How to Measure Scalability of Distributed Stream Processing Engines?

Sören Henning
Kiel University
Kiel, Germany
soeren.henning@email.uni-kiel.de

Wilhelm Hasselbring
Kiel University
Kiel, Germany
hasselbring@email.uni-kiel.de

ABSTRACT

Scalability is promoted as a key quality feature of modern big data stream processing engines. However, even though research made huge efforts to provide precise definitions and corresponding metrics for the term scalability, experimental scalability evaluations or benchmarks of stream processing engines apply different and inconsistent metrics. With this paper, we aim to establish general metrics for scalability of stream processing engines. Derived from common definitions of scalability in cloud computing, we propose two metrics: a load capacity function and a resource demand function. Both metrics relate provisioned resources and load intensities, while requiring specific service level objectives to be fulfilled. We show how these metrics can be employed for scalability benchmarking and discuss their advantages in comparison to other metrics, used for stream processing engines and other software systems.

CCS CONCEPTS

• **General and reference** → *Metrics*; *Measurement*; • **Software and its engineering** → *Data flow architectures*; *Cloud computing*; *Software performance*.

KEYWORDS

scalability, metrics, stream processing, cloud computing

ACM Reference Format:
Sören Henning and Wilhelm Hasselbring. 2021. How to Measure Scalability of Distributed Stream Processing Engines?. In *Companion of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21 Companion), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3447545.3451190

1 INTRODUCTION

Over the last decade, architectures, models, and algorithms for processing continuous streams of data across multiple computing nodes became an active field of research, both in academia and industry. As a result, several state-of-the-art stream processing engines such as Flink [3], Spark [21], Storm [18], or Kafka Streams [17] emerged. All of these engines promote scalability as a key feature to cope with the volume and velocity of big data workloads.

ICPE '21 Companion, April 19–23, 2021, Virtual Event, France

ACM ISBN 978-1-4503-8331-8/21/04…$15.00
https://doi.org/10.1145/3447545.3451190

Lots of research exists on improving performance, correctness, or fault-tolerance of stream processing engines while preserving this scalability. Empirical evaluations or benchmarks of different engines or configurations are, therefore, often evaluating scalability [7]. However, even though such studies share similar fundamental understandings of scalability, they no not apply common metrics or measurement methods for scalability. At the same time, research in software performance engineering made huge efforts to provide precise definitions and metrics for scalability in distributed systems [1, 4, 9] and, more recently, in cloud computing [8, 14, 16].

The goal of this paper is to build a solid foundations for scalability evaluations and benchmarks of stream processing engines. We follow up on general definitions of scalability in cloud computing and transfer them to stream processing. Based on these definitions, we derive our two Theodolite scalability metrics:

(1) Our *demand* metric describes how resource demands evolve with increasing load.
(2) Our *capacity* metric describes how load capacity evolves with increasing provisioned resources.

After a brief summary on scalability and stream processing in Section 2, we formally define our metrics in Section 3. We discuss our proposed metrics in detail and compare them with scalability metrics of related work in Section 4. Section 5 concludes this paper.

2 BACKGROUND

In this section, we summarize fundamental scalability definitions and present how state-of-the art frameworks process continuous streams of data in a distributed fashion.

2.1 Definition of Scalability

Initial definitions for scalability of distributed systems where presented by Bondi [1] and Jogalekar and Woodside [9], which were later generalized by Duboc et al. [4]. More recently, such definitions have been specified to target the peculiarities of scalability in cloud computing [8, 14, 16].

A definition of scalability in cloud computing is, for example, given by Herbst et al. [8], which states that "scalability is the ability of [a] system to sustain increasing workloads by making use of additional resources". In a subsequent work [20], the authors further specify this and highlight that scalability is characterized by the following three attributes:

Load intensity is the input variable a system is subject to. Scalability is evaluated within a range of load intensities.
Service levels objectives (SLOs) are measurable quality criteria that have to be fulfilled for every load intensity.

Provisioned resources can be increased to meet the SLOs if load intensities increase.

A software system can be considered scalable within a certain load intensity range if for all load intensities within that range it is able to meet its service level objectives, potentially by using additional resources. Such a definition targets both horizontal and vertical scalability [20].

