Hasso Plattner Institute for Digital Engineering
at the University of Potsdam
Enterprise Platform and Integration Concepts Research Group

A Benchmark for Enterprise Stream Processing Architectures

Dissertation
submitted in partial fulfillment of the requirements for the degree of

Doctor rerum naturalium (Dr. rer. nat.)

in the academic discipline of *practical computer science*
to the Digital Engineering Faculty
at the University of Potsdam

Guenter Hesse, M.Sc.

Supervisors:

Prof. Dr. h.c. mult. Hasso Plattner, University of Potsdam
Prof. Dr. Manfred Hauswirth, Technical University of Berlin
Prof. Dr. Matthias Weidlich, Humboldt University of Berlin

Potsdam, November 2021

Published online on the
Publication Server of the University of Potsdam:
https://doi.org/10.25932/publishup- 56600
https://nbn-resolving.org/urn:nbn:de:kobv:517-opus4- 566000

ABSTRACT

Data stream processing systems (DSPSs) are a key enabler to integrate continuously generated data, such as sensor measurements, into enterprise applications. DSPSs allow to steadily analyze information from data streams, e.g., to monitor manufacturing processes and enable fast reactions to anomalous behavior. Moreover, DSPSs continuously filter, sample, and aggregate incoming streams of data, which reduces the data size, and thus data storage costs.

The growing volumes of generated data have increased the demand for high-performance DSPSs, leading to a higher interest in these systems and to the development of new DSPSs. While having more DSPSs is favorable for users as it allows choosing the system that satisfies their requirements the most, it also introduces the challenge of identifying the most suitable DSPS regarding current needs as well as future demands. Having a solution to this challenge is important because replacements of DSPSs require the costly re-writing of applications if no abstraction layer is used for application development. However, quantifying performance differences between DSPSs is a difficult task. Existing benchmarks fail to integrate all core functionalities of DSPSs and lack tool support, which hinders objective result comparisons. Moreover, no current benchmark covers the combination of streaming data with existing structured business data, which is particularly relevant for companies.

This thesis proposes a performance benchmark for enterprise stream processing called *ESPBench*. With enterprise stream processing, we refer to the combination of streaming and structured business data. Our benchmark design represents real-world scenarios and allows for an objective result comparison as well as scaling of data. The defined benchmark query set covers all core functionalities of DSPSs. The benchmark toolkit automates the entire benchmark process and provides important features, such as query result validation and a configurable data ingestion rate.

To validate ESPBench and to ease the use of the benchmark, we propose an example implementation of the ESPBench queries leveraging the *Apache Beam* software development kit (SDK). The Apache Beam SDK is an abstraction layer designed for developing stream processing applications that is applied in academia as well as enterprise contexts. It allows to run the defined applications on any of the supported DSPSs. The performance impact of Apache Beam is studied in this dissertation as well. The results show that there is a significant influence that differs among DSPSs and stream processing applications. For validating ESPBench, we use the example implementation of the ESPBench queries developed using the Apache Beam SDK. We benchmark the implemented queries executed on three modern DSPSs: *Apache Flink*, *Apache Spark Streaming*, and *Hazelcast Jet*. The results of the study prove the functioning of ESPBench and its toolkit. ESPBench is capable of quantifying performance characteristics of DSPSs and of unveiling differences among systems.

The benchmark proposed in this thesis covers all requirements to be applied in enterprise stream processing settings, and thus represents an improvement over the current state-of-the-art.

ZUSAMMENFASSUNG

Data Stream Processing Systems (DSPSs) sind eine Schlüsseltechnologie, um kontinuierlich generierte Daten, wie beispielsweise Sensormessungen, in Unternehmensanwendungen zu integrieren. Die durch DSPSs ermöglichte permanente Analyse von Datenströmen kann dabei zur Überwachung von Produktionsprozessen genutzt werden, um möglichst zeitnah auf ungewollte Veränderungen zu reagieren. Darüber hinaus filtern, sampeln und aggregieren DSPSs einkommende Daten, was die Datengröße reduziert und so auch etwaige Kosten für die Datenspeicherung.

Steigende Datenvolumen haben in den letzten Jahren den Bedarf für performante DSPSs steigen lassen, was zur Entwicklung neuer DSPSs führte. Während eine große Auswahl an verfügbaren Systemen generell gut für Nutzer ist, stellt es potentielle Anwender auch vor die Herausforderung, das für aktuelle und zukünftige Anforderungen passendste DSPS zu identifizieren. Es ist wichtig, eine Lösung für diese Herausforderung zu haben, da das Austauschen von einem DSPS zu teuren Anpassungen oder Neuentwicklungen der darauf laufenden Anwendungen erfordert, falls für deren Entwicklung keine Abstraktionsschicht verwendet wurde. Das quantitative Vergleichen von DSPSs ist allerdings eine schwierige Aufgabe. Existierende Benchmarks decken nicht alle Kernfunktionalitäten von DSPSs ab und haben keinen oder unzureichenden Tool-Support, was eine objektive Ergebnisberechnung hinsichtlich der Performanz erschwert. Zudem beinhaltet kein Benchmark die Integration von Streamingdaten und strukturierten Geschäftsdaten, was ein besonders für Unternehmen relevantes Szenario ist.

Diese Dissertation stellt *ESPBench* vor, einen neuen Benchmark für Stream Processing-Szenarien im Unternehmenskontext. Der geschäftliche Kontext wird dabei durch die Verbindung von Streamingdaten und Geschäftsdaten dargestellt. Das Design von ESPBench repräsentiert Szenarien der realen Welt, stellt die objektive Berechnung von Benchmarkergebnissen sicher und erlaubt das Skalieren über Datencharakteristiken. Das entwickelte Toolkit des Benchmarks stellt wichtige Funktionalitäten bereit, wie beispielsweise die Automatisierung den kompletten Benchmarkprozesses sowie die Überprüfung der Abfrageergebnisse hinsichtlich ihrer Korrektheit. Um ESPBench zu validieren und die Anwendung weiter zu vereinfachen, haben wir eine Beispielimplementierung der Queries veröffentlicht. Die Implementierung haben wir mithilfe des in Industrie und Wissenschaft eingesetzten Softwareentwicklungsbaukastens *Apache Beam* durchgeführt, der es ermöglicht, entwickelte Anwendungen auf allen unterstützten DSPSs auszuführen. Den Einfluss auf die Performanz des Verwendens von Apache Beam wird dabei ebenfalls in dieser Arbeit untersucht. Weiterhin nutzen wir die veröffentlichte Beispielimplementierung der Queries um drei moderne DSPSs mit ESPBench zu untersuchen: *Apache Flink*, *Apache Spark Streaming* und *Hazelcast Jet*. Der Ergebnisse der Studie verdeutlichen die Funktionsfähigkeit von ESPBench und dessen Toolkit. ESPBench befähigt Performanzcharakteristiken von DSPSs zu quantifizieren und Unterschiede zwischen Systemen aufzuzeigen.

Der in dieser Dissertation vorgestellte Benchmark erfüllt alle Anforderungen, um in Stream Processing-Szenarien im Unternehmenskontext eingesetzt zu werden und stellt somit eine Verbesserung der aktuellen Situation dar.

ACKNOWLEDGEMENTS

All good things must come to an end, this dissertation being no exception. It would not have been possible without the many inspiring and supporting people I met along the way. First, I would like to thank my advisor Prof. Hasso Plattner, who enabled my research through his extraordinary commitment and support for the academic world. His tremendous energy and experiences in the area of enterprise software gave me great motivation for my research endeavor.

I further want to thank Prof. Felix Naumann, Prof. Tilmann Rabl, Prof. Manfred Hauswirth, and Prof. Matthias Weidlich for their support and valuable feedback on my work. Moreover, I want to thank the many people at the *Enterprise Platform and Integration Concepts* chair that supported and inspired me during the last couple of years. Thank you Dr. Matthias Uflacker and Dr. Michael Perscheid for constantly fostering my research as chair representatives, thanks to all my fellow research associates for the motivation and discussions. I especially want to thank Stefan Halfpap, Christoph Matthies, and Jan Kossmann, with whom I shared the office during the last couple of years as well as many great moments in private gatherings. I also want to thank Benjamin Reissaus, whom I could give advice and support for his master thesis, Kelvin Glaß for his support on the Apache Beam analysis, as well as Marilena Davis, Andrea Lange, Marcus Wacke, and all the other great people at the chair who made sure that everything is running smoothly.

I finally want to thank my former colleagues at SAP, my friends who have not been mentioned so far, my partner Dr. Milena Quittnat, as well as my family, who supported me in challenging and fun times of my life. I especially want to thank my parents Veronika and Heinz Hesse, who always gave me the freedom to pursue whichever projects I chose.

CONTENTS

1

INTRODUCTION

Due to the increasing volumes of generated streams of data caused by accelerating digitization and automation, new challenges and opportunities in the area of data processing arise. An example of this development with respect to data is the media services provider *Spotify*, which processes different kinds of events such as user registrations or any other kind of interaction with Spotify apps. While Spotify encountered a data stream with about 1.5 million events/second in 2016, this number increased to 8 million events/second in 2018 [JS19, Ser19]. Such large occurring data masses are not phenomena that are exclusively visible at Spotify. On a daily basis, a single sensor on a *General Electric* jet engine creates 500 GB of data [DD13], the sensors of an oil field generate more than 1 TB [HSK+19], and *Facebook's* data warehouse receives 600 TB [VW14]. These examples highlight the huge data masses that companies across different industry sectors face today.

In certain industry sectors, such as mechanical engineering or automotive, Industry 4.0 and Internet of Things (IoT) have gained traction to describe developments that bring new possibilities with respect to business models. New IoT technologies are being created, sensor accuracy increases, and new analytical IT systems are being developed that allow to query huge amounts of data within seconds. On the economic side, a substantial and steady price decrease for sensor IoT equipment has occurred, which is expected to continue in the following years [McK15, CMN18]. These developments foster an increased deployment of IoT technologies in companies and as a result, more IoT data is available to enterprises [WL14]. Monetarily expressed, the total global worth of IoT technology is expected to reach USD 6.2 trillion by 2025 [Int14].

One of the sectors involved most in IoT is industrial manufacturing [Int14]. That is the case because manufacturers see great potential in unlocking further efficiency potential and reducing costs at their production facilities through these technologies [MCB+15]. An example of a factory that already captures high volumes of data with high velocity is the *General Electric* battery production plant in New York State. There are 10,000 different data attributes created at this plant, some as often as every 250 ms [WL14]. An ultrasonic sensor production plant in Hungary creates about 170 GB of data per day [NOE+18]. Manufacturing equipment, such as a single saw, generates 50,000 messages or 1.2 GB of data on a daily basis[1]. Injection molding machines even produce up to multiple terabytes of sensor data in 24 hours [HVN16]. Such data streams provide detailed information about the current state of machines and allow timely reactions to events, e.g., failures or altered temperatures. However, added value can only be created if the data stream is analyzed within an adequate timeframe. The value and relevance of high-frequency streaming data rapidly decrease with the amount of time that has passed since its creation. In case of a failure event, actions need to be taken as soon as possible to minimize its negative impact. Despite the importance of IoT data, industry studies reveal that most of the data gathered today is not used at all. Furthermore, the remaining data that is incorporated and analyzed is not exploited in its entirety [MCB+15].

Analyzing captured IoT data and combining it with existing structured business data leads to a holistic view of the value chain, i.e., of the entire process of creating value in companies. Examples of such business data are supplier or customer information. This combination of data from different technical levels is known as vertical data integration. A practical example of vertical integration is a printing machine, where the humidity is regulated depending on sensor measurements (streaming data) as well as the used colors and paper (structured manufacturing data) to optimize the print quality and reduce the number of defective goods. Another example is the healthcare service provider *BJC HealthCare*, which adopts IoT technologies for their inventory and supply chain management to enable a live-tracking of their stocks. This detailed inventory overview allows for more need-specific purchases and resulted in a reduction of onsite stock at each facility by 23 % [Ltd19].

Additional to IoT, Industry 4.0 is the other related term in the context of manufacturing that gained attention in the past years. This is due to the large estimated added value achievable for businesses, especially when integrating available data, both horizontally and vertically. Such data integrations in the

[1] `https://crate.io/customers/senseforce-iot/`, accessed: 2020-11-10

context of Industry 4.0 are conceptually visualized in Figure 1.1. Horizontal data integration is a traditional topic in which software companies and their customers have a comparatively high degree of experience. More recent developments exist in the area of inter-company integrations. Besides, there is the vertical integration of technical data and business data. The technical data, such as sensor measurements, is thereby produced in high volumes over a short period of time. Since sensor values only have a local semantic, they need to be integrated with enterprise data to get a global semantic, and to gain in business value.

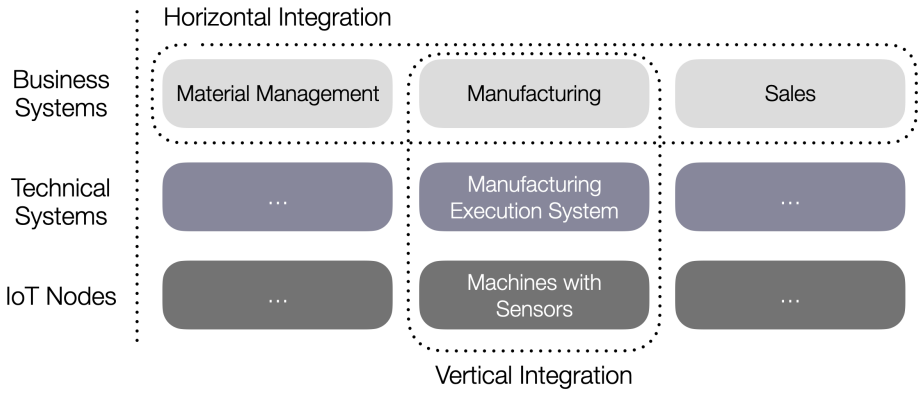


Figure 1.1: Conceptual overview of horizontal and vertical data integration in the context of Industry 4.0 ([HSMU19])

A survey conducted by McKinsey & Company in January 2016 amongst companies in the United States, Germany, and Japan with at least 50 employees, highlights the significance of Industry 4.0 [McK16]. Particularly, the assessments of 300 domain experts on this topic were gathered using 23 questions, which asked for, e.g., importance or significance ratings. The involved companies cover various industries, such as automotive suppliers, chemicals, and healthcare. As one outcome, the study reveals that the majority of companies expect Industry 4.0 to increase their competitiveness [McK16], which highlights its business value. According to a more recent survey of more than 400 companies worldwide, the COVID-19 pandemic even caused an increase regarding the perceived value of Industry 4.0 at most of the companies [ADKM21]. One of the identified key challenges for adopting Industry 4.0 concepts is the integration of different data sources. Especially with the emerging importance of IoT data, the fairly old challenge of integrating disparate data sources becomes highly relevant [McK16, RS94]. Data Stream Processing Systems (DSPSs) represent a technology suitable for tackling the challenge of data integration.

1.1 Data Stream Processing and the Need for a New Performance Benchmark

DSPSs are capable of handling streams of data with high volume and velocity that are created by, e.g., IoT devices. These systems analyze streams of data on the fly using continuously running queries, i.e., queries that are executed for a potentially infinite time and accordingly produce results during this period. DSPSs filter, aggregate, or further transform data streams and can combine other data sources, such as databases storing business data, with streaming data. Figure 1.2 visualizes a typical stream processing architecture. This architecture enables the analysis of streaming data and its combination with business data. Moreover, data transformed by a data stream processing application can be persisted in, e.g., another message queue, a database, or a data lake.

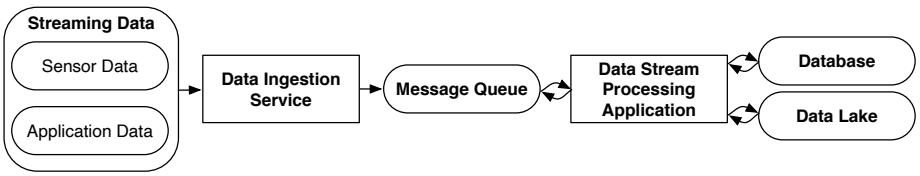


Figure 1.2: A common data stream processing architecture visualized in Fundamental Modeling Concepts (FMC[KGT05], based on[2])

High volumes of streaming data, such as sensor measurements or application data (e.g., user click events), are ingested into a message queue, which is used to make the data broadly available. A stream processing application running on a DSPS retrieves the streaming data via the queue, analyzes or alters it, and stores results in a storage solution if desired. DSPSs enable the management of continuously generated data and thus, making use of this data. This capability of handling potentially indefinite data sets has led to the crucial role DSPSs play in the data-intensive scenarios faced by enterprises today.

The increased importance of streaming data, and thus DSPSs, fostered the development of a multitude of new systems and frameworks in recent years [IAM+19, HL15], such as *Kafka Streams*[3], *Hazelcast Jet*[4], *Apache*

[2] `https://www.ververica.com/what-is-stream-processing`, accessed: 2020-11-30
[3] `https://kafka.apache.org/documentation/streams/`,accessed:2021-03-08
[4] `https://jet-start.sh`, accessed: 2020-11-10

Flink [CKE+15], *Apache Spark Streaming* [ZDL+13], and *Apache Apex*[5]. Next to these open-source systems, there are also commercial closed-source DSPSs, like *IBM Streams*[6], and *Google Cloud Dataflow*[7]. Contrary to the group of recently developed DSPSs, *Aurora* [ACÇ+03] and *STREAM* [ABB+03b], for instance, are early DSPSs that were already presented in the early 2000's.

While having more DSPSs is favorable for users as it allows choosing the system that satisfies their requirements the most, it also introduces the challenge of identifying the most suitable DSPS regarding current needs as well as future demands. Having a solution to this challenge is important, as replacements of DSPSs require costly re-writes of applications. The common way of quantitatively comparing systems to decide which one to choose is performance benchmarking. However, quantifying performance differences between DSPSs is currently a difficult task, because existing benchmarks fail to integrate all core functionalities of DSPSs and lack tool support, which hinders objective result calculations. Moreover, no current benchmark covers the combination of streaming data with structured business data, which is particularly relevant for companies.

A reason for the lack of satisfying DSPSs benchmarks is complexity. Compared to the thematically close domain of database benchmarks, where many well-established and standardized performance benchmarks exist, a DSPS benchmark requires more tool support. For instance, one or multiple data streams need to be created to simulate a stream processing scenario. The streams have to ingest data at reproducible rates in order to enable comparable benchmark results. For scaling this data ingestion and decoupling it from the system under test (SUT), performance benchmarks often incorporate a message queue. This additional component also adds to the benchmark's complexity. Moreover, there is not only a single query result that needs to be checked with respect to its correctness, but multiple results as the queries run and produce results continuously. These requirements lead to an increased effort for building a performance benchmark for DSPSs compared to benchmarks of related domains, such as database management systems.

We propose to close this gap with *ESPBench*, a new performance benchmark for enterprise stream processing architectures that is presented within this dissertation.

[5] `https://apex.apache.org/docs/apex/`, accessed: 2020-11-10
[6] `https://www.ibm.com/de-en/marketplace/stream-computing`, accessed: 2020-11-10
[7] `https://cloud.google.com/dataflow/`, accessed: 2020-11-10

1.2 Research Questions

The identified need for a new enterprise stream processing benchmark leads to the following research question:

What is a suitable architecture of an enterprise stream processing benchmark that satisfies the demands on relevance, portability, scalability, and simplicity, i.e., the four requirements of good benchmarks?
These requirements on good benchmarks defined by Jim Gray [Gra93] impact, e.g., the incorporated data, the benchmark workload, the toolkit, and the performance result calculation. Moreover, these demands influence the overall benchmark architecture, such as the question of whether or not to incorporate a message broker. Based on this research question, the following related questions evolve.

- *How does the usage of an abstraction layer for the development of stream processing applications impact the performance regarding latencies?*
  Businesses and academia make use of such abstraction to gain flexibility regarding the choice of DSPS, i.e., to diminish the required migration efforts for exchanging a DSPS. It is of importance to understand the resulting performance impact of incorporating an abstraction layer for application development. A central performance indicator in this context is latency, i.e., the time difference between retrieving an input record and outputting the corresponding application result.

- *What are the performance limitations of the benchmark components and its toolkit with regard to the configurable data ingestion rate?*
  The complexity of DSPS benchmarks requires to employ components next to the SUT, e.g., for providing data streams. Furthermore, DSPS benchmarks often support scalability with respect to the data ingestion rate. Accordingly, the performance limits of the benchmark components and tools need to be acquainted with people who use the benchmark to ensure an intended benchmark process. If benchmark tools are overburdened, their lack of performance is studied instead of the SUT's performance.

1.3 Contributions

In the following, we briefly highlight the main contributions of this dissertation and present the corresponding peer-reviewed publications, which describe substantial parts of this work.

A Benchmark for Enterprise Stream Processing Architectures
We propose ESPBench, a performance benchmark for enterprise stream processing architectures that satisfies the mentioned requirements on a good benchmark: *relevance, portability, scalability*, and *simplicity*. By enterprise stream processing, we refer to the combination of streaming and structured business data. Our benchmark design represents real-world scenarios and allows for an objective result calculation as well as scaling of data. The defined benchmark query set covers all core functionalities of DSPSs that are described in detail in Chapter 2. The benchmark toolkit automates the entire benchmark process and provides important features, such as query result validation and a configurable data ingestion rate. To validate ESPBench and to ease the use of the benchmark, we propose an example implementation of the ESPBench queries leveraging the *Apache Beam* software development kit (SDK). Apache Beam is an abstraction layer for developing stream processing applications, that is applied in academia and enterprise contexts. It allows for the execution of the defined programs on any of the supported DSPSs. We use the example query implementation to benchmark three modern DSPSs. The results of the study prove the functioning of ESPBench and its toolkit. ESPBench is capable of quantifying performance characteristics of DSPSs and of unveiling relevant differences among systems.

The following publications contain work related to the contribution that is described in this dissertation:

- *Conceptual Survey on Data Stream Processing Systems* [HL15]
  This publication conceptually compares selected state-of-the-art DSPSs. The comparison focuses on architectural aspects of the systems. Chapter 2 of this dissertation presents the technological background. It includes the description of DSPSs relevant for this thesis, which is partly influenced by the publication.

- *Application of Data Stream Processing Technologies in Industry 4.0 - What is Missing?* [HSMU19]
  This work highlights the aspects that are missing for a broader application of data stream processing technologies. One facet is the need for a satisfying benchmark for DSPSs. Furthermore, it presents industry insights derived from site inspections at manufacturing companies with a focus on Industry 4.0. The work mainly influenced Chapter 1 of this dissertation.

- *Adding Value by Combining Business and Sensor Data: An Industry 4.0 Use Case* [HMSU19]
  This publication presents an Industry 4.0 demo application, which includes

the generation of business and sensor or streaming data. It allows for ad-hoc data exploration as well as horizontal and vertical data integration. This prototype highlights the potential of Industry 4.0 and sophisticating stream processing technologies. Parts of the paper were incorporated in Chapter 1 as well as in Chapter 3.

- *A New Application Benchmark for Data Stream Processing Architectures in an Enterprise Context: Doctoral Symposium* [HMRU17]
  This work presents the identified gap in the area of stream processing benchmarks. It further illustrates the idea of a new benchmark for data stream processing architectures. The article highlights related research questions and the importance of this work. Moreover, first architectural ideas are introduced that are partly taken up in Chapter 3.

- *Senska - Towards an Enterprise Streaming Benchmark* [HRM+17]
  This publication introduces *Senska*, which is the theoretical blueprint behind ESPBench. It presents basic concepts and ideas of ESPBench and thus, its content can be found mainly in Chapter 3.

- *ESPBench: The Enterprise Stream Processing Benchmark* [HMP+21]
  This work presents ESPBench, a new performance benchmark for enterprise stream processing architectures. Its intended functioning is validated by benchmarking three state-of-the-art DSPSs, using a developed example implementation of the benchmark queries. The developed benchmark concepts and experimental evaluation results are described in Chapter 3 and Chapter 4.

Study of the Performance Impact of Apache Beam

Apache Beam is a popular abstraction layer for developing data stream processing applications. We analyze how the use of Apache Beam impacts performance. Our analysis compares the latencies of a selected microbenchmark workload, implemented using native system SDKs with their counterparts developed using the Apache Beam SDK on three different systems. We evaluate the performance penalties when using Apache Beam depending on the system, query, and level of parallelism. The results serve as a reference point for estimating the performance decrease that needs to be accepted for the gain in flexibility regarding the choice of DSPS. The results are discussed in Chapter 4.

The following publication contains substantial parts of the contribution:
Quantitative Impact Evaluation of an Abstraction Layer for Data Stream Processing Systems [HMG+19]

Empirical Evaluation of the Achievable Apache Kafka Input Rate and Timestamp Characteristics
Apache Kafka [KNR11] is a popular messaging system that we employ in the ESPBench architecture. Specifically, it provides the input data streams for the SUT without being part of the SUT. We analyze how many input records per second Apache Kafka can handle with the configuration settings employed in ESPBench. Moreover, we study the impact of selected configuration and execution options that Apache Kafka and its tools provide. In the context of ESPBench, it is crucial to ensure that Apache Kafka does not become a bottleneck when benchmarking the SUT, because this would lead to benchmarking Apache Kafka instead of the SUT.

ESPBench further uses the *LogAppendTime* feature of Apache Kafka, which, if enabled, makes Apache Kafka store a timestamp along with a data record that represents the time when the record is appended to the Apache Kafka log. Due to batching mechanisms, these timestamps can be delayed and thus influence the calculated latencies. We analyze how large these delays are and how selected configuration options impact them. The content of this work mainly influenced parts of Chapter 4.

The following publication contains substantial parts of the contribution:
How Fast Can We Insert? An Empirical Performance Evaluation of Apache Kafka [HMU20]

1.4 Outline

The remainder of this dissertation is structured as follows:

Chapter 2 describes the background of data stream processing and performance benchmarking.

Chapter 3 depicts our proposed benchmark. It highlights the design objectives that are taken into account, the benchmark scenario, and its architecture. Furthermore, the chapter presents the benchmark process, the used data, and the queries defined by ESPBench.

Chapter 4 illustrates the experimental evaluations of this dissertation. These analyses comprise the validation of ESPBench and its functioning by benchmarking three modern DSPSs in combination with the provided example query implementations, which were developed using the Apache Beam software development kit. Further experimental evaluations study the performance impact

of the abstraction layer Apache Beam as well as the achievable data ingestion rates of Apache Kafka.

Chapter 5 presents related work in the area of data stream processing benchmarks, with respect to the performance impact analysis of Apache Beam, as well as in the context of the conducted Apache Kafka study.

Chapter 6 concludes the dissertation with a summary as well as an outlook on future research directions.

2

BACKGROUND

This chapter describes the background of this thesis, which includes an introduction to data stream processing and related technologies as well as to the area of performance benchmarking.

2.1 Data Stream Processing and Related Technologies

This section gives an overview of data stream processing in general and presents selected data stream processing systems (DSPSs) as well as related technologies that are relevant for the area of stream processing and the conducted experiments in this thesis. Particularly, next to the DSPSs, we describe Apache Beam, the abstraction layer for developing stream processing applications, and the messaging system Apache Kafka.

Stream processing can be described as the processing of unbounded data sets, i.e., the permanent processing of data with an unbound or possibly infinitely emitting data source. A data stream S is an unbound data set which contains values v associated with a timestamp t. Multiple values might have the same timestamp [KKLLC15]. Accordingly, a data stream S can be defined as follows:

$$S := \{(v_1, t_1), (v_2, t_2), (v_3, t_3), ...\}.$$

DSPSs have the purpose to perform stream processing, frequently referred to as executing *continuous queries* [ABW04, BW01]. Timestamps play a crucial

role in this context as they define an order among records that might arrive disorderly at the DSPS. Systems often distinguish between the following three time categories [Rab17]:

- **Event time:** when the data is created
- **Ingestion time:** when the data is received (system time)
- **Processing time:** when the data is processed (system time).

Typically, there is a skew between event time and processing time, i.e., the delta between both is not constant over time [ABC+15, Rab17]. Such a skew is exemplarily visualized in Figure 2.1. Applications may want to consider this fact depending on the use case. So, e.g., if a streaming application works with time-based windows using processing time, results could look different compared to the same application making use of event time for its window calculation. Data records with a high skew at the end of a window would not be considered for a corresponding processing time window. Besides, being aware of these differences between time categories is important for performance benchmarking. E.g., using processing times when calculating latencies could, depending on the DSPS implementation, lead to ignoring queuing times and thus, lead to latencies that do not reflect an end users experience.

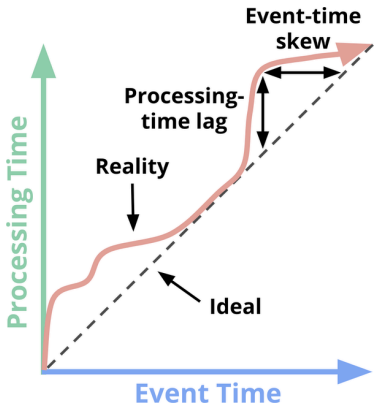


Figure 2.1: Exemplary visualization of the skew between event time and processing time in data stream processing systems ([ACL18])

Stream processing applications can be visualized as a *dataflow*, which is a directed graph. It consists of nodes and edges, representing operators or data and connections between nodes, respectively. Depending on the DSPS as well as the requirements and preferences of the software engineer, such stream processing applications can be developed using a high-level, often SQL-like,

interface or lower-level interfaces offered in programming languages such as Java or Scala [CGH+17]. However, the latter is currently more common since high-level abstractions are not offered by every DSPS. Moreover, there is no SQL-like dialect yet that is generally accepted among DSPSs [HMG+19].

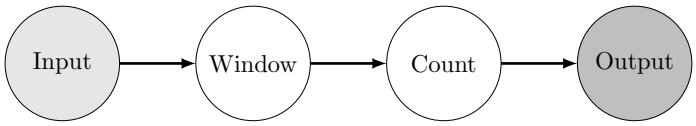


Figure 2.2: Example of a dataflow representing a stream processing application that counts elements of a window (based on [CGH+17])

An example of a dataflow is visualized in Figure 2.2. The first and the last node represent the data source and the data sink, respectively. The application transforms the incoming data into windows, whose elements are counted afterward. That can be useful, e.g., for counting the number of click events on an e-commerce website during the last ten seconds to analyze a user's activity level and provide assistance if needed. A window can be defined using various criteria, e.g., based on a timespan or the number of its elements. It is an operator that splits the stream into smaller batches. That is often required as a potentially infinite data stream does not fit into the memory of a server. Two major sub-categories of time-based windows are *tumbling* and *sliding windows*. While tumbling windows do not overlap, there is an intersection between consecutive sliding windows. The decision for a certain kind of window is up to the application developer and depends on the use case, i.e., on the question which data needs to be analyzed [CGH+17, Rab17].

The fields of application for stream processing technologies are manifold and generally exist where high volumes of data need to be managed with high velocity. Examples range from the manufacturing industry, which is the domain of the benchmark presented in this dissertation, over social media analyses [NWZ+19], to Semantic Web, i.e., *Resource Description Framework* (RDF) stream processing for querying heterogeneous streams of data [DBE15, DTC+19].

2.1.1 Data Stream Processing Systems

This section gives a brief historical overview of DSPSs, highlights the core functionalities of these systems, and conceptually describes selected systems that are relevant to the area of data stream processing. Particularly, the DSPSs Apache Flink, Apache Spark, Apache Apex, and Hazelcast Jet are presented in the following. All of these systems are used in performance investigations that are part of Chapter 4.

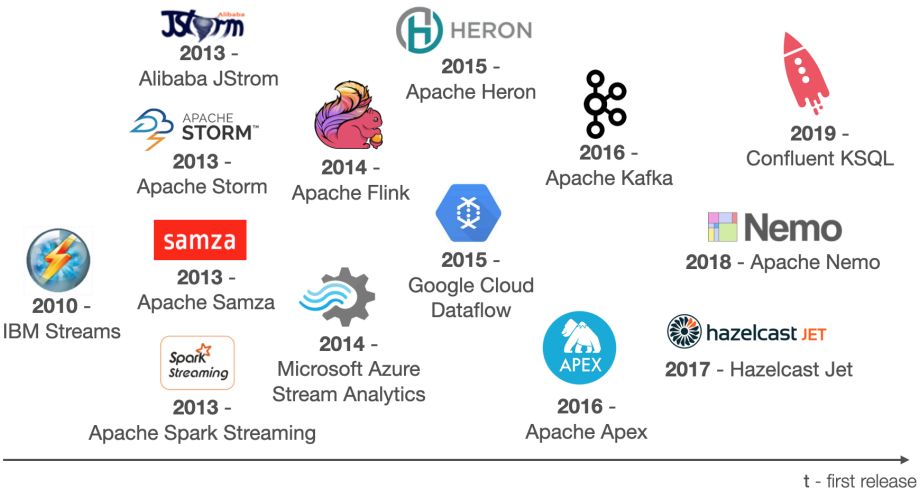


Figure 2.3: Excerpt of the release dates of new data stream processing systems between 2010 and 2020 (image sources[1])

History of Data Stream Processing Systems

One of the early DSPSs, which aimed at processing data in a streaming fashion rather than using an approach based on data batches, is *Aurora* [ACÇ+03]. While this system was designed for a single-server deployment, distributed DSPSs emerged soon after its release. One example is the succeeding project called *Borealis* [AAB+05], which built on top of it. Comprehensive scale-out abilities were introduced by Google's programming model *MapReduce* [DG04], which focused on batch processing. These ideas were adapted in later developed DSPSs, such as *Apache Storm* [TTS+14]. Afterward, stream processing increasingly gained attention, and many more distributed DSPSs were developed, an excerpt of them being depicted in Figure 2.3. The systems shown in Figure 2.3 cover both, open-source and closed-source DSPSs [LKT18].

Core Functionalities

We use the core set of operations for event processing systems presented by Mendes [Men14] as a basis for defining the functionality that should be covered by DSPSs and thus, by the workload of ESPBench. Although this list of op-

[1] all images accessed 2020-12-04: IBM Streams: https://rhc4tp-cms-prod-vpc-76857813.s3.amazonaws.com/s3fs-public/ibm-streams-logo-readme-lg.png, Alibaba JStorm: http://cdn.huodongxing.com/logo/org/201405/3941603602732/781790435449755.jpg, Apache Storm: https://storm.apache.org/images/logo.png, Apache Samza: https://upload.wikimedia.org/wikipedia/en/thumb/f/fb/Apache_Samza_Logo.svg/1200px-Apache_Samza_Logo.svg.png, Apache Spark Streaming: https://dvirgiln.github.io/assets/images/spark_streaming.png, Apache Flink: https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSnv7if12_Rvz3Za9Phpw3hmjQQsaalL7eCzA&usqp=CAU, Microsoft Azure Stream Analytics: https://pbs.twimg.com/profile_images/932734446062641153/vvYmgGZ-_400x400.jpg, Apache Heron: https://raw.githubusercontent.com/apache/incubator-heron/master/website2/docs/assets/HeronTextLogo.png, Google Cloud Dataflow: https://codelabs.developers.google.com/codelabs/cloud-dataflow-starter/img/62b0919755804bea.png, Apache Kafka: https://upload.wikimedia.org/wikipedia/commons/thumb/0/05/Apache_kafka.svg/1200px-Apache_kafka.svg.png, Apache Apex: https://upload.wikimedia.org/wikipedia/commons/thumb/3/3c/Apache_Apex_Logo.svg/1200px-Apache_Apex_Logo.svg.png, Confluent KSQL: https://pbs.twimg.com/profile_images/1197163687217266688/MEIl-K0w.png, Apache Nemo: https://nemo.apache.org/img/nemo-logo.png, Hazelcast Jet: https://repository-images.githubusercontent.com/48046454/d0528f00-7b37-11e9-81bc-c62e692b8ef3

erations is defined for event processing systems, it is applicable to data stream processing in general [HRM+17]. To incorporate the benchmark's enterprise character and increase relevance, we extend the original list. Particularly, we broaden the term pattern detection by altering it to machine learning. Furthermore, we add the aspects of transforming data, which is also included in an earlier work of Mendes et al. [MBM09], and combining streaming with historical data. In the course of this dissertation, historical data is defined as data that is durably stored, e.g., business data persisted in a database. The challenge of integrating stored data is also mentioned as one out of eight requirements of real-time stream processing by Stonebraker, Çetintemel, and Zdonik [SÇZ05], which emphasizes its importance for the stream processing domain. The specific list of operations is shown in the following:

1. **Windowing**
   Analyzing data organized in windows, i.e., in sets of data defined by certain characteristics, e.g., time.

