Scalability Benchmarking of Kafka Streams Applications

Simon Ehrenstein
stu200776@mail.uni-kiel.de

Kiel University

Abstract. Stream processing frameworks have gained popularity in the past years. In contrast to the isolated analysis of individual operations we design four benchmarks for common stream processing use cases that are derived from realistic production scenarios. Therefore, we define relevant topologies that cover the realistic scenarios, scalability dimensions, metrics, and workloads that lead to our benchmark definitions. Additionally, we execute one of our benchmarks for the stream processing framework Kafka Streams. Our results show that Kafka Streams scales linearly with the workload for the given configuration.

Keywords: Stream Processing · Kafka · Kafka Streams · Benchmarking

1 Introduction

Data processing software systems can be divided into batch- and stream processing systems [3]. While in batch processing systems, whole chunks of incoming data are processed sequentially in a single step at a certain time after its arrival, in stream processing systems, the data is processed continuously with its arrival. In times of the Internet of Things (IoT) and huge amounts of data, stream processing is gaining importance [8]. A key requirement for stream processing is the ability of applications to handle larger loads by adding resources which is referred to as scalability [13]. Also, scalability is fundamental to allow auto-scaling of the application, e.g., in elastic environments. A realistic scenario for stream processing is the monitoring of the power consumption in industrial facilities [4]. In this case, sensors for measuring the power consumption are the source of the incoming data which is processed continuously to allow real-time visualization and analytics. When the load of input data changes and exceeds the maximum load the application can handle with its current configuration, the application must by scaled in order to cope with the changing requirements. Therefore, it is important to know how the system reacts to changing input data, meaning it should be conceivable how the system scales. There are various stream processing frameworks out there which contribute to the design and implementation of stream processing applications by providing reusable structures. To enable developers to find the appropriate stream processing framework for their applications, it is useful to have knowledge about how different frameworks scale in various scenarios. Therefore, in this paper, we define a set of specification-based benchmarks [10] that can be used as a basis to compare different stream processing frameworks in terms of their scalability, or to compare the same stream processing framework in different environments, for example, for different cloud providers. Additionally, we focus on the implementation and execution of one of our designed benchmarks for the stream processing framework *Kafka Streams*.

The remainder of the paper is structured as follows: In Section 2, we give a short overview over Kafka Streams. After that, in Section 3, we explain our benchmarking approach which consist of the description of topologies and the definition of scalability dimensions, metrics, and workloads. Finally, in Section 4 we execute one of our benchmarks and discuss the results, before we come to our conclusions in Section 5.

2 The Stream Processing Framework Kafka Streams

Kafka Streams [12] is a framework which allows implementing stream processing applications in combination with Apache Kafka [11], simply referred to as Kafka. Kafka allows clients to publish streams of data entries to a distributed store and to asynchronously consume data from the store. Thereby, data is considered to be a continuous stream of key-value pairs that are stored in ordered data structures called topics. Topics are partitioned in order to allow parallel processing for achieving scalability. Moreover, partitions can be replicated over multiple Kafka brokers for achieving fault-tolerance. Kafka Streams utilizes Kafka and provides abstractions for performing common operations in stream processing such as mapping operations, aggregations or joins. In Kafka Streams, an operation is either stateless or stateful. Each application of a stateless operation, for example, a filter operation, does only depend on one record, whereas the application of a stateful operations, e.g. counting, depend on multiple records.

3 Benchmark Design

3.1 Definition

Benchmarking of software can be defined as a standardized process used for the competitive comparison of systems [10]. Based on this definition we want to provide a specification for benchmarking competing stream processing frameworks in order to compare their scalability.

