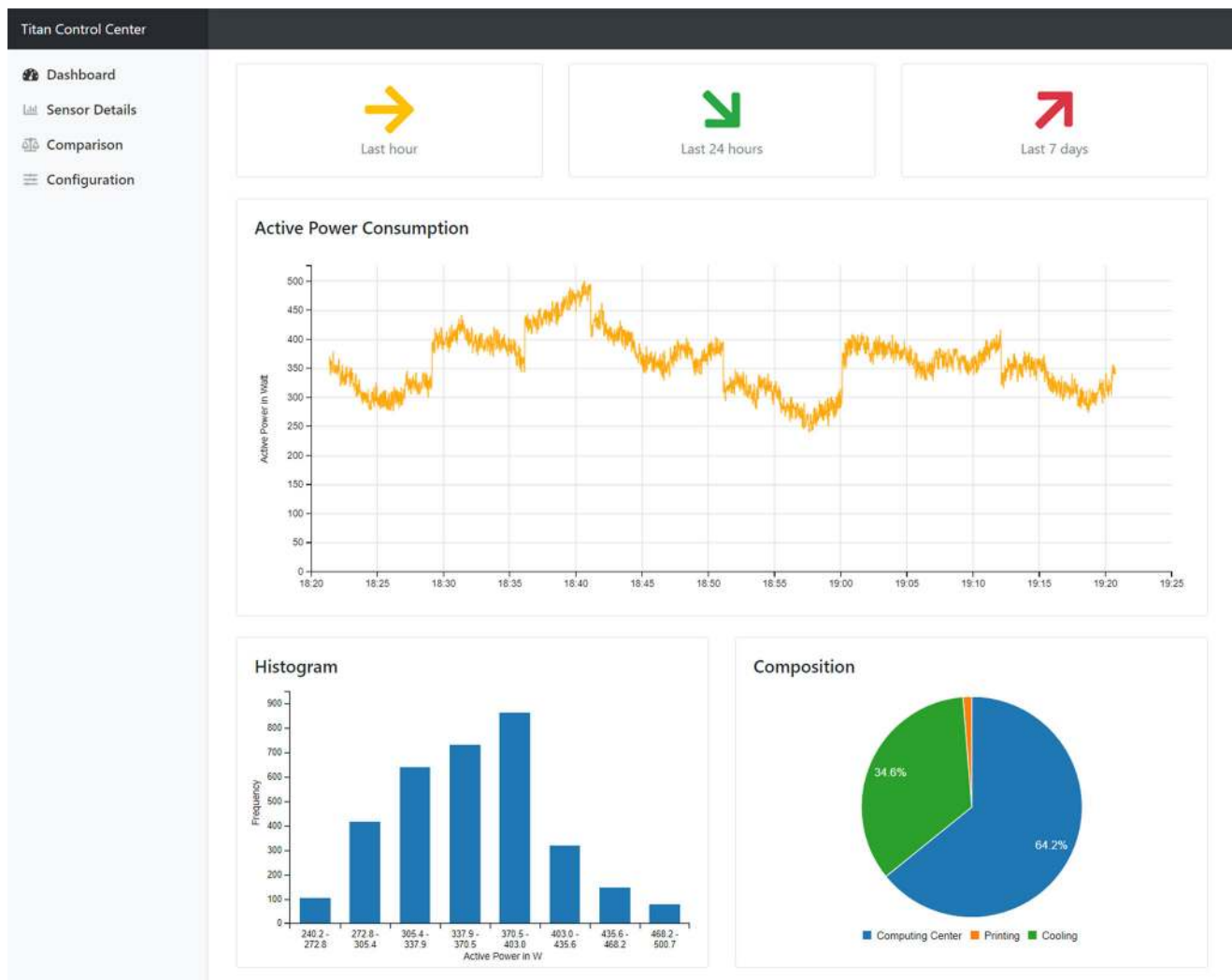


Fig. 3 Screenshot of the Titan Control Center dashboard visualizing the energy consumption in manufacturing enterprises

Specifying the proposed architecture with the dual streaming model allows for a straightforward implementation in Kafka Streams. Nevertheless, as the dual streaming model is more abstract than the Kafka Streams API, we had to introduce some additions: In some places, streams had to be explicitly converted into tables via *reduce* operations. The data type of aggregated records does not match the data type for sensor measurements, thus, an additional *map* operation before merging with the sensor data stream was necessary. To avoid emitting subtract events from the table of aggregations, these events have to be explicitly filtered out. The operator duplicating records for each parent had to be implemented as a custom *flatTransform* operator, as Kafka Streams currently does not support *flatMap* operations on tables. Using Kafka Streams promises a high degree of scalability and reliability, as deploying multiple instances of this microservice causes Kafka Streams to balance the load among all instance based on the topology and the number of configured Kafka topic partitions.