2. **Transformation**
   Modifying incoming data, e.g., through calculations or string operations.

3. **Aggregation/Grouping**
   Combining multiple incoming values into a single characteristic, e.g., by calculating an average or a count over data streams.

4. **Filtering (Selection/Projection)**
   Excluding certain data fields or records based on defined criteria, e.g., based on a certain value included in the data.

5. **Correlation/Enrichment (Join)**
   Combining multiple data streams to a single stream, e.g., measurements of different sensors.

6. **Machine Learning**
   Applying machine learning algorithms to data streams, e.g., for detecting outliers.

7. **Combination with Historical Data**
   Combining streaming data with historical data stored in a database, e.g., to get a better understanding of processes.

Apache Flink

Apache Flink is an open-source DSPS with batch and stream processing capabilities. It resulted out of *Stratosphere* [ABE+14], an open-source research project for big data analytics. In 2014, the creators of Apache Flink founded the company *data Artisans* in Berlin. The startup was acquired in 2019 for $103 million by *Alibaba*, which uses Apache Flink in their system's landscape. A few weeks after Alibaba acquired data Artisans, it got renamed to *ververica* [ABE+14, Rus19, Tzo19].

Apache Flink offers comprehensive Java and Scala application programming interfaces (APIs) for developing applications. There are libraries built on top of Apache Flink that extend its functionalities, such as the library *Gelly* for graph processing [CKE+15, HL15].

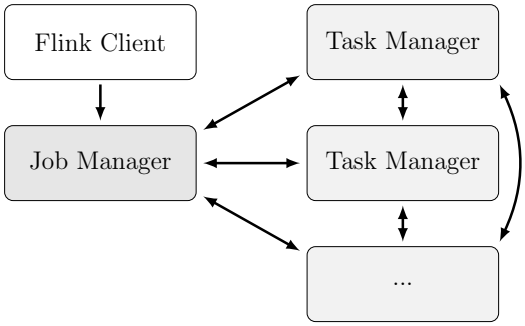


Figure 2.4: Architecture of an Apache Flink cluster with one Job Manager, multiple Task Manager instances, and one connecting client (based on[CKE+15, HL15])

The conceptual architecture of an Apache Flink cluster is visualized in Figure 2.4. It shows an Apache Flink client, a *Job Manager*, and *Task Manager* instances. When deploying a program to the Apache Flink system, the client first transforms it into a dataflow graph, i.e., a directed acyclic graph (DAG), and sends it to the Job Manager. The client itself is not part of the program execution and can, after transmitting the dataflow graph, either disconnect from the Job Manager or stay connected in order to receive information about the execution progress [CKE+15, HL15, HMG+19]. The Job Manager or master is responsible for scheduling work amongst the Task Manager instances and for keeping track of the execution. There can be multiple Job Manager instances whereas only one Job Manager can be the leader. Others are standby and take over in case of a failure [CKE+15, HL15, HMG+19].

The Task Manager instances execute the assigned parts of the program. Technically, a Task Manager is a Java Virtual Machine (JVM) process. There must be at least one Task Manager within an Apache Flink deployment. Task Managers exchange data amongst each other where needed. Each of them provides at least one task slot in which subtasks are executed in multiple threads. A task slot can be shared by multiple subtasks as long as they belong to the same application, even if they are part of different tasks. While one task is executed by one thread, Apache Flink chains multiple operator subtasks into a single task, such as two subsequent map operations. A benefit of this optimization is, e.g., reduced overhead for inter-thread communication [CKE+15, HL15, HMG+19].

Every task slot is assigned a subset of the resources that belong to its corresponding Task Manager. Particularly, the available memory is split amongst task slots. Corresponding CPU separation is not supported in the current Apache Flink version [CKE+15, HL15, HMG+19].

Apache Spark Streaming

Apache Spark is another open-source system for distributed data processing. Its initial version was developed in 2009 at the University of California Berkeley (UC Berkeley). Members of this research project founded the company *Databricks* in 2013. It builds commercial services around Apache Spark [2] [HL15, HMG+19].

Apache Spark offers, next to batch processing functionalities, stream processing features as part of its library *Apache Spark Streaming*. However, stream processing is implemented using micro-batches, i.e., it is not a tuple-by-tuple processing. This concept distinguishes Apache Spark Streaming from many other DSPSs, such as Apache Flink. Apache Spark Streaming applications can be written in, e.g., Java, Scala, or Python. Next to the stream processing library, there are other add-ons built on top of Apache Spark, e.g., for machine learning and graph processing [AXL+15, HMG+19, SS15, ZXW+16].

The architecture of an Apache Spark deployment is shown in Figure 2.5. An application is executed in the form of multiple independent processes distributed across a cluster. The *SparkContext* coordinates these processes. This coordinator is an object in the *main()* function of the application, which is called *Driver Program*. Moreover, the SparkContext connects to a Cluster Manager that takes care of resource allocation.

Currently, there are four Cluster Managers supported by Apache Spark - *Spark Standalone*, *Apache Mesos* [HKZ+11], *Apache Hadoop YARN* (Yet Another Resource Negotiator) [VMD+13], and *Kubernetes* [Bre15]. As soon as a

[2] `https://databricks.com/spark/about`, accessed: 2020-12-08

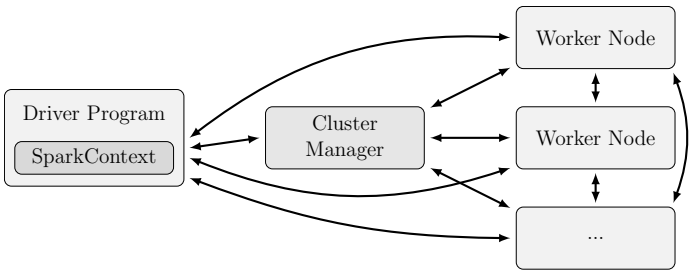


Figure 2.5: Architecture of an Apache Spark deployment in cluster mode with multiple Worker Nodes (based on [HL15, HMG+19])

connection is established, the SparkContext acquires so-called executors on the Worker Node instances. Each executor is a process belonging to exactly one application. It stores data and performs computations. So different applications running on the same Apache Spark cluster are executed in different JVMs, which separates it, e.g., from the execution concept of Apache Flink [HMG+19].

Once executors are acquired, the SparkContext transmits the program in the form of a JAR or Python file to them. Afterward, it sends tasks to the executor processes. One process can run multiple tasks in several threads [HL15, LWI+14]. Figure 2.6 visualizes a worker process architecture that also fits this Apache Spark concept.

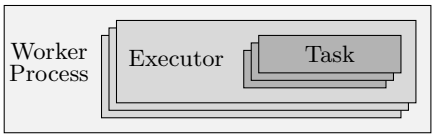


Figure 2.6: Architecture of an Apache Spark Worker Process (based on [HL15, TTS+14])

A central data structure that is used in Apache Spark is the *Resilient Distributed Dataset* (RDD). An RDD is a distributed memory abstraction. To be more concrete, it is a partitioned and read-only collection of records. Apache Spark Streaming employs a processing model called *discretized streams* (DStreams). Such a DStream is a sequence of RDDs. An incoming data stream is divided into batches stored in RDDs. Data transformations are then performed on these RDDs as visualized in Figure 2.7. The output is again represented as a DStream [HL15, ZCD+12, ZDL+13].

Apache Apex

Apache Apex is a stream and batch processing system, which was originally developed at the company *DataTorrent* in 2012. In 2015, Apache Apex became

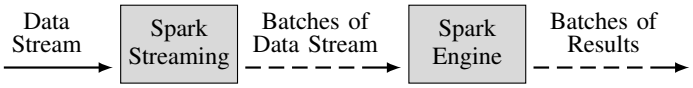


Figure 2.7: Stream processing concept in Apache Spark Streaming (based on [HL15, ZDL+13])

an incubator project of the Apache Software Foundation. In September 2019, the project retired[3]. Apache Apex is technically based on Apache Hadoop, an open-source implementation of the *MapReduce* programming model, which allows to store and process large amounts of data [VMD+13]. Hadoop comprises the components Apache Hadoop YARN and *Hadoop Distributed File System* (HDFS)[4]. Similarly to, e.g., Apache Flink, stream processing is implemented in a tuple-by-tuple-processing fashion [Bha16, DF16, HMG+19].

The high-level architecture of *Apache Hadoop 2* is depicted in Figure 2.8. The distributed file system HDFS at the bottom serves as the storage layer. Apache Hadoop YARN acts as a resource manager on top of HDFS. Multiple data processing frameworks can run on top of these two layers. They provide different functionality, such as batch or stream processing capabilities. Two of them are *Hadoop MapReduce* and Apache Apex [SSS+15]. Apache Spark[5] and Apache Flink[6] are further systems that can run on Apache Hadoop YARN as one of multiple deployment options they offer.

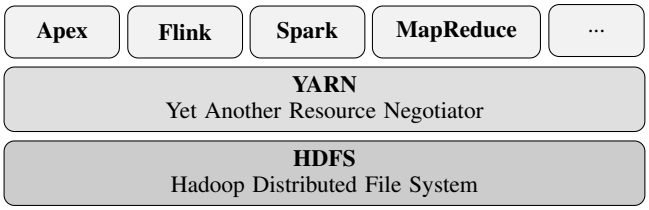


Figure 2.8: Architecture of Apache Hadoop 2 (based on [HMG+19, SSS+15])

Apache Hadoop YARN's internal architecture is illustrated in Figure 2.9. The depicted installation runs a single application. The application components are marked with dashed lines. A central element within Apache Hadoop YARN that exists once per cluster is the *Resource Manager*. Technically, it is a daemon process running on a dedicated machine. It represents the interface

[3] `https://attic.apache.org/projects/apex.html`, accessed: 2020-12-08
[4] `http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html`, accessed: 2020-12-08
[5] `https://spark.apache.org/docs/latest/running-on-yarn.html`, accessed: 2020-12-08
[6] `https://ci.apache.org/projects/flink/flink-docs-master/deployment/resource-providers/yarn.html`, accessed: 2020-12-08

for client applications and monitors the status of the cluster nodes. Moreover, the Resource Manager decides about resource distribution among programs. It thereby allocates and leases resources in form of *Containers*. A container can be viewed as "a logical bundle of resources (e.g., <2 GB RAM, 1 CPU>) bound to a particular node" [VMD+13].

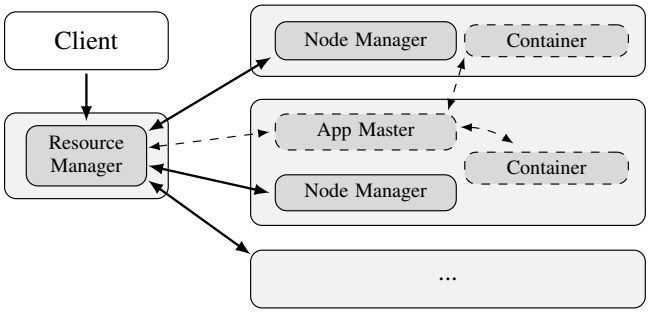


Figure 2.9: Architecture of an Apache Hadoop YARN cluster with one client (based on [HMG+19, VMD+13])

A *Node Manager* daemon is running on each node for managing the available cluster resources. It keeps track of the node's resources and notifies about failures should the occasion arise. The Resource Manager and the Node Manager communicate using a heartbeat mechanism [VMD+13].

After the Resource Manager accepts an application submitted by a client, it allocates containers and starts the *Application Master* within it. This master process also sends heartbeats to the Resource Manager. There may be additional message exchange on application level between the Application master and its assigned containers [HMG+19, VMD+13].

The Application Master manages the program execution with respect to aspects such as resource needs and fault management. That covers coordinating the logical execution plans by requesting resources and generating the physical execution plans according to the actually assigned resources. To get new resources, an Application Master has to send a request to the Resource Manager. As soon as a resource lease on behalf of an Application Master is created, the corresponding container is pulled by the Master's next heartbeat [HMG+19, VMD+13].

Apex Malhar is a library built on top of the Apache Apex core. It provides different input, output, and compute operators, which are used for developing stream processing applications running on Apache Apex. These operators provide capabilities for, e.g., communicating with Apache Kafka [HMG+19].

Hazelcast Jet

Hazelcast Jet is the name of Hazelcast's stream processing engine. Hazelcast itself is a distributed in-memory data grid (*Hazelcast IMDG*, short Hazelcast). Similar to, e.g., Apache Spark and Apache Flink, Hazelcast is developed using a JVM language, namely Java. The development of Hazelcast started as an open-source project under the Apache License 2.0. The same-named *Hazelcast incorporation* added commercially licensed enterprise features in 2013. In 2014, the company bundled these additional functionalities into *Hazelcast Enterprise*. Currently, Hazelcast Inc. offers the commercial versions named *Hazelcast IMDG Enterprise* and *Hazelcast IMDG Enterprise HD*. The latter one extends Hazelcast IMDG Enterprise by, e.g., a high-density memory store. Additionally, there is a Hazelcast Jet Enterprise component. It extends Hazelcast Jet by, e.g., security features and a lossless restart functionality[7] [Luc15].

From an architectural point of view, Hazelcast Jet is different from both, Apache Spark and Apache Flink, due to its so called masterless design. The oldest node a cluster represents the de facto leader. As such, it manages, e.g., data responsibilities within the cluster. Hazelcast organizes the data in shards or partitions, which are distributed equally among the cluster. Furthermore, it keeps data backups at multiple nodes to prevent data loss in case of a node failure [Joh15].

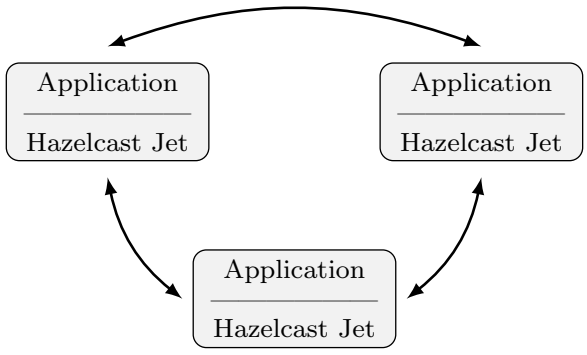


Figure 2.10: Architecture of a Hazelcast Jet deployment in the embedded mode with three nodes (based on[7],[Joh15])

There are two deployment modes supported for both, Hazelcast IMDG and Hazelcast Jet: the *embedded mode* and the *client-server mode*. Figure 2.10 and Figure 2.11 visualize the embedded and client-server deployment respectively. Each architecture illustration comprises three Hazelcast Jet nodes. Both figures

[7] `https://docs.hazelcast.org/docs/3.12.4/manual/html-single/index.html`, accessed: 2020-12-09

show the mentioned absence of a dedicated master within a Hazelcast cluster. In the embedded deployment mode of Figure 2.10, both the application as well as Hazelcast Jet share a single JVM. This design has the advantage of, e.g., low-latency data access due to the tight coupling of application and Hazelcast Jet. A downside of the embedded mode is its inability to scale the application and data processing engine/data persistence independently[7][Joh15].

Figure 2.11 visualizes a client-server deployment with three nodes and two applications. The Hazelcast Jet cluster can be created, scaled, and managed independently from any application that is supposed to run on this cluster. This setup brings isolation or separation of application and cluster but also introduces challenges. For instance, one has to pay more attention to the classpaths of both, application and cluster nodes [Joh15].

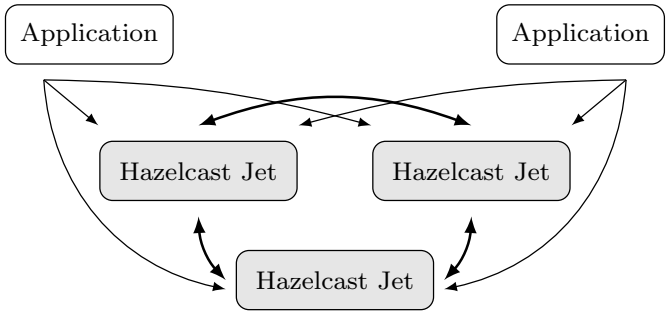


Figure 2.11: Architecture of a Hazelcast Jet deployment in the client-server mode with three nodes and two applications (based on[7],[Joh15])

Comparison of the Presented Data Stream Processing Systems

In the following, we compare the presented DSPSs with regard to selected aspects to create a consolidated overview.

Organisational Background

Except for Hazelcast Jet, all DSPSs represent a project of the Apache Software Foundation. All of them are open-source or at least there is an open-source version of the system. Apache Flink and Apache Spark originate from university projects, while the others have evolved from the industry. Hazelcast already started as an open-source project. Furthermore, behind Apache Flink, Apache Spark, Apache Apex, and Hazelcast Jet, there is or used to be a company making contributions to the system or monetizing an extended enterprise version or services around it.

Programming Languages

Each of the presented DSPSs is mainly written in a JVM language. While Apache Spark is mainly written in Scala, all other systems are primarily developed using Java. For application development, all systems offer comprehensive Java APIs. Writing stream processing programs in Scala or Python is also supported by, e.g., Apache Spark Streaming and Apache Flink. SQL can be used to a limited functional degree within Apache Flink[8], Apache Spark Streaming[9], and Apache Apex[10]. Hazelcast Jet has no plans to enable SQL support as it focuses on Java as the language for application development[11]. All of the presented DSPSs support the abstraction layer Apache Beam for the development of stream processing programs[12]. With respect to programming languages, Apache Beam offers a Java, Python, Go, and Scala software development kit (SDK)[13]. Section 2.1.2 introduces the abstraction layer Apache Beam in more detail.

System Architecture

The vast majority of the presented DSPSs follow a master-worker pattern, meaning there is a kind of master node with a coordinating character and multiple worker nodes that perform the data processing. The only exception is Hazelcast Jet. In a Hazelcast cluster, all nodes are of the same type, i.e., a cluster only consists of worker nodes.

Data Processing Model

All presented DSPSs except for Apache Spark Streaming analyze data in a tuple-by-tuple fashion. Apache Spark Streaming processes data in micro-batches.

Data Processing Guarantees

All presented systems guarantee exactly-once processing, i.e., each input tuple is processed exactly once. This ensures correct results also in recovery scenarios [HMG+19].

[8] `https://ci.apache.org/projects/flink/flink-docs-release-1.11/dev/table/`, accessed: 2020-12-09

[9] `https://spark.apache.org/docs/latest/streaming-programming-guide.html`, accessed: 2020-12-09

[10] `https://apex.apache.org/docs/malhar-3.7/apis/calcite/`, accessed: 2020-12-09

[11] `ttps://jet.hazelcast.org/faq/`, accessed: 2020-12-09

[12] `https://beam.apache.org/documentation/runners/capability-matrix/`, accessed; 2020-12-09

[13] `https://beam.apache.org/get-started/beam-overview/`, accessed; 2020-12-09

2.1.2 Apache Beam

Apache Beam describes itself as a unified programming model, which allows defining batch and stream processing applications. Multiple Apache Beam SDKs are provided for this purpose. Currently, there are SDKs for four programming languages: Java, Python, Go, and Scala[13]. The Apache Beam project itself mainly contains Java code[14].

Instead of developing an application for a single DSPS, Apache Beam allows writing programs that are compatible with any of the execution engines it supports. Engine-specific *runners* translate the Apache Beam code to the target runtime. Using such an abstraction layer theoretically allows for an arbitrary exchange of engines without the need for code adaption.

Next to Apache Flink, Apache Spark, Apache Apex, and Hazelcast Jet, there are further frameworks that support the execution of applications developed with the Apache Beam SDK[12]. These are, e.g., Apache Samza [NPP+17], IBM Streams[15], and Google Cloud Dataflow[16]. So the group of supported execution engines covers both, closed-source systems, such as Google Cloud Dataflow and IBM Streams, as well as open-source systems. Apache Beam itself resulted out of the donation of the Cloud Dataflow SDKs and programming model [ABC+15] to the Apache Software Foundation[17].

For supporting Apache Beam, a DSPS needs to have a corresponding Apache Beam *runner*. This runner transforms an Apache Beam application to a program, which can be executed on the corresponding DSPS. The runner only provides the asynchronous method *run(Pipeline)*, which takes and executes the developed Apache Beam application. The function returns a runner-specific *PipelineResult* object, which is an application descriptor for the DSPS that offers functionality for, e.g., checking the program's status or cancelling it[18].

The Apache Beam project provides central components for application development. Selected core components are, based on [ABC+15, HMG+19] and online ressources[19,20], presented in the following:

- **Pipeline** represents the entire application definition, including data input, transformation, and output.

[14] `https://github.com/apache/beam`, accessed: 2020-12-09
[15] `https://www.ibm.com/de-en/marketplace/stream-computing`, accessed: 2020-12-15
[16] `https://cloud.google.com/dataflow/`, accessed: 2020-12-15
[17] `lhttps://cloud.google.com/blog/products/gcp/cloud-dataflow-apache-beam-and-you`, accessed: 2020-12-17
[18] `https://beam.apache.org/contribute/runner-guide/`, accessed: 2020-12-17
[19] `https://beam.apache.org/documentation/programming-guide/`, accessed: 2020-12-17
[20] `https://github.com/apache/beam/blob/master/website/src/contribute/runner-guide.md`, accessed: 2020-12-17

- **PCollection** embodies an immutable distributed data set that can be either bounded or unbounded. The latter is used for data stream processing applications. It can be created through, e.g., reading from an external source, such as Apache Kafka. Different *Pipeline* objects cannot share a *PCollection* object. Random access of data elements is not supported.

- **PTranform** represents a data transformation. It receives one or more *PCollection* objects and applies a transformation on each input record. The transformation itself is provided as a function object and might be distributed across worker nodes, depending on, e.g., the runner and execution engine. Applying a transformation to data leads to an output of zero or more *PCollection* objects. Moreover, read or write operations on external storage systems are realized using *PTransform* objects. Apache Beam provides some core transforms as well as composite transforms, which are a combination of core transforms. Examples for the latter are functions for counting or combining collection records. It is also possible to create custom composite transforms. Selected standard Apache Beam core transforms are outlined in the following:

  - **ParDo** is a parallel element-by-element processing of data. The processing of a single element can lead to zero or more elements in the output *PCollection*. The actual transformation needs to be defined as a *DoFn* object, which represents a distributed processing function in the context of Apache Beam. Such a distributed processing function, which is technically a subclass of *DoFn*, has to fulfill two requirements due to Apache Beam's concept of distributed data processing:

    * *Serializability*, as the serialized function needs to be transmitted to remote worker nodes. Although the parent class is serializable, it is important as non-serializable attributes must not be added in the subclass. It impacts application design as, e.g., loading large amounts of data into fields should be avoided prior to serialization for performance reasons.
    * *Thread-Compatibility* - the Apache Beam SDK is not thread-safe. A function object is only accessed by one thread at a time, but a subclass can create its own threads. Due to this absence of thread-safety in the Apache Beam SDK, thread synchronization is up to the application developer.

    Additionally, it is recommended but not required to implement a

DoFn function object in an idempotent fashion. Doing so ensures it can be executed multiple times without causing unintended effects. That is since Apache Beam does not provide guarantees on the number of executions.

These two outlined requirements also hold true for the *Combine* and *Window* transforms of Apache Beam, i.e., for subclasses of *CombineFn* as well as *WindowFn*, respectively. Both of them are explained later on.

- **GroupByKey**, as the name already states, processes key-value pairs and collects all values belonging to the same key. It is an aggregation operation that outputs pairs consisting of a key and a collection of values that belong to this key. For use with streams, one must use an aggregation trigger or non-global windowing to enable the grouping to be applied to a finite data set.
- **CoGroupByKey** is similar to *GroupByKey*. The difference is that there are two or more key/value *PCollection* objects, on which *CoGroupByKey* performs a relational join. Regarding its use with unbounded data sets, the same holds true as for *GroupByKey*.
- **Combine** melds a collection of values. This can either done in an entire *PCollection* or by key, e.g., in succession of a *GroupByKey* operation to combine multiple values associated with a certain key to a single value. For applying the *Combine* transform, a corresponding function has to be passed. For advanced combine algorithms, Apache Beam provides the *CombinFn* class, similar to the previously described *DoFn* class. So subclasses can be developed, for which the same requirements exist as for *DoFn* subclasses.
- **Flatten** merges the data of multiple *PCollection* objects that contain data of the same type into a single *PCollection*.
- **Partition** allows to split a single *PCollection* object that contains data of the same type into multiple smaller *PCollection* objects based on a passed partitioning function. The number of partitions needs to be determined at graph construction time, i.e., it cannot be defined or changed based on calculations within the application.

Moreover, Apache Beam supports windowing, which divides a *PCollection* based on, e.g., its elements' timestamps. Aggregating transforms like *GroupByKey*, *CoGroupByKey*, and *Combine* implicitly work with windows as they can only process finite data sets. A *trigger* defines when to emit results of

an aggregation on a data stream. Apache Beam provides standard window functions, but also allows for creating individual ones. For that, it offers the *WindowFn* class, which has the same restrictions and design principle as *DoFn* and *CombineFn*. Provided window functions are, e.g.,:

- **Fixed Time Window** is used to divide a *PCollection* into tumbling, i.e., not overlapping windows of a defined period of time, e.g., 60 seconds. Assuming the application starts at 12:00:00, the first window contains all elements with a timestamp up to but not including 12:01:00. Accordingly, the second window comprises elements with timestamps belonging to the interval [12:01:00, 12:02:00).
- **Sliding Time Window** is like a fixed time window, but with a possible overlap, i.e., elements might belong to more than one window. An example is a window of 60 seconds, which is recalculated every 30 seconds. Referring to the previous example, the first window is identical, with the timestamp interval [12:00:00, 12:01:00). However, the second window contains elements belonging to the interval [12:00:30, 12:01:30). Consequently, there could be elements belonging to both windows.

Besides these two common window types, Apache Beam supports, e.g., a single global window as well as session and calendar-based windows. The latter represents a time-based window with large time spans, particularly days, months, or years [21].

2.1.3 Messaging System Apache Kafka

Apache Kafka was originally developed at LinkedIn in 2010 [Kos16] and is often employed in combination with a DSPS for storing data, e.g., at Bouygues Telecom [Abd15] and Zalando [VL16]. Kreps, Narkhede, and Rao [KNR11] described Kafka in 2011 as a distributed messaging system for log processing. To be more concrete, the core of Apache Kafka is a publish-subscribe system implemented as a distributed commit log, in which applications can store data. Over the years, Apache Kafka enlarged its scope through extensions that were built around or on top of it. An example is Kafka Streams[22], a client library for developing stream processing applications. When talking about Apache Kafka within this dissertation, we refer to its message broker core.

On a technical level, an Apache Kafka cluster consists brokers and is connected to producers, i.e., applications that send data to Apache Kafka, and

[21] `https://beam.apache.org/releases/javadoc/2.0.0/org/apache/beam/sdk/transforms/windowing/CalendarWindows.html`, accessed: 2021-07-17

[22] `https://kafka.apache.org/documentation/streams/`, accessed: 2020-12-18

consumers, i.e., application that retrieve data from Apache Kafka. A cluster consists of multiple, numbered Apache Kafka brokers. These brokers store data assigned to topics. Data producers send data to a certain topic, stored in the cluster. Multiple messages can be published with a single send request for performance reasons. Consumers subscribe to a topic and are forwarded new values sent to this topic as soon as they arrive. Similar to send requests, a data pull can also retrieve multiple messages, although Apache Kafka consumer APIs might make it look like there is a single request per message [HMU20, KNR11].

Data stored in Apache Kafka is organized in *topics*, which are divided into partitions. The number of topic partitions can be configured at the time of topic creation. Partitions of a single topic can be distributed across different brokers of a cluster. Additionally, it is possible to define a replication factor for each topic, with one being the minimum. Using this approach, data loss in the case of a broker failure can be prevented. In the context of replication, Apache Kafka defines a so-called leader and followers for each partition. The leader handles all reads and writes for the corresponding topic partition, whereas the followers copy or replicate the inserted data.

An example architecture of an Apache Kafka cluster with three brokers, two topics, three producers, and three consumers is visualized in Figure 2.12. While topic1 has two partitions, named *part1* and *part2*, *topic2* only has one partition. The leading partitions are marked in bold type. The remaining partitions represent replicas. While *topic1* has a replication factor of one, meaning one replica for each partition, the second topic has two replicas.

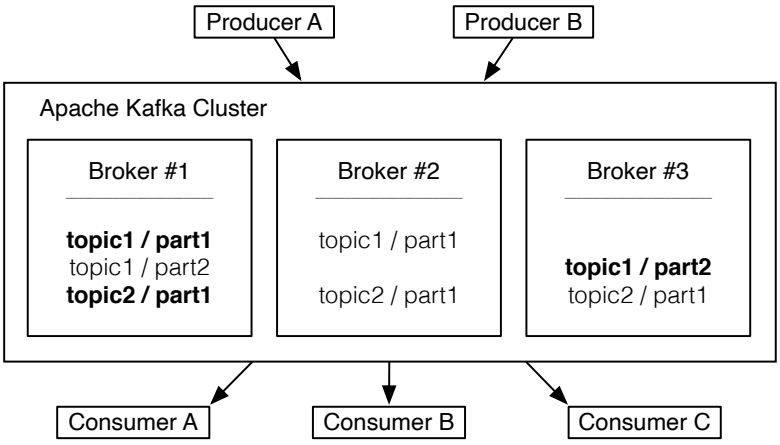


Figure 2.12: Example architecture of an Apache Kafka cluster with three brokers, two topics, two producers, and three consumers (based on [KNR11])

[23] `https://kafka.apache.org/documentation/`, accessed: 2020-12-18

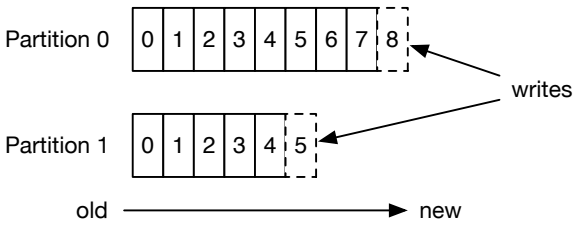


Figure 2.13: Storage concept of an Apache Kafka topic with two partitions (based on[23])

Figure 2.13 visualizes the structure of an Apache Kafka topic with two partitions. Each of these partitions is an ordered and immutable record sequence where new values are appended. A sequential number is added to each record within a topic partition, which is referred to as the *offset* in the context of Apache Kafka. The absence of an explicit message id reduces overhead for Apache Kafka, as, e.g., auxiliary random-access index structures become obsolete. A message order across partitions is not guaranteed by Apache Kafka [HMU20, KNR11].

Apache Kafka provides the topic *__consumer_offsets* for storing the offsets. However, consumers must manage their offset. They can commit their current offset either automatically in certain intervals, or manually. The latter can be done either synchronously or asynchronously. A consumer needs to pass the offset to the cluster when polling new records. Apache Kafka returns all messages with a greater offset, i.e., all new messages that have not already been sent to this consumer. As the consumer has control over its offset, it can also decide to start from the beginning and so to reread messages[23].

Not managing consumer offsets at the brokers reduces their complexity. Consequently, Apache Kafka brokers do not know which messages have been already polled by a consumer, which otherwise could be a criterion for deleting messages from the log. Instead, messages are deleted either based on their age, meaning there is a configurable retention time, or after a certain configurable amount of data has been reached [HMU20, KNR11].

Technically, a topic partition consists of a set of segment files of approximately the same size, e.g., 1 GB. New data will be appended to the last segment file. To increase performance, the segment files are not necessarily flushed to disk each time after a new record is appended. Particularly, flushing is configurable and done based on the number of published messages or the elapsed time. In this context, it is important to take into account that only after messages are flushed, they become available for consumption [HMU20, KNR11].

Apache Kafka provides the *LogAppendTime* setting, which induces Apache Kafka to assign a timestamp to each message once it is appended to the log. The existing alternative is *CreateTime*, which represents the default mode. In this setting, the timestamp created by the Apache Kafka producer when creating the message, i.e., before sending it, is stored along with the message. For transmitting messages, a producer can require multiple retries, which increases the difference between the timestamp assigned to a message and the time when it is appended to the log and thus, made available for consuming applications. This potential delay can influence design of consuming applications.

All in all, Apache Kafka provides flexible message storing capabilities, which are useful for data stream processing scenarios. With its distributed architecture, it further provides scale-out possibilities for adapting to changed workloads.

2.2 Performance Benchmarking

Performance benchmarking is a wide research area that is omnipresent in scientific discussions and industry. Since the 1960s, people know about the need for evaluation methodologies for computer systems. Nowadays, we can find performance benchmarks in various areas. Be it the newest graphics performance numbers or the results of car tests comparing acceleration capabilities and the range limits of electric vehicles, the performance of systems is often analyzed and compared. This is crucial to get a better understanding of how such systems behave and relate to each other, which impacts decision-making processes, e.g., the decision of which one to buy. This section introduces the topic of performance benchmarking by elaborating on its definition, benchmark classifications, and design principles for benchmarks [LC85].

2.2.1 Definition of Benchmarking

A popular book in the area of performance benchmarking for information technology (IT) is *The Benchmark Handbook - For Database and Transaction Processing Systems*, which was written by Jim Gray [Gra93]. It inspired the work presented in this dissertation. Gray describes the question performance benchmarks in the area of IT are trying to answer as: "What computer should I buy?" [Gra93]. Moreover, he elaborates on the answer to this question. According to him, the computer or system with the lowest total cost of ownership (TCO) that achieves the needed performance should be chosen. In contrast to Gray, Joslin [Jos65] disregards the costs and additionally mentions that costs,

as well as other factors, become irrelevant if "the system cannot do the workload" [Jos65]. For Joslin, the most important question any computer evaluation should try to answer is about how long the system will need to process the given workload [Jos65]. However, assuming that multiple systems or configurations can process a workload in a reasonable time, regardless of how that is defined, money may become a relevant factor when deciding which system to buy. This dissertation uses the definition of Vieira et al. [VMSK12] because of its universal character. They describe performance benchmarks as "standard procedures and tools aiming at evaluating and comparing different systems or components in a specific domain (e.g., databases, operating systems, hardware, etc.) according to specific performance measures" [VMSK12]. The domain of the benchmark presented in this work is enterprise stream processing architectures.