3.2 Topologies

In contrast to benchmarking individual stream processing operations in an isolated manner, our approach is based on realistic production scenarios from a real-time stream processing application [4] for the monitoring of the power consumption in industrial facilities based on sensor measurements. Therefore, we build our benchmarks on top of different topologies that map to relevant use cases. The first benchmark is based on a linear topology which is composed out of stateless operations. This maps to use cases where we read messages from a stream, manipulate them and finally store them in a database. In the second benchmark, we look at an example for hierarchial aggregation that allows to compose values from certain sources to higher order results with respect to a given hierarchial data structure. An example use case for this form of aggregation is a system, where we want to determine the total power consumption of machines, based on sensors that are part of these machines, where the machines themselves are grouped together to higher order units, resulting in a nested tree structure. For the two remaining benchmarks, where we examine the aggregation based on non-overlapping and overlapping windows. The two types of windows are visualized in Figure 1. In Kafka terminology, non-overlapping windows are also called *tumbling* windows (a), while overlapping windows are referred to as *hopping* windows (b) [12]. In general, windows are defined by their range, representing the span of the window, and their period, determining the interval how often a new window begins. When the period of a window is less than its range, this means multiple windows of that form are overlapping when they are continuously created, since data points are assigned to all parallel windows that exist at the time of observation. For the special case, where range and period are equal, the windows are considered to be non-overlapping, since there only exist one definite window at a time where each data point can be assigned to. This means, tumbling windows can be seen as a special case of hopping windows. Accordingly, we define the four topologies *(1) Sink*, *(2) Hierarchial Aggregation*, *(3) Non-Overlapping Windowed Aggregation*, and *(4) Overlapping Windowed Aggregation* as the basis for our benchmarks.

The first topology *(1) Sink* corresponds to a typical linear pipeline of stateless operations. First, the data is read from a Kafka topic and a **map** operation is performed to manipulate the data. Consecutively, in a **for Each** operation, we log the corresponding message. In a realistic scenario, here the data can be, e.g., be forwarded to a persistent storage such as an database or alternatively to another kafka topic.

The second topology *(2) Hierarchial Aggregation* constitutes an hierarchial, key-based aggregation, as proposed by [5] which reuses previously computed aggregation results. Further, we define the following two benchmarks based on two similar topologies for time-based aggregation. As these topologies are quite similar to each other, they are visualized together in Figure 2.

The third benchmark *(3) Non-Overlapping Windowed Aggregation* constitutes an windowed aggregation with non-overlapping windows: After reading input records from a topic, we group our records of the stream by their keys in a **group By Key** operation. After that, we transform the grouped stream to a windowed stream with non-overlapping windows and perform the aggregation, before we publish the aggregation results to the output topic.

The benchmark*(4) Overlapping Windowed Aggregation* allows a key-based aggregation with overlapping windows: Similarly to the third benchmark, we read records from a topic, but afterwards, we perform a **select Key** operation,

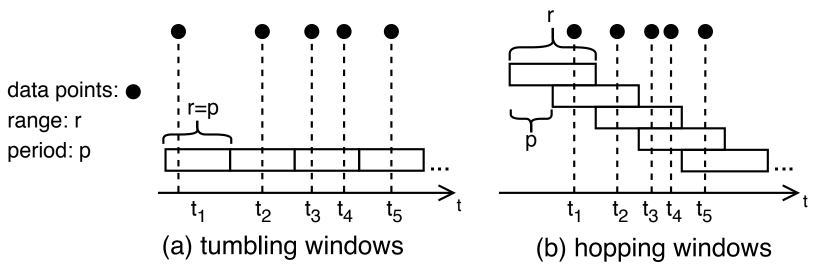

Fig. 1. Window types for aggregation in Kafka Streams.

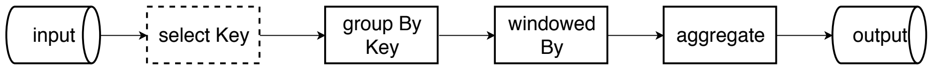

Fig. 2. Visualization of the topologies for overlapping and non-overlapping windowed aggregation. The select key operation is only used in the third benchmark for overlapping windowed aggregation.

where we modify the key, based on the record timestamp. This leads the amount of possible keys to grow linearly with the size of some number. For example, if it is desired to process records with the same key for each weekday for the window, the domain of keys grows by a factor of seven, as each original key gets mapped to one of seven possible keys, each for one weekday. In the next step, we perform the **group By Key** operation and the remaining stream operations analogously to the third benchmark.

3.3 Relevant Dimensions for Scalability

Stream processing applications can have various characteristics, such as types and format of input data or the internal structure of the application. Hence, each of these characteristics constitutes a dimension that potentially impacts the scalability of the system, resulting in a large degree of variability [1] for the scalability benchmarking. In the following, we define a set of dimensions that affect the scalability of stream processing engines. This enables us to evaluate with our benchmarks, whether the system scales with the corresponding dimensions.