In cloud computing also the distinction between scalability and elasticity is important. Elasticity also takes temporal aspects into account and describes how fast and how precise a system adapts its provided resources to changing load intensities [8]. Scalability, on the other hand, is a prerequisite for elasticity, but only describes whether increasing load intensities can be handled in principle.

2.2 Distributed Stream Processing

Modern stream processing engines [5] process data in jobs, where a job is defined as a dataflow graph of processing operators. They can be started with multiple instances (e.g., on different computing nodes, containers, or with multiple threads). For each job, each instance processes only a portion of the data. Whereas isolated processing of data records is not affected by the assignment of data portions to instances, processing that relies on previous data records (e.g., aggregations over time windows) requires the management of state. Similar to the MapReduce programming model, keys are assigned to records and the stream processing engines guarantee that all records with the same key are processed by the same instance. Hence, no state synchronization among instances is required. If a processing operator changes the record key and a subsequent operator performs a stateful operation, the stream processing engine splits the dataflow graph into subgraphs, which can be processed independently by different instances.

It is quite common that stream processing engines read and write data from and to a messaging system. Some messaging systems such as Apache Kafka have the additional advantage that they already partition data according to keys.

3 THE THEODOLITE SCALABILITY METRICS

In this section, we derive our Theodolite [7] scalability metrics and show how they can be used to benchmark the scalability of distributed stream processing engines.

3.1 Scalability in Stream Processing

As summarized in Section 2, scalability can be described by the attributes load intensity, provisioned resources, and service level objectives. In the following, we characterize these attributes in the context of stream processing.

Load intensity. Load on a stream processing application corresponds to messages coming from a central messaging system. Load can have multiple dimensions, such as number of messages per unit time or size of messages. Depending on the stream processing engine, it is likely that they scale differently depending on the load dimension. As stream processing engines employ partitioning based on keys as primary means for parallelization, a sensible load dimension is, for example, the number of distinct message keys per unit time. We denote the set of possible load intensities for a given dimension with L and the range of load intensities scalability should be evaluated for with $\hat{L} \subseteq L$.

Provisioned resources. Modern stream processing engines are mainly scaled horizontally by varying the number of instances. Traditionally, this is the amount of virtual or physical computing nodes. Nowadays with containerized deployments, the underlying hardware is further abstracted and stream processing engines are often scaled with the amount of (e.g., Docker) containers. We generalize different resource scaling options and denote the set of resources that can be provisioned with R.

Service levels objectives (SLOs). From a user-perspective, often the only requirement is that all[1] messages are processed in time. As a measure of this, we propose our lag trend metric (Section 3.3), which describes how the number of queued messages evolves. However, also other or additional SLOs can be used.

We define the set of all SLOs as S and denote an SLO $s \in S$ as Boolean-valued function $\text{slo}_s : L \times R \to \{\text{false}, \text{true}\}$ with $\text{slo}_s(l, r) = \text{true}$ if a stream processing engine with r resource amounts does not violate SLO s when processing load intensity l.

3.2 Scalability Metrics

Based on the previous characterization of scalability, we propose two functions as metrics for scalability. In many cases, both functions are inverse to each other. However, we expect both metrics to have advantages, as discussed in Section 4.2.

Resource Demand Metric. The first function maps load intensities to the resources, which are at least required for processing these loads. We denote the metric as demand : $\hat{L} \to R$, defined as:

$$\forall l \in \hat{L} : \text{demand}(l) = \min\{r \in R \mid \forall s \in S : \text{slo}_s(l, r) = \text{true}\}$$

The *demand* metric shows, for example, whether the resource demand increases linearly, whether disproportionately many resources are required (e.g., exponentially), or whether a system only scales up to a certain point, i.e., there is a load which can not be handled even though further resources are added.