The idea behind the TCO mentioned by Gray is to consider not only the purchase price when evaluating a possible acquisition of a good, but also to include additional costs that will occur when buying it. Such costs could be associated with the use or maintenance of the product to purchase, such as training or service and support expenditures. Costs can be distinguished in many ways, e.g., as fixed and variable costs or direct and indirect costs. However, the calculation of TCO is often complex as certain aspects are hard to quantify. This also holds true for the TCO of data stream processing landscapes when it comes to, e.g., programming or operations costs. Moreover, many factors cannot be assessed in isolation. The existing workforce and their skillsets or the availability of experts are examples of aspects that impact the costs of using and operating a certain system. Other costs, such as hardware or software costs, are easier to determine [ES93].

The performance can be expressed in, e.g., response times or throughput key performance indicators (KPIs). In enterprise contexts, a service level agreement (SLA) is often used for assuring a certain performance, for instance regarding system availability. If performance results are satisfying, benchmarks are also used for marketing purposes, so that companies can claim that their system are superior to the systems of their competitors. However, this imposes the dangers of optimizing a benchmark for a certain system or optimizing a system for a specific benchmark to improve on their results. Therefore, benchmarks should be viewed critically regarding, e.g., whether they test the core set of system functionalities or only a selected subset, and how the computations and results fit the benchmark purpose [Gra93].

Besides, benchmarks are also used for system development. By using them, efforts in the area of performance improvement can be assessed or the overall

performance characteristics can be monitored over time. For instance, each code contribution can trigger a benchmark run and the results can be compared to the previous ones to ensure that there is no performance decrease introduced with the new changes [Gra93].

2.2.2 Benchmark Classifications

There are different types of performance benchmarks that distinguish based on independent aspects such as workload complexity, domain, or standardization. Thus, classifications are multidimensional and benchmarks can belong to more than one group. Selected benchmark categories are presented in the following to give a brief overview of the domain. This enumeration does not aim to be a complete summary.

One way of classifying benchmarks is by using workload characteristics and targeted systems under test. In the following, we present selected benchmark categories, which are determined by these aspects.

Micro and Functional Benchmarks

Micro benchmarks investigate the performance of systems when executing rather low-level atomic operations, such as file system operations. Functional benchmarks are located on a slightly higher level. They evaluate, e.g., a join operation in a database context or applying a filter to a data stream. However, one can find different classifications in literature. While, e.g., Rabl et al. [RFD+15] consider the evaluation of a sorting algorithm as a functional benchmark, Huang et al. [HHD+10] classify sorting as a micro benchmark. To overcome the question of correct classification, we regard micro benchmarks and functional benchmarks as one group and use both terms synonymously in the following.

Micro benchmarks are well-suited for evaluating very distinct functionality. They allow, e.g., for comparing different implementations of a certain operator on the same system without having to implement much code around. This would be unnecessary code for the desired goal, i.e., benchmarking only the existing operator implementations. The simplicity that comes with such a relatively simple workload eases result interpretation and benefits result credibility.

The situation looks different when aiming at more complex systems, such as business applications. These programs usually do not consist of a single operator call. Micro and functional benchmarks provide limited insights in such a case, because they measure only a small subset of the overall system. Having a great performance in smaller benchmark scenarios does not necessarily result

in an equally good performance when these atomic operations are combined in a business application.

An example for a micro benchmark is *HiBench* [HHD+10], which is a benchmark suite for Apache Hadoop that contains various benchmarks workloads. *StreamBench* [LWXH14] is another example, which aims at benchmarking DSPSs and is presented in more detail in Chapter 5.

Application Benchmarks

Application benchmarks, in contrast to micro benchmarks, compare rather comprehensive programs, such as business applications. This analysis of real-world scenarios introduces complexity, e.g., when it comes to implementing and using the benchmark. Moreover, complex applications could make it difficult, e.g., to understand performance differences between benchmark scenarios or interpret results correctly.

An example for such an application benchmark is *TPC Benchmark C* (TPC-C) [Cou90], a standard application benchmark for on-line transaction processing (OLTP) defined by the *Transaction Processing Performance Council* (TPC). *LinearRoad* [ACG+04] is another example, which represents one of the most popular benchmarks for DSPSs. Chapter 5 presents the idea and characteristics of *LinearRoad*.

Genre-Specific Benchmarks

Another classification variant for benchmarks is the division by its genre or the domain of systems to be benchmarked. Examples are the *Graph 500* [MWBA10] benchmark focusing on graph data, *MLBench* for machine learning workloads [LZZ+18], and benchmarks focusing on DSPSs, such as the afore-mentioned *LinearRoad* and *StreamBench*.

Standardized Benchmarks

To create reliability and credibility, standard bodies emerged for defining performance benchmarks. Having such bodies is supposed to reduce the danger of developing a benchmark in a way that benefits the developer's own system most. Popular standard bodies are, e.g.,:

- *Business Application Performance Consortium (BAPCo)*
  This non-profit consortium was founded in 1991 and aims to develop objective performance benchmarks for popular application and industry standard operating systems (OSs). A popular BAPCo benchmark is *SYSmark*, *SYSmark 2018* being its latest version. It focuses on office-centric user activities. The BAPCo website shows about 300 *SYSmark 2018* results.

The list of BAPCo members includes, e.g., Intel, Microsoft, and Samsung[24,25,26].

Although BAPCo is a standard body, they got accused by Advanced Micro Devices (AMD) of favoring Intel in SYSmark in the early 2000s. About ten years later, AMD left the BAPCo consortium and Intel got fined for false advertising, amongst others, related to the BAPCo SYSmark benchmark [Smi10, Ung16].

- *Standard Performance Evaluation Corporation (SPEC)*
  SPEC is a non-profit corporation founded in 1988. Similar to BAPCo, it develops performance benchmarks and publishes benchmark results on its website. SPEC comprises about 120 organizations, including, e.g., AMD, Intel, and educational institutions[27]. It is organized into four groups:

  - **Graphics and Workstation Performance Group (GWPG)** develops graphics and workstation benchmarks and is subdivided into three groups. These project groups are the application, graphics, and workstation performance characterization groups. *SPECviewperf* is one of the belonging benchmarks, which evaluates 3D graphics performance on systems using the *Open Graphics Library* (OpenGL) and the *Microsoft DirectX* APIs[27,28].
  - **High Performance Group (HPG)** targets high-performance system architectures with their benchmarks. The *SPEC ACCEL* benchmark suite is one example of a benchmark belonging to HPG. This benchmark analyzes the performance of a system when executing computationally intense applications using hardware accelerators and corresponding APIs for target offloading[27,29].
  - **Open Systems Group (OSG)** develops benchmarks for workstations or servers running open operating system (OS) environments. *SPEC CPU* is one benchmark belonging to this group, which contains a CPU or compute intense workload[27,30].
  - **Research Group (RG)** aims to promote research in the area of performance benchmarking for both, established as well as emerging

[24] `https://bapco.com/about/`, accessed: 2020-12-20
[25] `https://bapco.com/products/sysmark-2018/`, accessed: 2020-12-20
[26] `https://results.bapco.com/results/benchmark/SYSmark_2018`, accessed: 2020-12-20
[27] `https://www.spec.org/spec/faq/`, accessed: 2020-12-20
[28] `https://www.spec.org/gwpg/gpc.static/vp13info.html`, accessed: 2020-12-20
[29] `https://www.spec.org/accel/`, accessed: 2020-12-20
[30] `https://www.spec.org/cpu2017/`, accessed: 2020-12-20

technologies. Moreover, the group's objective is to foster collaboration between academia and industry. The RG is again divided into multiple working groups based on their focus topics, e.g., big data and cloud[27,31].

- *Transaction Processing Performance Council (TPC)*
  TPC is another non-profit corporation. It was founded in 1988 to overcome the issue of inadequate use of benchmark results for marketing reasons. However, even after the publication of the first standardized benchmark, there were still complaints about misleading uses of benchmark results. In response, TPC introduced a review process for performance benchmarks. This process introduced requirements for extensive documentation as well as an independent audit. All imposed qualifications need to be satisfied before benchmark results get officially published. Nowadays, there are certified TPC auditors for verifying the validity of such benchmark results [NLW+09].

  Regarding the benchmark domain, TPC's original objective was to create transaction processing and database benchmarks, for which it is well-known. Organizationally, TPC is headed by the *General Council*, which consists of all TPC member companies. This group includes, e.g., AMD, Intel, and IBM. Underneath, there are standing and technical subcommittees. The standing committee takes care of administrative, public relations, and documentation topics. The technical group's responsibilities include proposing new benchmarks and maintaining existing ones[32] [NLW+09].

  One of the most popular TPC benchmarks is the before-mentioned TPC-C benchmark, which describes itself as an OLTP benchmark. Even though TPC-C got already approved in 1992, it is still relevant. It is still being applied by academia and industry, which is indicated by, e.g., the TPC-C results website. The website lists multiple recent result publications from 2019[33] [Cou90]. However, Krüger et al. showed through analyzing an SAP Business Suite system that the TPC-C workload is at least not representative for all enterprise OLTP systems, as it contains too many write operations [KKG+11]. This analysis showed that also standardized benchmarks can fail to represent real-word scenarios.

[31] `https://research.spec.org/en/working-groups.html`, accessed: 2020-12-20
[32] `http://www.tpc.org/information/who/whoweare5.asp`, accessed: 2020-12-20
[33] `http://www.tpc.org/tpcc/results/tpcc_last_ten_results5.asp`, accessed: 2020-12-20

2.2.3 Design Principles for Performance Benchmarks

Jim Gray [Gra93] defined four criteria a benchmark designed for a certain domain should fulfill. These criteria influenced other publications that provide benchmarks or guidelines in the area of benchmark development, which illustrates their impact [Hup09, FAS+12, BKD+14]. The benchmark proposed in this dissertation is also based on these aspects, which are:

- **Relevance**, meaning that the workload should represent typical operations of the corresponding domain,
- **Portability**, meaning that a benchmark implementation on different systems and architectures should not be an issue,
- **Scalability**, meaning a benchmark should be compatible with small as well as large architectures, and
- **Simplicity**, meaning a benchmark needs to be understandable to ensure credibility [Gra93].

There is also newer work in the area of benchmark design, such as the work by Karl Huppler [Hup09]. He defines five characteristics of good benchmarks and mentions that it is not possible to be perfect in every aspect. According to him, it is rather usual to have strengths in one or two of these areas. Specifically, Huppler states that a good benchmark is:

- **Relevant**, i.e., results should convey they are showing something of importance,
- **Repeatable**, i.e., multiple runs should result in the same results,
- **Fair**, i.e., all systems involved can equally participate, including that a benchmark should, e.g., be portable and not be optimized for a certain environment,
- **Verifiable**, i.e., there should be confidence that the benchmark results actually represent the system under test's (SUT's) performance, and
- **Economical**, i.e., it should be affordable to implement or use the benchmark, not only considering the monetary costs but also, e.g., implementation efforts [Hup09].

Another work by v. Kistowski et al. [vKAH+15] analyzes multiple definitions and summarizes the relevant aspects very similar to Huppler. The only major

distinction is related to the last characteristic, i.e., the requirement of being economical. Particularly, v. Kistowski et al. view this criteria from a broader perspective summarized as *usability*. This usability aspect reflects the need to allow benchmark users to execute a benchmark in their test environments, without facing major obstacles. Economical hurdles can be seen as an example of such an obstacle.

Overall, all descriptions of desired benchmark characteristics are free of contradiction and have certain overlaps, e.g., with respect to workload relevance or portability. Moreover, it is likely that the simpler a benchmark, the easier to implement or use, and hence, more economical due to lower implementation efforts. So there are aspects from the mentioned authors that might be viewed differently, but that closely relate to each other.

3

ESPBENCH - THE ENTERPRISE STREAM PROCESSING BENCHMARK

This chapter introduces ESPBench, the enterprise stream processing benchmark that is one of the main contributions of this thesis. After describing the benchmark scenario, we introduce input data.. Subsequently, we present ESPBench's architecture, the benchmarking process, and the benchmark queries. At the end of this chapter, we recapitulate the design objectives for developing a good benchmark and how they apply for ESPBench.

The included artifacts, such as the ESPBench toolkit and the example query implementations, have been published in a public software repository[1] as well as in an open-access research repository [HM20].

3.1 Scenario

The benchmark scenario used in the following is inspired by the *Grand Challenge* published in 2012 at the *ACM International Conference on Distributed and Event-Based Systems* (DEBS) [JHF+12]. The scenario simulates the setting of a manufacturing company that purchases materials and semi-finished goods, which are being processed and assembled to a finished product at certain workplaces. Along this process, high-tech manufacturing machines are used for production.

The company aims to improve monitoring and analytical possibilities regarding its manufacturing processes through the use of data stream processing. It wants to reduce the effect of irregularities by, e.g., identifying them earlier and

[1] `https://github.com/hpi-epic/ESPBench`

having more information on their impact and character. Added values, e.g., a better understanding of processes, are also created through the combination of sensor and business data, often referred to as data integration or vertical integration in the context of Industry 4.0. While horizontal integration describes a data integration along the value chain, e.g., by combining business data through foreign key dependencies within the corresponding database tables, vertical integration is different. Vertical integration depicts the combination of data from different technical levels. It stands for the connection of business data stored in an enterprise resource planning (ERP) system with technical data, such as sensor measurements captured at manufacturing machines. In between, there might be other data sources that can be integrated, like machine execution systems (MES) [HSMU19].

For realizing such a detailed production overview, sensors are employed to the technical equipment. They keep track of multiple aspects, e.g., the amount of electrical power consumed or the state of a release valve for a chemical additive. Data stream processing systems (DSPSs) are a natural fit for such a use case where continuous data analysis is needed. In order to allow interpretations of measurements and further sense-making, the captured streaming data needs to be integrated with traditional business data.

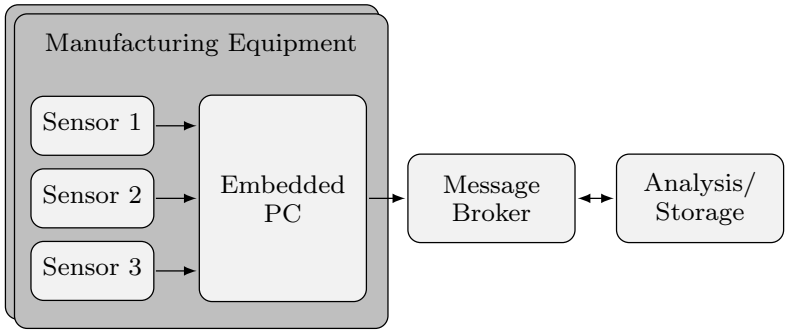


Figure 3.1: Sensor architecture at the manufacturing equipment in ESPBench (based on [JHF+12])

Figure 3.1 visualizes the sensor arrangement at the manufacturing equipment. Multiple sensors are installed on a single machine. The measured data is collected using an embedded PC, which provides the data for further analyses. Within the benchmark scenario, there are two such machines sending sensor values of the same structure.

3.2 Data

The data used for ESPBench comprises two major kinds of data, traditional business data as well as sensor data. The ESPBench repository contains instructions on how to obtain these types of data, which is by either downloading them from the internet or generating them using the provided benchmark tools.

Both types of data distinguish themselves from each other in multiple aspects, such as volume and velocity. A general overview of how the two types of data compare with respect to selected characteristics is given in Table 3.1.

Characteristic	Sensor Data	Business Data
Volume and Velocity	Up to multiple terabytes created by a single manufacturing machine, daily [HVN16]	Multiple terabytes in total, e.g., for a 20 years old SAP ERP installation at a leading Canadian energy company [Sou17]
Data Quality	Measurement errors, lost data	Correctness crucial for business
Data Manipulations	No updates	Updates exist
References	Strong time and location reference	Strong business process reference
Value for Enterprises	Usually not crucial for daily business	Essential for daily business

Table 3.1: Conceptual comparison of sensor and business data (based on [HRM+17])

3.2.1 Sensor Data

ESPBench uses two types of data streams that need to be processed by the SUT. The first one is based on the data set used at the *DEBS Grand Challenge* 2012, which contains measurements from multiple sensors combined to single records. It is collected using an embedded PC as shown in Figure 3.1. There are two machines sending this kind of data. The record's structure is shown in Table 3.2, which is almost identical to the structure used at the *DEBS Grand Challenge* 2012. ESPBench extends the existing structure by a single column containing the generated *workplace id*, which identifies the workplace where the data is captured. It enables combining sensor and business data.

The original *DEBS Grand Challenge* 2012 data contains measurements from analog as well as binary sensors installed at high-tech manufacturing equipment. The latter kind produces either 0 or 1 as a value. Analog sensors send data either

#	Technical Information	Description
1	required fixed64 *ts*	timestamp
2	required fixed64 *index*	message index
3	required fixed32 *mf01*	electrical power main phase 1
4	required fixed32 *mf02*	electrical power main phase 2
5	required fixed32 *mf03*	electrical power main phase 3
6	required fixed32 *pc13*	anode current drop detection cell 1
7	required fixed32 *pc14*	anode current drop detection cell 2
8	required fixed32 *pc15*	anode current drop detection cell 3
9	required uint32 *pc25*	anode voltage drop detection cell 1
10	required uint32 *pc26*	anode voltage drop detection cell 2
11	required uint32 *pc27*	anode voltage drop detection cell 3
12	required uint32 *res*	unknown
13-18	required bool *bm05-bm10*	chemical additive information
19-66	optional bool *pp01-pp36*, *pc01-pc06*, *pc19-pc24*	unknown
67	required fixed32 *workplaceid*	workplace ID

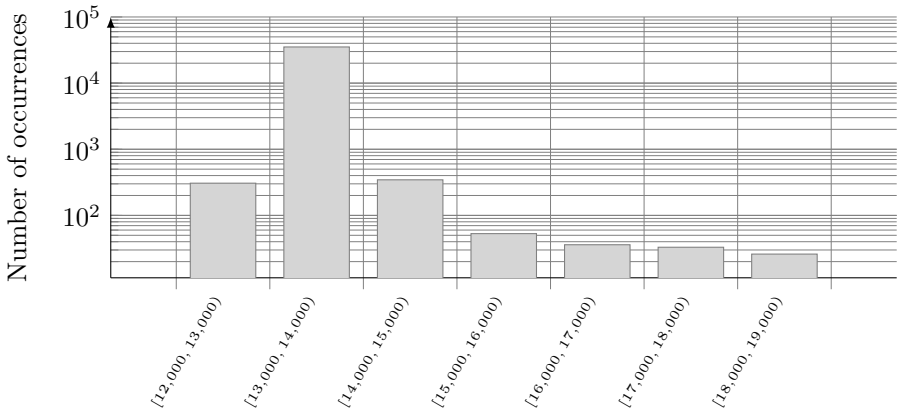
Table 3.2: Data structure of the sensor measurements stream (based on [JHF+12])

in the range of 0 to $2^{16} - 1$ or $-(2^{15})$ to 2^{15}. The actual data can be obtained online[2] and only requires minor benchmark-specific modifications. Particularly, the file needs to be duplicated and each of the resulting files needs to be extended by the *workplace ID* column. That can be done via a command line operation, which is explained in the ESPBench software repository. One resulting data file belongs to workplace *one*, while the other one is associated with workplace *two*.

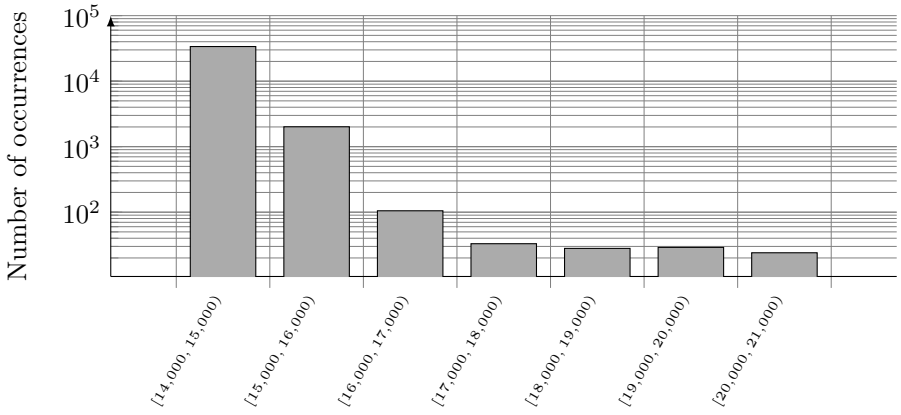
The columns used for the first stream of sensor data for ESPBench queries are *mf01*, *mf02*, and *mf03*, i.e., the electrical power on main phases one, two, and three. It represents the energy consumption of the manufacturing tools. This data is valuable for, e.g., identifying irregularities in the production process. The data distributions of these three columns are visualized in Figure 3.2. It is important to note that all sub-figures use a logarithmic y-axis.

The value distribution for column *mf01* in Figure 3.2a shows that most values are in the range of [13,000, 14,000). ESPBench query 3 for instance filters for

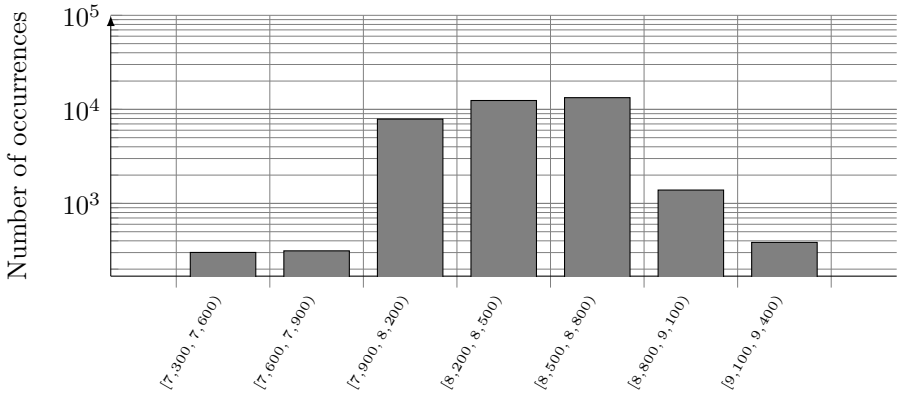
[2] `ftp://ftp.mi.fu-berlin.de/pub/debs2012/`



(a) Histogram visualizing the value distribution of column *mf01* - electrical power on main phase 1



(b) Histogram visualizing the value distribution of column *mf02* - electrical power on main phase 2



(c) Histogram visualizing the value distribution of column *mf03* - electrical power on main phase 3

Figure 3.2: Histograms visualizing the value distributions of columns *mf01* – *mf03* - electrical power on main phases 1-3

values higher than about 15,000. That is where the 99.5^{th} percentile is located. Figure 3.2a depicts that such values exist, while not being the majority.

Figure 3.2b shows a slightly different value distribution for *mf02* with the majority of values belonging to the first bin, i.e., to the range [14,000, 15,000). In contrast to that, the value distribution of *mf03*, which is visualized in Figure 3.2c, shows only a few values in the first two bins as well as in the last bin. Most of the values of *mf03* concentrate in the middle three to four bins. However, the power distribution of main phase three distinguishes from the other two as its overall range is smaller, so the power consumption on main phase three is less volatile. While the other two span a range of about 7,000, a range of 2,100 is already enough to cover all values existing for main phase three. Most of the values of *mf03* are distributed across three bins. However, they only span a range of [7,900, 8,800), which is a spread even smaller than one bin in the previous two histograms.

The second type of data stream coming from the manufacturing equipment consists of data about the production times, i.e., no measurements of energy usage or comparable environment characteristics. Having these data allows for combining sensor and business data. The data set is not part of the *DEBS Grand Challenge*, but created by the ESPBench data generator tool. The data stream's structure is depicted in Table 3.3. It contains the *order id*, *order line number*, *production order line number*, and a column that indicates whether the corresponding product entered or left the workplace.

#	Technical Information	Description
1	required uint32 *pt_o_id*	order id
2	required uint32 *pt_ol_number*	order line number
3	required uint32 *pt_pol_number*	production order line number
4	required bool *pt_is_end*	indicates whether a product is entering or leaving a workplace

Table 3.3: Structure of the sensor data used for production time determination

The structure of the business data, which is visualized in Figure 3.3, reveals that the first three columns shown in Table 3.3 are the primary key of the table *PRODUCTION_ORDER_LINE*, and thus can identify the workplace. Data specifying when products enter and leave a workplace are needed for time-based vertical data integration. Such a scenario is, e.g., linking sensor measurements at manufacturing machines to products, which are or have been processed at these machines. Having such a holistic view of the value chain with detailed production information is a significant asset for businesses, as it enables them

to gather valuable insights. These insights can lead to competitive advantages, e.g., by providing a better customer experience through offering more details on the production process with regard to customer orders [HMSU19].

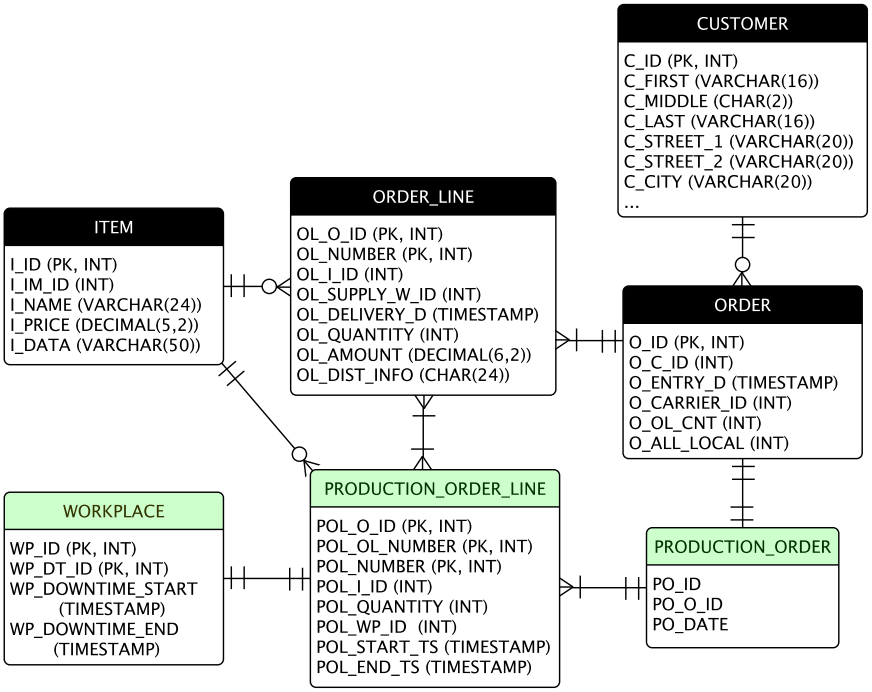


Figure 3.3: Business data in ESPBench in Crow's Foot Notation [Hit02] - adapted TPC-C schema (based on [HMP+21])

3.2.2 Business Data

The schema of the business data is depicted in Figure 3.3. It is based on the TPC-C benchmark data schema [LD93]. The data model covers basic relations, such as *CUSTOMER* and *ORDER*, that are representative for any manufacturing company as well as for other industries like, e.g., consumer goods and energy. In the context of ESPBench, we simplified the TPC-C table design without impacting the query costs as we removed tables not used by the benchmark queries. Inspired by modern ERP systems, we also added new relations, which incorporate the domain character of industrial manufacturing. In particular, we removed the tables *WAREHOUSE*, *STOCK*, *DISTRICT*, *HISTORY*, and *NEW-ORDER*. We extended the schema by the tables *PRODUCTION_ORDER*, *PRODUCTION_ORDER_LINE*, and *WORKPLACE*, which are highlighted in green in Figure 3.3. In the default configuration, to have a representative data

size, the business data is generated with a *scale factor* of three, which equals a TPC-C setting with three warehouses. This scale factor impacts the overall business data size and can be altered for scaling reasons.

The *WORKPLACE* contains information about scheduled downtimes. These data allow to distinguish planned downtimes from irregularities that require reactions. The other two added tables store data about the production orders, which are linked to the customer orders and workplaces. Storing business entities, such as sales or production orders, in a header and an item table is a common and thus a relevant concept in modern ERP systems [HMSU19, Pla09].

3.3 Architecture

Figure 3.4 shows the high-level overview of our idea of an architecture for a performance benchmark focused on DSPSs. Knowing about this simplified view helps getting a better understanding of the idea beyond ESPBench's architecture. Figure 3.4 shows three main components: the *data sender*, the *system under test*, and the *validator and result calculator*.

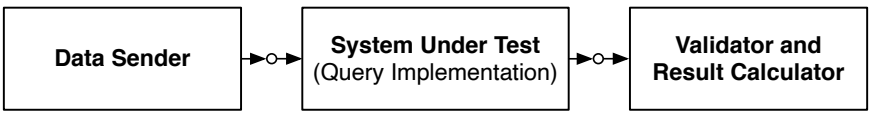


Figure 3.4: General architecture of a data stream processing benchmark in FMC (based on [HMRU17])

The data sender is responsible for transmitting data to the system under test (SUT), i.e., for creating the data stream. In the context of ESPBench, or stream processing benchmarks in general, the SUT processes incoming data and responds according to the defined queries. Produced results are evaluated by a result validator in order to ensure the correctness of the query implementations. This component can also calculate benchmark result metrics, e.g., the latencies. Computing results independently from metrics that might be provided by an SUT ensures objectivity and comparability, as metric calculations among SUTs might differ.

The detailed architecture of ESPBench is visualized in Figure 3.5. The components labeled as toolkit are provided as part of ESPBench. They simplify the application of the benchmark and ensure objective results. The single components are introduced in detail in the following.

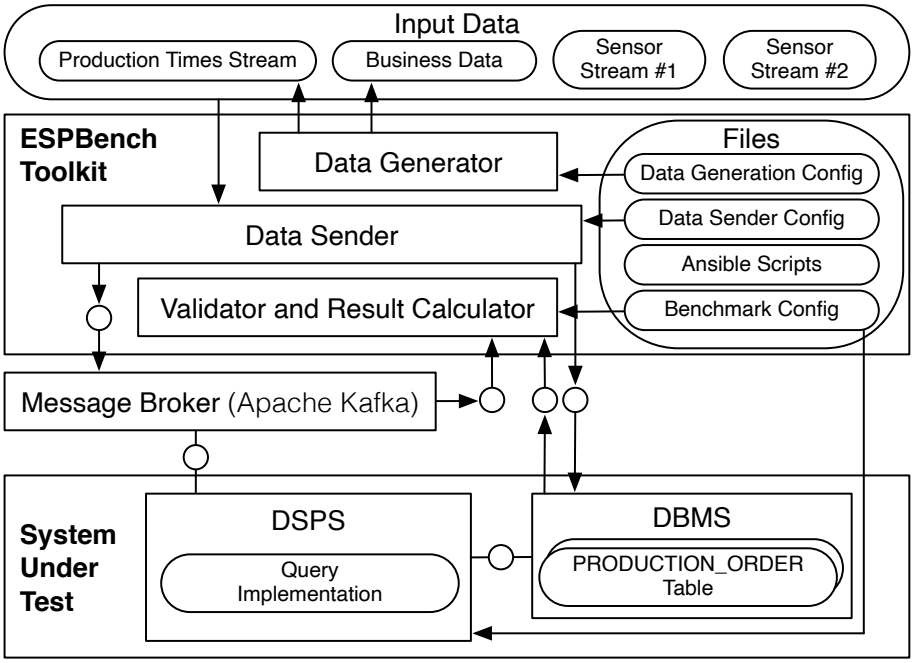


Figure 3.5: Architecture of ESPBench in FMC ([HMP+21])

3.3.1 Input Data

The input data is present as files in comma-separated values (CSV) format. Section 3.2 describes the data characteristics and how to obtain the data. The *Sensor Stream #1* and *Sensor Stream #2* files illustrated in Figure 3.5 are slightly modified versions of the *DEBS Grand Challenge* data set, which can be found online[3]. The other two files, i.e., the *Business Data* and the *Production Times Stream*, are generated by the provided *Data Generator* tool.

3.3.2 Data Generator

The *Data Generator* is one of the provided benchmark tools. It is responsible for creating two types of data as mentioned before, *Business Data* and the *Production Times Stream* data. This data generation is fully automated by ESPBench. The benchmark defines a configuration file for data generation that determines the scale factor as well as the output directory for the CSV files.

[3] `ftp://ftp.mi.fu-berlin.de/pub/debs2012/`, accessed: 2021-08-28

3.3.3 Data Sender

The *Data Sender* is another part of the ESPBench toolkit. It is used for two tasks: the import of the business data into the DBMS, which is part of the SUT, as well as the ingestion of streaming data into the *Message Broker*. There is also a configuration file for the data sender. It is used for defining parameters, such as the data input rate or the locations of the CSV files that serve as input data.

3.3.4 Message Broker

ESPBench incorporates the message broker *Apache Kafka* as the interface to the SUT. This combination of a message broker with data processing systems, e.g., a DSPS, can be found in various existing data processing landscapes in academia as well as industry [HRM+17]. Thus, the architectural decision to include a message broker represents real-world environments. The message broker's role in ESPBench reflects reality and so adds relevance to the benchmark. Besides, message brokers are also part of other performance benchmark designs [HMRU17, HRM+17, LWXH14, vDdP20].

An additional reason for using Apache Kafka is scalability with respect to ingesting data. If the data sender was to directly send data via sockets to the SUT, a change in the number of sockets would require changing the query implementations, since the additional connections would need to be handled by the SUT. Apache Kafka topics provide a solution to this challenge. As mentioned in Section 2.1.3, an arbitrary number of Apache Kafka producers can send data to a certain topic, which is internally distributed across the cluster by Apache Kafka. The SUT application receives data from a topic, allowing the number of producers to be adapted independently. The use of Apache Kafka topics therefore allows for an arbitrary number of data senders, e.g., for scaling data ingestion rates, without the need to modify query implementations. To ensure the correct order of records within Apache Kafka topics, we use topics with a single partition, which allows order preservation in Apache Kafka. This design is visualized in Figure 3.6. This single partition allows for sufficiently high ingestion rates as shown in Section 4.3. If higher data ingestion rates are required and result order is not of importance, ESPBench also allows for multiple partitions. The modification only requires an adaption of the result validator tool.