3.4 Number of Message Keys

Depending on the stream processing framework, the decision which record belongs to which partition of a topic is made with respect to the key. For example, in Kafka Streams, messages with the same key are always assigned to the same partition. As in Kafka Streams, the amount of partitions impacts the degree of parallel processing within a topic, the amount of messages with different keys must be large enough, such that, the messages can be distributed evenly over all partitions in order to exploit the full potential of parallelization. To an instance that produces messages with the same key we refer as a message generator.

Message Frequency A fundamental workload dimension is the frequency of the sent messages. We define the message frequency as the number of messages with the same key that a generator produces within a certain time span. As a consequence, the number of generators and the message frequency together determine the total rate with which messages are produced within the system.

Hierarchy Structure For topologies that perform hierarchial aggregation, it can be analyzed how the application scales with the structure of underlying hierarchy. On the one hand, the complexity of a hierarchy is affected by its height and on the other hand by the number of children that each inner node within the hierarchy has. Therefore, we can analyze how the application scales with both, the height of the hierarchy and the number of children for individual nodes. Depending on the interpretation, the hierarchial structure can be closely related to the number of message keys, e.g., each leaf of the hierarchy can be a message generator. This also means, that the hierarchy also affects the number of message keys.

Time Window Structure For topologies that use windowed aggregation, the structure of the time window, the aggregation is based, on is crucial. This includes the range of the window and additionally the period how often a new window is initialized. Depending on these parameters, there might exist multiple partially overlapping windows at the time which means that records can belong to more that one window.

3.5 Base Metrics for Scalability

In this section, we define the base metrics that are prerequisite for the definition of the composed scalability metrics presented in Section 3.6.

Latency The latency for records in a stream processing application lets us assess whether the processing of data within the system is sufficient. Karimov et. al.[9] define latency as the difference between the time a message has been processed completely and the initial time of the event, or the time the system has started processing the message. However, calculating the latency as described for stateful operations, e.g. windowed aggregations, yields to distorted results, since some records are not processed immediately. This can be encountered by defining the latency for stateful operations as the distance between the time a record has been processed completely and the maximum event time over all records contained in the respective stateful context. With these definitions of latency, we can conclude that our application is able to handle the load efficiently as long as the latency remains mainly constant.

Consumer Lag Measuring the latency often is appropriate for assessing the scalability of an application, as it does not require a deeper understanding of the examined application. However, the latency can be influenced by other factors, such as network bottlenecks, buffering, or waiting for the completion of stateful operations. A more precise approach is to analyze the consumer lag which often is the main factor affecting the latency in stream processing. During the lifetime of an application, producers and consumers continuously commit offsets that represent how many messages they have produced or consumed to a certain point. Practically, the consumer lag is determined by the number of records consumers are behind from producers for a topic. This means, the consumer lag can be determined by the distance between producer and consumer offset. Naturally, consumers are always be behind producers, since messages can only be consumed after they are produced. Thus, an increasing consumer lag is an indication for a lack of the application to cope with the incoming rate of data.

Throughput We define the throughput of an application as the number of records, a streaming application can process in a fixed amount of time. Together with the latency and the consumer lag, the throughput is the base for the following definition of our more composed scalability metrics.

3.6 Scalability Metrics

In this section, we present a set of more complex metrics that allow us to measure the scalability of our application for the different topologies with respect to the different workload dimensions.

Sustainable Throughput We define the sustainable throughput as the maximum throughput a stream processing application can handle while the consumer lag is not increasing significantly. Accordingly, given a threshold for the upward slope, the sustainable throughput is determined by the maximum throughput where the increase of the consumer lag is less than the threshold. In practice, the sustainable throughput is an indication for the point where an application has to be scaled. This definition of sustainable throughput is similar to the definition from Imai et. al. [7] who define the metric over the amount of data that accumulates within the stream processing topology.

Appropriate Hierarchy- and Time Window Structure Analogously to the sustainable throughput, we define metrics that allow to relate the consumer lag to the superordinate parameters of the topology. Specifically, we define the appropriate hierarchy structure as the maximum height of the hierarchial structure where the consumer lag does not increase significantly. The appropriate hierarchy structure for the number of children per inner node is defined analogously. Similarly, we define the appropriate time window structure for a time window with a range r and a period p as the maximum value of p where the consumer lag does not increase significantly.