Experimental Scalability and Reliability Evaluation

We experimentally evaluate the scalability and the reliability of the proposed stream processing architecture. For these evaluations, we simulate different numbers of power metering sensors and aggregate their data streams to predefined groups using the Titan Control Center (see above). Each simulated sensor emits one measurement per second. We record both the number of sensor measurements per sensor and the number of computed aggregation results per second as well as the average event-time latency per second. The event-time latency is the time passed between sensor record generation and obtaining the result of an aggregation [3]. The experimental setup is deployed in a Kubernetes cluster of 4 nodes, each equipped with 384 GB RAM and 2×16 CPU cores providing 64 threads (overall 256 parallel threads).

Evaluation of Scalability

A software system is considered scalable if it is able to continue processing an increasing workload with additional resources provided [15]. In order to assess the scalability of our proposed approach, we therefore evaluate how an increasing amount of input data can be handled by an increasing number of aggregation instances. For this purpose, we identify the number of instances required to aggregate the data streams of a given workload. We group 8 simulated sensors into one group and group 8 of such groups again into one larger group. We evaluate 4 workloads, where we simulate 2, 3, 4, or 5 nested groups, resulting in a total

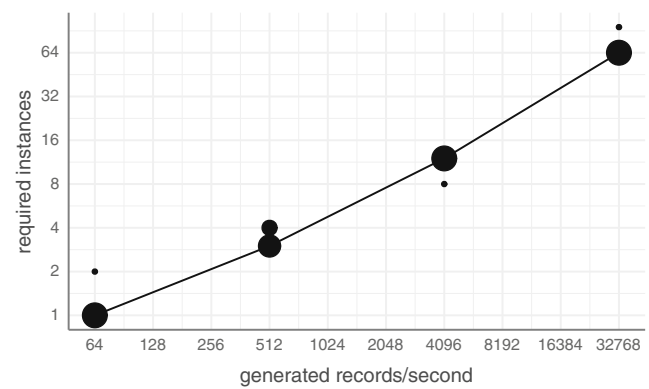


Fig. 4 Number of required instances for aggregation in relation to the number of generated records per second. The size of points indicates the frequency of the respective observation. The black line connects the median numbers of required instances per workload

amount of sensors of 8^2 , 8^3 , 8^4 , and 8^5 and a corresponding number of records per second. For each scenario, we deploy different numbers of instances ranging from 1 to 128.

We consider a deployment (i.e., a certain set of deployed instances) as being sufficient to handle the given workload if all simulated data can be processed such that no data records are piling up between their generation and the aggregation. To obtain this information, we measure the event-time latency between generation and aggregation. If this latency remains largely constant, we conclude that records are aggregated at approximately the same speed as they are generated. We apply linear regression to compute a trend line and consider the processing to be largely constant if the trend line's slope is less than 100 ms.¹ For each evaluated workload, we determine the minimum number of instances, which is required to handle that workload, i.e., shows an event-time latency trend line with a slope of less than 100 ms. This evaluation is repeated 10 times.

Figure 4 shows the median over all repetitions of the required number of instances per workload. We observed that the required number of instances scales linearly with the amount of data to be aggregated.

Evaluation of Reliability

In order to assess the reliability of our proposed approach, we evaluate how the architecture behaves when components fail. We generate the workload described in the scalability evaluation, which simulates 4 nested groups of sensors, and aggregate the data with 24 instances of the aggregation component. After 10 min of processing, we inject a

¹Due to Kafka Streams' task model, the throughput is subject to large fluctuations. The calculated trend line can therefore be inaccurate, suggesting a more conservative threshold for the slope of 100 ms. Lower thresholds return similar median results, but produce more outliers.

failure during operation by shutting down 18 instances and starting them again after 5 more minutes. We measure both the number of generated messages and the amount of aggregation results during the entire evaluation. Since both values fluctuate strongly, we additionally calculate a moving average over a 60-s window.