Load Capacity Metric. Our second metric maps provisioned resource amounts to the maximum load, these resources can process. We denote this metric as capacity : $R \to \hat{L}$, defined as:

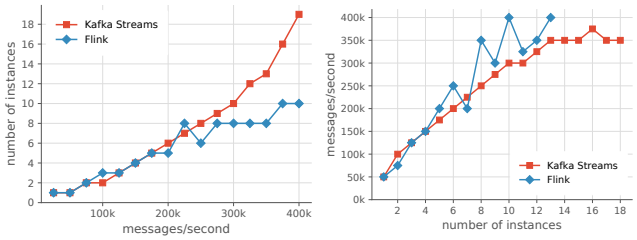
$$\forall r \in R : \text{capacity}(r) = \max\{l \in \hat{L} \mid \forall s \in S : \text{slo}_s(l, r) = \text{true}\}$$

Analogously to the *demand* metric, the *capacity* metric shows at which rate processing capabilities increase with increasing resources. It also allows determining whether a system only scales to a certain point, which is when with additional resources, the load capacity does not further increase or is even decreasing.

3.3 Lag Trend Metric

The lag of a stream processing job describes how many messages are queued in the messaging system, which have not been processed yet. Our lag trend metric describes the average increase (or decrease) of the lag per second. It can be measured by monitoring the lag and

[1]Note that in stream processing the requirement to process *all* messages is often relaxed to *preferably all* to increase performance.



(a) Resource demand **(b) Load capacity**

Figure 1: Scalability of Kafka Streams and Flink with our Theodolite scalability metrics

computing a trend line using linear regression. The slope of this line is the lag trend.

The lag trend metric can be used to define an SLO, whose function evaluates to true if the lag trend does not exceed a certain threshold. Ideally, this threshold should be 0 as a non-positive lag trend means that messages can be processed as fast as they arrive. However, it could make sense to allow for a small increase as even when observing an almost constant lag, a slightly rising or falling trend line will be computed due to outliers.

We expect that in most cases, checking the lag trend alone suffices as an SLO. The architectures of modern stream processing engines make it unlikely that SLOs such as a maximum tolerable processing latency can be fulfilled by scaling provisioned resources.

3.4 Benchmarking Example

In the following, we show how our proposed metrics can be used for benchmarking scalability of stream processing engines. We employ our Theodolite benchmarking framework [7] to benchmark the scalability of Kafka Streams and Flink. The Theodolite framework tests a set of load intensities against a set of resource amounts and validates whether configured SLOs are met. Hence, it approximates our proposed scalability metrics.

Figure 1 shows the benchmark results of both engines for our *demand* metric as well as for the *capacity* metric. Load intensities $\hat{L}$ are specified as messages with distinct keys per second. Provisioned resources R are specified as number of stream processing engine instances, each executed in an own Kubernetes Pod, restricted to one CPU core. We use only one SLO, which is based on our lag trend metric and evaluates to true if the lag trend does increase by more than 2 000 records per second. The shown benchmark results are extensions of the *downsampling* benchmarks from our previous study [7], which also provides further information regarding the experimental setup.

4 DISCUSSION WITH RELATED METRICS

In this section, we discuss our proposed scalability metrics and relate them to metrics in related work on evaluating scalability.

4.1 Scalability as a Function

As in most studies [2, 15, 19], both our metrics describe scalability as a function instead of a scalar. Sanders et al. [16] highlight that scalability is a function as usually capacity does not grow at a constant rate with additional resources [2, 7, 19]. However, they remark that scalability can be measured as a scalar within a range.

A downside of having a function metric is that it makes it difficult to create a rating, which orders systems by their scalability. Such a rating is desired for benchmarking scalability of different stream processing engines. A possible solution is to cluster systems with similar functions (e.g., systems that scale linearly) and then compare their derivative or axis intersection.

The Universal Scalability Law [6] describes a general performance model of system scalability. It is based on the assumption that scalability of arbitrary systems can be described using a non-linear rational function with two system-specific coefficients, representing contention and coherency. If applicable to stream processing, these coefficients could serve to rate engines. However, it remains unclear how well these coefficients can be derived from empirical measurements when considering capacity as discrete values.

4.2 Function of Load vs. Function of Resources

While with our *demand* metric, scalability is described as a function of load, our *capacity* metric describes it as a function of resources. Although often both metric functions are inverse to each other, each metric has specific advantages.

We observe that in experimental studies, evaluating scalability as a function of resources is more frequent than evaluations as a function of load [2, 10, 15, 19]. An issue with those metrics is that they contradict common scalability definitions, which concern the system's behavior if subject to increasing load (see Section 2). Our *demand* metric overcomes this and shows explicitly whether a system is able to handle increasing load intensities.