3.7 Workload Definition

Figure 3 gives an overview over the topologies, the associated scalability dimensions, and metrics. Considering which dimension is relevant for each topology and which metric is suitable for each dimension, we define the workloads for our benchmarks. Hereby, each benchmark is composed by multiple experiments, each with different parameters.

For the first benchmark that corresponds to the topology *(1) Sink*, the relevant scalability dimension are the number of generators and message frequency, and the corresponding metric is the sustainable throughput. We design the workload, such that we have a variable number of generators from 2^{12} to 2^{17} that each produce records with a constant frequency of 1000 ms. We estimate the sustainable throughput by the maximum measured throughput of the application where the consumer lag does not increase more than 100 records over the duration of the benchmark execution, meaning that the difference between the consumer lag at the beginning and the end of the benchmark is less than 100 records.

For the second benchmark that is based on the topology *(2) Hierarchial Aggregation*, the relevant scalability dimension are the hierarchy structure and the message frequency, and the relevant metric is the appropriate hierarchy structure. For our workload, we only consider full hierarchies, meaning that each inner node within the hierarchy has a total number of 10 children. Consequently, the total number of leaves of the hierarchy is equal to 10^k, where k is the height of the hierarchy. For each leaf, we create a message generator, meaning that the workload defines multiple experiments with 10^1 to 10^5 generators, each producing records with a frequency of 1 record per second.

For the third benchmark based on the topology *(3) Non-Overlapping Windowed Aggregation*, the most relevant scalability dimension is the number of generators, as the number of keys determines the size key space that the windowed aggregation is based on, but also the message frequency can be considered. The corresponding is the sustainable throughput. Our workload contains different amounts of generators, from 2^{12} to 2^{17}, each generating messages with message frequencies from 1 record per second to 6000 messages per second. For each of the amounts of keys, the workload consists of a constant time window with a range and period of one minute. As a consequence, there always exists exactly one window at a time.

For the fourth benchmark, based on the topology *(4) Overlapping Windowed Aggregation*, the dimension is the time window structure, defined by the range and period of the window, but as for the previous benchmark, also the message frequency might be considered. The relevant metric is the sustainable time-window structure. Our workload defines a time window with a fixed range of 1 minute and a variable period between 1000 ms and 10ms. As a consequence, the number of overlapping windows is at least 60 and at most 6000. Moreover, for each incoming message, we select a new key, where the new key depends on the record timestamp in seconds modular seven, resulting the key space to increase by the factor of seven. Although it would also be reasonable to consider

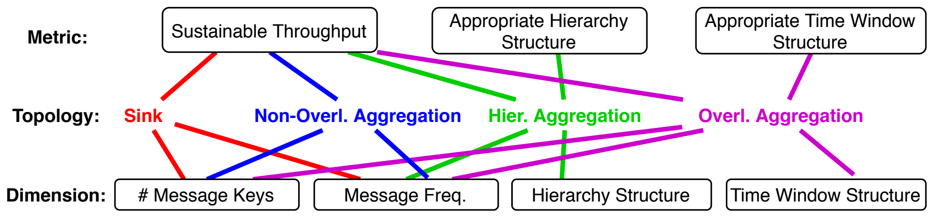

Fig. 3. Overview over the topologies, the associated scalability dimensions, and metrics.

variable numbers of message keys and message frequencies, within the bounds of this benchmark we set the number of message keys to 10.000 and the message frequency to 10 records per second.

Similarly to the first benchmark, for the other three benchmarks, we estimate the appropriate hierarchy- and time window structure, as well as the sustainable throughput with a threshold of 100 records.

4 Benchmark Execution

Within the bounds of this paper, we only execute our benchmark for the first topology *(1) Sink*. Therefore we implement the topology for our first benchmark as a Kafka Streams application. In this section, we describe the general setup, environment and the results of the execution.