Another reason for using Apache Kafka is latency measurements. In order to achieve latency results that are as correct and comparable as possible,

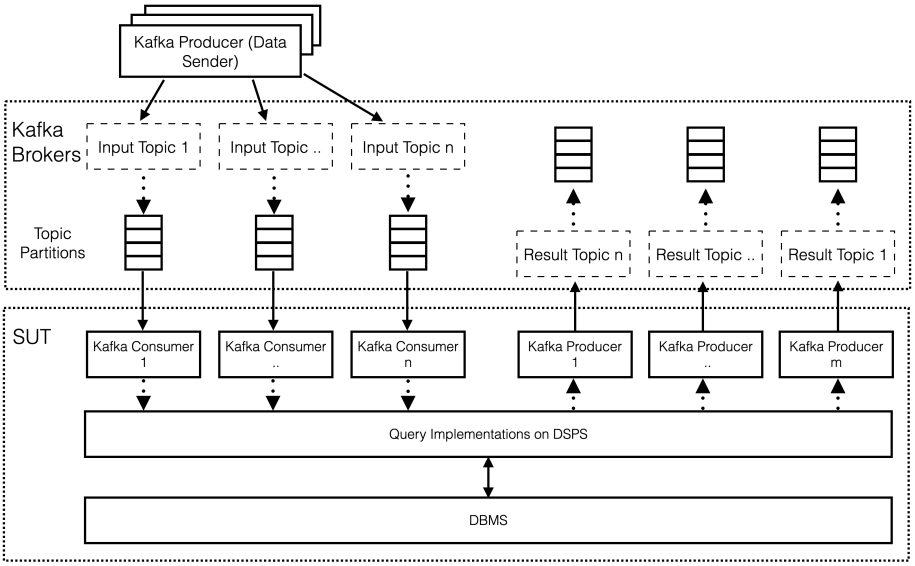


Figure 3.6: Detailed view on the role of Apache Kafka within the ESPBench architecture as data source and result data storage for the SUT (based on [HRM+17])

we leverage Apache Kafka's timestamp functionality. Specifically, we use the *LogAppendTime* configuration described in Section 2.1.3. The correspondingly captured timestamps, which represent the time when input data is made available to the SUT and when results are stored, are taken into account for latency calculations. By doing so, it is possible to keep those calculations independent of the SUT. Thus, no modifications of query implementations for result calculation reasons are needed. Additionally, system-dependent differences or variations in terms of time measurements can be preempted. That allows retrieving objective and comparable results.

The downside of this approach is the included overhead time that is needed for transferring messages from the Apache Kafka broker to the SUT and back. This overhead time does not reflect the actual computation time of the SUT. However, we do not consider that as an issue for the latency measurements of ESPBench. Since all benchmarked systems follow the same approach, the overhead is included in all measurements and so results remain comparable in similar environments. This situation is given if influencing parts, such as the network connection between Apache Kafka and the SUT, stay constant. Another potential influence on the latency measurements using the proposed approach is the batching mechanism of Apache Kafka producers. However, our analyses presented in Section 4.3.2 indicate that the impact is neglectable. Thus,

the presented latency measurement approach is an objective way of calculating latencies in the context of ESPBench. It allows for benchmarking any implementation that is able to retrieve and send data to and from Apache Kafka.

Besides, it is crucial to ensure that the message broker does not become a bottleneck. This has to be prevented since the objective of a benchmark is to analyze the SUT and not any of the surrounding tooling components. So in the context of ESPBench, an input rate that Apache Kafka can manage needs to be configured.

3.3.5 System Under Test

The SUT is responsible for answering the defined benchmark queries and is supposed to comprise a DSPS and a DBMS. *PostgreSQL* [SR86], as a well-known and widely used DBMS, is the default database for ESPBench. It can be exchanged by any other DBMS, e.g., Hyrise [GKP+10] or SAP HANA [FCP+11], which requires minor adaptions in the toolkit. Particularly, the components communicating with the DBMS, i.e., the data sender and validator, need to be adapted accordingly. The SUT reads the input data from Apache Kafka and, if needed, from the DBMS. It writes results back to either Apache Kafka or the DBMS, depending on the query. The flexible benchmark architecture allows analyzing applications, which are able to read from and write to Apache Kafka and the used DBMS. So the concrete SUT architecture is up to the ESPBench user, which means the user has a certain degree of freedom with respect to design decisions.

3.3.6 Validator and Result Calculator

To determine the correctness of the query answers and to calculate objective benchmark results, we provide the *Validator and Result Calculator*. This component runs after the data sender and the SUT have finished. It leverages the *Akka*[4] framework for processing data streams. The application rereads and reprocesses the input data from Apache Kafka and, if needed, PostgreSQL. It then calculates the query results and compares them to the output created by the SUT.

Moreover, the *Validator and Result Calculator* computes latencies, i.e., timestamp differences. It uses the Apache Kafka timestamps or DBMS timestamps for that, so the times taken when a record is written to the Apache Kafka log or DBMS. By doing so, we ensure objectivity as we do not rely on perfor-

[4] `https://akka.io`, accessed: 2020-12-22

mance measurements of DSPSs, which can conceptually differ. Additionally, performance characteristics created by systems might ignore queuing, which would not represent the real experiences of system users. That can be avoided by using Apache Kafka and DBMS timestamps as previously mentioned. This concept is similar to the separation of the driver, i.e., the data sender in the context of ESPBench, and the SUT as presented by Karimov et al. [KRK+18].

3.4 Benchmark Process

The activity diagram shown in Figure 3.7 gives a brief overview of the ESPBench benchmark process. The visualized process steps are exchangeable and can be extended. The entire process sums up the *ansible*[5] scripts that we developed for automating ESPBench in order to simplify its usage. One benchmark run corresponds to executing all steps of Figure 3.7. Starting a benchmark run only requires the execution of a single script. The modularization in different files eases process adaptions, such as re-orderings, extensions, or replacements of benchmark steps. The published ESPBench repository contains all ansible files.

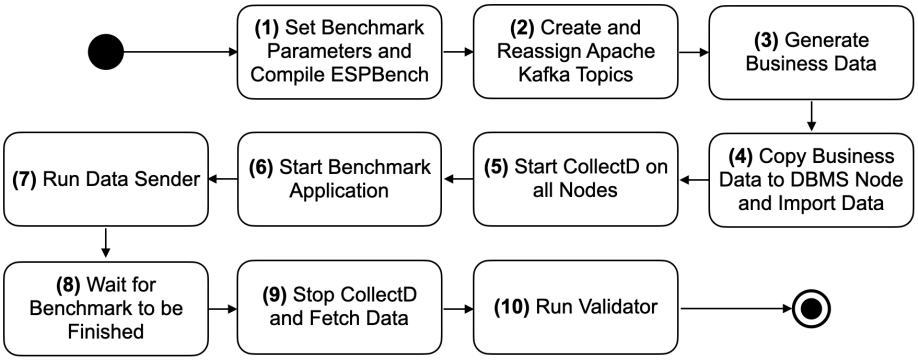


Figure 3.7: ESPBench process visualized as an Activity Diagram [DtH01] ([HMP+21])

The single process steps of Figure 3.7 are described in the following:

1. At the beginning, the script sets benchmark parameters and compiles the ESPBench project to a fat *Java Archive* (JAR) file using the *sbt-assembly*[6] plugin. This tool is an extension to *sbt*[7], a build tool for, e.g., Scala and Java projects. The parameter setting is only relevant if parameters are passed to the ansible script as command line parameters. If not, all parameters are read from the configuration files defined by ESPBench.

[5] `https://www.ansible.com`, accessed: 2020-12-22
[6] `https://github.com/sbt/sbt-assembly`, accessed: 2020-12-22
[7] `https://www.scala-sbt.org`, accessed: 2020-12-22

2. In the second step, the script creates the needed Apache Kafka topics according to the configuration and the naming schema defined by ESPBench. Furthermore, it reassigns the created topics to assure an even topic distribution across the Apache Kafka brokers.

3. The ansible script triggers the data generator tool to perform the generation of business data according to the configuration.

4. The created business data files are copied to the DBMS server and imported into the database system using the data sender tool. If tables that are to be imported already exist in the database schema, e.g., due to previous benchmark runs, they are truncated.

5. Afterward in the fifth step, the tool *collectd*[8] is started on all servers to gather system data during the benchmark run. Doing so allows to investigate detailed differences between systems or system configurations, e.g., with respect to the memory consumption or CPU utilization. Collecting such data can also benefit the analyses of discovered bottlenecks.

6. Step six starts the benchmark query or queries that are to be executed in the current benchmark run.

7. After a few seconds wait time for allowing the DSPS to process the submitted application, the script invokes the data sender tool. It sends the streaming data to the corresponding Apache Kafka topic(s). The data sender runs for the configured period of time. The ESPBench naming convention allows query implementations to identify the needed topic names with the data available in the configuration files.

8. Once the data sender tool finishes, the ansible script waits for some seconds for the benchmark query or queries to finish data processing.

9. *collectd* is stopped and the recorded data is fetched to the server where the ansible script was invoked, i.e., to the location where all logs and result files are gathered.

10. At the end, the *validator and result calculator* tool is executed. It checks the query result correctness and computes the benchmark results. The tool determines aggregated results, such as the mean latency and percentiles, as well as the single latencies for each output record. It writes the output to log and CSV files, which can be used for further analysis, such as plotting of single query result latencies.

[8] `https://collectd.org`, accessed: 2020-12-22

3.5 Queries

This section presents the benchmark queries designed by ESPBench that the SUT is tasked with. When defining benchmark queries, relevance and simplicity need special consideration. Having easily understandable queries is a crucial requirement for, e.g., result credibility reasons. Without knowing what is happening during benchmark runs, it is hard to draw conclusions based on the benchmark results. Moreover, trusting results becomes a challenge if it is unclear how they arose. Another aspect influenced by simplicity is the application of the benchmark. Understanding the workload is essential for implementing the queries for other stream processing architectures, and thus for using the benchmark. Regarding the design goal of query relevance, the proximity of the benchmark workload to real-world scenarios and the coverage of important stream processing functionality need to be considered. To ensure the latter aspect, we cover all core functionalities of DSPSs presented in Section 2.1.1.

Table 3.4 shows the queries defined by ESPBench. They fully cover the before-mentioned core operations of DSPSs. The complete coverage ensures the relevance of the benchmark and adds to the conducted validation of the queries with industry partners from the industrial manufacturing domain. Next to the use case behind each query, the covered DSPS functionalities are referenced in Table 3.4. A query definition and a description are also part of this table. Queries are defined using a syntax inspired by the *continuous query language* (CQL) [ABW06]. Example implementations using the Apache Beam software development kit (SDK) are available in the ESPBench software repository.

Query 1 - Check Sensor Status
The query *Check Sensor Status* monitors the sensor value *mf01*, i.e., the electrical power in main phase one. This provides insights into irregularities to operators as soon as possible by calculating useful and up-to-date key performance indicators (KPIs). The query computes the average, minimum, maximum, and the overall number of sensor values in tumbling windows of one second. The result records contain the four calculated values separated by a comma.

Query 2 - Determine Outliers The second query determines outliers based on the values *mf01* and *mf02*, i.e., the electrical power measurements for main phases one and two of the manufacturing equipment. The query results give hints on possible irregularities in the manufacturing equipment. We employ the *stochastic outlier selection* algorithm [Jan13] on count-based tumbling windows of 500 elements. The output consists of records with a probability of being

#	Use Case	Tested Aspects*	Query Definition	Description
1	Check Sensors	1;2;3	**SELECT** AVG(mf01), MIN(mf01), MAX(mf01), COUNT(mf01) **FROM** STREAM_SENSOR **TUMBLING WINDOW** 1 **SECONDS**	Calculate *avg, min, max, count* for the last 1sec for *mf01* for monitoring.
2	Determine Outliers	1;6	**SELECT** STOCHASTIC_OUTLIERS(mf01, mf02), outlier_probability **FROM** STREAM_SENSOR **CUSTOM WINDOW** 500 **ELEMENTS** **WHERE** threshold >= 0.5	Calculate outliers using Stochastic Outlier Selection [Jan13] for combination of *mf01* and *mf02*. Output records that are an outlier with at least 50% probability.
3	Identify Errors	4	**SELECT** * **FROM** STREAM_SENSOR **WHERE** mf01 > 14,963	Log if sensor value electrical power main phase 1 exceeds limit of 14,963.
4	Check Machine Power	5;7	**SELECT** * **FROM** STREAM_SENSOR1 **AS** s1, STREAM_SENSOR2 **AS** s2, DB_TABLE_1 **AS** t **WHERE** (s1.M_ID = t.M_ID **AND** s1.mf03 < 8,105 **AND** (s1.TS > t.DOWNT_END **OR** s1.TS < t.DOWNT_START)) **OR** (s2.M_ID = t.M_ID **AND** s2.mf03 < 8,105 **AND** (s2.TS > t.DOWNT_END **OR** s2.TS < t.DOWNT_START))	Log if any machine is in an unscheduled phase of being turned off or on stand-by (assumption: there is always the next downtime stored in DB_TABLE_1).
5	Persist Processing Times for Products	4;7	**UPDATE** PRODUCTION_ORDER_LINE **IF** (STREAM_TIMES.PT_IS_END == 0) { **SET** POL_START_TS = (**SELECT** TIMESTAMP **FROM** STREAM_TIMES) } **ELSE** { **SET** POL_END_TS = (**SELECT** TIMESTAMP **FROM** STREAM_TIMES) } **WHERE** POL_O_ID = STREAM_TIMES.PT_O_ID **AND** POL_OL_NUMBER = STREAM_TIMES.PT_OL_NUMBER **AND** POL_NUMBER = STREAM_TIMES.PT_POL_NUMBER	Whenever a product enters or leaves a workplace, log the time to the corresponding DBMS entry (PRODUCTION_ORDER_LINE table).

Table 3.4: Query set of ESPBench (based on [HRM+17, HMP+21])

* 1. Windowing, 2. Transformation, 3. Aggregation/Grouping, 4. Filtering (Selection/Projection), 5. Correlation/Enrichment (Join), 6. Machine Learning, 7. Combination with Historical Data

an outlier that is equal to or higher than 50%. This allows to identify possible irregularities. Structurally, the output is represented as the corresponding input sensor record plus the outlier probability correct to two decimal places, which is separated from the corresponding sensor record by a comma.

Query 3 - Identify Errors

Query three reports actual errors. These errors are defined by the unusually high power consumption of a manufacturing machine, i.e., a value greater than 14,963, the 99.5 percentile in the main phase one, see Figure 3.2a. The output of query three are the corresponding input sensor records that indicate an error situation.

Query 4 - Check Machine Power

The fourth query checks the machine power, i.e., if the electrical power is unexpectedly low. Such a situation requires action. As input, this query gets two structurally identical input streams from two workplaces, i.e., two manufacturing machines. If any of the machines are in an unplanned phase of being shut-down or on standby, the corresponding record needs to be logged. This is the case if the electrical power in main phase three, i.e., column *mf03*, falls below the value of 8,105, the 9^{th} percentile, and there is no downtime planned for the machine. Planned downtimes have to be looked up in the DBMS, which is part of the SUT. Query four creates a holistic view of a part of the manufacturing process by combining sensor data and historical business data. The table *WORKPLACE* stores the beginning (*WP_DOWNTIME_START*) and the end (*WP_DOWNTIME_END*) of the next scheduled downtime for any workplace, which is identifiable by its ID (*WP_ID*). This information is also part of the sensor data as presented in Table 3.2.

Query 5 - Persist Processing Times

Query five represents another use case where sensor data and business data are combined. It stores time information in the database, which allows for the mentioned data integration, i.e., for connecting sensor data with business data by using a timestamp-based approach. Precisely, the DBMS relation *DB_PRODUCTION_ORDER_LINE*, which contains information about the factory's production orders, needs to be updated in query five. So contrary to query four, which reads business data, query five manipulates data stored in the DBMS, specifically the production times. The data input for this query is the second type of data stream with its structure shown in Table 3.3. This data indicates when a product or a part of it entered or left a workplace or machine.

The current timestamp needs to be set in the corresponding table column of *PRODUCTION_ORDER_LINE*, i.e., either the start or end timestamp column needs to be updated, depending on the incoming sensor record.

3.6 Review of Design Principles

The ESPBench design decisions are based on the four criteria defined by Gray [Gra93], namely relevance, portability, scalability, and simplicity. Although these aspects defined by Gray were already published in the early 90s, several later work builds upon them, and thus they are still valid for state-of-the-art benchmarks [Hup09, FAS+12, BKD+14, HRM+17]. Moreover, the great popularity of Gray's criteria indicates their value for the area of performance benchmarking. The four benchmark characteristics and their translation to the context of DSPS benchmarks are illustrated in the following.

Relevance
To satisfy the aspect of relevance, the benchmark architecture aims to represent real-world scenarios and system environments as realistically and representatively as possible. This covers data characteristics, the defined benchmark queries, as well as involved benchmark components.

One major aspect with respect to data is the incorporation of business data to fully reflect data characteristics at enterprises. Furthermore, relevance can be achieved by employing real-world data in the benchmark, which ideally reflects the characteristics of all industries. However, getting sufficient rights to use real-world data, especially when it comes to enterprise business data, is a challenge as many companies keep their data as a secret. That might help, e.g., to keep or enhance possibly existing competitive advantages or to be less predictable for competitors. Not publishing data used within a benchmark hinders the reproduction of results and the application of the benchmark in other scenarios. A lack of available data results in low result credibility and an overall limited value of a benchmark for the community. This is why ESPBench only employs data that is accessible without any restriction.

If no fitting data set is obtainable, one can generate synthetic data. This data should be as close to real-world data as possible for relevance reasons. It can be achieved through generating data based on previously collected real-world data sets, which alone, i.e., without the generated part of data, would not be sufficient regarding their size. This technique is applied by several benchmarks as presented in Chapter 5. The details of ESPBench's input data are described in Section 3.2.

With respect to the data input rate, i.e., the number of incoming records per time unit which need to be processed by the SUT, ESPBench can be configured according to a user's needs. This freedom of choice ensures a lasting relevance as the user of ESPBench can adapt the data input rate in accordance with altered environments. Technology developments that might lead to increasing input rates in real-world scenarios thus can be benchmarked using ESPBench in its current version.

Another area, where relevance needs to be considered, is the queries, i.e., the logic that is going to be executed by the SUT. These queries have the objective to answer useful and practically beneficial questions, while covering the core functionalities of DSPSs. The ESPBench queries satisfy these requirements. The relevance in terms of closeness to real-world scenarios is ensured by, e.g., validating the queries with two industry partners from the corresponding benchmark domain, namely industrial manufacturing. Although the benchmark focuses on a single domain, query characteristics and used functions, such as filtering or aggregating values of a data stream, are applicable to domains apart from manufacturing. Hence, benchmark results are also beneficial for users from different industries [HRM+17]. Moreover, ESPBench's architecture and tools can be leveraged as a platform, which can be used for defining a new benchmark workload belonging to another domain.

Portability

To recap the portability definition, a benchmark shall be as independent from the operating system and system as possible. As one step towards this direction, ESPBench enables implementing the benchmark for as many different environments as possible. As a consequence, a potentially large number of implementations or usages can be reached, due to the obsolete or comparatively low porting efforts. A high number of implementations contributes to a high relevance and result credibility.

ESPBench further ensures portability by allowing for a free choice of operating systems or employed technologies for the benchmark implementation. Although DSPSs might seem like a natural fit for data stream processing, a DSPS can potentially be exchanged with any other system or implementation that is able to answer the defined queries. ESPBench also provides freedom of choice when it comes to the DBMS as a recommended part of the SUT.

While minor adaptions in the toolkit are required when changing data processing systems used as the SUT, aspects like the CPU architecture or the server operating system can be altered without the need for code adaptions. With re-

spect to the benchmark toolkit itself, it is compatible with many operating systems and platforms as it is developed using the Java Virtual Machine (JVM) language Scala.

Scalability

To satisfy the requirement of scalability, ESPBench supports smaller as well as bigger systems with regard to scale-up and scale-out architectures. Adding main memory or CPUs to an SUT, i.e., scaling it up, might benefit the system's performance, while increasing the price for the tested architecture. The same holds true for scaling out, i.e., for distributing the load on more server nodes inside the SUT. Both scenarios are supported by ESPBench's architecture and its tools. ESPBench leverages a message broker as the interface to the SUT and the result calculator. Query implementations simply subscribe to topics to retrieve data and publish query results to a topic. This design decision creates single endpoints. So neither a change of message broker nodes nor of SUT nodes would require code adaptions, which means ESPBench ensures effortless scalability in this context. In other words, ESPBench can handle any degree of parallelism an SUT incorporates without the need for changes in the toolkit. Additional servers result in an increased price for the benchmark architecture, similar to upgraded hardware in scale-up scenarios.

The benchmark tools further allow to scale with regard to the size of business data, the duration of a benchmark run, and the data input rate of the streams, i.e., how many records are sent on the stream per time unit.

Simplicity

Simplicity is one of the most important requirements in our opinion. One action to keep ESPBench as simple as adequate is the provision of a comprehensive and benchmark toolkit. The toolkit comprises applications for data ingestion, query result validation, and latency calculation. Moreover, ESPBench comes with scripts for automating the entire benchmark process. These scripts include an optional component for capturing system variables, such as used main memory or CPU utilization. This functionality helps to get full insights into a system's behavior, without the need for benchmark users to develop their own monitoring framework.

As a second pillar next to the toolkit, ESPBench provides an example implementation of the benchmark queries. This implementation is used for evaluating ESPBench by benchmarking selected DSPSs. Publishing an example implementation of the queries not only allows for getting an extensive query description,

but also represents an artifact that can already be used for benchmarking different architectures with only minor adoptions required. Multiple DSPSs can be benchmarked with this implementation as we used the abstraction layer Apache Beam for development. It allows executing applications on any of the supported DSPSs.

Consequently, ESPBench is usable out-of-the-box, i.e., there is no need to develop a data sender, result validator, latency calculator, or even to implement the benchmark queries. Furthermore, the entire benchmark process is fully automated. These design decisions largely contribute to ESPBench's simplicity.

4

EXPERIMENTAL EVALUATION

This chapter presents the experimental evaluations of this dissertation. It is divided into three main sections. The first section of this chapter shows the experimental performance evaluation of selected data stream processing systems (DSPSs) using ESPBench and the provided example query implementations. The employed example implementations are developed using the Apache Beam software development kit (SDK). The system under test comprises PostgreSQL as the database management system (DBMS). Moreover, three state-of-the-art DSPSs, namely Apache Flink, Apache Spark Streaming, and Hazelcast Jet, are benchmarked and the benchmark results are discussed. The objective of these experiments is the validation of ESPBench. To be more concrete, this section showcases the functioning of the overall benchmark concept and its toolset to gain credibility.

The second section analyzes the performance impact of Apache Beam, the abstraction layer used in academia and industry, which we also applied for the provided example implementation of the ESPBench queries. We study the performance differences of Apache Beam applications compared to implementations using native DSPS SDKs.

The third section evaluates the performance capabilities of Apache Kafka, the message broker employed in ESPBench's architecture. We study Apache Kafka's capabilities with respect to sustaining data ingestion rates. Knowing about the performance limits of Apache Kafka is crucial for a correct benchmark execution. If Apache Kafka was a bottleneck within ESPBench, Apache Kafka itself would be benchmarked instead of the actual system under test (SUT).

Furthermore, we analyze the impact of batching on the timestamps taken by Apache Kafka. As these timestamps are used for latency calculations within ESPBench, it is crucial to understand if the batching mechanisms employed by Apache Kafka can distort benchmark results. If so, that has to be taken into account when calculating or analyzing latencies computed by ESPBench.

A section discussing the threats to validity with regard to the proposed performance benchmark concludes this chapter.

4.1 Validation of ESPBench

This section presents the results of the experimental evaluation of ESPBench. After giving an overview of the benchmark setup, we present the benchmark results in detail as well as the lessons learned.

4.1.1 Benchmark Setup

The overall setup contains eight server nodes. We initiate the benchmark start from one dedicated server. Particularly, we start the ansible script on this node, which automates the benchmark execution. The DBMS PostgreSQL is deployed on another dedicated machine. Three nodes build the Apache Kafka cluster and the remaining three nodes contain the DSPS.

Characteristic	Value
Operating System	Ubuntu 18.04 LTS
CPU	Intel(R) Xeon(R) CPU X7560 @ 2.27 GHz, 8 cores (2x), Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60 GHz, 8 cores (1x)
RAM	57 GB (2x), 32 GB (1x)
Network	1Gbit/s: - measured bandwidth between nodes: 117.5 MB/s - measured bandwidth intra-node transfer: 908 MB/s
Disk	13 Seagate ST320004CLAR2000 in RAID 6, access via Fibre Channel with 8 Gbit/s: measured write performance about 70 MB/s
Hypervisor	VMware ESXi 6.7.0
Apache Kafka Version	2.3.0
Scala Version	2.12.8
Java Version	OpenJDK 1.8.0_222

Table 4.1: System characteristics of the Apache Kafka brokers for the ESPBench validation

Property	Value
key-serializer-class	org.apache.kafka.common.serialization.StringSerializer
value-serializer-class	org.apache.kafka.common.serialization.StringSerializer
batch-size	16,384 bytes
buffer-memory-size	33,554,432 bytes
acks	1
linger-time	0 ms

Table 4.2: Apache Kafka producer properties used by the data sender for the ESPBench validation

The system characteristics of the Apache Kafka servers are listed in Table 4.1. Table 4.2 shows the Apache Kafka producer properties that are applied by the data sender. To ensure that Apache Kafka does not become a bottleneck, i.e., to make sure that the SUT is benchmarked as intended, we use selected data input rates. Particularly, we use data rates that can provenly be handled by Apache Kafka in a setup as used. The study presented in Section 4.3 indicates that Apache Kafka can easily handle 10,000 messages/second, which is a satisfying ingestion rate for many current real-world scenarios. Additionally, the study shows that input rates in such a range with the used input data characteristics clearly do not saturate the network capacities [HMU20].

The system characteristics of the SUT nodes are listed in Table 4.3. The evaluation benchmarks the query implementations developed using the Apache Beam SDK, which are executed on Apache Flink, Apache Spark Streaming, and Hazelcast Jet, each in combination with PostgreSQL. Specifically, we use Apache Beam 2.16.0. The level of effort put into Apache Beam support by the DSPSs is likely to be reflected in the results. Using Apache Beam for application development is a relevant approach, which is employed by companies, such as Lyft [KW19] and Spotify [Li17], in their data processing landscapes. However, it is important to keep in mind that implementations using system SDKs are likely to show a different performance [HMG+19]. The main objective of the experiments is to validate the functioning of ESPBench with its queries and tools.

For evaluating ESPBench, we use Hazelcast Jet's client-server deployment as this deployment mode provides the greatest flexibility. Due to the separation of applications from the Hazelcast Jet cluster, one can, e.g., scale independently regarding the application and so easily adapt to changing environments. As scalability and flexibility are common requirements for enterprise applications, we view the client-server deployment as the more suitable deployment option for ESPBench and choose it over the embedded mode [Joh15]. Apache Spark

Characteristic	Value
Operating System	Ubuntu 18.04 LTS
CPU	Intel(R) Xeon(R) CPU E5450 @ 3.00 GHz, 8 cores
RAM	57 GB
Network	1Gbit/s: - measured bandwidth between nodes: 117.5 MB/s - measured bandwidth intra-node transfer: 908 MB/s
Disk	13 Seagate ST320004CLAR2000 in RAID 6, access via Fibre Channel with 8 Gbit/s: measured write performance about 70 MB/s
Hypervisor	VMware ESXi 6.7.0
Apache Flink Version	1.8.2
Hazelcast Jet Version	3.0
Apache Spark Version	2.4.4
PostgreSQL Version	9.6.12
Scala Version	2.12.8
Java Version	OpenJDK 1.8.0_222

Table 4.3: System characteristics of the SUT nodes for the ESPBench validation

Streaming and Apache Flink, which, contrary to Hazelcast Jet, both follow a master-worker pattern as explained before, use two nodes as workers and one node as the master. The Hazelcast Jet installation is also distributed across three nodes, i.e., all DSPS deployments use the same server resources. The detailed system configurations are part of the ESPBench software repository.

4.1.2 Benchmark Results

This section presents the results of the experimental evaluation performed using ESPBench. All implementation details can be obtained in the published benchmark artifacts, e.g., the Apache Beam query implementations or the result validator implementation. Next to latencies, we analyze the observed memory usage and system loads of the involved servers in order to get an understanding of how each DSPS utilized the available resources.

Result Overview

Table 4.4 shows the overall results, i.e., the latencies for the different ESPBench queries, which were introduced in Table 3.4, and benchmark settings. Each query was executed three times on every system and with each data input rate, which is sufficient due to the low variance of latency results. Table 4.4 presents the averages of all three runs. We benchmarked data input rates of 1,000 and 10,000 messages/second, both of which are proven to be manageable by Apache

Query	Input Rate	System	90%tile in s	Min in s	Max in s	Mean in s
1 - Sensor Status	1,000 msg/s	Apache Flink	10.659	0.049	18.591	4.269
		Hazelcast Jet	0.024	0.009	0.691	0.020
		Apache Spark	n/a	n/a	n/a	n/a
	10,000 msg/s	Apache Flink	16.492	0.048	33.423	5.767
		Hazelcast Jet	0.036	0.012	1.030	0.029
		Apache Spark	n/a	n/a	n/a	n/a
2 - Outliers	1,000 msg/s	Apache Flink	615.078	9.352	676.535	358.076
		Hazelcast Jet	533.177	5.353	590.170	304.689
		Apache Spark	n/a	n/a	n/a	n/a
	10,000 msg/s	Apache Flink	8,175.784	40.446	9,147.738	4,599.666
		Hazelcast Jet	7,425.443	24.564	8,282.022	4,140.149
		Apache Spark	n/a	n/a	n/a	n/a
3 - Errors	1,000 msg/s	Apache Flink	0.011	0.001	0.045	0.005
		Hazelcast Jet	0.021	0.004	0.158	0.017
		Apache Spark	0.534	0.121	1.248	0.387
	10,000 msg/s	Apache Flink	14.979	0.002	19.058	4.581
		Hazelcast Jet	0.016	0.005	0.795	0.014
		Apache Spark	1.557	0.137	5.380	0.780
4 - Machine Power	1,000 msg/s	Apache Flink	0.717	0.003	2.792	0.251
		Hazelcast Jet	0.371	0.006	4.082	0.195
		Apache Spark	1.008	0.141	1.966	0.644
	10,000 msg/s	Apache Flink	470.689	1.936	517.291	275.096
		Hazelcast Jet	87.299	6.008	94.599	56.236
		Apache Spark	303.432	4.255	325.951	188.158
5 - Processing Times	1,000 msg/s	Apache Flink	106.892	0.506	114.750	65.261
		Hazelcast Jet	88.006	1.823	96.316	51.278
		Apache Spark	102.736	0.803	112.815	61.820
	10,000 msg/s	Apache Flink	2,028.137	2.274	2,211.899	1,136.910
		Hazelcast Jet	2,129.287	6.202	2,345.790	1,170.944
		Apache Spark	1,863.259	1.941	2,061.002	1,041.930

Table 4.4: Latency overview of the experimental analysis using ESPBench and query implementations based on Apache Beam

Kafka in the employed settings [HMU20]. Thus, we ensure that the SUT is benchmarked and not Apache Kafka. Every benchmark run lasted exactly five minutes.

Overall, the latency results are diverse, with Hazelcast Jet often performing best with respect to the 90^{th} percentile of mean latencies. In the following, we elaborate on the results of the single queries.

Query 1 - Check Sensor Status

The overall results shown in Table 4.4 reveal that Hazelcast Jet performed significantly better than Apache Flink. Although Apache Flink's minimum latency is very low with a value of 49 ms, we also experienced remarkably higher response times. The maximum response time of the Apache Flink runs is greater than 18 s. The 90^{th} percentile exceeds 10 s. In contrast, Hazelcast Jet's worst response time is about 700 ms.

Apache Spark's response times are not taken into consideration, because the query results were wrong, i.e., the query outputs differed from the expected results. This is the case although we used the same Apache Beam application

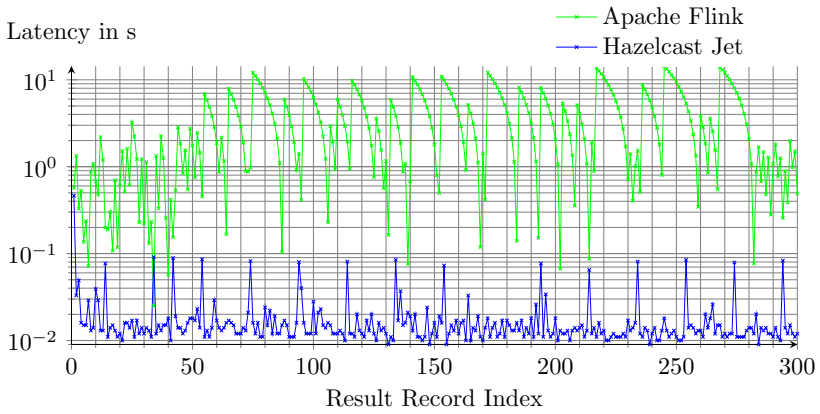


Figure 4.1: Latencies for *query 1 - check sensor status* on Apache Flink and Hazelcast Jet for a data input rate of 1,000 messages/second - Apache Spark results not shown due to wrong query results

for all systems. It is an important finding, which highlights the necessity of having a query result validation for performance benchmarks as published with ESPBench.

An explanatory hypothesis for the false results can be found in Apache Spark's data architecture, i.e., the use of micro-batches. This concept distinguishes Apache Spark Streaming from the other DSPSs. Through batching mechanisms, windows might be different. If a micro-batch represents the finest granularity and cannot be split, it could be left out of a window even though most of the contained records semantically belong to the window, depending on the window semantics of the DSPS. The work by Botan et al. [BDD+10] studies the heterogeneity in window semantics that exists among DSPSs.

Figure 4.1 visualizes the result latencies of two benchmark runs with an input rate of 1,000 messages/second, one executed on Apache Flink and one on Hazelcast Jet. It becomes visible that there are upward outliers for both systems, while the overall latencies on Apache Flink are significantly higher as mentioned before. After the Apache Flink latency reaches a new local maximum, the latencies slowly decline step-by-step. This behavior is different from the latency development that we can observe for Hazelcast Jet runs. For these runs, the latency immediately jumps back to normal, i.e., the value range to which most latencies belong, after facing an upward outlier. Figure 4.2 shows the results for an input rate of 10,000 messages/second. It shows a very similar result. One difference is the comparatively steady and high latencies at the beginning of the benchmark run with Apache Flink.

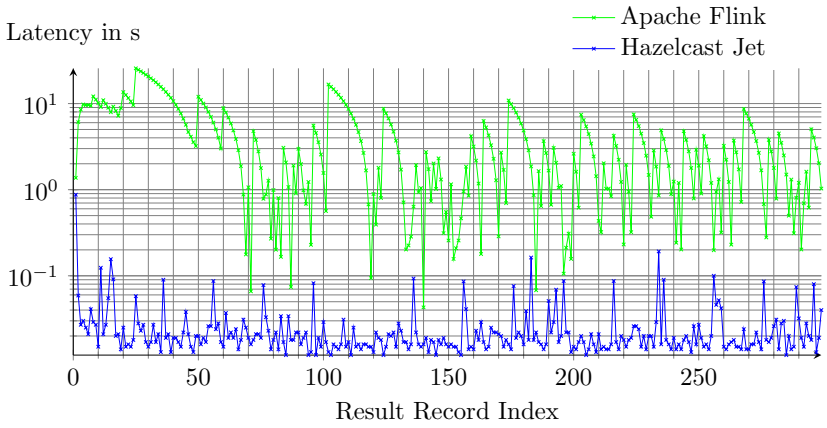


Figure 4.2: Latencies for *query 1 - check sensor status* on Apache Flink and Hazelcast Jet for a data input rate of 10,000 messages/second - Apache Spark results not shown due to wrong query results

Query 2 - Determine Outliers

The overall latencies of Table 4.4 show significantly higher values for the second query than for query one. Hazelcast Jet again outperforms Apache Flink. However, there is only a relatively small difference between both systems. Executing the query implementation for Apache Spark revealed a valuable finding. The Apache Spark system throws an exception when submitting the Apache Beam query implementation to it: *java.lang.IllegalStateException: No TransformEvaluator registered for UNBOUNDED transform View.CreatePCollectionView.* This exception is related to the missing support for side inputs for Apache Spark[1]. The observation overall indicates that the DSPS exchangeability of Apache Beam applications is limited.