The average number of processed records per second over time is presented in Fig. 5. It can be seen that the amount of performed aggregations decreases sharply during the simulated failure. This is reasonable as there are not enough resources available anymore to process all data. However, if the stopped instances are replaced by new instances, the amount of processed data increases again. Furthermore, as we set the number of processing instances twice the number actually necessary (cf. scalability evaluation), the piled up data is also processed.

Dataflow Architecture Variations

While the basic architecture described in “[Basic Dataflow Architecture](#)” provides correct results when all measurements from the input stream arrive in the order of their timestamps, out-of-order records would produce wrong aggregation results. According to the dual streaming model, a late-arriving record would cause a recomputation of all consecutive steps. While this is in principle desired, the corresponding lookup in the last values table only yields the most recent value causing wrong results. A simple, yet in many cases sufficient solution is to simply reject late-arriving records. A more complex, but also correct solution is to explicitly consider measurement times during the aggregation process. In the following, we present two architecture variations and compare them with our basic architecture in Table 1.

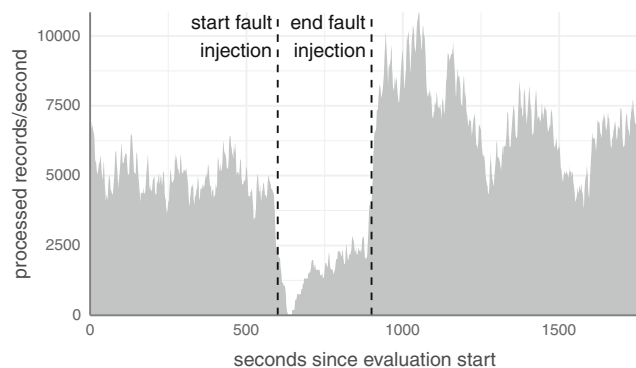


Fig. 5 Number of processed records per second over a moving average of 60 s in the course of time. The two dashed vertical lines indicate the point in time at which the simulated failure starts or ends

Tumbling Window–Based Dataflow Architecture

In order to support out-of-order records, we extend our previously described architecture by a windowing operator located after the duplicate step and before the creation of the last values table (see Fig. 6). This operator splits the absolute timeline into tumbling windows, that is, time windows of constant but configurable size [9], and associates each record with the time window it was recorded in. The windowing operator transforms each input record by extending its key by the corresponding window, where a window is represented by a pair of start and end timestamp.

For the consecutive last value table, this means the table does not only store the last value per sensor-group-pair anymore, but instead stores this value for each time window. However, to not let the table grow indefinitely, a decision has to be made for how long data of passed time windows have to be retained. This can be configured via a “grace period,” which effectively specifies for how long out-of-order records are considered by the architecture since records, which arrive after the grace period of their corresponding window has expired, are discarded. The following grouping and aggregation is then not exclusively made by the key’s “group” part, but by its “group” and “window” attribute. Afterwards, a result is only emitted when the window ended. In contrast to our basic architecture, a result for a group is thus no longer published to the aggregations stream with every update of a sensor, but only at the end of each window. This is achieved by using a *suppression* function [16] called before publishing to the aggregation result streams. Suppressing record forwarding is not part of the dual streaming model but is implemented as an operator in Kafka Streams.

With this extended architecture, out-of-order records are associated with their corresponding time window. Thus, they neither have to be discarded nor their consideration has influence on the aggregation results for other windows. In comparison with our basic architecture of “[Basic Dataflow Architecture](#),” the memory usage of the last value table in this architecture is larger by a constant factor, which corresponds to the number of simultaneously opened windows. The number of simultaneously open windows at any time corresponds to the number of windows whose grace period has not expired until that time and, thus, only depends on the choice of the grace period (see Table 1).

The major drawback of this architecture is that an emit rate has to be chosen in advance. Since for computing an aggregation result for a group of sensors only measurements within the respective time window are considered, this architecture requires each sensor to provide at least one measurement per window. The frequency sensors delivering data with must therefore be known in advance (or at least an upper estimate), so that the window size is