4.1 Setup

The general setup for our benchmarking approach is visualized in Figure 4. The setup consists of the implementation of the *Sink* topology in form of a Kafka Streams application, horizontally scaled to the number of instances for the respective experiment. Additionally, we have a *Workload Generator*, based on [5], which is continuously producing data according to the respective workload definition and publishing this data to a Kafka topic. Subsequently, the data is processed by the topology. We use the *Java Management Extensions* (JMX) which enable the Kafka and Kafka Stream applications to expose their metrics. As a consequence, we are able to collect the metrics with the time series database *Prometheus*[1] and visualize them with the monitoring tool *Grafana*[2] in form of dashboards. As, according to Figure 3, the relevant metric for the first topology is the sustainable throughput, we use two JMX metrics[3] to determine the throughput and the average consumer lag across all partitions of a topic.

We analyze the behaviour of our application for 1, 2, 4, 8, and 16 application instances and we run each of our experiments for 360 seconds, with an additional

[1] https://prometheus.io/
[2] https://grafana.com/
[3] kafka_streams_stream_metrics_process_rate
kafka_consumer_consumer_fetch_manager_records_lag_avg

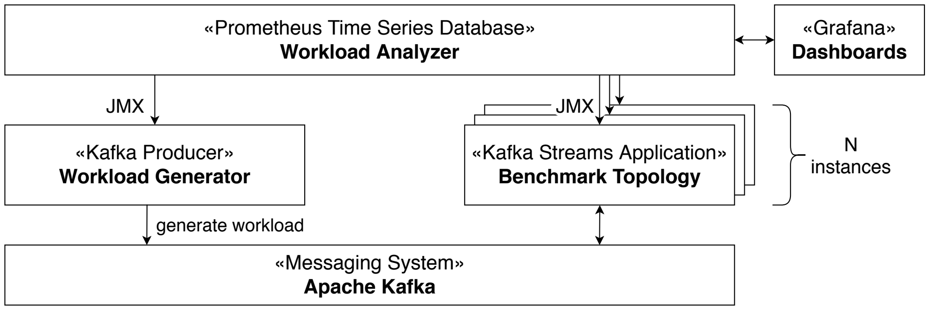

Fig. 4. Setup for the execution of the Benchmarks. All components are deployed on a 4 node Kubernetes cluster.

warmup time of 180 seconds for the *Java Virtual Machine* (JVM) [6]. Moreover, we limit the computational resources per instance to 2 GiB of memory, to prevent the workload generator from becoming a bottleneck. For the replication of our benchmark, we provide our source code and containerized executables [2].

4.2 Environment

We execute our benchmark on a 4 node Kubernetes cluster where each node is equipped with two *Intel Xeon Gold 6130 (2.1 GHz, 16 Cores, 64 threads)* CPUs with a total of 384 GB RAM. This way, we ensure that our Kafka Streams application can be scaled properly.

4.3 Results

The results of our benchmark are shown in Figure 5. Each blue point constitutes the sustainable throughput for respective number of instances. For 1 instance the sustainable throughput is 2^{13} and for 2, 4 and 8 instances it is 2^{13}, 2^{15} and 2^{16}. Additionally, the red trend line, computed with linear regression, allows us to estimate the sustainable throughput for amounts of instances between our measurements. We could not determine the sustainable throughput for 16 instances, since the workload generator became a bottleneck for 2^{17} messages per second.

In general, we can see that our Kafka Streams application scales linearly with the total number of sent messages per second. Although, for 1 instance and 2 instances, the sustainable throughput is the same. This may be caused by the gap between the individual amounts of messages per second in the different experiments growing too fast. Therefore, we plan to adjust our workload definition so that we consider a more fine grained series for the amount of messages per second.

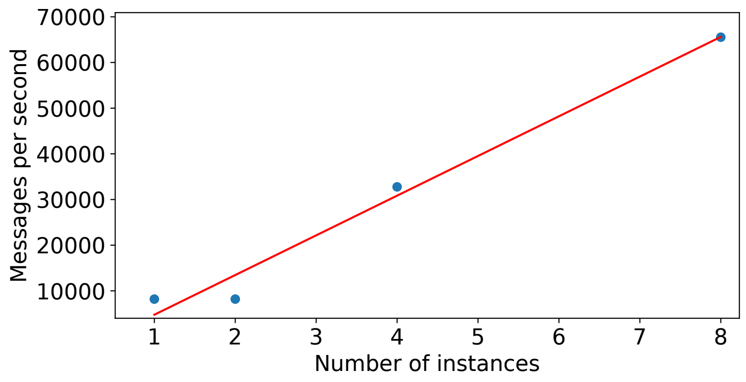

Fig. 5. The results of our benchmark execution. The blue points constitute the sustainable throughput for individual experiments. The red line represents an estimation of functional form of the scalability, computed with linear regression.