Figure 4.3 shows the latencies for the two 1,000 messages/second runs on Apache Flink and Hazelcast Jet. Due to the high number of almost 21,000 result records, only every 500^{th} latency is plotted. This figure illustrates another interesting finding. The latencies of both systems show the same pattern, a steadily growing latency, which is an indicator for records queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This fact does not become visible when looking at the overall latency numbers shown in Table 4.4, which indicates the importance of the ESPBench validator feature to output single record latencies.

[1] `https://issues.apache.org/jira/browse/BEAM-1564`, accessed: 2021-08-28

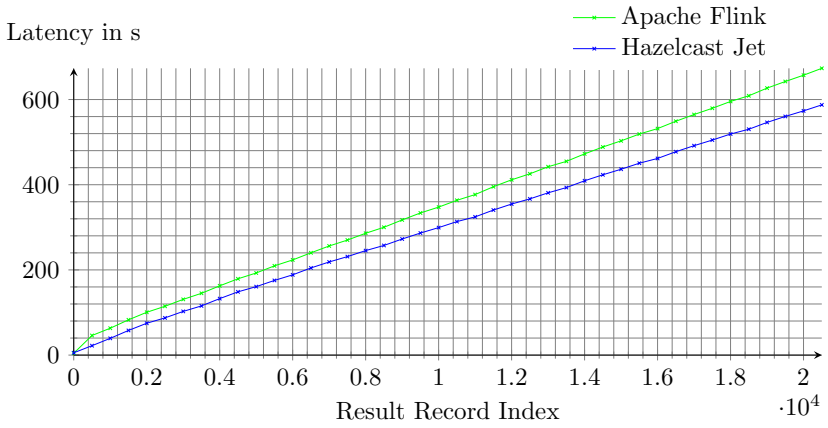


Figure 4.3: Latencies for *query 2 - determine outliers* on Apache Flink and Hazelcast Jet for a data input rate of 1,000 messages/second - Apache Spark results not shown due runtime exception

Query 3 - Identify Errors

The conducted benchmark runs for query three output about 1,250 errors and 12,250 errors for the input rates of 1,000 messages/second and 10,000 messages/second, respectively. Figure 4.4 visualizes the latencies for the smaller input rate, particularly every tenth latency for readability reasons. The results of Figure 4.4 show that there are ups and downs, while all systems stay in a certain range. Apache Spark Streaming even shows a pattern-like trend, which might be caused by its use of micro-batches.

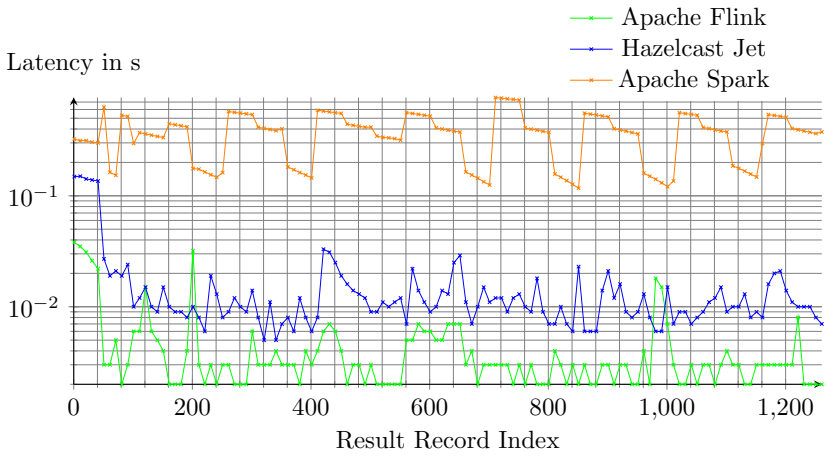


Figure 4.4: Latencies for *query 3 - identify errors* for a data input rate of 1,000 messages/second

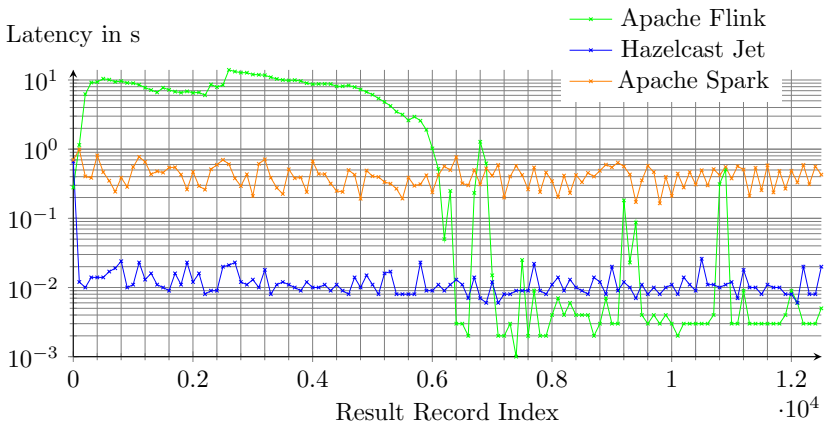


Figure 4.5: Latencies for *query 3 - identify errors* for a data input rate of 10,000 messages/second

Figure 4.5 shows the latencies for an input rate of 10,000 messages/second on a logarithmic scale, this time every 100^{th} latency. Hazelcast Jet and Apache Spark Streaming again show ups and downs within a comparatively small range. The previously observable stepwise pattern for Apache Spark Streaming is not present anymore.

Apache Flink shows different behavior. For about half of the five-minute benchmark run, the latencies are on a relatively high level. After this period, the latencies drop and stay on the new level, although with relatively high swings. This unique progress was observed in all Apache Flink runs for this data input rate and query. It again indicates the importance of capturing and investigating single latencies to get full insights.

Query 4 - Check Machine Power

Figure 4.6 visualizes the latencies for a run with an input rate of 1,000 messages/second. Relatively small peaks can be identified for all systems, which may be caused by garbage collection runs. Overall, there is not a surprising insight. While peaks for Apache Spark are comparatively high and constant throughout the five minute runtime, the peaks for Apache Flink and Hazelcast Jet are bigger at the beginning with a decreasing trend. For 10,000 messages/second, steadily increasing latencies can be observed, similar to the latency trend for query two visualized in Figure 4.3. It again indicates that records were queuing up, i.e., the SUT could not keep up with the data input rate in the benchmarked setting.

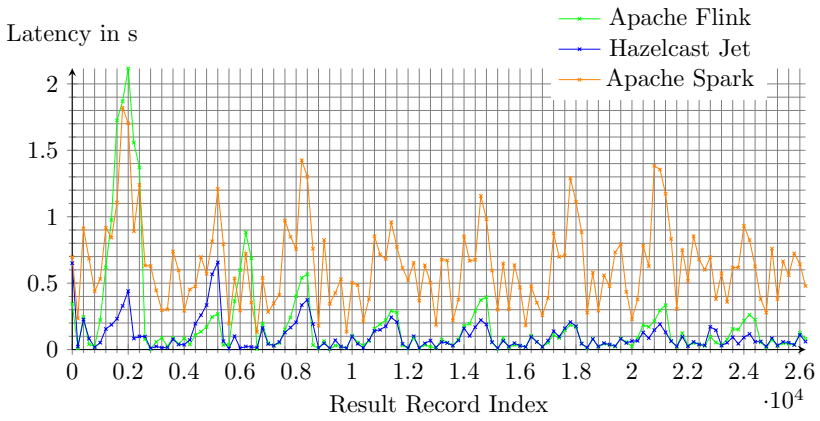


Figure 4.6: Latencies for *query 4 - check machine power* for a data input rate of 1,000 messages/second

Query 5 - Persist Processing Times

The latencies for query five are visualized in Figure 4.7. The experimental evaluation shows steadily increasing latencies as for query two, whose latencies are visualized in Figure 4.3. This result again shows that the SUT cannot handle the load properly. Furthermore, the gathered results reveal that the DBMS is the bottleneck for this write-heavy query with about 300,000 required updates for the five-minute runs with 1,000 messages/second as data input rate.

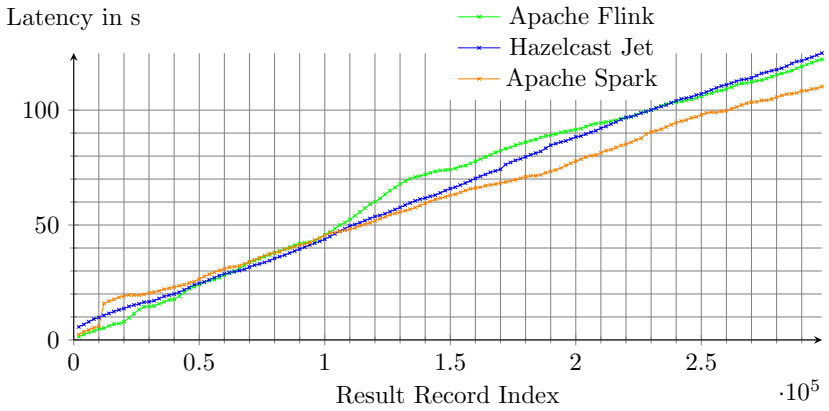


Figure 4.7: Latencies for *query 5 - persist processing times* for a data input rate of 1,000 messages/second

System Utilizations

This section presents the measured system utilizations for the benchmark runs highlighted before. Particularly, we analyze the *system load* and the *used memory* for query runs with an input rate of 10,000 messages/second. The measurements were captured using *collectd*, which collected corresponding values every ten seconds in the used settings. All figures in this subsection show 36 values, i.e., represent a period of 360 seconds or six minutes. This allows to investigate the systems' behavior for the conducted five-minute benchmark runs. Values for Apache Spark Streaming runs are not presented for the first two queries, because these settings failed to produce correct results as explained previously in Section 4.1.2 and Section 4.1.2. Out of the multiple benchmark runs, an exemplary run is plotted in the following. As we did not discover noticeable outliers or differences between benchmark runs, the plots can be considered representative for the measured setting.

System Load

The system load gives an overview over the CPU and I/O utilization of a server, i.e., also reflecting performance limits regarding disk writes. It is defined as the number of processes demanding CPU time, specifically processes that are ready to run or waiting for disk I/O. The following figures show one-minute-averages of the system loads. This key performance indicator (KPI) is also known as *short-term system load*. As we are using servers with an eight-core CPU, each as described in Table 4.3, it is desirable that no node exceeds a system load of eight to not over-utilize a machine [Gre17].

Overall, there is no over-utilization recognizable in the analyzed scenarios. The highest system load is generated by Hazelcast Jet. Among the queries, the greatest load of about five was observed for the first query.

Figure 4.8 visualizes the short-term system loads for the SUT server nodes while executing the first query of ESPBench. Two major differences between Apache Flink and Hazelcast Jet become visible. Firstly, all Hazelcast Jet nodes show a higher system load than the Apache Flink node with the highest utilization. Secondly, while Hazelcast Jet utilizes all three nodes, which results in a load between approximately three and five, there is only one Apache Flink node that shows a system load considerably greater than zero. Specifically, the Apache Flink node with the highest utilization shows system loads between one and two most of the time. The better system utilization of Hazelcast Jet is likely to be a reason for the lower latencies that are associated with it for query one compared to Apache Flink, cf. Table 4.4.

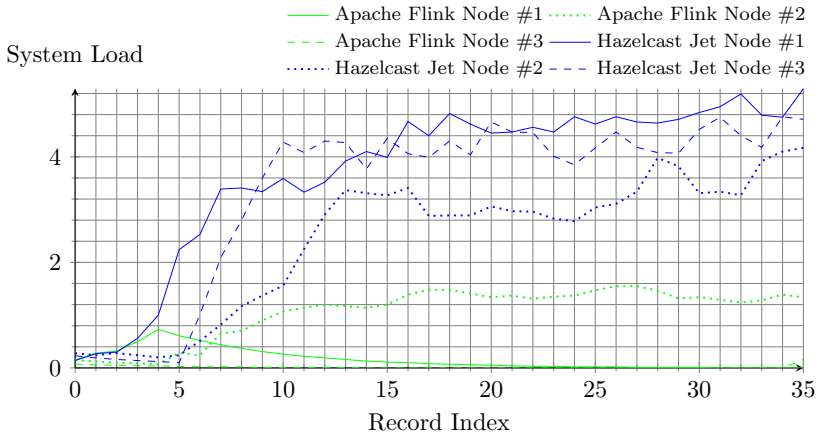


Figure 4.8: Short-term system load for query 1

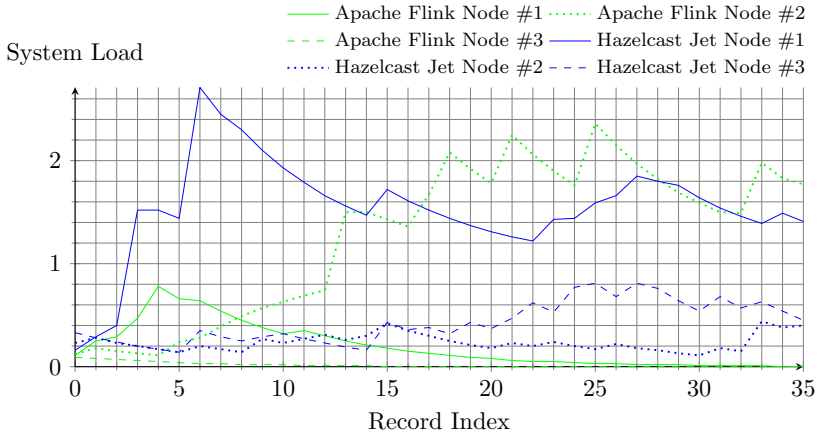


Figure 4.9: Short-term system load for query 2

Figure 4.9 shows a slightly different picture. The highest values for the short-term system loads are overall smaller compared to the observations related to the first query, with the maximum system load being below three. The server nodes maximally utilized by Apache Flink and Hazelcast Jet show about the same system load level on average. While the third node is almost not utilized by Apache Flink, the first node shows a peak at the beginning. Particularly, this peak amounts to a system load of about 0.8. From there, it drops to utilization of about zero. Hazelcast Jet utilizes the further two nodes slightly more, with values fluctuating between 0.1 and 0.8. The overall schema is similar among the systems, i.e., both DSPSs have a higher utilized node and two nodes with lower system loads.

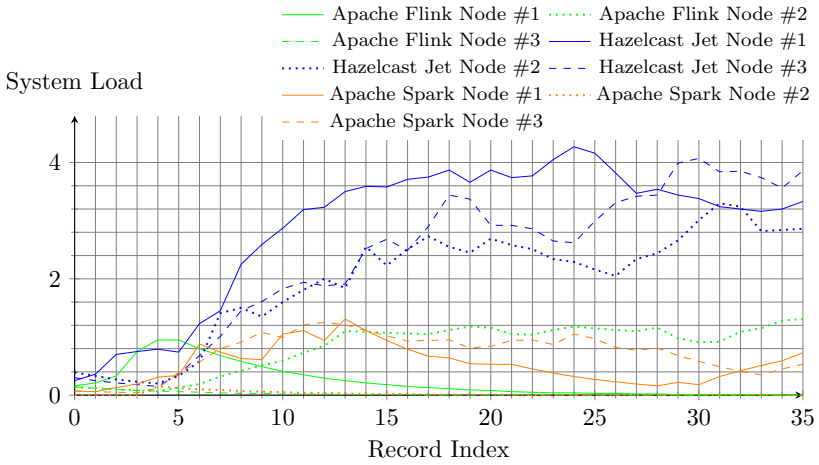


Figure 4.10: Short-term system load for query 3

Figure 4.10 visualizes the system loads for query three. The utilization of the Hazelcast runs is similar to the one shown in Figure 4.8, i.e., higher utilization of all three nodes. Moreover, the utilization of each Hazelcast node is greater than the highest node utilization of all other runs. The system loads for the Hazelcast nodes fluctuate between 2 and 4.4 most of the time. Apache Flink again only causes a system load noticeably higher than zero at one node. Apache Spark utilizes two nodes on about the same level as the Apache Flink node with the highest system load. The third Apache Spark node shows a load of approximately zero.

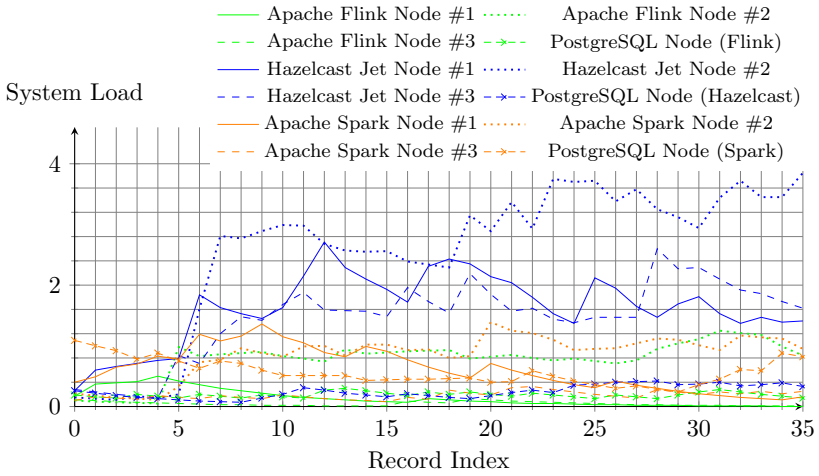


Figure 4.11: Short-term system load for query 4

Figure 4.11 and Figure 4.12 visualize the short-term system loads for query four and five, respectively. Additionally to the content of the figures before, these illustrations also show the utilizations of the database node, which is incorporated for these two queries.

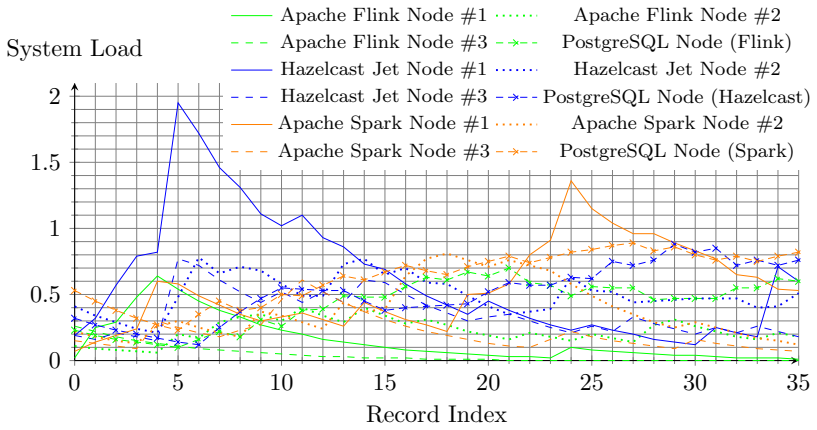


Figure 4.12: Short-term system load for query 5

Figure 4.11 draws a similar picture as Figure 4.10. One difference is the utilization of the Hazelcast nodes, which is slightly lower than before. The PostgreSQL node has very low utilization of less than one for all settings. Figure 4.12 shows overall lower loads with a maximum below two. Both, a Hazelcast node and an Apache Spark node show a small peak during the run. The PostgreSQL node load never surpasses a value of one for query five as well. So for none of these two workloads, PostgreSQL caused a significant increase of the system load on the server it was running on.

Used Memory

The following figures show the amount of main memory used by the SUT nodes during the query execution. As shown in Table 4.3, each SUT node is equipped with 57 GB of main memory. Overall, none of the nodes has used more than 18 GB RAM for any benchmark run. For every query, the overall highest memory usage can be observed for a Hazelcast Jet run. To be more concrete, query two executed on Hazelcast Jet resulted in the greatest usage of roughly 18 GB.

Figure 4.13 shows the used main memory for the execution of query one. The picture is similar to the corresponding system load charts, meaning Hazelcast Jet utilizes the nodes most. While the used memory stays approximately constant for two out of three nodes for both, Apache Flink and Hazelcast Jet, a slight increase can be observed for one node in both settings. The used memory of

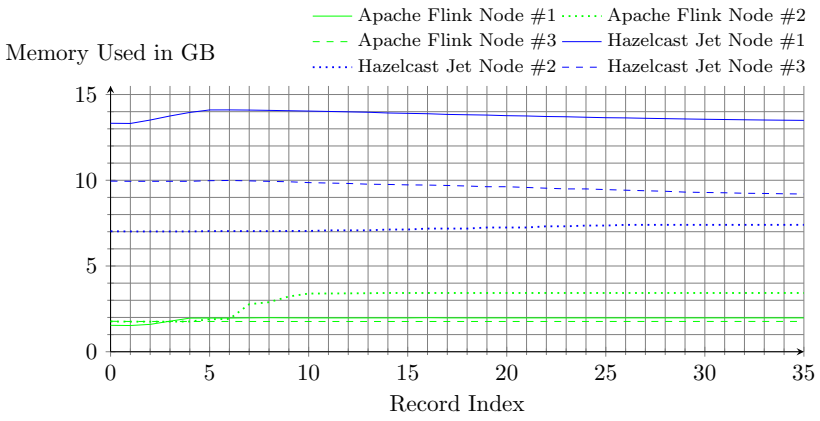


Figure 4.13: Used main memory - query 1

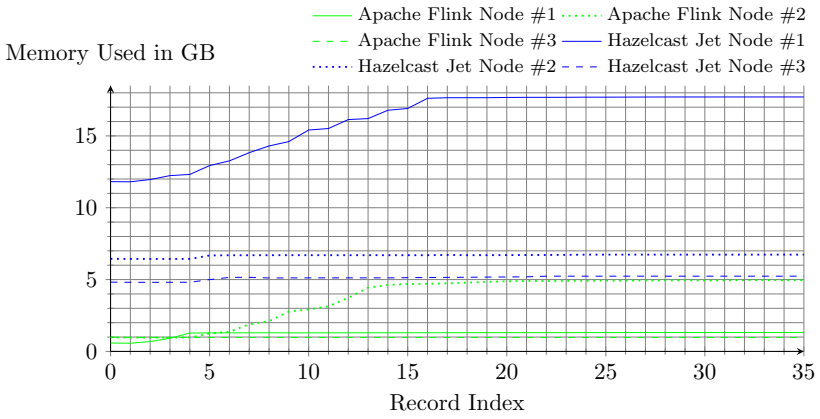


Figure 4.14: Used main memory - query 2

Hazelcast Jet nodes ranges from 7 to 16 GB. The Apache Flink cluster uses between 1.5 to 3.5 GB of main memory on each node.

Figure 4.14 visualizes the memory usages for query two. We see a similar pattern as before with two nodes having a constant memory usage and one node's usage increasing. However, the increase is slightly bigger than in Figure 4.13, which results in a maximum memory usage of about 18 GB for Hazelcast Jet and 5 GB for Apache Flink.

Figure 4.15 also looks very similar to Figure 4.13 with respect to Apache Flink and Hazelcast Jet. The Apache Spark nodes show different behavior the other two DSPSs. The memory consumption of node two stays almost constant at close to 5 GB. Node one shows a slight increase at the beginning, which can be observed for a Hazelcast Jet and Apache Flink node as well. Node three shows

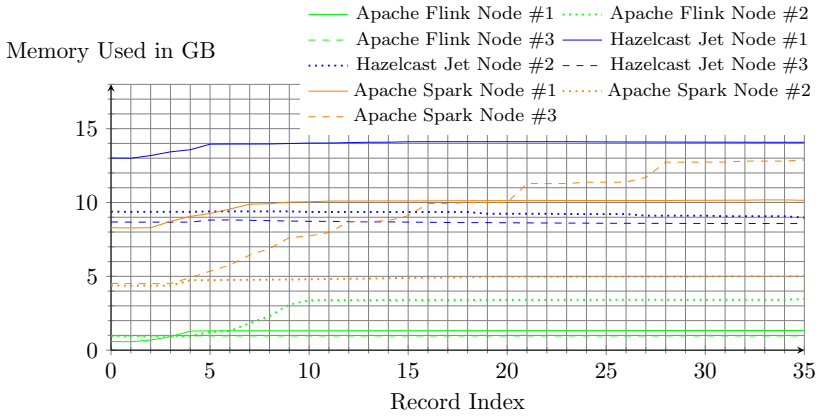


Figure 4.15: Used main memory - query 3

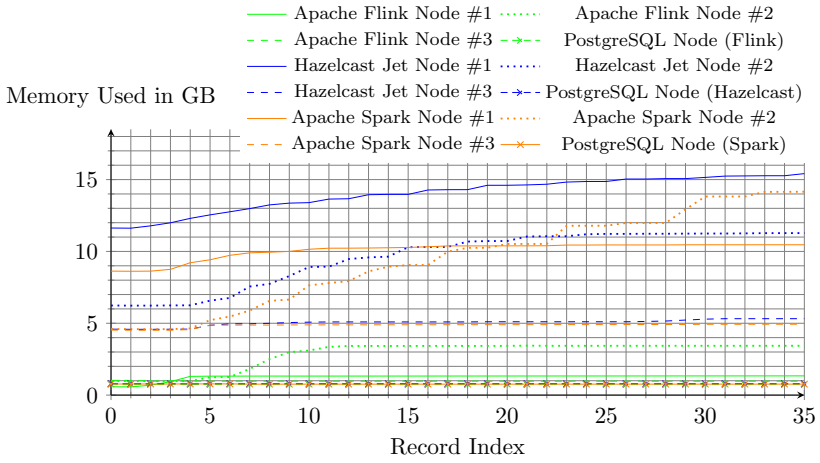


Figure 4.16: Used main memory - query 4

a new behavior, a step-wise increase of used memory over time, which results in a relatively high overall increase. Specifically, the memory consumption grows from about 4.5 GB to about 13 GB.

The memory consumptions for query four are shown in Figure 4.16. While the pattern for Apache Spark nodes looks similar to Figure 4.15, Hazelcast Jet and Apache Flink show new patterns. Particularly, these two systems also caused a step-wise increase in memory usage for query four. Overall, Apache Flink nodes again use the least memory. Hazelcast Jet nodes occupy slightly more memory than its Apache Spark counterparts. The memory usage of PostgreSQL is identical for all setups, i.e., constant on a low level of about 1 GB.

The results for query five displayed in Figure 4.17 reveal a memory consumption for the PostgreSQL nodes that is identical to query four. Thus,

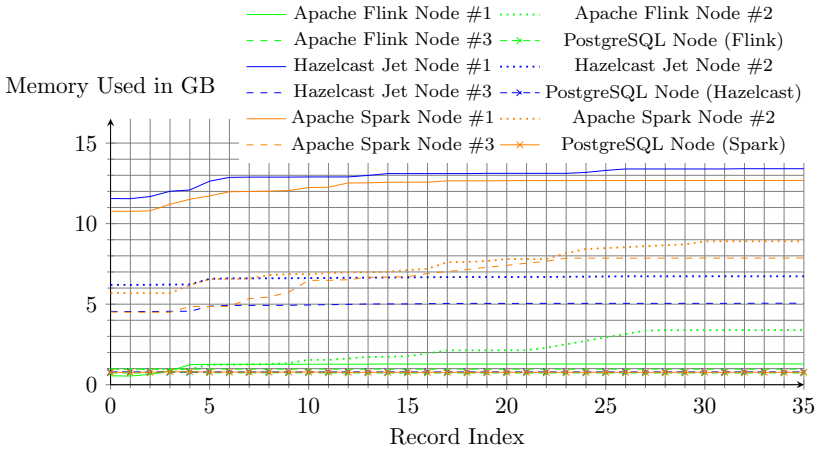


Figure 4.17: Used main memory - query 5

these two queries did not noticeably impact the memory consumption of the database node. The remaining memory consumption trends are also similar to Figure 4.16, with a lower overall increase.

4.1.3 Lessons Learned

There are three main lessons we learned from evaluating ESPBench. The first is the importance of having a result validation. Unfortunately, most of the existing benchmarks lack concepts or proper tool support for this aspect. We highlight the importance of result validation, e.g., through pointing to differing results for the same application executed on different systems. Depending on the use case, this might or might not be acceptable. However, it is crucial to be aware of such a system's behavior.

A second lesson we learned is that the portability of Apache Beam applications is not always given, i.e., it is not guaranteed that the paradigm 'write once, execute anywhere' holds true. In particular, we logged an exception when running the application for query two with Apache Spark, despite successfully executing it on Apache Flink and Hazelcast Jet.

The third main learning is that it is crucial to look at single response times. Aggregated KPIs, such as a mean latency, often do not allow us to understand the system's behavior and thus, can lead to misleading conclusions. That is the case, e.g., if latencies are steadily growing, i.e., a system cannot handle the load and queues incoming records. It can be falsely assumed that the mean latency determined after a limited benchmarking period is the one that can be expected in a productive deployment, i.e., when the application is running permanently.

4.2 Performance Impact of Apache Beam

This section describes the conducted performance analysis of Apache Beam, which compares stream processing applications developed using the Apache Beam SDK with applications developed using a DSPS SDK. After presenting the benchmark setup, the performance results are discussed. The section ends with the lessons learned. Query implementations, as well as further applications and scripts used for the study, can be obtained online[2].

4.2.1 Benchmark Setup

This section presents the general benchmark architecture and process for analyzing the performance impact of Apache Beam. That includes details on the data ingestion concept, the query execution, and the result calculation. Moreover, the employed benchmark queries are described.

Benchmark Architecture and Process

The overall benchmark architecture is depicted in Figure 4.18. The benchmark process is divided into three separate and consecutive phases. The architectural components, which are involved in the corresponding part of the process, are marked in the figure by dashed curly brackets.

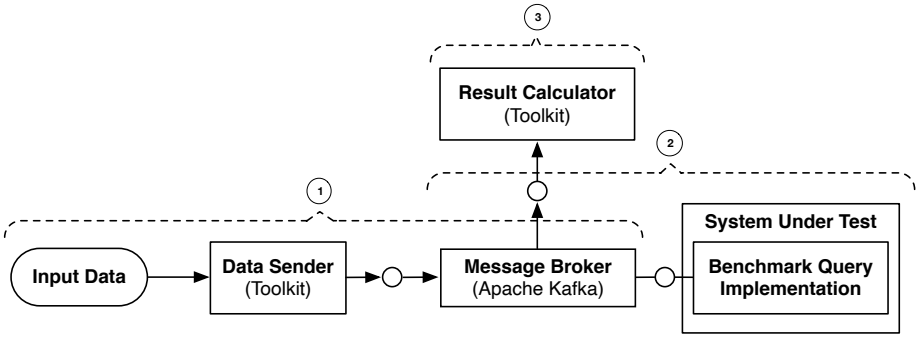


Figure 4.18: Overview about the benchmark architecture and process for analyzing the performance impact of Apache Beam visualized using Fundamental Modeling Concepts (FMC) (based on [HMG+19])

On the left-hand side of Figure 4.18, there is the *input data*. This data is present as a text file. The *data sender* reads this data and forwards it to the *message broker*, which in particular is Apache Kafka. The data sender tool is taken from ESPBench. So it allows to define certain configuration parameters,

[2] `https://github.com/guenter-hesse/ApacheBeamImpactEvaluation`

such as the data ingestion rate or the level of applied Apache Kafka Producer acknowledgments. The *system under test* on the right-hand side, i.e., the DSPS to be benchmarked, executes the implemented queries. Thereby, it reads the input data from and writes query results to the message broker. Moreover, there is a *result calculator* tool developed in Scala. It reads the query output from the message broker and, identically to ESPBench, leverages the timestamps captured by Apache Kafka for the calculation of execution times. The three different benchmark process steps marked in Figure 4.18 are described in the following:

1. Data Ingestion
Firstly, the data sender inserts the input data into an Apache Kafka topic. Particularly, the 1,000,001 records of the *AOL Search Query Log*[3] dataset are ingested. This dataset is also used in the *StreamBench* [LWXH14] benchmark. The input topic is created with a replication factor of one and one partition in order to ensure the correct order of messages as Apache Kafka only guarantees the correct order for entries within a single partition. Structurally, the data file consists of records with five tab-separated columns. These columns contain a user ID, the query issued by the user, the time at which the query was issued, the search result rank the user clicked on if applicable, and the search result Uniform Resource Locator (URL) the user clicked on if applicable.

2. Query Execution
During the execution phase, each query runs ten times for each execution setup. Meanwhile, there are no other programs executed on the system. Each DSPS is restarted at the beginning of this benchmarking step. The stream processing program computes the output and sends it to an Apache Kafka topic. This result topic is also created with a replication factor of one and one partition for the same reasons as mentioned previously. Each query is executed with a parallelism of one and two, ten times each. Moreover, every query is implemented using the APIs provided by the DSPS as well as using the Apache Beam SDK. So for each query, there are twelve different query execution setups: *three DSPSs* × *two parallelisms* × *two SDK options per system*.

The mentioned parallelism is set differently depending on the system and the used APIs. Regarding Apache Flink, it is configured using the command line option *-p* or *--parallelism*. This parameter can be passed when sub-

[3] `http://www.researchpipeline.com/mediawiki/index.php?title=AOL_Search_Query_Logs`, accessed: 2020-12-29

mitting an application for specifying the desired degree of parallelism[4]. For programs executed on Apache Spark Streaming, the configuration parameter *spark.default.parallelism*[5] is used. Apache Apex does not explicitly provide an option for configuring the parallelism. So instead, we set the number of *VCOREs* accordingly in the Apache Hadoop YARN configuration[6] as well as in the Apache Apex application as a directed acyclic graph (DAG) attribute[7]. The approach of using this configuration option is also applied for the programs running on Apache Apex, which are developed using the Apache Beam SDK. All details can be found in the published artifacts[2].

3. Result Calculation

Lastly, the result records are read from Apache Kafka for each query and the time difference between the firstly inserted and the lastly inserted record is computed. Apache Kafka is configured to use its *LogAppendTime* feature, i.e., the timestamp when a record is appended to the Apache Kafka log is stored together with the record itself. For the calculation of execution times, we use these timestamps, which allow keeping the measurements application- and system-independent. That is a crucial benefit with respect to result correctness, as definitions of performance criteria vary among systems. Consequently, one cannot rely on performance data provided by DSPSs [KRK+18]. The overhead between having the correct result computed within the SUT and having it appended to Apache Kafka log is identical for every system and hence, results are comparable.

With regard to the hard- and software setup, virtual machines are used for all nodes. The Apache Kafka version 2.11-0.10.1.0 is installed on a three-node cluster with 64 GB main memory and an Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60 GHz CPU with eight cores each. The DSPSs are installed on two server nodes, where both nodes act as worker nodes or the equivalent. These two nodes are identical to the Apache Kafka nodes with regard to both, main memory and CPU. Ubuntu 14.04 is installed as the operating system on all servers. Regarding system and framework versions, Apache Apex 3.7.0, Apache Hadoop 2.7.3, Apache Spark 2.3.0, Apache Flink 1.4.0, and Apache Beam 2.3.0 are used. The configuration files for the different systems can be found in the published repository[2].