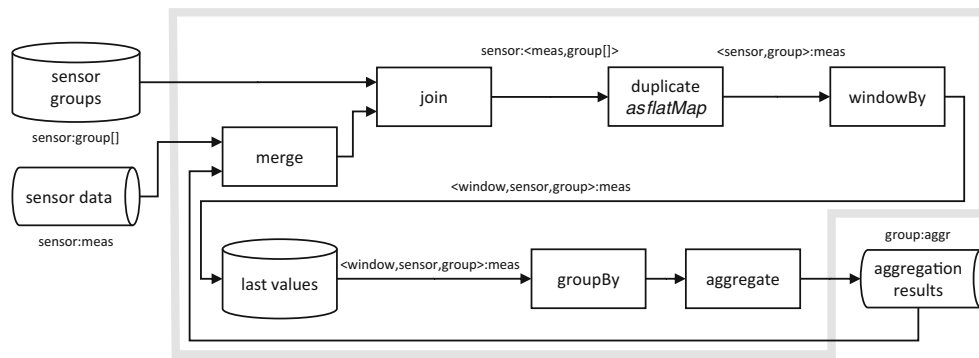


Fig. 6 Extended dataflow architecture, which allows considering the recording time of measurements to calculate correct results for out-of-order records. The notation of architectural elements corresponds to

the one in Fig. 2. The added data type *window* represents a time window consisting of an inclusive start timestamp and an exclusive end timestamp

chosen sufficiently large. Furthermore, this frequency automatically corresponds to the frequency of publishing aggregation results. However, these two frequencies may conflict if individual sensors generate measurements with different frequencies but the frequency of generating aggregation results should not be bounded by the lowest of these frequencies.

Hopping Window–Based Dataflow Architecture

Our second architecture variation is a modification of the tumbling window–based architecture depicted in Fig. 6. In order to support out-of-order records as well as to allow configuring the aggregation emit rate independently from the lowest measurement frequency, it modifies the windowing operator in a sense that it creates hopping windows [9] instead of tumbling windows. That is, the windowing operator does not split the timeline in successive windows, but instead in windows, which overlap by a configurable duration. Thus, a measurement does not belong to one window anymore but to multiple windows, causing the windowing operator to not emit one windowed record per input anymore but instead to produce multiple windowed records. The semantics of the following operators do not change. However, duplicating a record for different windows causes the following grouping and aggregate operations to be performed for each open window. The final suppress operation, however, causes an aggregation result to be emitted only when the window closes. Thus, on average only one of the aggregation results is emitted per measurement.

The hopping window–based architecture thus allows to choose the window size according to the maximal expected duration between two measurements of the same sensor. On the other hand, it allows to specify an additional emit rate, which can be significantly smaller and configures the interval with which aggregation results are emitted.

Consequently, if no new measurements for a sensor were received since emitting the last aggregation result, this architecture allows to rely on previous measurements, which were recorded before the last emission of an aggregation result (but are still within the window size). Unlike to our basic architecture but identically to the tumbling window–based architecture, the hopping window–based architecture only emits results at certain points in time (i.e., whenever a window closes). In contrast to the tumbling window–based architecture, however, the emit rate can be set arbitrary small without compromising correctness. However, the greater the difference between window size and emit rate is, the more records will be generated. The windowing operator duplicates each measurement as often as there are overlapping windows. According to this factor, records are created that need to be processed by the consecutive streaming operators and the last value table has to store accordingly more values. “[Experimental Comparison of Architectures](#)” presents our experimental comparison of how the choice of the window size influences the required computing resources.

Experimental Comparison of Architectures

In this section, we experimentally evaluate and compare the performance of the individual architectures presented in “[Basic Dataflow Architecture](#)” and “[Dataflow Architecture Variations](#).” For this purpose, we reuse the experiment setup from “[Experimental Scalability and Reliability Evaluation](#)” and simulate 512 sensors organized in a pre-defined hierarchy with 3 nested levels of groups. Again, each sensor emits one measurement per second and we use the Titan Control Center to aggregate sensor measurements according to the hierarchy. The Kafka Streams application executing the stream processing architecture is configured with a *commit interval* of 1 s meaning that Kafka Streams stores

and forwards intermediate results in the stateful operations every second. We evaluate different numbers of Kafka partitions and set the number of Kafka Streams threads to 4 times the number of partitions ensuring that each Kafka Streams task can be executed in a separate thread. For the two window-based architectures, we set the emit rate to 1 s and the grace period to 10 s. For the hopping window-based architecture, an additional window size has to be configured, which we set in individual experiments to 10 s as well as to 60 s. Each configuration is executed for 10 min, where we discard all monitoring results for the first minute and we repeat each execution 10 times. The experimental setup is deployed on the same Kubernetes cluster as described in “[Experimental Scalability and Reliability Evaluation](#).” However, in contrast to the previously described evaluations of scalability and reliability, we only deploy one aggregation instance but configure it to use multiple threads.