5 Conclusions and Future Work

In this paper we presented a specification that allows to benchmark stream processing frameworks in terms of scalability. Specifically, we designed four benchmarks that cover common use cases in realistic applications of stream processing. In addition to the description of respective topologies, this included the definition scalability dimensions, workloads and suitable metrics for the evaluation. Further, we implemented one of our benchmarks for the stream processing framework Kafka Streams and we executed on a Kubernetes cluster. Our results show that our benchmark allows us to assess that the implemented Kafka Streams application scales linearly with the amount of incoming messages. However, it may be appropriate to adjust the definition of the workload of the benchmark in order to achieve more precise information how the application scales with small numbers of instances. Further, it would be reasonable to design the workload generator as a distributed application to allow to produce much larger workloads without the generator becoming a bottleneck. This way, it would be possible to analyze the scalability for more than 8 instances. For the future, we plan to implement and execute the remaining three benchmarks also for Kafka Streams, in order to evaluate how the framework scales for more complex stream processing topologies. For this, the general architecture of our setup, using JMX in combination with Prometheus and Grafana can be used as a basis.

References

1. Campbell, R.H., Kamhoua, C.A., Kwiat, K.A.: Scalability, Workloads, and Performance: Replication, Popularity, Modeling, and GeoDistributed File Stores, pp. 133–159. IEEE (2018). https://doi.org/10.1002/9781119428497.ch5
2. Ehrenstein, S.: Seminar artifacts for: Scalability Benchmarking of Kafka Streams Applications (Feb 2020). https://doi.org/10.5281/zenodo.3665658

3. Harvan, M., Locher, T., Sima, A.C.: Cyclone: Unified stream and batch processing. In: 45th ICPPW. pp. 220–229 (2016). https://doi.org/10.1109/ICPPW.2016.42
4. Henning, S., Hasselbring, W., Möbius, A.: A scalable architecture for power consumption monitoring in industrial production environments. In: IEEE ICFC. pp. 124–133 (2019). https://doi.org/10.1109/ICFC.2019.00024
5. Henning, S., Hasselbring, W.: Scalable and reliable multi-dimensional aggregation of sensor data streams. In: IEEE International Conference on Big Data (2019)
6. Horký, V., Libič, P., Steinhauser, A., Tůma, P.: Dos and don'ts of conducting performance measurements in Java. In: Proceedings of the 6th ACM/SPEC ICPE. pp. 337–340. ACM (2015). https://doi.org/10.1145/2668930.2688820
7. Imai, S., Patterson, S., Varela, C.A.: Maximum sustainable throughput prediction for data stream processing over public clouds. In: 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. pp. 504–513 (2017). https://doi.org/10.1109/CCGRID.2017.105
8. Jeon, Y., Lee, K., Kim, H.: Distributed join processing between streaming and stored big data under the micro-batch model. IEEE Access **7**, 34583–34598 (2019). https://doi.org/10.1109/ACCESS.2019.2904730
9. Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., Markl, V.: Benchmarking distributed stream data processing systems. In: IEEE 34th ICDE. pp. 1507–1518 (2018). https://doi.org/10.1109/ICDE.2018.00169
10. v. Kistowski, J., Arnold, J.A., Huppler, K., Lange, K.D., Henning, J.L., Cao, P.: How to build a benchmark. In: Proceedings of the 6th ACM/SPEC ICPE. pp. 333–336. ACM (2015). https://doi.org/10.1145/2668930.2688819
11. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: Proceedings of the NetDB (2011)
12. Sax, M.J., Wang, G., Weidlich, M., Freytag, J.C.: Streams and tables: Two sides of the same coin. In: Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics. pp. 1:1–1:10. BIRTE '18, ACM (2018). https://doi.org/10.1145/3242153.3242155
13. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. SIGMOD Rec. **34**(4), 42–47 (2005). https://doi.org/10.1145/1107499.1107504