[4] `https://ci.apache.org/projects/flink/flink-docs-release-1.6/ops/cli.html`, accessed: 2020-12-29

[5] `https://spark.apache.org/docs/latest/configuration.html`, accessed: 2020-12-29

[6] `http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-common/yarn-default.xml`, accessed: 2020-12-29

[7] `https://ci.apache.org/projects/apex-core/apex-core-javadoc-release-3.6/com/datatorrent/api/Context.OperatorContext.html`, accessed: 2020-12-29

Query	Description
Identity	Read the input and output it, without performing any data transformation. Can be seen as a baseline query with respect to computational complexity.
Sample	Read input and output only a certain percentage of data that is randomly chosen. The number of output tuples is as big as about 40% of the number of input tuples.
Projection	Read input and output only a certain column of the input record. In the presented measurements, the values of the first column are chosen for being included in the output.
Grep	Read input and output only records that match a certain regex. The search string used for the measurements is "test", which leads to an output of 3,003 records or about 0.3 % of the number of input records.

Table 4.5: Overview of the stateless StreamBench queries used for evaluating the Apache Beam performance impact (based on [LWXH14])

Benchmarked Queries

The executed queries are taken from the *StreamBench* [LWXH14] performance benchmark. StreamBench defines seven different queries. Four of these are stateless, i.e., it is not required for an application to keep a state for producing correct answers. The remaining three queries are stateful. The stateless queries used for the benchmark presented in this paper are listed in Table 4.5. Stateful queries are excluded as Apache Beam, at the time this study was conducted, did not support stateful processing when programs were executed on Apache Spark [HMG+19].

4.2.2 Performance Results

This section illustrates the performance results with regard to the measured execution times. Moreover, the standard deviations of these execution times and the performance impacts of using the Apache Beam SDK are presented in detail.

Execution Times

The following charts visualize the measured average execution times for the four benchmarked queries. On the y-axis, the combinations of system, parallelism, and kind of implementation, i.e., using Apache Beam or system APIs, are listed. The x-axis shows the times in seconds. The letter P denotes the degree of parallelism.

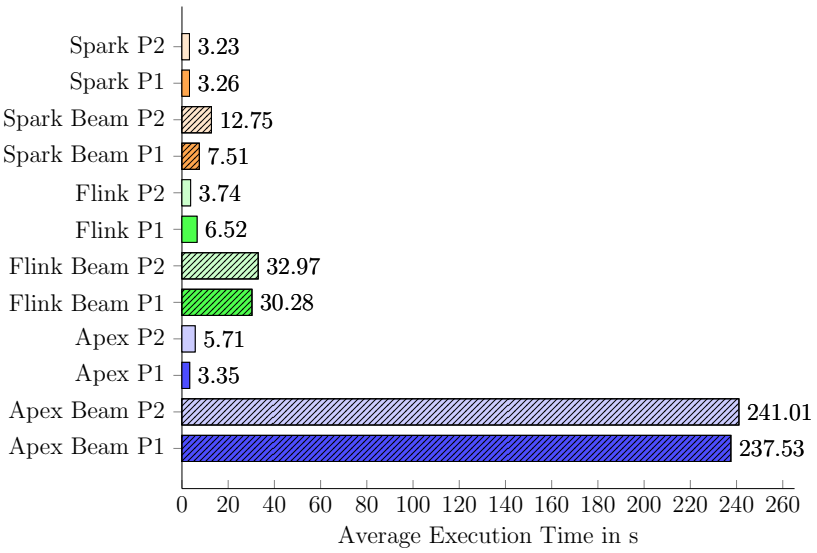


Figure 4.19: Average execution times - identity query

Figure 4.19 shows the results for the identity query. The average execution times range from 3.23 s for Apache Spark Streaming with a parallelism of two, to 241.01 s for the Apache Beam implementation running on Apache Apex with the same parallelism factor. It can be seen that the query implementations using Apache Beam are slower compared to the implementations using the APIs provided by the corresponding systems in all cases. That is true for almost all measurements presented in the following.

The overall shortest execution times belong to queries run on Apache Spark Streaming, closely followed by Apache Apex and Apache Flink, both of which have a noticeable slower average runtime for one kind of parallelism. That could be due to outliers in the series of runs. The absolute and relative standard deviations are shown later on in Figure 4.23 and Figure 4.24, respectively.

When looking at the runtimes of the queries implemented using the Apache Beam SDK, differences are significantly larger. While these queries are again fastest when running on Apache Spark, with times of 7.51 s and 12.75 s for parallelisms of one and two respectively, Apache Flink follows with times between 30 s and 33 s. Apache Beam queries running on Apache Apex have by far the highest execution times with around 240 s. So the differences between the execution times of the analyzed systems are significantly higher for the queries implemented using Apache Beam compared to those developed using native system APIs. In comparison to the size of these variances, distinctions between parallelism factors are very small. These observations lead to the conclusion

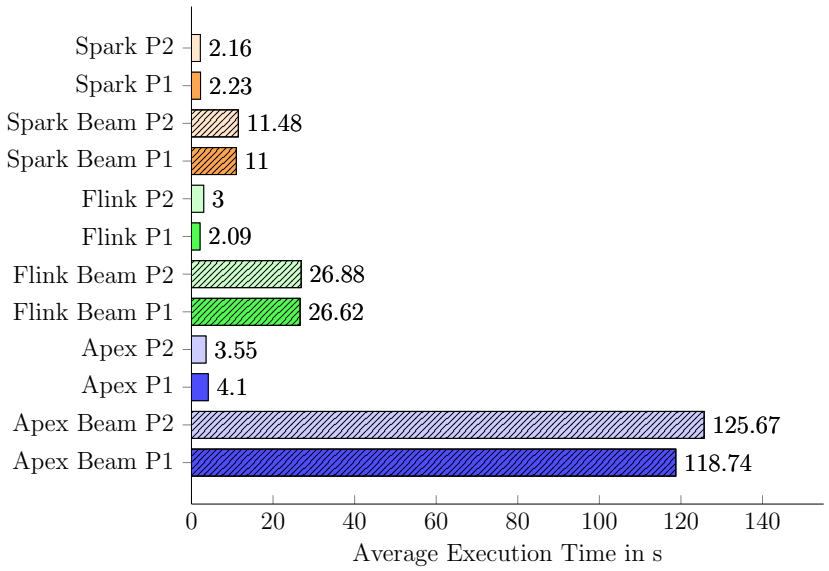


Figure 4.20: Average execution times - sample query

that the translation of the Apache Beam runner for Apache Apex created an execution plan that is far away from what is possible with the DSPS with respect to performance. While there is also a performance gap for the other systems, the one for Apache Apex is by far the largest one.

Besides, the response time difference between parallelism factors of the Apache Beam query running on Apache Spark Streaming is noticeable. The average execution time for the parallelism of two is close to 70 % higher compared to these for the parallelism of one. As the average relative standard deviation for these benchmark runs is low as illustrated later on, the high execution time is not caused by outliers. A reason for this observation could be the introduced overhead regarding, e.g., data transfer, that comes with the split-up of tasks due to parallelization, which may not pay off for simple queries like the identity query.

Figure 4.20 displays the results for the sample query. Again, it can be seen that the query implementations using native system APIs outperform these using the Apache Beam SDK. Moreover, the execution times of the programs that were developed using the system APIs do not differ significantly between the analyzed systems and parallelism factors. Compared to identity query results, times are slightly lower overall, which could be a result of the lower number of output records. So for, e.g., Apache Spark Streaming, the average execution times are about 1 s lower for the sample query compared to the results for the identity query. The Apache Beam query implementation executed on Apex is

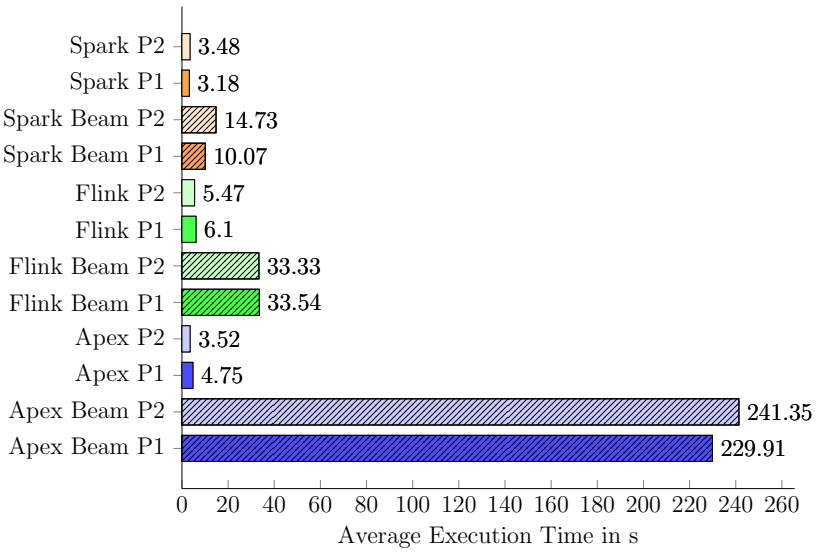


Figure 4.21: Average execution times - projection query

an exception as there is a major difference recognizable. Specifically, the average execution times for the sample query amount to only about 50 % of the identity query times.

With average execution times of about 2.09 s and 3 s for the sample query developed using Apache Flink APIs for parallelisms of one and two respectively, these numbers are below the corresponding times for Apache Apex. Thus, the performance ranking between systems for the sample query is identical to the ranking for the identity query. So concretely, the times for Apache Spark are lowest, followed by these of Apache Flink and Apache Apex for both kinds of implementation.

The results of the projection query are visualized in Figure 4.21. They are similar to the numbers for the identity query in all aspects. This closeness leads to the conclusion that splitting a string and accessing one column of the resulting list does not introduce a noticeable overhead. Regarding the number of output tuples, both queries are identical. However, the size of the result records differs. The tuple size for the output of the projection query is smaller, as only a subset of the columns is sent to the output topic. This reduction in output size does not have a noticeable impact on the query performances as the results reveal.

Figure 4.22 visualizes the measurements for the grep query. These execution times are overall the lowest ones. There are again differences between systems and used APIs. Especially the implementations using the native APIs offered by Apache Spark Streaming and Apache Flink have relatively low execution times

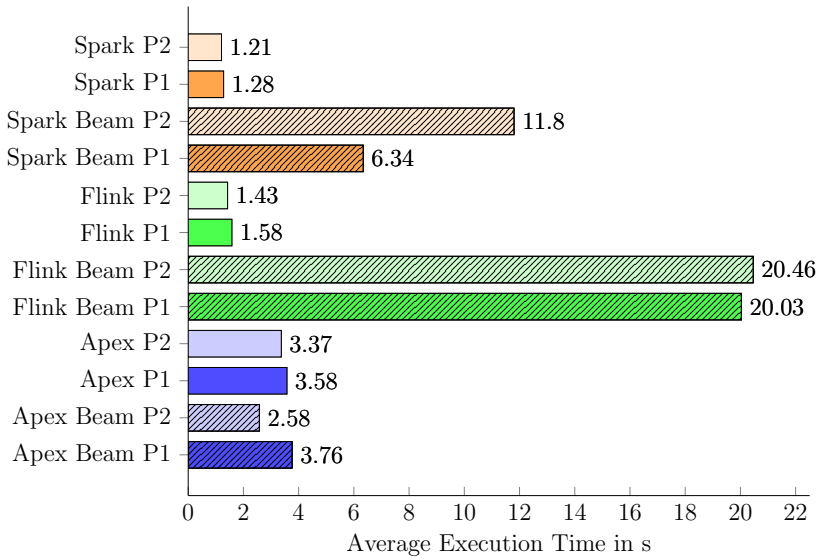


Figure 4.22: Average execution times - grep query

in comparison to the corresponding numbers for the other three queries. With about 20 s, the Apache Beam version for Apache Flink is close to 7 s faster than the corresponding sample query result with about 27 s. There is no noticeable difference between parallelism factors.

Contrary to Apache Flink, the average execution times for queries developed using Apache Beam and running on Apache Spark differ amongst parallelism factors. Similar to the corresponding times for the identity query depicted in Figure 4.19, the average execution time for a parallelism factor of two is noticeably higher. In particular, with absolute times of about 11.8 s and 6.34 s, a parallelism factor of two slows down the average execution time by more than 85 % in comparison to the time measured for a parallelism of one. Reasons for that are likely the same as described for the identity query results.

A surprising result is the Apex Beam performance for Apache Apex. While the times for the native Apache Apex implementation are about on the same level as the corresponding results for all other queries, the ones for the query developed using Apache Beam are remarkably lower. For the projection and the identity query, Apex Beam results are approximately between 230 s and 240 s. With about 120 s, the sample query performance is already significantly better. However, with 2.58 s and 3.76 s, the execution times for the Apache Beam version of the grep query executed on Apex are orders of magnitude lower.

A reason for the relatively low execution times could lie in the number of output records and the resulting smaller effort that is needed for emitting query

outcomes. To be more concrete, the output for the grep query is significantly lower than for the other three queries, though, the sample query already outputs fewer tuples than the projection and the identity query.

Standard Deviations in Measured Execution Times

Figure 4.23 visualizes the absolute standard deviations with respect to the execution times, while Figure 4.24 shows the corresponding relative standard deviations. The absolute standard deviations are a valuable metric to directly asses how large the deviations might be that a user of a DSPS application has to face. All deviation values are calculated for every system-query-SDK combination. The dimension SDK distinguishes between using the Apache Beam SDK or native system APIs for application development. Deviations for the two parallelism factors are averaged and condensed in this way. This is done since separate visualizations for different parallelisms would not reveal any further insights. Additionally, the reduced number of values simplifies the analysis of standard deviations.

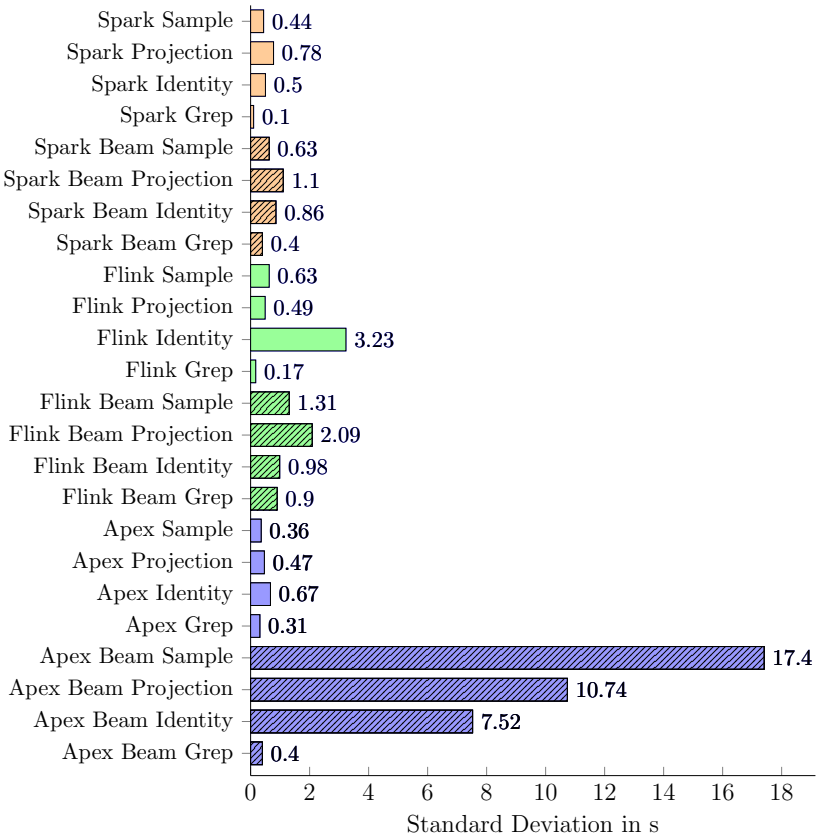


Figure 4.23: Standard deviations for system-query-SDK combinations

When looking at Figure 4.23, it can be seen that there are noticeable differences amongst systems, queries, and implementation variants. The largest absolute standard deviations are found for queries executed on Apache Apex and implemented using Apache Beam, the grep query being the only exception. Between the absolute standard deviations for these three queries, there are also noticeable differences. The sample query implemented using Apache Beam and executed on Apache Apex shows the highest absolute standard deviation with about 17.4 s. The projection query and the identity query follow with deviations of 10.74 s and 7.52 s respectively. Apart from the Apache Apex-Apache Beam combinations, the highest absolute standard variation belongs to the native Apache Flink identity query. In summary, the highest absolute standard deviations vary by multiple seconds and belong to the highest overall average execution times.

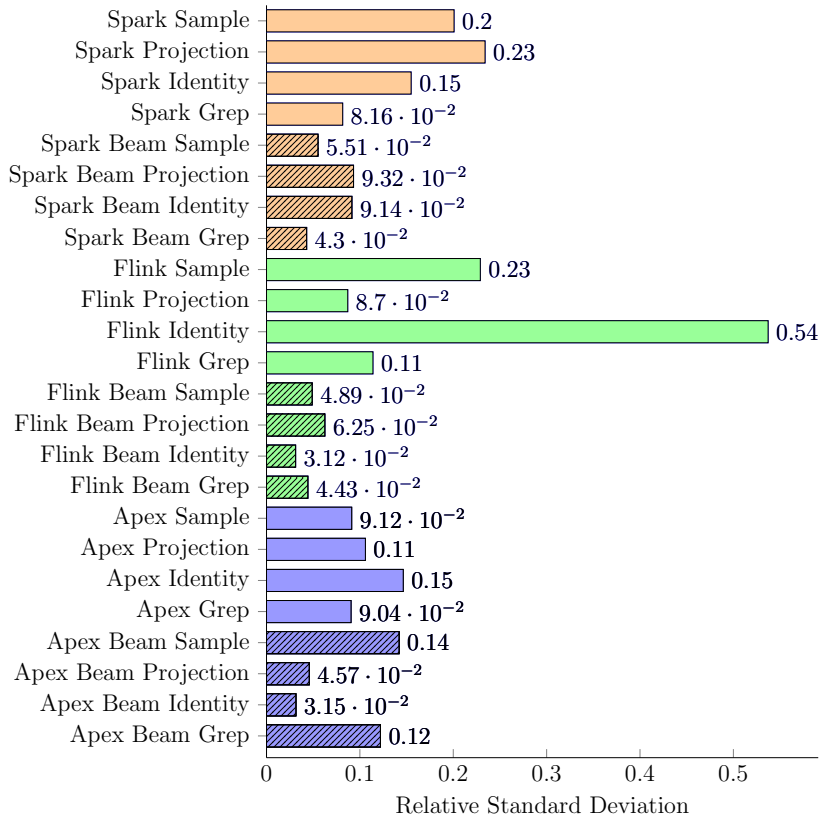


Figure 4.24: Relative standard deviations for system-query-SDK combinations

Figure 4.24 visualizes the relative standard deviations for the measurements. There is one value that is notably higher than others, which belongs to the identity query executed on Apache Flink. That is not a surprising result as the

absolute standard deviation was identified as one of the highest, while the execution times are relatively low. Figure 4.19, which is visualizing the execution times of the identity query, makes visible that there is a noticeable difference between the numbers for the two parallelism factors for Apache Flink. Particularly, the execution time for Apache Flink with a parallelism of one is almost 75 % higher than the one for Apache Flink with a parallelism of two. Although it seems to be plausible at a first glance that higher parallelisms lead to better performance, this correlation is absent for other results.

Number of Run	Parallelism = 1	Parallelism = 2
1	6.25 s	4.15 s
2	21.56 s	3.77 s
3	3.42 s	2.71 s
4	3.31 s	5.29 s
5	3.73 s	3.00 s
6	12.69 s	3.93 s
7	3.90 s	2.90 s
8	3.96 s	3.66 s
9	3.42 s	3.57 s
10	3.01 s	4.45 s

Table 4.6: Execution times for the identity query implemented using the native system APIs and executed on Apache Flink

The analysis of the corresponding execution times provides a more detailed picture of this setting. Table 4.6 shows the execution times for the benchmark runs of the identity query on Apache Flink, i.e., numbers for the corresponding ten runs with a parallelism of one as well as for the ten runs with a parallelism of two. When looking at these measurements, it becomes clear that there are two to three outliers that cause the observed relatively high coefficient of variation. While the results for the higher parallelism are relatively homogeneous, there are outliers in the list of execution times for runs with a parallelism of one. Particularly, seven out of ten execution times range from three to four seconds. The times for the remaining benchmark runs differ significantly. To be more concrete, these runs lasted about 6 s, 12.5 s, and 21.5 s. So the greatest execution time is more than seven times higher than the lowest one. These outliers are the reason for the comparatively high relative standard deviation. Apart from the identity query executed on Apache Flink, there are no further values that stand out in Figure 4.24. These findings overall indicate a higher variance of response times for the discussed experimental setting.

Performance Impact of Apache Beam

The performance impact factors are calculated based on the arithmetic means of the measured execution times.

The averages are determined as follows:

$$\bar{t}(dsps, query, k, p) = \frac{1}{N_{run}} \sum_{r=1}^{N_{run}} t(dsps, query, k, p, r),$$

where $\bar{t}(dsps, query, k, p)$ denotes the average over the execution times for a certain DSPS, query, kind of implementation, i.e., using Apache Beam or native system APIs, and certain parallelism. Variable k represents the mentioned kind of implementation and p denotes the used degree of parallelism. The number of benchmark runs is expressed as N_{run}, which is equal to ten for the context of this paper. The execution time for a single query run of a certain benchmark scenario is shown as $t(dsps, query, k, p, r)$.

The slowdown factor calculation makes use of these arithmetic means. Specifically, it is computed as follows:

$$sf(dsps, query) = \frac{1}{N_p} \sum_{p=1}^{N_p} \frac{\bar{t}(dsps, query, Beam, p)}{\bar{t}(dsps, query, native, p)},$$

where $sf(dsps, query)$ denotes the slowdown factor for a given DSPS and a given query. N_p depicts the number of parallelisms tested, which equals two in the previously discussed benchmark scenario. Particularly, we applied a parallelism factor of one as well as a parallelism of two. So in simplified terms, the ratio of average execution times for Apache Beam implementations and these using native system APIs is calculated and again averaged over parallelisms, all for a given query and DSPS combination.

The average execution times for a certain system, query, and parallelism are determined, separately for the Apache Beam version as well as the implementation using the DSPS SDK. The average execution time belonging to the Apache Beam variant is then divided by the corresponding average for the native query. That is done for every parallelism. The resulting factors for each parallelism are finally averaged by dividing their sum by the number of parallelisms.

The result tells how much slower or faster the Apache Beam version for a query and DSPS performed in the conducted measurements. A result greater than one marks a slowdown, whereas a result smaller than one means that the Apache Beam implementation was faster than the one using native system APIs.

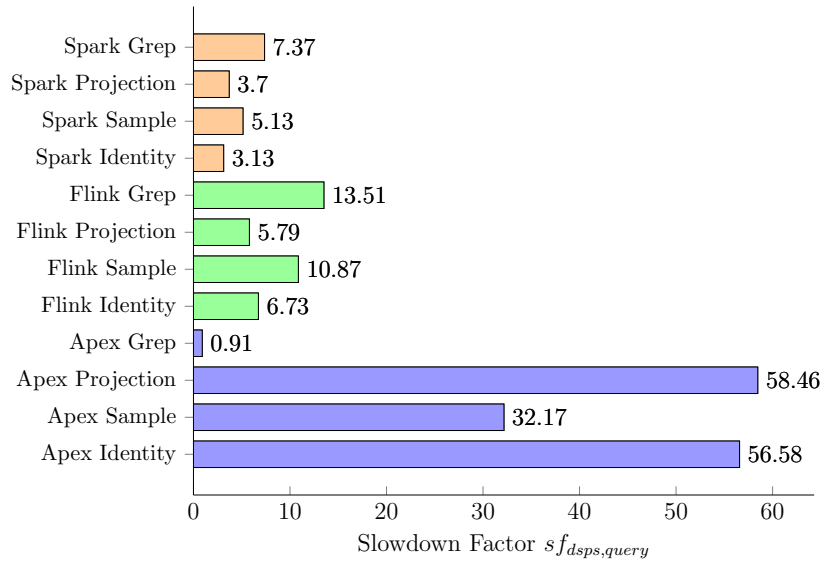


Figure 4.25: Slowdown factors for the analyzed stream processing systems and queries

The results for the computed slowdown factors are visualized in Figure 4.25. It can be seen that the Apache Beam implementations are slower for almost all DSPSs and queries in comparison to these developed using native system APIs. Generally, one can recognize differences between the studied systems and queries. When looking at Apache Flink and Apache Spark, the slowdown factors are similar. This especially holds true with respect to relative distinctions amongst queries. Particularly, the performance penalty for the fastest query, namely the grep query, is highest. Accordingly, it is the lowest for the longest-running queries projection and identity for both systems. Overall, the performance impact on Apache Flink is slightly higher.

In contrast, the Apache Apex results show a different pattern. The highest performance impact can be seen, contrary to Apache Flink and Apache Spark, for the longest-running queries projection and identity. The query with the shortest execution time, the grep query, is overall the only query where the Apache Beam implementation is even faster than the one using native system APIs according to the calculated slowdown factor. However, this speedup is very low, i.e., the Apache Beam query implementation is about as fast as the one using the SDK of Apache Apex. So with a higher number of executions, this difference might vanish.

When looking at the slowdown factors, there are also noticeable differences between Apache Apex and the other two analyzed systems. Except for the

grep query slowdown, all slowdown factors are significantly higher compared to Apache Flink or Apache Spark Streaming. The slowdown factor for the projection query, e.g., is about 58 and so more than four times higher than the highest slowdown factor for either Apache Flink or Apache Spark Streaming.

In summary, the conducted benchmark shows that Apache Beam has a negative performance impact for almost all scenarios. Averaged over systems, the performance penalty is lowest on Apache Spark, closely followed by Apache Flink. Patterns between these two systems and executed queries are similar. The performance impact on Apache Apex is different, meaning the impact is significantly higher in most of the cases. Furthermore, the previously mentioned pattern is vice-versa, i.e., the performance penalty on Apache Apex is highest when it is lowest for the other two DSPSs. The more output a query produces or the higher the execution time on Apache Apex, the higher the performance impact of Apache Beam. The grep query running on Apache Apex is an exception to that as explained before. Apart from this exceptional case, slowdown factors range from about three to almost 60. Thus, in most of the studied cases, Apache Beam has a significant influence on performance when looking at the calculated slowdown factors. Accordingly, with respect to the experimental evaluation of ESPBench shown before, this finding indicates that the execution of queries implemented with the DSPS SDKs would have let to different latencies.

To analyze the differences between the execution of queries developed with the Apache Beam SDK and with a DSPS SDK, we studied how they are executed. Figure 4.26 and Figure 4.27 visualize the execution plans for the grep query executed with a parallelism of one on Apache Flink, implemented without and with Apache Beam, respectively. These two plans serve as an example, which highlights the differences between the execution of an application developed using a DSPS SDK and one using the Apache Beam SDK. Information on execution plans are retrieved from the Apache Flink system and visualized using the Apache Flink *Plan Visualizer*[8].

The first execution plan depicted in Figure 4.26 contains three elements, a *data source*, an *operator*, and a *data sink*. Particularly, the source is shown as a *custom source*, the sink as an *unnamed sink*, and the operator is a *filter*. These details fit the definition of the grep query, as it basically filters data. Data is forwarded along these three elements.

The second execution plan is presented in Figure 4.27. It comprises seven elements in total. In particular, these elements are a *data source*, followed by six *operators*. The data source at the beginning is named *PTransformTransla-*

[8] `https://flink.apache.org/visualizer/`, accessed: 2020-12-30

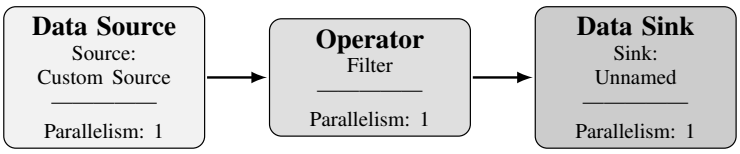


Figure 4.26: Execution plan of Apache Flink for the grep query implemented using the Apache Flink SDK

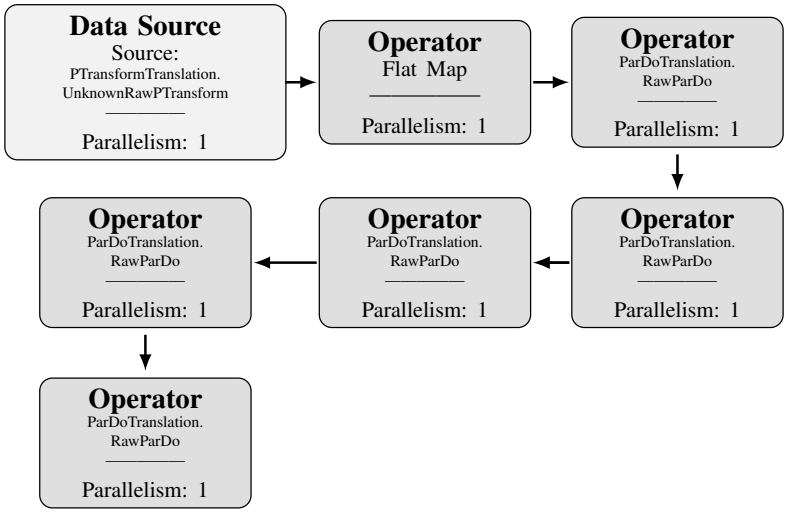


Figure 4.27: Execution plan of Apache Flink for the grep query implemented using Apache Beam

tion.UnknownRawPTransform. *PTransformTranslation* is a registry of familiar transforms and uniform resource names (URNs)[9]. A *PTransform* is used, e.g., for reading or writing to an external storage system.

The *data source* forwards data to the first operator, a *flat map*. This operator performs an action on each input value and produces zero or more output values. Its Apache Beam counterpart is the *read()* method of the *KafkaIO* class, which creates a *Read* PTransform. The remaining five *ParDoTranslation.RawParDo* operators follow the flat map. A *ParDoTranslation* comprises tools for working with instances of *ParDo*. A ParDo is one of the core transforms provided by Apache Beam. The first ParDo represents calling *withoutMetadata()* on the *Read* PTransform, which drops the Kafka metadata as it is not needed. Moreover, the method again returns a PTransform containing a *PCollection* of key-value pairs. The downstream operator represents the call of the *create()* method belonging to the class *Values*. This operator takes the previously created PCollection of key-value pairs and returns a PCollection containing only

[9] `https://beam.apache.org/contribute/runner-guide/`, accessed: 2020-12-30

the values. Further downstream, the grep query logic is applied and resulting values are sent to Apache Kafka[9] [ABC+15, HMG+19].

When comparing both execution plans, it becomes visible that the plan for the query implemented using Apache Beam is significantly larger, i.e., it contains more elements in comparison to its counterpart. That is due to the more complex management of communication with Apache Kafka and could cause lower performance. Both plans have in common that they start with a data source and that all elements are executed with a parallelism of one, due to the defined degree of parallelism. A dedicated data sink cannot be identified for the program developed using Apache Beam. Thus, the sink must be represented as an operator.

Overall, the performance of Apache Beam applications highly depends on the runner implementations. The effort put into this development is likely to vary between systems. The closeness of the DSPSs' programming model to the underlying concepts of Apache Beam also impacts the application execution. Further details, e.g., with respect to the concrete impact of the additional operators, could be uncovered through profiling applications. However, all measurements are a snapshot in time and results may differ with different versions of Apache Beam, other DSPS versions, or alternative system configurations. Moreover, changed workload characteristics might also influence performance results. More detailed analyses were out of scope for this thesis.

4.2.3 Lessons Learned

The benchmark results show that Apache Beam has a noticeable impact on the performance of DSPSs in almost all cases. Programs developed using Apache Beam suffered from a slowdown of up to a factor of 58 in the worst case. At the same time, there is one scenario where the query developed using Apache Beam is slightly faster as its counterparts using the APIs of the corresponding DSPS. However, for most scenarios, we observed a slowdown of at least a factor three.

The results lead to two major conclusions. Firstly, using Apache Beam as an abstraction layer for application development comes at a significant cost in terms of runtime performance. Secondly, the results of benchmarking different DSPSs using a program developed with Apache Beam are not likely to represent the performance differences, which are to be expected from a benchmark with programs developed using native system SDKs. So depending on what is supposed to be studied, benchmark users need to decide on how to implement the benchmark queries. While using Apache Beam certainly provides greater flexibility to switch underlying DSPSs with relatively low effort, users need to

be aware of the fact that this advantage comes with a negative impact on performance. This performance penalty varies among systems and applications and is currently hard to predict.

4.3 Performance Capabilities of Apache Kafka

Apache Kafka is a central component of ESPBench that does not belong to the SUT. Hence, it is important to understand the performance capabilities of Apache Kafka to ensure that the message broker does not become a bottleneck that distorts benchmark results. This Apache Kafka analysis section is divided into two parts. The first part studies the ingestion rates, which can be achieved with Apache Kafka and how configuration parameters impact these rates. The second part analyzes the impact of the batching mechanisms employed by Apache Kafka on the timestamps taken by the broker when a record is appended to the log.

4.3.1 Ingestion Rate Capabilities

These days, where data masses keep growing and applications move to the cloud, horizontal scalability becomes increasingly important. Message brokers play a central role in modern IT landscapes, as they allow us to adapt to data sources that face rises in volume or velocity. Moreover, they can be used to decouple disparate data sources from applications. Usage scenarios where message brokers are employed are manifold and reach from, e.g., machine learning [ZL19] and stream processing architectures [HL15, HMRU17, HRM+17] to general-purpose data processing [ZWL+17].

When using a system within an IT landscape, it is crucial to know if the functional and non-functional requirements for the scenarios it is supposed to be used for are met. If non-functional requirements related to performance are not satisfied, the system might become a bottleneck. This situation does not necessarily mean that the system is unfit for a given use case, but can also indicate a suboptimal system configuration. It is, therefore, a challenge for users to know about or be able to evaluate the capabilities of a system in certain environments and with distinct configurations. However, this knowledge is a prerequisite for making informed decisions about whether a system is suitable for the existing use cases. Additionally, having this information is also crucial for finding well-fitting system configurations.

At the beginning of this section, we introduce the benchmark architecture used for the Apache Kafka capability analysis. Afterward, we present the bench-

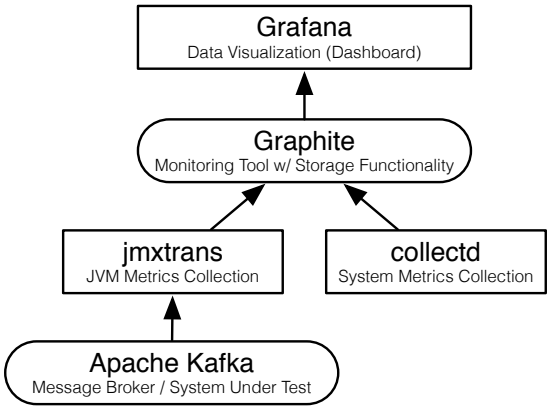


Figure 4.28: Monitoring architecture for the Apache Kafka analysis in FMC

mark process, followed by the data sender configurations. The ESPBench data sender tool is used for this study. Section 4.3.1 describes the benchmark results of the ingestion rate analyses. A brief summary follows at the end.