For each evaluated architecture, we continuously monitor the required computing resources as well as the average aggregation latency and the consumer lag. The aggregation latency is the time passed between receiving a sensor measurement and emitting the corresponding aggregation result. The consumer lag describes the difference between the number of stored records in Kafka and the number of already consumed records by the aggregation. For determining the required computing resources, we measure

the memory usage of the aggregation as well as Kafka Streams’ disk usage since Kafka Streams stores the operator’s state on the local disk but also uses in-memory caching. Furthermore, we determine the required number of threads by finding a configuration which does not lead to an increasing consumer lag over time.

Figure 7 shows the average value of all monitored properties. We provided sufficient memory on the experiment nodes, which explains the low usage of the local disk in comparison with main memory. It is remarkable that the resource consumption is almost the same for our basic architecture and the tumbling window-based one. This leads us to conclude, that for a better correctness no more hardware resources have to be provided. Also the aggregation latency scarcely differs. However, if larger aggregation window sizes are required, as enabled by the hopping window-based architecture, these can only be achieved with significantly increased resources. For example, an aggregation window that is 10 times larger than the emit rate requires 42% more memory compared with a window size that equals the emit rate. An aggregation window of 1 min for a 1-s emit rate requires even 69% more memory. Kafka Streams allows to store the state of operators at the local disk, which is typically cheap compared with main memory. Whereas the increasing memory usage is therefore probably not an issue for many use cases, the increasing amount of generated data requires accordingly more threads for processing them. An

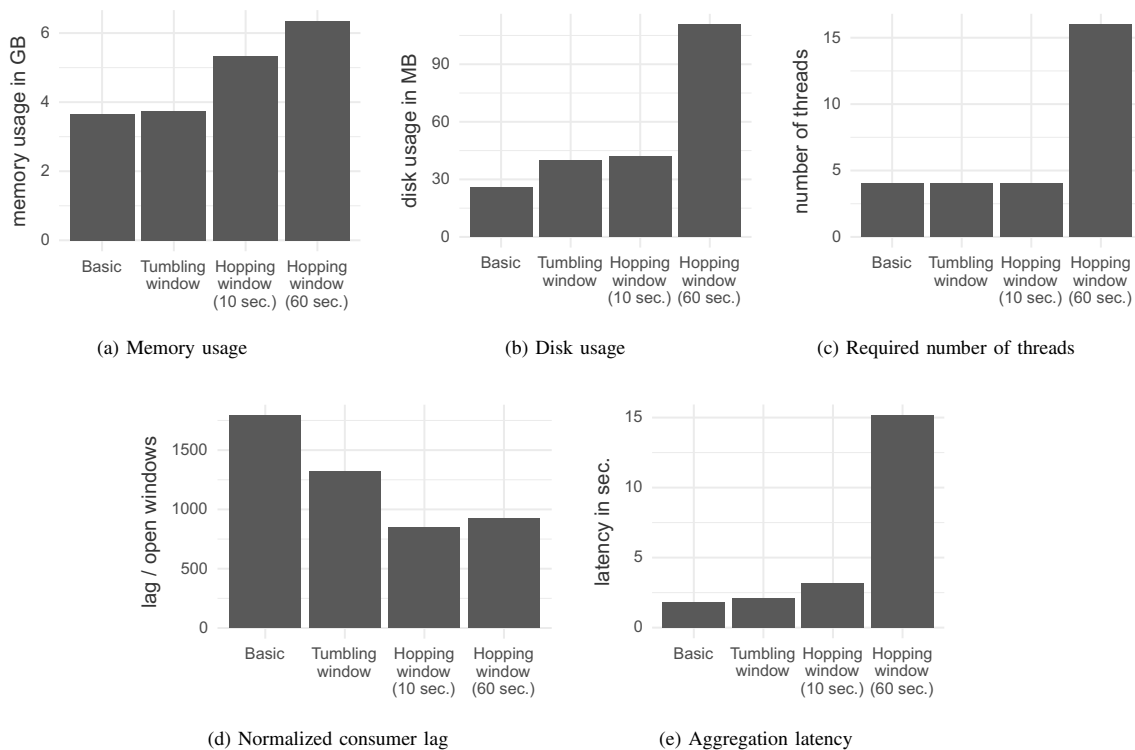


Fig. 7 Average monitored value of different performance attributes for the individual architecture variations. **a** Memory usage. **b** Disk usage. **c** Required number of threads. **d** Normalized consumer lag. **e** Aggregation latency

aggregation window of 60 s requires, for example, 4 times the amount of threads than an aggregation window of 10 s.