The main advantage of our *capacity* metric on the other hand is that it allows to express situations where capacity drops with increasing resources. This can occur if coordination between instances outweigh parallelization benefits [6] or if particular resource configurations are overly efficient.

The Universal Scalability Law [6] uses the scale-up metric. It is a function of resources and describes the relative capacity, defined as the percentage increase in capacity of n resources in comparison to 1 resource. Such normalization based on a reference value (e.g., 1 instance or a certain load intensity) can be applied to both our metrics. It could allow to exclude resource efficiency of systems.

4.3 Resources as a Function of Load

With our *demand* scalability metric, we describe scalability as how resource demands evolve with increasing load intensities. According to our definition of scalability, it is also sensible to describe it as how service levels evolve with increasing load intensity. For example, Kossmann et al. [12] evaluate how the amount of processed records evolve with increasing load of cloud services for transaction processing. Their study shows that while for some systems the throughput grows proportionally with increasing load, thus, all records are processed (SLO is met), for other systems, the amount of processed messages does not further increase (SLO is not met).

The authors evaluate cloud services, which are automatically scaled in the background by the cloud provider. In such deployments, measuring how the underlying resource demands evolve does usually not provide much benefit as resource scaling is out

of the user's control. When considering manual resource scaling deployments, however, provisioned resources are explicitly chosen to meet SLOs. Thus, we expect it to be more relevant to measure how resource demands evolve.

4.4 Capacity as Discrete Values

With our *capacity* scalability metric, we propose to determine the load capacity as a discrete value from a given set of load intensities. In many scalability studies, however, capacity is measured as a continuous value. While this might be feasible for databases [13] or batch processing [6], we consider it to be difficult to achieve in stream processing. To determine capacity as a continuous value, we observe basically two options, both having weaknesses.

The first option is to generate a constant load and measure the throughput, i.e., how much of this load is processed. Although not explicitly described, it looks like this technique was applied in the scalability evaluations of stream processing engines by Karakaya et al. [10] and Nasiri et al. [15] A first weakness of this approach is that the generated load must be sufficiently high as otherwise the throughput would be bounded by the generated load intensity instead of by the capacity of the provisioned resources. Further, it is unclear how much higher the load has to be. As the throughput may vary strongly [7, 11], it may temporarily be higher than the generated load, causing the stream processing engine to not operate at its maximum. Finally, this approach is based on the assumption that for a given amount of provisioned resources the throughput is always the same, independent of the generated load. However, this assumption is questionable, unless explicitly evaluated. It is likely that a high load on the messaging system also influences response times or chunk sizes.

A second option to measure load capacity as continuous values is to steadily increase the load intensity for a given resource amount and determine at which load intensity SLOs are not fulfilled anymore. A similar method is taken by Karimov et al. [11] for their sustainable throughput metric. A weakness of this approach is that dependencies between different load intensities are difficult to rule out. We observe that even with constant load the throughput of stream processing engines varies strongly and, additionally, increases after some warm-up period [7]. To determine a reasonable load capacity, the monitored throughput has therefore to be averaged over some period of time. Furthermore, when increasing the load without restarting experiments, stream processing engines or related software infrastructure might perform optimizations for lower load intensities, which are not ideal for higher loads.

When measuring scalability with our proposed metrics, we therefore strongly recommend to evaluate the slo_s functions in isolated experiments for one resource configuration and a constant load.

5 CONCLUSIONS AND FUTURE WORK

With this paper, we propose two metrics for scalability of distributed stream processing engines, derived from common scalability definitions. While our *demand* metric describes how resource demands evolve with increasing load, our *capacity* metric describes how load capacity evolves with increasing provided resources. Both metrics share the explicit definition of SLOs that always have to be fulfilled and the definition of load intensity and provisioned resources as discrete spaces. We discuss advantages of these decisions in comparison to other metrics as well as compared to each other.

We expect these metrics to lay a solid foundation for benchmarking scalability of stream processing engines. For future work, we also plan to evaluate whether the Universal Scalability Law can be applied to create a ranking of such engines.

ACKNOWLEDGMENTS

This research is funded by the German Federal Ministry of Education and Research (BMBF, grant no. 01IS17084B).