Benchmark Architecture

The architecture of the monitoring system is shown in Figure 4.28. We use *Grafana*[10], an open-source tool for creating and managing dashboards and exporting data, as the graphical interface to the user. The presented benchmarks employ version 5.4.5 of its *docker* image[11]. Operating system-level (OS-level) virtualization through docker is used for the ease of installation and replicability of results. The OS base image used in this image enables a simple time zone configuration via an environment variable, which is important for time synchronization among all systems. Later versions of the image contain a different OS, specifically *Alpine Linux*[12], which no longer supports this feature.

Grafana fetches the data to display from *Graphite*[13], an open-source monitoring tool. It consists of three components: *Carbon*, *Whisper*, and *Graphite-web*. *Carbon* is a service that retrieves time-series data, which is stored in *Whisper*, a persistence library. *Graphite-web* includes an interface for designing dashboards. However, these dashboards are not as appealing and functionally comprehensive as the corresponding components of *Grafana*, which is why *Grafana* was included. For the installation of *Graphite*, the official *docker* image in version 1.1.4[14] is used, again for time zone configuration reasons.

[10] `https://grafana.com`, accessed: 2020-12-30
[11] `https://grafana.com/grafana/download/5.4.5?platform=docker`, accessed: 20200-12-30
[12] `https://alpinelinux.org`, accessed: 2020-12-30
[13] `https://graphiteapp.org`, accessed: 2020-12-30
[14] `https://hub.docker.com/r/graphiteapp/graphite-statsd/tags?page=1&ordering=last_updated&name=1.1.4`, accessed: 2020-12-30

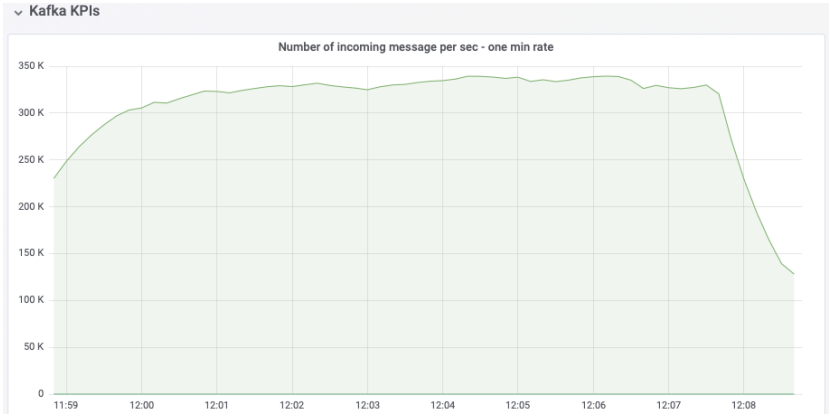


Figure 4.29: Excerpt of the developed Grafana dashboard for the Apache Kafka analysis

Graphite receives its input from two sources: *collectd* and *jmxtrans*[15]. The former runs on the broker's machines in the described setup. The tool offers plugins for gathering OS-level measurements, such as memory usage, system load, and received packages over the network. *Jmxtrans* is a tool for collecting Java Virtual Machine (JVM) runtime metrics. These metrics are provided via *Java Management Extensions* (JMX)[16]. Using *jmxtrans*, we tracked internal metrics, such as JVM memory usage, the number of incoming bytes, and the number of messages entering Apache Kafka per time unit.

Apache Kafka is the system under test (SUT) in the conducted evaluation. However, it can be exchanged for any other system running in a JVM, i.e., the proposed architecture is not limited to Apache Kafka or message brokers in general. The information gathered in *Graphite* is summarized in a *Grafana* dashboard. Exports of the collected *Grafana* data enable further analyses. An excerpt of the developed Grafana dashboard is visualized in Figure 4.29. It shows a graph presenting the one-minute rate of the number of incoming messages per second.

Apache Kafka is installed on three virtual machines with identical hardware and configurations, which are shown in Table 4.7. We use a commodity network setup, whose bandwidth we determined using *ipfer3*[17]. The write performance is measured using the Unix command-line tool *dd*[18], specifically with the command: *dd if=/dev/zero of=/opt/kafka/test1.img bs=1G count=1 oflag=dsync* .

[15] `ttps://www.jmxtrans.org`, accessed: 2020-12-30

[16] `https://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html`, accessed: 2020-12-30

[17] `https://github.com/esnet/iperf`, accessed: 2020-12-30

[18] `https://www.gnu.org/software/coreutils/manual/html_node/dd-invocation.html`, accessed: 2020-12-30

Characteristic	Value
Operating system	Ubuntu 18.04.2 LTS
CPU	Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz, 8 cores
RAM	32GB
Network bandwidth	1Gbit: - measured bandwidth between nodes: 117.5 MB/s - measured bandwidth intra-node transfer: 908 MB/s
Disk	min. 13 Seagate ST320004CLAR2000 in RAID 6, access via Fibre Channel with 8Gbit/s: measured write performance about 70 MB/s
Hypervisor	VMware ESXi 6.7.0
Kafka version	2.3.0
Scala version	2.12.8
Java version	OpenJDK 1.8.0_222

Table 4.7: Server characteristics of the Apache Kafka broker nodes used for the Apache Kafka Analysis

The data sender, taken from the ESPBench toolkit, is a Scala application compiled to a fat *jar* file. It is executed using OpenJDK 1.8 with the default parallel garbage collector (*ParallelGC*). The data sender is assigned an initial memory allocation pool of 1 GB, while the maximum size of this pool amounts to about 14 GB. Apache Kafka uses an initial and maximum memory allocation pool of 1 GB and the *Garbage-First garbage collector* (G1 GC).

Benchmark Execution Process

Each analysis run lasts ten minutes. The main characteristic studied is the number of incoming or ingested messages. Particularly, the one-minute rate of this KPI is analyzed, i.e., the number of incoming messages during the last minute. If not stated otherwise, the data sender is executed on the broker server where the topic is stored.

To reduce the number of manual steps needed, *ansible* is used for automation. Starting the ansible script triggers a build of the data sender project, the creation of a topic, and the assignment of this topic to the first of our three Apache Kafka brokers. For all measurements, we use topics with a single partition and a replication factor of one. Having one partition is a setting used for scenarios in which the order of data is crucial. That is the case as Apache Kafka only makes guarantees for the correct message order within a single partition.

After the Apache Kafka topics are prepared, the data sender is started. Subsequently, a rise in the number of incoming messages of Apache Kafka can be

observed using the Grafana dashboard. Once the configured sending period has passed, the ansible script stops and the dashboard charts adapt correspondingly. The dashboard data is then exported as CSV. The timeframe of these exports is configurable in Grafana.

We incorporate the same data set as used in ESPBench as input, i.e., the data of the *Grand Challenge* published 2012 at the conference *Distributed and Event-Based Systems* (DEBS) [JHF+12]. When the end of the input file is reached, the data sender starts again from the beginning.

Data Sender Configuration

Table 4.8 shows the default configuration parameters, which the data sender applies to the Apache Kafka producer. Unless otherwise stated, these are the parameters employed in the presented measurements in Section 4.3.1. An Apache Kafka producer batches messages to lower the number of requests, thereby increasing throughput. The *batch-size* property limits the size of these message packages. The initially used value is the default of 16,384 bytes, as defined in the Apache Kafka documentation[19]. The *acks* producer property determines the level of acknowledgments for sent messages. There are three different options for the *acks* configuration:

- *0*: The producer does not wait for any acknowledgment and counts the message as sent as soon as it is added to the socket buffer.
- *1*: The leader will send an acknowledgment to the producer as soon as the message is written to its local log. The leader will not wait until its followers, i.e., other brokers, have written it to their log.
- *all*: The leader waits until all in-sync replicas acknowledge the message before sending an acknowledgment to the producer. By default, the minimum number of in-sync replicas is set to one.

In addition to the configuration of the Apache Kafka producer, the developed data sender tool also provides configuration options. One example is the *read-in-ram* Boolean setting, which determines how the input data is read. If *read-in-ram* is not set, the data source object returns an iterator object of the records. If it is set, the source object first loads the entire data set into memory by converting it into a list, and then returns an iterator object for the created data structure. Unless otherwise stated, *read-in-ram* is enabled in the presented results. Furthermore, the number of mes-

[19] `https://kafka.apache.org/documentation/`, accessed: 2020-12-30

Property	Value
key-serializer-class	org.apache.kafka.common.serialization.StringSerializer
value-serializer-class	org.apache.kafka.common.serialization.StringSerializer
batch-size	16,384 bytes
buffer-memory-size	33,554,432 bytes
acks	0

Table 4.8: Default properties applied to the Apache Kafka producer for the Apache Kafka analysis

sages the data sender should send per time unit can be adapted using the `java.util.concurrent.ScheduledThreadPoolExecutor` class. It can execute a thread periodically by applying a configurable delay. By using this parameter, we can determine how many messages are to be sent per time unit. Each execution sends a single message to the Apache Kafka broker. So a configured delay of, e.g., 10,000 ns, leads to an input rate of 100,000 messages/second (MPS).

Ingestion Rate Analysis

This section presents the ingestion rate (*ir*) analysis performed for Apache Kafka. It comprises analyzing three selected input rates with varying configurations regarding *acks* levels, *batch size*, data sender locality, *read-in-ram* option, and data sender processes.

Result Overview

Figure 4.30 shows the maximum achieved ingestion rates of Apache Kafka for the analyzed configurations. The ingestion rates illustrated in all figures are the one-minute rates of incoming messages/second, a KPI provided by Apache Kafka. For all benchmark scenarios with the configured input rate of 1,000,000 messages/second, we selected the runs with the most stable rate.

The highest input rate with about 421,000 messages/second was achieved with two distinct data sender processes, each sending 250,000 messages/second. However, this is less than the configured input rate. With a single data sender configured to send 1,000,000 messages/second, the input rates are lower. The results for the *acks* levels of *1* and *all* are similar with input rates around 340K MPS. Surprisingly, sending messages without waiting for acknowledgment, i.e., *acks* set to *0*, decreased the achieved input rate. The maximum is at about 294,000 messages/second with an increased default batch size. In contrast to the other benchmark scenarios, the achievable input rate with acknowledgments disabled could be positively influenced by a higher batch size without harming the stability of the input rate.

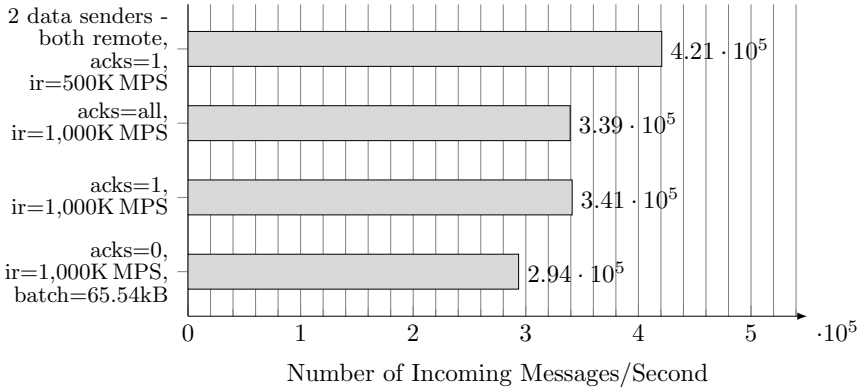


Figure 4.30: Result overview of the achieved numbers of ingested messages/second into Apache Kafka - one-minute rate

Input Rate of 100,000 Messages/Second

Figure 4.31 shows the one-minute rates of incoming messages/second for a configured data input rate of 100,000 messages/second. The parameters under investigation for this benchmark series are the data sender locality, the *acks* level, and the *read-in-ram* option. Similar to all other observations, an increase in the number of incoming messages/second can be seen at the beginning. This is when the data sender is started and the one-minute rate begins to adapt accordingly. Likewise, a sudden decrease in ingested messages per second is present at the end of all charts. This is due to the termination of the data sender after the configured period of time has been reached. Consequently, the most interesting part of the evaluations is the data presented in the center of the plots.

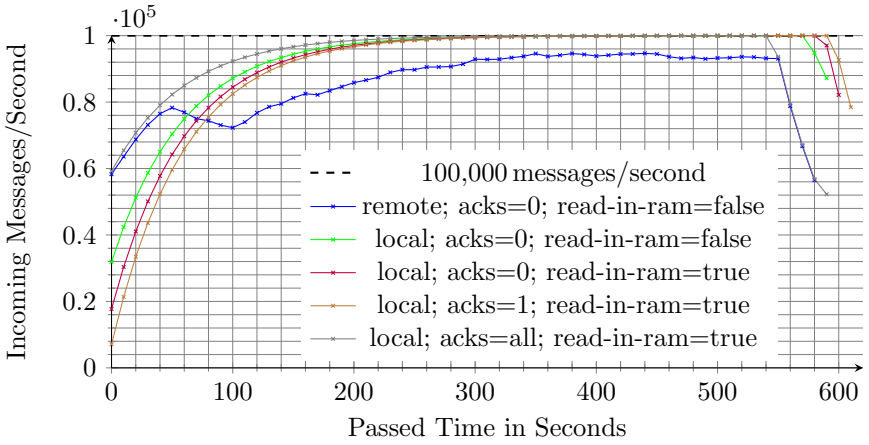


Figure 4.31: Ingested messages/second - one-minute rate, configured ingestion rate of 100,000 messages/second

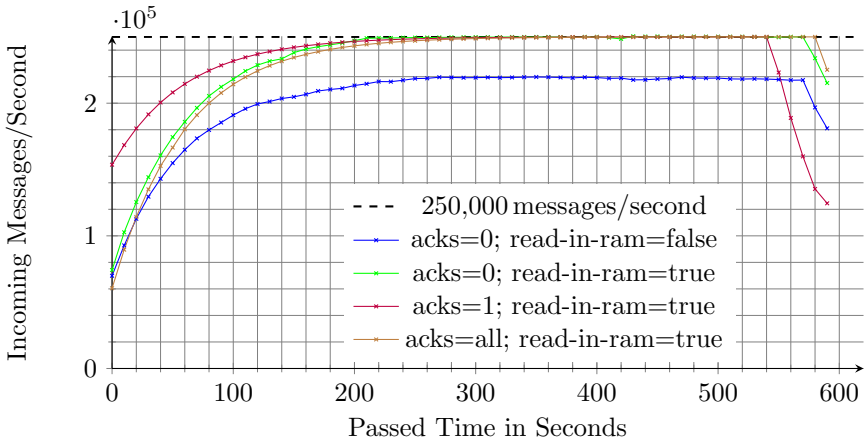


Figure 4.32: Ingested messages/second - one-minute rate, configured ingestion rate of 250,000 messages/second

Figure 4.31 further visualizes that almost all chosen settings reach the configured input rate of 100K messages/second. Particularly, all tested *acks* level and *read-in-ram* combinations reach the configured rate of 100K messages/second. The only exception is the *remote; acks=0; read-in-ram=false* configuration, which is represented by the blue line. In this setting, the data sender was executed remotely, specifically on a 2015 Apple MacBook Pro, which was connected to the network using Ethernet. For all other benchmark runs in the following, the data sender was executed on the broker where the topic is stored.

Input Rate of 250,000 Messages/Second

Figure 4.32 shows the results for an input rate of 250,000 messages/second. As we already identified the limits of the commodity hardware in Figure 4.31, we do not pursue further tests with the laptop configuration. Figure 4.32 highlights the significance of the *read-in-ram* configuration. Particularly, the three configurations where *read-in-ram* is set to *true* reach the configured input of 250,000 messages/second, whereas the run where *read-in-ram* is set to *false*, represented in blue, does not. This configuration, where *read-in-ram* is not active, is not able to handle more than about 220,000 messages/second. Thus, enabling *read-in-ram* has a positive influence on the achievable number of ingested messages per second as the latency for accessing the main memory is lower than for accessing the disk. It is evident that with *read-in-ram* disabled, there is a bottleneck at the data sender side at this configured input rate. The data cannot be read as fast as it is required to achieve an ingestion rate of 250,000 messages/second.

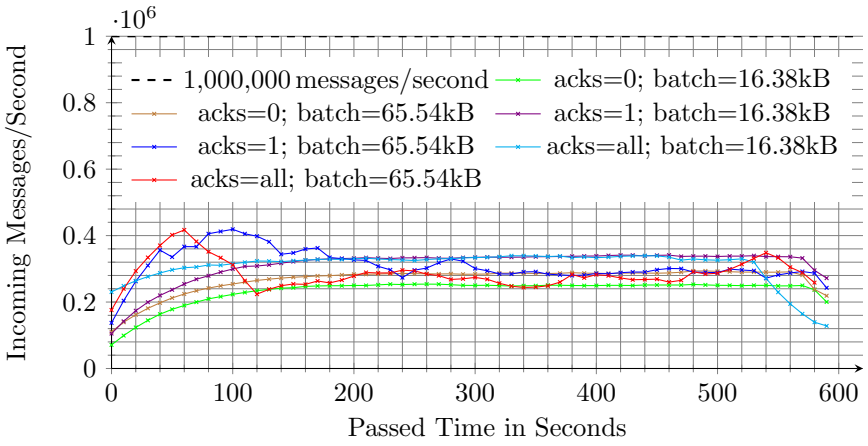


Figure 4.33: Ingested messages/second - one-minute rate, configured ingestion rate of 1,000,000 messages/second

Input Rate of 1,000,000 Messages/Second

Figure 4.33 visualizes the ingestion rate results for an input rate of 1,000,000 messages/second. As we discovered the limits of configurations with *read-in-ram* set to *false* previously, the parameter is enabled for all following measurements. Additionally to testing different *acks* levels, we analyze the effects of changes to the *batch size*. Particularly, we study the default size and a *batch size* increased by a factor of four, which amounts to a batch size of 65.54 kB.

None of the configurations reaches the configured ingestion rate. While the highest ingestion rates peak at about 420,000 messages/second for a short period, the lowest one is at about 250,000 messages/second. The two configurations that achieve this maximum peak are the ones with a *batch size* of 65.5 4kB and *acks* set to *1* and *all*. However, these are also the only two scenarios where no steady ingestion rate could be established. The *acks* level of *0* combined with the default *batch size* reached the lowest ingestion rate. Changing *acks* to either *1* or *all* resulted in a rise to a rate of about 320,000 messages/second. Concerning the *batch size*, the increase resulted in a higher ingestion rate for the scenarios without acknowledgments. Specifically, a rise of more than 20K messages/second can be observed. For the other acknowledgment settings, the raised *batch size* led to an unstable ingestion rate.

Figure 4.34 shows the incoming data rates in megabyte (MB)/second. The data volumes ingested are provided by Apache Kafka as the metric *BytesInPerSec*. The chart fits the corresponding ingestion rates shown in Figure 4.33. The highest peaks are at about 90 MB/second. The measured maximum network

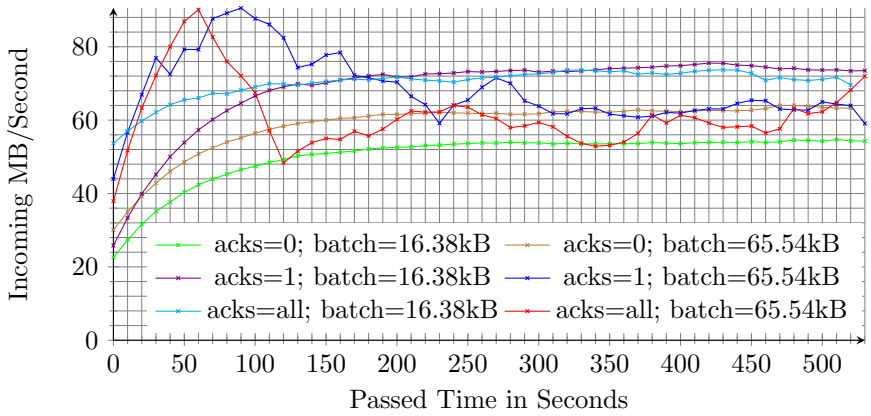


Figure 4.34: Incoming MB/second - one-minute rate, configured ingestion rate of 1,000,000 messages/second

bandwidth between the Apache Kafka brokers is about 117.5 MB/second, see Table 4.7. Therefore, if further network traffic is created, which is not captured by the *BytesInPerSec* metric, the bandwidth of the employed commodity network could be a limiting factor in peak situations if data is sent from a remote host. As we executed the data sender on the node storing the corresponding topic partition, there was intra-node transfer and we used the loopback interface with its higher bandwidth of about 908 MB/second, which is not a bottleneck. The determined write performance of about 70 MB/second described in Table 4.7 is even closer to the observed limits in Figure 4.34. Depending on how optimized Apache Kafka writes to disk, e.g., using compression techniques, the achievable performance might be higher than the measured disk performance limit. Nevertheless, the observations lead to the conclusion that the ingestion rate is likely to be disk-bound in the viewed benchmark setting.

Figure 4.35 shows the short-term system loads of the Apache Kafka broker containing the topic partition, which is also the server where the data sender is executed. The figure reveals that in all settings, which led to a steady input rate, the broker node has a system load lower than eight. Thus, the broker has not been over-utilized from a system load perspective, since we employed servers equipped with an eight-core CPU each as described in Table 4.7.

The two remaining scenarios show system loads with a maximum value close to 15, which indicates an over-utilization that could limit the achievable ingestion rate. Interestingly, the system load is not proportional to the corresponding ingestion rates. At the time of the peak ingestion rate, e.g., the highest system load has not reached its maximum. This observation is an indicator of a growing number of waiting write operations.

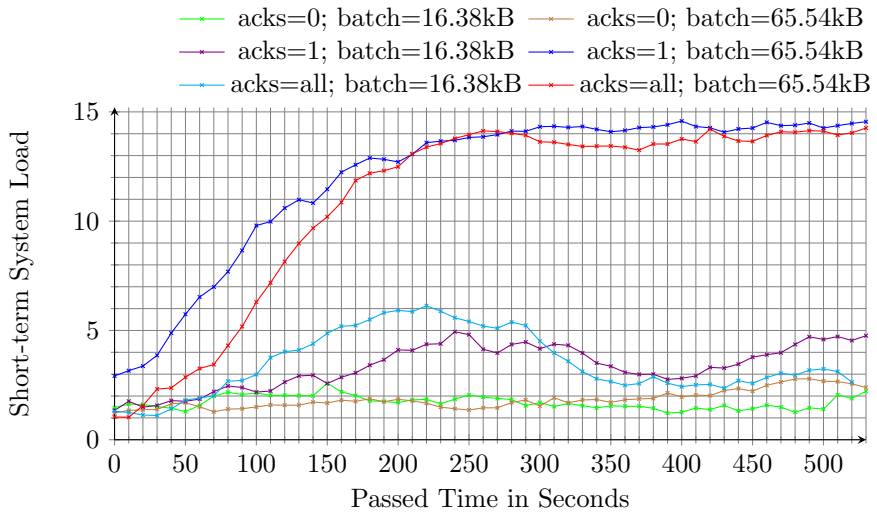


Figure 4.35: Short-term system load of the Apache Kafka broker containing the topic partition - one-minute rate, configured ingestion rate of 1,000,000 messages/second

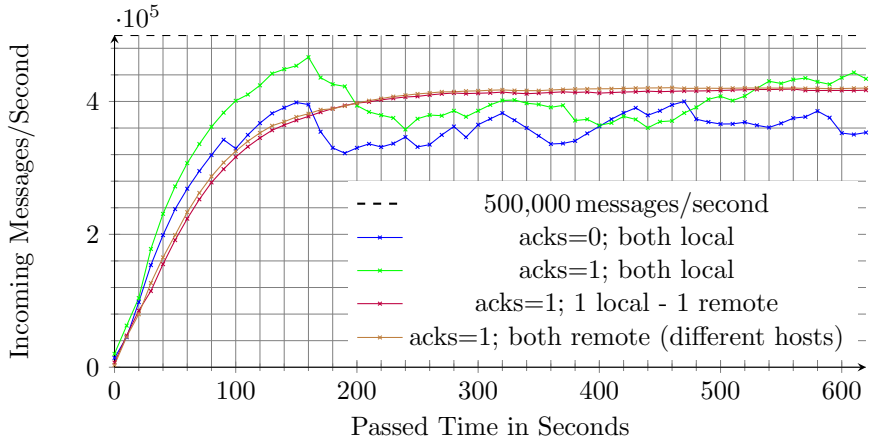


Figure 4.36: Ingested messages/second - one-minute rate, configured ingestion rate of 500,000 messages/second in total with two data senders

Input Rate of Overall 500,000 messages/second with Two Senders

To see if server resources regarding CPU are a limiting factor, we distributed the data sender. We included the default *batch size* and left out the measurements where *acks* are set to *all* as they are, similarly to the previously presented results, practically identical to the runs with *acks* set to *1*. Figure 4.36 shows the achieved ingestion rates. The blue and green lines illustrate the runs where both senders run locally, i.e., on the broker node containing the topic partition.

The blue line visualizes the results for disabled *acks* and the green line those for an *acks* level of *1*. The purple line in Figure 4.36 represents the run where one data sender is invoked on the broker that stores the topic, and one data sender at another broker. The brown line shows the results for the run where the data senders are executed on the two brokers that do not store the topic.

Our measurements show that the two settings where at least one data sender is executed remotely lead to the same result: a steady input rate of about 420,000 messages/second. The benchmark runs having both data senders run locally have a different outcome. Similar to the previous results, the *acks* set to *1* overall outperforms *acks* set to *0*. However, neither configuration reaches a steady input rate. Both have the highest spike at the beginning, which is a behavior observed before.

Next to almost identical trends regarding the ingestion rates compared to the two benchmark settings presented before, the system loads are also equal with a value consistently approaching 15 as visualized in Figure 4.35. This again indicates an over-utilization of the server. For the two configurations where at least one data sender is executed remotely, the system load never exceeds a value of three on any server. Nevertheless, the input rates for the setting with two data senders are the highest on average compared to the previous figures, with a maximum input rate of about 460,000 messages/second. Figure 4.37 shows what these message input rates mean regarding data size. The amount of incoming data in MB/second is visualized for the setting where both data senders were executed locally and remotely with *acks* set to *1*. It can be seen that the maximally achieved input rate of Figure 4.36 corresponds to a data volume-wise input rate of about 100 MB/s. For the constant input rate, where both data senders were executed remotely, a size-wise input of close to 92MB/s is reached. The amount of incoming bytes per second exceeds the measured maximum write performance mentioned in Table 4.7, which could be due to increased parallelization or an optimized way of storing messages implemented in Apache Kafka. As the fully distributed setting uses the *eth0* interface to the broker, the network bandwidth of about 117.5 MB/second applies. Since the reported number of incoming bytes is close to this limit and metadata or further traffic might not be captured, the network represents a potential bottleneck.

Figure 4.38 visualizes the number of packages received on interface *eth0* exemplary for three benchmark runs with a logarithmic scale on the y-axis. This interface is the only physical one, next to the loopback interface, on the used servers. The figure highlights the differences caused by changes in data sender locality. While the number of received packages is not impacted if data is only

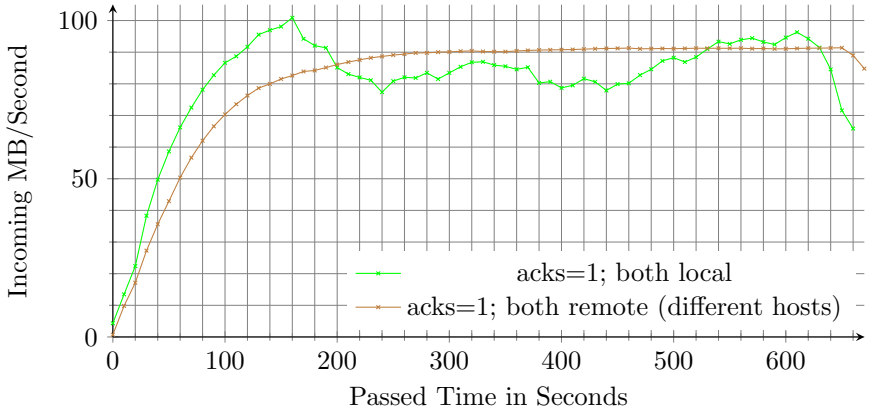


Figure 4.37: Incoming MB/second - one-minute rate, configured ingestion rate of 500K messages/second in total with two data senders

sent from the node where the corresponding target topic is stored, transmitting data from a remote host significantly increases this KPI. Specifically, not having remote data senders results in between 25 and 60 received packages on *eth0*. Having one remote data sender amounts to about 30,000 received packages/second.

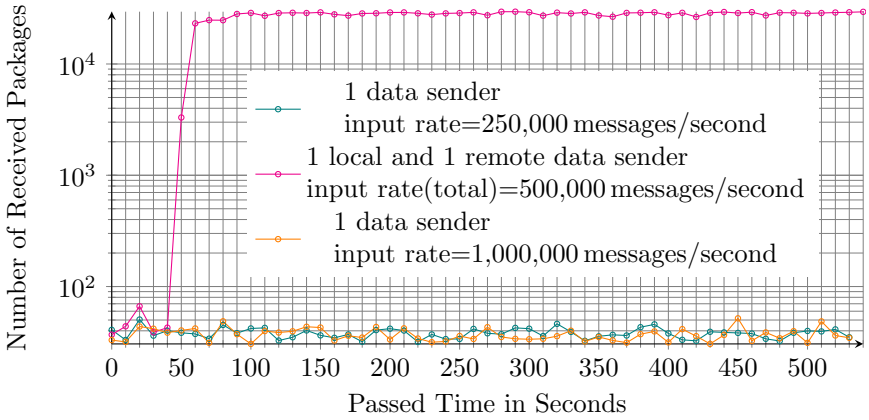


Figure 4.38: Packages received/second on interface eth0 - one-minute rate, *acks=1*

Summary of Ingestion Rate Analysis

Our benchmark results reveal two main insights: firstly, although a single data sender can create an input rate of 250,000 messages/second as shown in Figure 4.32, two independently executed data senders do not lead to an input rate of 500,000 messages/second. Secondly, we see that the influence with respect to where data senders are invoked is noticeable. If two data senders are executed

in parallel on the same host, they are able to overwhelm the server or impede each other, as the observed system load of about 15 indicates, see Figure 4.35. Another limiting factor can be found in the write-to-disk performance of the used server and the network bandwidth when sending data from a remote host. The observed memory usage was never close to its limits for any of the presented benchmark scenarios.

The most promising configuration in the employed setup, which led to stable input rates, has the default *batch size* and *acks* set to *1* or *all*. A stable rate is desirable as it leads to predictable system behavior. Moreover, multiple data senders distributed across nodes are able to increase the achievable ingestion rates. An input rate of about 250,000 messages/second to Apache Kafka can be achieved using a single data sender.

4.3.2 Delay Evaluation of Apache Kafka Log Timestamps

Apache Kafka can be configured to store a timestamp once a record is appended to its log. This timestamp can then be used by consuming applications. For instance, it can be interpreted as the record creation time or used for calculating latencies as the difference between an input record and output record timestamp. The second example describes how the result calculator of the ESPBench toolkit determines latencies. However, batching effects used by Apache Kafka could distort such measurements. In order to get a better understanding of these batching effects, we analyze their impact on the captured log append timestamps. That includes studying the impact of different data input rates and Apache Kafka Producer batch sizes. This section introduces the evaluation setup as well as the results.

Evaluation Setup

We use the data sender that is part of the ESPBench benchmark proposed in this thesis. Moreover, we employ the previously described Apache Kafka cluster with three brokers that are installed on nodes with the system characteristics listed in Table 4.7.

For evaluation, we use the default data sender properties shown in Table 4.8, unless otherwise stated. First, we send the DEBS Grand Challenge 2012 data set to an Apache Kafka topic. Afterward, we consume these records from the corresponding topic and extract the timestamps.

Timestamp Evaluation Results

All presented figures in this section show timestamp differences of a certain number of consecutive records for a certain data sender configuration. Regarding the timestamps, only the difference to the first timestamp is displayed on the y-axis for readability reasons.

Figure 4.39 shows 100 timestamps of consecutive messages sent at a rate of 1,000 messages/second. It can be seen that the trend is clearly linear, i.e., no significant batching effects can be observed for this setting.

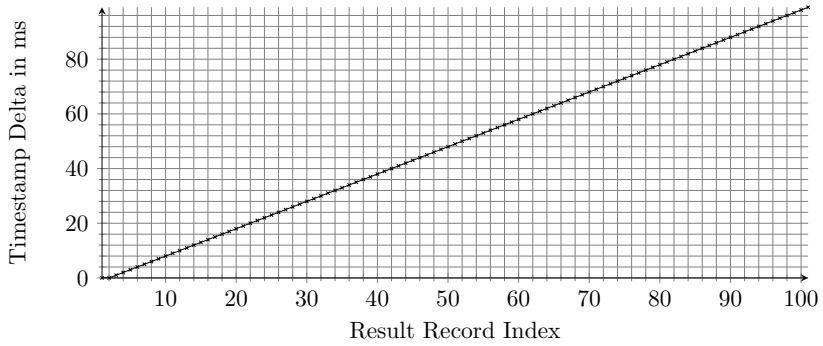


Figure 4.39: 100 consecutive Apache Kafka timestamps, ingestion rate of 1,000 messages/second

Figure 4.40 shows 100 consecutive timestamps for a higher data ingestion rate of 10,000 messages/second. This higher pressure on Apache Kafka reveals a stepwise trend, i.e., the batching applied in Apache Kafka becomes visible by looking at the timestamps plotted in Figure 4.40. However, the time differences between steps are very small. Specifically, there is only a difference of 1 ms between each step and every step or batch contains about 11 data records. These observations show that the batching influence is marginal. This fact becomes especially clear when looking at the result excerpt that is visualized in Figure 4.40. This figure only contains 100 records, while 10,000 messages/second were ingested.

Figure 4.41 visualizes the timestamps of 500 consecutive records, which is still a small amount given the fact that 10,000 records are ingested every second. While a stepwise trend is still visible, the steps are already much smaller compared to Figure 4.40. The more records are shown, the more the graph seems to represent a linear trend.