We normalize the consumer lag by dividing the measured lag by the amount of parallel open windows, which corresponds to the number of updates triggered per measurement (see Table 1). Compared with the number of measurements generated per second, the consumer lag stays on a fairly low level for all architectures. From the basic to the tumbling window-based and from the tumbling window-based to the hopping window-based architecture with a 10-s window, the aggregation latency only increases slightly. In further experiments, we observed that with a shorter commit interval the latency can be reduced at the cost of increased resource consumption. In order to reduce the amount of intermediate data, our Kafka Streams implementation uses a suppress operator, which ensures that aggregation results are only emitted after the aggregation time window passes. Unfortunately, with this configuration, late-arriving records (i.e., those arriving during the grace period) cause significantly delayed aggregation updates, which probably explains the large delay for the architecture with a hopping window of 60 s.

Industrial Case Study

Our implementation is used in production with two manufacturing enterprises to apply Industrial DevOps [17]. In these enterprises, the aggregation results are used, for example, to gain insights into how much energy is used by certain types of machines (e.g., the overall air conditioning), how big the difference is between measured company-wide energy consumption and the sum of all known consumers, or how much an individual machine contributes to the overall consumption of all machines of that type. Figure 8 shows a screenshot of the Titan Control Center's comparison view. It provides a continuously updating view of the total electrical power consumption of our partner *Kieler Nachrichten Druckzentrum*, a newspaper printing

company, in comparison with the consumption of its major consumers.

Related Work

Analyzing sensor data, often from IoT device, is a frequent use case for stream processing [18]. In such analytics systems, aggregating individual measurements to reduce the overall amount of data is a common task [19].

The primary type of data aggregation studied in literature is aggregating data streams along the dimension of time. In this case, records within the same time window having the same key are aggregated to a new record. Depending on the actual requirements, records can be aggregated within fixed windows, sliding windows, or session-based windows [12]. In addition to research on the efficient computation of window aggregations [20–22], there are also publications proposing software architectures for a temporal aggregation platform, for example by Twitter [23]. Temporal aggregations are compatible with our approach, which means that temporal aggregated streams of sensor data can be further aggregated to groups and streams for sensor groups can be further aggregated over time.

Joins in stream processing (analogous to those in SQL) [24] allow to connect different or identical streams. As the join operation is a bivariate function and thus aggregating multiple streams would require a chain of join operations, a corresponding pipeline can only statically be created and not dynamically adjusted at runtime. This contrasts our approach, which allows reconfigurations at runtime by changing the sensor groups table or an underlying stream.

As most approaches applying stream processing, our presented approach is primarily based on data parallelism [2], meaning that (sub)topologies exist in multiple instances, each processing a portion of the data. Pipe-and-Filter frameworks such as TeeTime [25] employ task parallelism, where the individual filters (operators) are executed in parallel. Since in this way all data pass each filter, the identified

Table 1 Comparison of our three proposed architectures. k corresponds to the number of different sensor-group-pairs, e is the emit rate, w is the window size, and g is the grace period

Architecture	Basic	Tumbling window-based	Hopping window-based
Support for out-of-order records	×	✓	✓
Unknown record frequency	✓	×	✓
Updates triggered per record	1	1	$\frac{w}{e}$
Memory usage	$\mathcal{O}(k)$	$\mathcal{O}(k \frac{g+e}{e})$	$\mathcal{O}(k \frac{g+w}{e})$

Note that for the tumbling window-based architecture, e equals w . Whereas our basic architecture does not support out-of-order records and the tumbling window-based architecture requires the measurement frequency of sensors to be known beforehand, the hopping window-based architecture is able to handle both. However, in contrast to the former two, it generates significantly more intermediate records and thus uses more memory in the last value table.