REFERENCES

[1] A. B. Bondi. 2000. Characteristics of Scalability and Their Impact on Performance. In *Proc. International Workshop on Software and Performance*. https://doi.org/10.1145/350391.350432
[2] G. Brataas, N. Herbst, S. Ivanšek, and J. Polutnik. 2017. Scalability Analysis of Cloud Software Services. In *Proc. International Conference on Autonomic Computing*. https://doi.org/10.1109/ICAC.2017.34
[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. 2015. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).
[4] L. Duboc, D. Rosenblum, and T. Wicks. 2007. A Framework for Characterization and Analysis of Software System Scalability. In *Proc. ESEC/FSE*. https://doi.org/10.1145/1287624.1287679
[5] M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos. 2020. A Survey on the Evolution of Stream Processing Systems. arXiv:2008.00842 [cs.DC]
[6] N. J. Gunther, P. Puglia, and K. Tomasette. 2015. Hadoop Superlinear Scalability. *Commun. ACM* 58, 4 (2015). https://doi.org/10.1145/2719919
[7] Sören Henning and Wilhelm Hasselbring. 2021. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures. *Big Data Research* 25 (2021), 100209. https://doi.org/10.1016/j.bdr.2021.100209
[8] N. R. Herbst, S. Kounev, and R. Reussner. 2013. Elasticity in Cloud Computing: What It Is, and What It Is Not. In *Proc. Int. Conference on Autonomic Computing*.
[9] P. Jogalekar and M. Woodside. 2000. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems* 11, 6 (2000). https://doi.org/10.1109/71.862209
[10] Z. Karakaya, A. Yazici, and M. Alayyoub. 2017. A Comparison of Stream Processing Frameworks. In *Proc. International Conference on Computer and Applications*. https://doi.org/10.1109/COMAPP.2017.8079733
[11] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl. 2018. Benchmarking Distributed Stream Data Processing Systems. In *Proc. International Conference on Data Engineering*. https://doi.org/10.1109/ICDE.2018.00169
[12] D. Kossmann, T. Kraska, and S. Loesing. 2010. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. In *Proc. SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/1807167.1807231
[13] J. Kuhlenkamp, M. Klems, and O. Röss. 2014. Benchmarking Scalability and Elasticity of Distributed Database Systems. *Proc. VLDB Endow.* 7, 12 (2014). https://doi.org/10.14778/2732977.2732995
[14] S. Lehrig, H. Eikerling, and S. Becker. 2015. Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics. In *Int. Conf. Quality of Software Architectures*. https://doi.org/10.1145/2737182.2737185
[15] H. Nasiri, S. Nasehi, and M. Goudarzi. 2019. Evaluation of distributed stream processing frameworks for IoT applications in Smart Cities. *Journal of Big Data* 6, 52 (2019). https://doi.org/10.1186/s40537-019-0215-2
[16] R. Sanders, G. Brataas, M. Cecowski, K. Haslum, S. Ivanšek, J. Polutnik, and B. Viken. 2015. CloudStore – Towards Scalability Benchmarking in Cloud Computing. *Procedia Comput. Sci.* 68 (2015). https://doi.org/10.1016/j.procs.2015.09.225
[17] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag. 2018. Streams and Tables: Two Sides of the Same Coin. In *Proc. International Workshop on Real-Time Business Intelligence and Analytics*. https://doi.org/10.1145/3242153.3242155
[18] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. 2014. Storm@twitter. In *Proc. SIGMOD International Conference on Management of Data*. https://doi.org/10.1145/2588555.2595641
[19] Vikash, L. Mishra, and S. Varma. 2020. Performance evaluation of real-time stream processing systems for Internet of Things applications. *Future Generation Computer Systems* 113 (2020). https://doi.org/10.1016/j.future.2020.07.012
[20] A. Weber, N. Herbst, H. Groenda, and S. Kounev. 2014. Towards a Resource Elasticity Benchmark for Cloud Environments. In *Proc. International Workshop on Hot Topics in Cloud Service Scalability*. https://doi.org/10.1145/2649563.2649571
[21] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. 2013. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proc. Symposium on Operating Systems Principles*. https://doi.org/10.1145/2517349.2522737