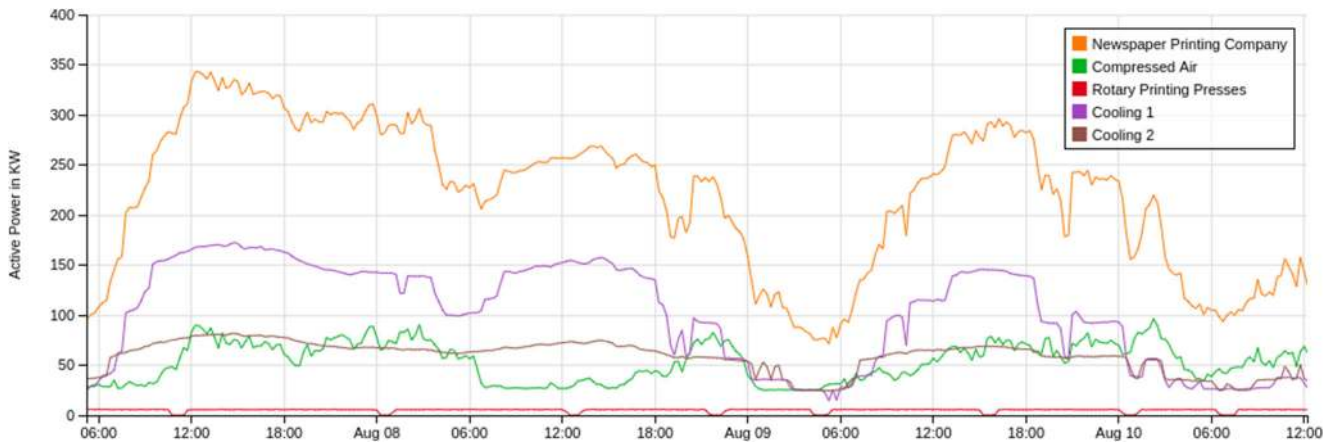


Fig. 8 Screenshot of the Titan Control Center's comparison view. It shows the overall power consumption of the *Kieler Nachrichten Druckzentrum* in comparison with the main subconsumers. The overall

consumption was continuously computed by aggregating the measurements of all subconsumers

requirements can be realized in a single filter. In contrast to our solution, however, scalability would be significantly compromised.

Conclusions

Software systems, which analyze or react on sensor data streams, often have to process not only raw measurements but also aggregated data for groups of sensors. In this paper, we presented a stream processing dataflow architecture for continuously aggregating sensor data. It supports the aggregation of hierarchical groups, multiple such groups in parallel, allows for reconfigurations at runtime and can be integrated (e.g., as a microservice) into existing big data architectures. We presented three variations of this dataflow architecture having different characteristics regarding correctness and performance. Our experimental comparison assists adopters in choosing an appropriate variation. We provide an implementation of this architecture for power consumption data and show how our implementation can be integrated into an analytics platform used in industry. Furthermore, in an experimental evaluation, we showed that our proposed architecture scales linearly with the amount of sensors and tolerates faults during operation. A replication package and experimental results provided as supplemental material allow to repeat and extend our work [26].

For future work, we plan to optimize the hopping window-based architecture in terms of its resource usage, while still being able to handle out-of-order records and tolerate varying measurement frequencies. Furthermore, future work may explore how our architecture can be implemented with other stream processing engines, for example, by considering recent trends towards uniform stream query

languages [27]. As we were not able to discover any scalability limitations in our conducted experimental evaluation, we also plan to conduct experiments with even larger amounts of sensors.

Funding This research is funded by the German Federal Ministry of Education and Research (BMBF) as part of the Titan project (<https://www.industrial-devops.org>, grant no. 01IS17084B). Open Access funding is enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. G. Cugola, A. Margara, Processing flows of information: from data stream to complex event processing. *ACM Comput. Surv.* **44**(3), 15:2-15:62 (2012). <https://doi.org/10.1145/2187671.2187677>
2. H. Röger, R. Mayer, A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput. Surv.* **52**(2), 36:1–36:37 (2019). <https://doi.org/10.1145/3303849>
3. J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, V. Markl, Benchmarking distributed stream data processing systems. in *Proc. IEEE International Conference on Data Engineering*, (2018)
4. J. Lin, The Lambda and the Kappa. *IEEE Internet Comput.* **21**(5), 60–66 (2017). <https://doi.org/10.1109/MIC.2017.3481351>

5. W. Hasselbring, Software architecture: past, present, future. in *The Essence of Software Engineering*, Springer, ed. by V. Gruhn, R. Striemer, (2018)
6. W. Hasselbring, G. Steinacker, Microservice architectures for scalability, agility and reliability in e-commerce. in *Proc. IEEE International Conference on Software Architecture Workshops*, (2017)
7. M. Fowler, Event sourcing. <https://martinfowler.com/eaDev/EventSourcing.html> (2005)
8. S. Henning, W. Hasselbring, Scalable and reliable multi-dimensional aggregation of sensor data streams. in *Proc. IEEE International Conference on Big Data*, (2019)
9. M.J. Sax, G. Wang, M. Weidlich, J.-C. Freytag, Streams and tables: two sides of the same coin. in *Proc. International Workshop on Real-Time Business Intelligence and Analytics*, (2018)
10. D.J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: a new model and architecture for data stream management. *The VLDB Journal* **12**(2), 120–139 (2003). <https://doi.org/10.1007/s00778-003-0095-z>
11. T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, Millwheel: fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.* **6**(11), 1033–1044 (2013). <https://doi.org/10.14778/2536222.2536229>
12. T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R.J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, S. Whittle, The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.* **8**(12), 1792–1803 (2015). <https://doi.org/10.14778/2824032.2824076>
13. G. Wang, J. Koshy, S. Subramanian, K. Paramasivam, M. Zadeh, N. Narkhede, J. Rao, J. Kreps, J. Stein, Building a replicated logging system with Apache Kafka. *Proc. VLDB Endow.* **8**(12), 1654–1655 (2015). <https://doi.org/10.14778/2824032.2824063>
14. S. Henning, W. Hasselbring, A. Möbius, A scalable architecture for power consumption monitoring in industrial production environments. in *Proc. IEEE International Conference on Fog Computing*, (2019)
15. N.R. Herbst, S. Kounev, R. Reussner, Elasticity in cloud computing: what it is, and what it is not. in *Proc. International Conference on Autonomic Computing*, (2013)
16. J. Roesler, Kafka streams' take on watermarks and triggers. <https://www.confluent.io/blog/kafka-streams-take-on-watermarks-and-triggers> (2019)
17. W. Hasselbring, S. Henning, B. Latte, A. Möbius, T. Richter, S. Schalk, M. Wojcieszak, Industrial DevOps. in *Proc. IEEE International Conference on Software Architecture Companion*, (2019)
18. A. Shukla, S. Chaturvedi, Y. Simmhan, RIoTbench: an IoT benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience* **29**(21), e4257 (2017). <https://doi.org/10.1002/cpe.4257>
19. A.B.A. Alaasam, G. Radchenko, A. Tchernykh, Stateful stream processing for Digital Twins: microservice-based Kafka Stream DSL. in *Proc. International Multi-Conference on Engineering, Computer and Information Sciences*, (2019)
20. J. Li, D. Maier, K. Tufté, V. Papadimos, P.A. Tucker, No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.* **34**(1), 39–44 (2005). <https://doi.org/10.1145/1058150.1058158>
21. J. Traub, P. Grulich, A.R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, V. Markl, Efficient window aggregation with general stream slicing. in *Proc. International Conference on Extending Database Technology*, (2019)
22. L. Benson, P.M. Grulich, S. Zeuch, V. Markl, T. Rabl, Disco: efficient distributed window aggregation. in *Proc. International Conference on Extending Database Technology, OpenProceedings.org*, (2020)
23. P. Yang, S. Thiagarajan, J. Lin, Robust, scalable, real-time event time series aggregation at Twitter. in *Proc. International Conference on Management of Data*, (2018)
24. A. Arasu, S. Babu, J. Widom, The CQL continuous query language: semantic foundations and query execution. *VLDB J.* **15**(2), 121–142 (2006). <https://doi.org/10.1007/s00778-004-0147-z>
25. C. Wulf, W. Hasselbring, J. Ohlemacher, Parallel and generic pipe-and-filter architectures with TeeTime. in *Proc. IEEE International Conference on Software Architecture Workshops*, (2017)
26. S. Henning, W. Hasselbring, Replication package for: scalable and reliable multi-dimensional sensor data aggregation in data-streaming architectures. <https://doi.org/10.5281/zenodo.3736689> (2020)
27. E. Begoli, T. Akidau, F. Hueske, J. Hyde, K. Knight, K. Knowles, One SQL to rule them all – an efficient and syntactically idiomatic approach to management of streams and tables. in *Proc. International Conference on Management of Data*, (2019)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.