To see how a larger batch size impacts the observations, we doubled the default batch size to a value of about 32 kilobytes. The corresponding mea-

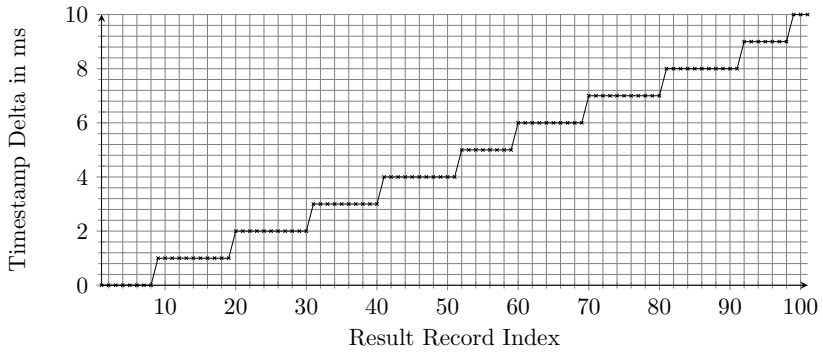


Figure 4.40: 100 consecutive Apache Kafka timestamps, ingestion rate of 10,000 messages/second

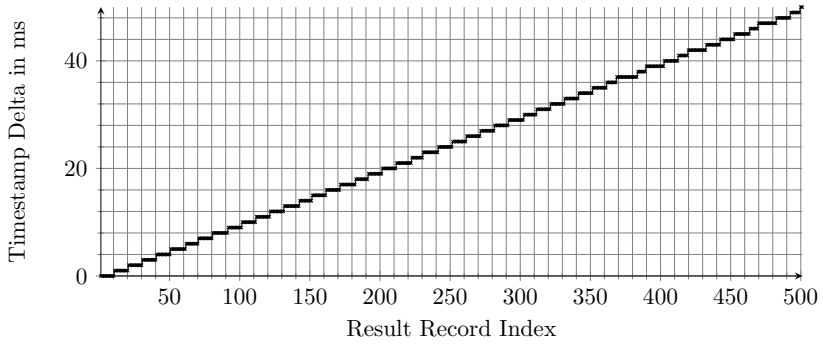


Figure 4.41: 500 consecutive Apache Kafka timestamps, ingestion rate of 10,000 messages/second

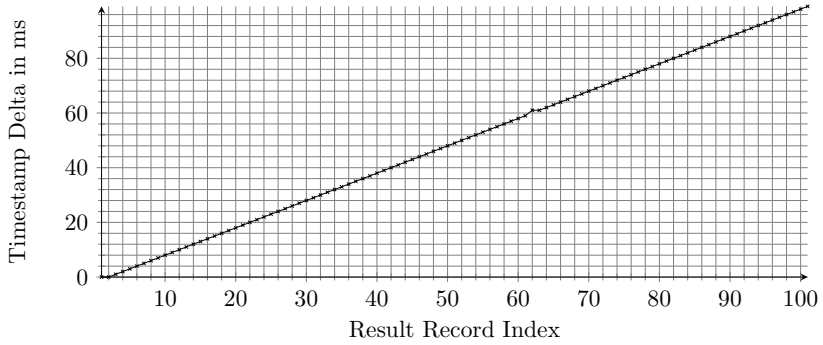


Figure 4.42: 100 consecutive Apache Kafka timestamps, ingestion rate of 1,000 messages/second, batch size of 32 kB

surements for input rates of 1,000 and 10,000 messages/second are visualized in Figure 4.42 and Figure 4.43, respectively. The figures show that the doubled batch size did not lead to a difference with respect to timestamp delays compared to the previous setting with the default batch size. While there is

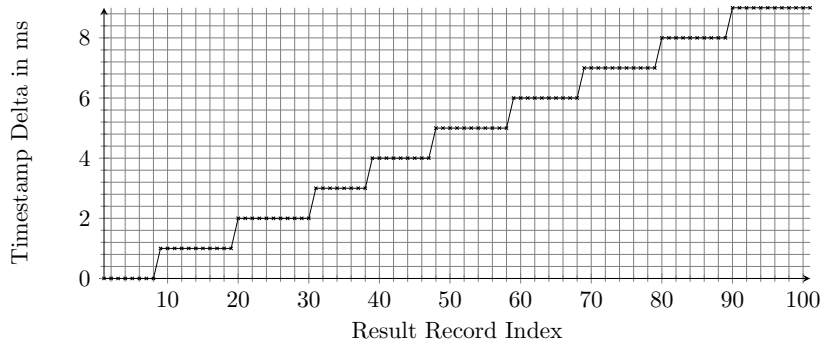


Figure 4.43: 100 consecutive Apache Kafka timestamps, ingestion rate of 10,000 messages/second, batch size of 32 kB

still a linear trend observable for the 1,000 messages/second input rate, the 10,000 messages/second run again reveals batches with about 11 records and a 1 ms difference between steps.

Summary of Timestamp Delay Analysis

Overall, the analyses show that the batching of Apache Kafka becomes visible when looking at the timestamps that Apache Kafka stores along with messages in its *LogAppendTime* settings. However, the results further show that the batching impact is marginal. The results reveal a negligible time difference of 1 ms between batches. Furthermore, the batches are very small with about eleven records on average. Having small batches reduces their overall impact on the timestamps. The smaller the batch size, the closer the behavior is with respect to making no use of batching at all. Doubling the default batch sizes does not change the observed behavior. With respect to ESPBench and its toolkit, the marginal impact on the timestamps, caused by the batching used in Apache Kafka, implies that the batching does not require special consideration when calculating latencies.

4.4 Threats to Validity

There are certain threats to validity related to the presented measurements. They are grouped into threats to *internal validity*, i.e., "the degree to which a study establishes the cause-and-effect relationship between the treatment and the observed outcome" [SDJ01], *external validity*, i.e., "the generalizability of findings from a study" [DGKL11], and *construct validity*, i.e., "the degree to which our scales, metrics and instruments actually measure the properties they are supposed to measure" [RT18].

4.4.1 Internal Validity

In the following, we elaborate on the internal threats to validity for the three major experimental evaluations presented in this dissertation.

Validation of ESPBench

One threat to internal validity is the existence of bottlenecks within the benchmark setup that prevents measuring the actual performance of the SUT. We analyzed the capabilities of the ESPBench data sender and Apache Kafka in selected settings, which indicates how far the data rate can be increased without creating an unintended bottleneck. However, different aspects, such as other operating systems, JVMs, or hardware, can influence the determined limits. Additionally, a high load not related to the benchmark on a server can impact the performance in an unintended way. In particular, either the SUT's performance can be negatively influenced or another system, such as a message broker, can become a bottleneck and thus, prevent a correct performance evaluation of the SUT. Monitoring the loads and processes on the investigated systems helps to identify such scenarios. ESPBench provides such monitoring capabilities by default. We did not encounter any anomalies in our measurements.

Another threat to internal validity is related to the *meaningfulness* of the benchmark results calculated by ESPBench. Particularly, DSPS developers could optimize their system according to the benchmark workload. Consequently, the system might show a great performance compared to other DSPSs. However, these results would have limited meaningfulness due to the optimizations, which some people might see as cheating. Installing certain barriers for preventing such benchmark optimizations is out of scope for the presented work.

The number of benchmark runs is another factor that influences the validity of benchmark results. ESPBench does not mandate a minimum number of runs. So having a high variance in latencies and only a few benchmark runs can result in benchmark results that are not representative of the actual cause-and-effect relationships. However, in order to keep benchmarking simple and economical, it is desirable to prevent unnecessary benchmark runs. In all of the presented experimental evaluations, we chose a number of runs appropriate to experienced result variances. Nevertheless, a threat to validity still exists, e.g., due to wrong assessments of variances.

Performance Impact Analysis of Apache Beam

For analyzing the performance impact of Apache Beam, we used two different ways to implement the used queries. Particularly, we developed each query

using the Apache Beam SDK and the corresponding DSPS SDK. Potential implementation flaws in one of the query implementations could lead to unfair comparisons and misleading performance results. However, the employed query set is rather simple with respect to the number of required computational steps per query, which makes them easy to implement and the implementations less prone to errors.

Apache Kafka Analysis

With our Apache Kafka analysis, we studied the ingestion rates that are achievable by Apache Kafka in certain settings. A threat to validity is that the changed configuration settings, e.g., an altered Apache Kafka producer buffer size, did not cause the variations observed in the results. To decrease this threat to validity, we performed all measurements multiple times and analyzed the variances. No anomalies were found in our studies.

4.4.2 External Validity

The threats to external validity for the three experimental evaluations of this dissertation are presented in the following.

Validation of ESPBench

One threat to external validity is related to the applicability or transferability of results produced by ESPBench to other domains, i.e., domains different from industrial manufacturing. The workload of ESPBench belongs to the manufacturing domain and is validated with companies from this industry sector. Thus, depending on the target domain, the employed combination of data and queries might have a varying value. Even within the manufacturing domain, there might be companies with significantly different workloads. However, as the queries cover all core functionalities of DSPSs, results of ESPBench give at least a general hint on a system's performance. Nevertheless, the existence of an unintended bias in the developed queries represents a related threat to validity.

Closely related to the queries, the data used in ESPBench is another threat to external validity. If users require to process data with different characteristics, e.g., streaming data that only consists of strings, performance results produced by ESPBench may have limited meaningfulness. However, ESPBench can be easily adapted to use other kinds of data.

ESPBench's overall scenario is based on the data and idea of the Grand Challenge from the DEBS conference of 2012. As this challenge was defined

about eight years ago, the *relevance* of this scenario in today's and the future's world embodies another threat to validity. However, we do not see any aspect within the data or benchmark situation that is outdated by today's standards. Our expert interviews with companies from the industrial manufacturing domain further confirmed this assessment.

Performance Impact Analysis of Apache Beam

External threats to validity of the Apache Beam study are related to the validity of results for different settings. Such variations can exist with respect to, e.g., software versions, software configurations, or hardware. Moreover, we employed rather simple queries from the *StreamBench* [LWXH14] micro benchmark. The applicability of results to more complex stream processing applications or different input data characteristics represents another threat to external validity.

Apache Kafka Analysis

The threats to external validity of the Apache Kafka analysis comprise the validity of results to different Apache Kafka versions, configurations, or hardware. Furthermore, the observed results might vary when using other input data characteristics. Different message brokers might also distinguish from Apache Kafka regarding the achievable data ingestion rates.

4.4.3 Construct Validity

In the following, we elaborate on the threats to construct validity of the conducted experimental evaluations.

Validation of ESPBench

Central threats to the construct validity are the ESPBench data sender and the message broker. Both are components that are employed in benchmark runs. If they become a bottleneck, these two components are benchmarked instead of the SUT and so the benchmark results would not represent the desired outcome. However, we ensured that this scenario did not take place through advanced analyses.

A second threat to construct validity is related to the determination of latencies. ESPBench leverages the timestamps stored by Apache Kafka for calculating the latencies. As a result, the time for transferring the data from Apache Kafka to the SUT and from the SUT to Apache Kafka is incorporated into these times. This overhead time is not wanted and represents a threat to the validity

of results, which are supposed to express the SUT's performance. However, our studies of these overhead times reveal that they are very low and neglectable compared to the overall query latencies.

Performance Impact Analysis of Apache Beam

The performance impact analysis of Apache Beam also relies on Apache Kafka timestamps. Due to this study construction, transfer times between the DSPS and Apache Kafka are reflected in the performance results. However, our analysis of these overhead times shows that they are insignificantly low compared to the overall performance measurements.

Apache Kafka Analysis

In our Apache Kafka study, we analyze the incoming message rate of the messaging system. This is a metric determined and provided by Apache Kafka. The trust in Apache Kafka to correctly measure this metric represents a threat to construct validity. However, Apache Kafka is a widely employed and mature system, whose performance metrics are applied in many applications. Moreover, all derived values for the incoming message rate were plausible in the studied scenarios.

5

RELATED WORK

This chapter presents related work. It covers three main topics: data stream processing benchmarks, the Apache Beam performance impact analysis, and the Apache Kafka capability evaluation.

5.1 Data Stream Processing Benchmarks

Only a few benchmarks for data stream processing architectures are available compared to the number of benchmarks for database management systems (DBMSs) [HMRU17]. In the following, we introduce the most prominent benchmarks for data stream processing scenarios.

Linear Road

Linear Road is an application benchmark proposed by Arasu et al. [ACG+04]. It is one of the most popular benchmarks focusing on data stream processing systems (DSPSs). *Linear Road* includes a benchmarking toolkit, which comprises a data generator, a data sender, and a result validator. However, the data sender is provided in an incomplete state, i.e., C/C++ coding is needed to use the tool[1]. With the execution of a benchmark implementation, a variable tolling system of a metropolitan area is simulated. This area covers a configurable number of expressways. The amount of accumulated tolls depends on multiple aspects of the traffic situation on these expressways.

[1] `https://www.cs.brandeis.edu/~linearroad/datadriverinstall.html`, accessed: 2020-12-31

The data sender emits the streaming data into the system under test (SUT). This input data comprises four different record types, from which position reports are by far the most abundant type. Depending on the overall situation on highways, car position reports may require the SUT to create an output or not. The remaining data consists of three record types, which express explicit user requests. These requests always demand an answer from the SUT.

With regard to the benchmark workload, Linear Road defines four different queries with corresponding output types. For complexity reasons, the implementation of the lastly presented query was even skipped in the two implementations described by Arasu et al. [ACG+04]. That indicates a lack of simplicity, at least for this query. Besides streaming data, historical data covering ten weeks of tolling history is generated and stored in files. It has to be incorporated for selected queries in order to produce correct answers.

As a benchmark result, Linear Road defines one overall metric called L-Rating. It indicates how many expressways a system can handle without violating the defined maximum response times for each query. The number of highways is a configurable parameter for the data generation step, which is influencing the amount of input data. However, it is not clear how much the data input rate can be increased before the data sender tool becomes a bottleneck.

StreamBench

Another popular benchmark is *StreamBench* [LWXH14]. It aims at benchmarking distributed DSPSs and can be categorized as a micro benchmark, i.e., it measures atomic operations, such as the execution of a projection rather than those of more complex applications, such as in Linear Road. Thus, when a system's performance for real-world scenarios or applications is to be evaluated, micro benchmark results only have limited validity. However, if, e.g., two distinct filter operators are to be compared, micro benchmarks have advantages over application benchmarks due to their simplicity. Measurements contain only the relevant parts without much overhead, which eases interpreting results.

StreamBench defines seven queries in total. They cover queries with single and multiple computational steps. Moreover, some queries require to keep a state in order to produce correct results, while others do not impose this requirement. Only one query uses numerical data, while all others work on textual data. Overall, the seven queries cover a variety of functionalities, although some typical streaming operations like window functions are not taken into account.

Additionally, StreamBench defines four workload suites, which influence the way the benchmark is executed. The suite has an impact on, e.g., data scales,

the executed query set, the existence of an intentional node failure, or employed benchmark result metrics. With respect to the input data, StreamBench makes use of two different real-world data sets. One of these contains textual data, while the other one comprises numerical information. The two data sets used in StreamBench serve as seeds for data generation.

Contrary to Linear Road, StreamBench employs a message broker, which is used for decoupling data generation and consumption. In particular, Apache Kafka is used as the message broker in StreamBench. This approach is similar to the benchmark architecture proposed with ESPBench in this dissertation. A benchmark tool for data ingestion, such as a data sender, which comes with Linear Road or ESPBench, is not described by the authors of StreamBench.

StreamBench defines different result metrics dependent on the workload suite. These include latency and throughput. The latter describes the average number of processed records per second and the amount of processed data in bytes per second. Both variants are calculated in total as well as per node. Moreover, three additional metrics are introduced: a durability index (uptime), a throughput penalty factor (assessing throughput change for node failure), and a latency penalty factor (assessing latency change after node failure). To the best of our knowledge, result validation with respect to the query outcome is not supported in this benchmark.

RIoTBench

RIoTBench [SCS17] is a benchmark that focuses on benchmarking distributed DSPSs. It defines 27 micro benchmark scenarios as well as four application benchmark use cases, which represent combined micro benchmarks. These cover Extract, Transform and Load (ETL) processes, statistics generation, model training, and predictive analytics scenarios.

As input data, RIoTBench uses scaled real-world data sets from different Internet of Things (IoT) domains, namely, smart city, smart energy, and health. A tool for ingesting data into the SUT or an application for query result validation is not provided by the benchmark. Next to latency, throughput as well as CPU and memory utilization, RIoTBench measures *jitter* as a metric. It is defined as the difference between the expected and the actual output rate during a certain time interval.

Yahoo! Streaming Benchmark

Another stream processing benchmark that emphasizes its real-world character is presented by *Yahoo!* [CDE+16] and often referred to as *Yahoo! Streaming Benchmark* (YSB). According to Chintapalli et al. [CDE+16], the benchmark

achieves an increased representativeness of real-world scenarios by incorporating Apache Kafka and *Redis*[2], an open-source in-memory key-value store. This design decision concerning the benchmark architecture is supposed to represent real-world data processing pipelines more realistically. The benchmark domain is advertisement analytics.

The benchmark workload consists of a single query, which reads events from Apache Kafka in *JavaScript Object Notation* (JSON) format. These records need to be parsed and irrelevant events have to be filtered out. The input represents advertisement events associated with advertisement campaigns. A windowed count of certain event data per campaign has to be persisted in the Redis key-value store.

While there are example implementations of the query provided[3], only rudimentary tool support exists. The repository describes the tools as *not polished*[3]. Moreover, a single query only reveals limited insights. Although the single application comprises multiple DSPS functionalities, performance analysis becomes difficult due to this architecture. Particularly, the more operators a streaming application contains, the more possible bottlenecks exist, while only one might be responsible for the overall latency limit. While historical data is briefly incorporated, it is stored in Redis, which does not represent a DBMS suitable or representative for persisting structured business data. The repository mentions intentions to expand the benchmark in the future, however, it seems that did not happen as there are no regular and recent development activities visible[3].

Open Stream Processing Benchmark

Van Dongen and Van den Poel [vDdP20] developed a benchmark named *Open Stream Processing Benchmark* (OSPBench), which also incorporates Apache Kafka for storing streaming data. In particular, the benchmark uses traffic sensor measurements captured in the Netherlands, which contain information of the quantity as well as the speed of vehicles at different locations within the country. The benchmark does not incorporate any historical data.

OSPBench defines a single query, but executes only certain parts of this query in different benchmark runs. So in the first run, the executed query only reads the input data. There is no data transformation performed at all. The following runs increase the complexity by adding further operators, e.g., for joining streams.

The benchmark determines latency, throughput, memory utilization, and CPU utilization as result KPIs. Next to the query, the benchmark defines differ-

[2] `https://redis.io`, accessed: 2020-12-31

[3] `https://github.com/yahoo/streaming-benchmarks`, accessed: 2020-12-31

ent workloads, which differ with respect to the data rate characteristics. While OSPBench provides some tool support, it includes no result validator. The experimental evaluation presented by Van Dongen and Van den Poel [vDdP20] includes the analyses of the relatively new client library *Kafka Streams*[4].

Theodolite

Theodolite [HH21] as the most recently developed performance benchmark pursues the objective to analyze the scalability of distributed DSPSs. It defines four queries from the Industrial Internet of Things (IIoT) domain, which get sensor data as input. None of the queries integrates historical business data. Theodolite further defines seven workload dimensions, i.e., dimensions that can be used for scaling the load. Examples for these dimensions are the input message frequency and the window size. Theodolite introduces three result KPIs, one of them being a scalability metric. It is defined as the required number of instances per workload.

Theodolite uses Apache Kafka as the messaging system, which decouples the data ingestion component from the SUT. Moreover, it is designed to work with stream processing provided as a microservice. In their experimental evaluation, the authors of the Theodolite paper [HH21] study Apache Flink as well as Kafka Streams. The query implementations for both DSPSs are published. However, there is no query result validator provided.

Summary

Table 5.1 compares the most prominent benchmarks for DSPSs and ESPBench in a brief overview. This summary reveals several weaknesses and obstacles in related work, which is why we see the need for a new stream processing benchmark. First, historical data was not or only barely taken into account in all major DSPS benchmarks. It was never incorporated with an ERP data-like character. We argue that this is a crucial aspect in most of the enterprise scenarios, since, to achieve the greatest added value, streaming data needs to be combined with business data. Second, the majority of current stream processing benchmarks lack satisfying tool support, e.g., for result validation or data ingestion. The experimental results of this dissertation highlight the importance of verifying the correctness of query results. Moreover, only two of the benchmarks support at least partial automation, which is a basic requirement in this context. This absence of proper tooling complicates the application of these benchmarks as well as retrieving objective and credible results. The tool support is particularly crucial for DSPS benchmarks as one or multiple data

[4] `https://kafka.apache.org/documentation/streams/`, accessed: 2021-03-08

	Linear Road [ACG+04]	StreamBench [LWXH14]	RIoTBench [SCS17]	YSB [CDE+16]	OSPBench [vDdP20]	Theodolite [HH21]	**ESPBench**
Benchmark type	application	micro	mixed	application	application	application	application
Historical data	briefly; one file with historical tolls	-	-	briefly; key-value store with advertisement data	-	-	extensive; business data, based on TPC-C schema
DSPS functionalities	partially covered	partially covered	partially covered	partially covered	partially covered	partially covered	fully covered
Data sender	stub provided	-	-	yes	yes	yes	yes
Result validator	yes	-	-	yes	-	-	yes
Automation	-	-	-	yes	partially	partially	yes
Query implementations published	-	-	yes	yes	yes	yes	yes

Table 5.1: Comparison of data stream processing benchmarks

streams need to be created reliably. This requirement adds complexity to data ingestion, query result validation, and response time calculations. Finally, the query workloads of existing performance benchmarks fail to cover the core DSPS functionalities, which leads to limited meaningfulness of their results.

Besides the selected performance benchmarks, there is further related work. *DSPBench* [BGM+20] presents a set of 13 applications that can be used for benchmarking DSPSs. Next to describing the applications, Bordin et al. [BGM+20] uses three of them to evaluate the performance characteristics of two DSPSs. Li et al. present *SparkBench* [LTW+17], a benchmark specifically designed for the DSPS Apache Spark. Karimov et al. [KRK+18] propose a benchmark framework with workloads inspired by the online gaming industry. They use this framework for benchmarking Apache Spark, Apache Storm, and Apache Flink. The work of Karakaya et al. [KYA17] also presents a performance analysis of these three DSPSs. Another work benchmarking these system has been published by Shahverdi et al. [SAS19]. However, they extend the list of analyzed stream processing solutions by Hazelcast Jet and Kafka Streams. The analysis itself is based on the Yahoo! Streaming Benchmark.

5.2 Performance Impact of Apache Beam

In the following, we present work related to Apache Beam and our conducted performance impact analysis. For supporting Apache Beam, a system has to implement a *runner*. The development of the runner for the DSPS *IBM Streams* is described by Li et al. [LGM+18]. Next to highlighting three implemented optimizations with regard to the IBM Streams runner, performance evaluations between IBM Streams, Apache Flink, and Apache Spark are presented.

Besides abstraction layers, such as Apache Beam, that allow for developing data stream processing applications using a programming language like Java, there is the idea of leveraging or extending SQL to be able to define stream processing programs. *Continuous Query Language* (CQL) [ABW06] is a comprehensive approach for such an SQL extension. CQL is a SQL-based language for defining continuous queries over data streams as well as updatable relations. It is not only a concept, but also integrated into the *STREAM* [ABB+03a] system, a DSPS developed at the Stanford University. Next to describing the CQL implementation in STREAM, the semantics of CQL is outlined and a comparison to other languages is included in the linked CQL paper.

Apache Calcite [BCH+18] is another approach towards CQL's direction, which was developed more recently. It is a framework that comprises different

functionalities, e.g., with respect to query processing, query optimization, and query language support. The latter feature represents the relevant aspect in this context. The Apache Calcite publication by Begoli et al. [BCH+18] describes its architecture as well as its developed SQL extensions for areas such as semi-structured data or geospatial queries. Furthermore, the work depicts the developed extensions for data stream processing queries, which are called *STREAM extensions*. They are inspired by the mentioned CQL and also explained on their website[5]. However, the website also mentions that not all the presented concepts have been implemented yet. A few DSPSs integrate with Apache Calcite, with Apache Apex and Apache Flink being two of them [BCH+18].

Jain et al. [JMS+08] discuss the differences between two SQL-based languages for defining stream processing queries, particularly *Oracle Continuous Query Language* (Oracle CQL)[6] and *StreamBase StreamSQL*[7]. Moreover, an approach for unifying both languages is proposed. However, they highlight that for achieving a complete standard, further challenges need to be tackled, e.g., the extension of the proposed model to additional core functionality such as pattern matching.

Next to such generic approaches, there are stream processing extensions to SQL for dedicated systems. Examples are the *Continuous Computation Language* (CCL) as the extended SQL version for *SAP HANA Streaming Analytics*[8], and *SamzaSQL* [PHPP16] as extended SQL of *Apache Samza* [NPP+17]. While these SQL extensions can be optimized for the corresponding DSPS, which might lead to better performance and less verbose query definitions, users do not have the flexibility a common standard or abstraction layer would provide. So executing a developed query on a different DSPS requires substantial porting efforts as SQL dialects between DSPSs differ significantly.

Apache Hop[9] or the *Hop Orchestration Platform* is an incubating Apache project that allows to define stream processing applications visually, i.e., by leveraging a graphical user interface. According to their website, designed programs are executable on any of the supported engines. For all systems except for the Hop native engine, this support is enabled by using Apache Beam.

Besides these approaches for unifying the definition of stream processing applications, there is also work in the area of benchmarking related to the

[5] `https://calcite.apache.org/docs/stream.html`, accessed: 2020-12-31

[6] `https://docs.oracle.com/cd/E16764_01/doc.1111/e12048/intro.htm`, accessed: 2020-12-31

[7] `https://docs.tibco.com/pub/sb-lv/2.1.8/doc/html/streamsql/ssql-intro.html`, accessed: 2020-12-31

[8] `https://help.sap.com/viewer/f1da0b944b1c4eae8137c9f913b66d44/2.0.04/en-US`, accessed: 2021-03-15

[9] `https://hop.apache.org`, accessed: 2021-03-09

Apache Beam performance impact analysis. *NEXMark*[10] is a benchmark using queries over data streams in the context of an online auction system. However, the benchmark was never finished, as the website still states that it is *work in progress*[10] and there is only a draft version of a paper, which was published a couple of years ago [TTPM02]. Nevertheless, in the context of Apache Beam, this benchmark was used as inspiration and foundation for a NEXMark-based benchmark suite[11]. This suite extends the eight existing NEXMark queries by five additional ones. A complete implementation of all queries for all runners is a work in progress according to the Apache Beam website[11].

The work presented by Marcu et al. [MCAP16] compares Apache Flink and Apache Spark. Their measurements include different queries, a grep query as used in the Apache Beam study of this thesis being one of them. One analyzed focus area is the scaling behavior of the DSPSs regarding the number of cluster nodes. However, both systems are studies regarding their batch processing performance in the performed measurements.

Lopez et al. [LLD16] compare Apache Storm, Apache Flink, and Apache Spark Streaming in their publication. Besides describing the architecture of these three systems, the performance is studied in a network traffic analysis scenario. Additionally, the behavior in case of a node failure is investigated.

Overall, there are several approaches creating a way to define stream processing applications that can be executed on multiple DSPSs, without the need for substantial porting efforts. One way could be the development of a commonly supported SQL standard, similar to the domain of database management systems.

5.3 Apache Kafka Capability Analysis

Francis and Merli [FM16] present Apache Pulsar, a message broker originally developed at *Yahoo!*. It makes use of the distributed storage service Apache BookKeeper[12]. Similar to Apache Kafka, Apache Pulsar employs the concept of topics to which producers can send messages and to which consumers can subscribe. Francis and Merli [FM16] also show a brief performance analysis. The throughput that was achieved in their study using an SSD is 1,800,000 messages/second. However, they do not give details about the test setup, making it hard to assess or reproduce the results.

[10] `http://datalab.cs.pdx.edu/niagara/NEXMark/`, accessed: 2021-03-09
[11] `https://beam.apache.org/documentation/sdks/java/nexmark/`, accessed: 2020-12-31
[12] `https://bookkeeper.apache.org`, accessed: 2020-12-31

Dobbelaere and Esmaili [DE17] compare Apache Kafka with RabbitMQ[13], which is another open-source message broker. In their work, they compare both solutions qualitatively and quantitatively. The impact of different acknowledgment levels is, similar to the studies conducted in Section 4.3, one factor amongst others that are evaluated in their study. However, contrary to our results, their findings do not show a clear difference in the achieved throughput between an acks level of *one* and *zero* in the analyzed setting. The usage of a different version of Apache Kafka is a potential reason for that.

Noac'h, Costan, and Bougé [NCB17] evaluate the performance of Apache Kafka in combination with DSPSs. They also study the influence of Apache Kafka configuration aspects, the producer batch size being one of them. In line with the results presented in Section 4.3, their findings reveal that a higher batch size does not necessarily lead to a higher throughput.

Kreps et al. [KNR11] present a performance analysis of the systems Apache Kafka, RabbitMQ, and Apache ActiveMQ[14]. Comparable to the related work presented before, they analyze the influence of the batch size of the Apache Kafka producer. Next to the producer, they study the Apache Kafka consumer behavior and compare it to RabbitMQ and Apache ActiveMQ. The achieved throughput for Apache Kafka presented by Kreps et al. [KNR11] is in a similar range as the results of this dissertation.

[13] `https://www.rabbitmq.com`, accessed: 2020-12-31
[14] `https://activemq.apache.org`, accessed: 2020-12-31

6

CONCLUSION

This last chapter presents a summary of the work outlined in this dissertation. Moreover, we highlight interesting topics for future work.

6.1 Summary

This dissertation introduced ESPBench, a benchmark for stream processing architectures in an enterprise context, where streaming data is combined with structured business data. In addition to this enterprise context, ESPBench covers all core functionalities of data stream processing systems, making its results relevant for all domains. As part of ESPBench, an example implementation using Apache Beam as well as a functionally comprehensive toolkit are published, which simplifies the use of the benchmark. In addition, the tools provide full automation of the entire benchmark process. The toolkit also allows for an objective result calculation, i.e., ESPBench does not rely on the differently measured performance metrics several systems may provide. We validated ESPBench using our example query implementation in an experimental evaluation. Particularly, we benchmarked three state-of-the-art DSPSs that execute the queries developed with the Apache Beam software development kit (SDK) along with a modern database management system (DBMS). We analyzed different data ingestion rates and their impact on latencies as well as system utilization. The benchmark results reveal that no system outperforms the others for all queries and input rates regarding latencies. The evaluation also shows that, when changing the DSPS which is executing an Apache Beam appli-

cation, one cannot rely on an identical application outcome. Even the error-free executability on another DSPS cannot be taken for granted as another finding discloses.

We further analyzed the performance impact which exists when developing stream processing applications with the Apache Beam SDK, compared to having the same programs developed using the native system SDKs. All benchmark implementations are provided in order to ensure reproducibility. The results of the impact analysis show that Apache Beam has a noticeable influence on the performance of DSPSs in almost all cases. Programs developed using Apache Beam suffered from a slowdown of up to a factor of 58 in the worst case. At the same time, there is one scenario where the query developed using Apache Beam is as fast as its counterparts using the APIs of the corresponding DSPS. However, for most scenarios, we observed a slowdown of at least a factor three.

The results lead to two major conclusions. Firstly, using Apache Beam as an abstraction layer for application development comes at a cost in terms of runtime performance compared to the applications developed using DSPS SDKs. Secondly, the performance penalty varies among DSPSs and stream processing applications. While using Apache Beam certainly provides greater flexibility to switch underlying DSPSs with relatively low effort, one needs to be aware of the fact that this advantage comes with a negative impact on performance.

As part of the Apache Kafka performance study of this dissertation, we proposed a monitoring architecture for applications running on a Java Virtual Machine (JVM) in general. Its design goals include the ease of setup and the usage of state-of-the-art technologies, such as Grafana, Carbon, jmxtrans, and collectd. We performed a comprehensive performance study on Apache Kafka using this suggested performance benchmarking landscape. Moreover, we evaluated and discussed the outcomes for selected Apache Kafka producer configurations in this work. The development and benchmarking artifacts, such as the data sender and the Grafana dashboard, are published to achieve transparency and reproducibility. In the defined setting with a single topic with one partition and a replication factor of one, we observed a maximum steady throughput of about 420,000 messages/second, which corresponds to about 92 MB/second. We identified and quantified the impact of the data producer batch size, acknowledgment level, data sender locality, as well as additional aspects on the input rate performance. A surprising finding is related to the influence of the acknowledgment level. Having acknowledgments enabled resulted in better performance, i.e., a higher message input rate, which was unexpected as it introduces an overhead.

Additionally, we analyzed the timestamps taken by Apache Kafka when a record is appended to its log. We studied how the batching mechanism of Apache Kafka impacts the timestamps regarding possible delays. Knowing that is crucial as a large delay can distort latency results calculated by ESPBench. The study reveals that a batching impact on timestamps is barely noticeable. While there is no influence identifiable for an input rate of 1,000 messages/second, the observed delays for 10,000 messages/second are marginal. Particularly, batches of about eleven records got assigned the same timestamp, while consecutive batches only had a timestamp difference of one millisecond. A doubled default batch size does not have an effect on these results.

To sum up, we presented a benchmark for data stream processing architectures in an enterprise context, i.e., where streaming data also needs to be combined with stored business data. We applied the benchmark on different architectures to prove its functioning and highlight the importance of functionalities, such as the query result validation and latency calculation. Additionally, we studied the performance impact of using the popular Apache Beam SDK for developing stream processing applications compared to making use of DSPS SDKs. To get a better understanding of the capabilities of Apache Kafka, the message broker employed in the proposed benchmark, we analyzed how many incoming records per second Apache Kafka can handle in selected configurations. These contributions presented in this dissertation add value to the area of performance benchmarking in the context of data stream processing.

6.2 Future Work

The studied area of performance benchmarking for data stream processing systems comprises several directions for future work. To broaden the use cases for ESPBench in the future, the validator can be extended to support ignoring the query result order, focusing only on the presence of results. This additional feature will allow extended scaling regarding the Apache Kafka topic partitions. In the current setting, we use topics with a single partition as Apache Kafka only guarantees the correct order of records within one partition. However, there might be use cases where a strict result order is not required which justify having this validator option. By scaling via Apache Kafka topic partitions, the maximum data input rate manageable by the message broker can be increased.

Another area of future work is the extension of the benchmark to calculate additional result KPIs that cover aspects such as the behavior in case of node failures. Besides that, the experimental evaluation can be broadened to further

topics. Possible areas of interest comprise analyses of, e.g., the scalability characteristics of the DSPSs and the impact of altered DBMSs or their configurations on the observed latencies. In this dissertation, the focus of our evaluation is on selected open-source systems. Results of comparisons with commercial systems, further open-source DSPSs, and more DSPS settings using ESPBench will lead to a more complete overview of the DSPS landscape.

Furthermore, ESPBench can be used to study other benchmark domains. For instance, geospatial data might be used or generated to test the performance behavior of relevant queries for this kind of data, e.g., for distance calculations.

Future work in the area of Apache Beam and related performance impact evaluations involves studying the reasons for performance differences in greater detail. Particularly, the stream processing applications and systems can be profiled to see how much time is spent in which part of the execution plans and, thus, to identify possible performance bottlenecks. Finding potential reasons for the comparatively large performance penalties when employing Apache Apex represents another interesting direction of future work. It might be possible to identify factors that influence the performance penalty to create a model which makes the performance predictable. Additionally, measurements can be extended regarding aspects such as the number of studied systems or query complexity as well as scaling, parallelism, or fault-tolerance behaviors. Further upcoming Apache Beam versions and other abstraction layers should be compared against the presented results to supplement the initial overview shown in this work.

The Apache Kafka analysis also revealed potential aspects of future research. Next to the study of selected configuration settings in Apache Kafka, a comparison to similar systems, such as Apache Pulsar, Amazon SQS, or RabbitMQ, are objectives that are valuable to the research community as there is currently a lack of such comparisons. Further work can integrate the analysis of data producers employed in data stream processing frameworks, such as Apache Flink or Apache Beam. These systems often provide their own Apache Kafka producer implementations or interfaces. It is currently unclear if these embedded producers perform differently in comparable settings regarding the achievable input rate. Besides, it is of interest how the achievable input rate behaves when scaling via, e.g., the number of topic partitions. With a growing number of partitions, scaling the number of broker nodes becomes an additional dimension. Its influence on the ingestion performance can be measured. The impact of higher replication factors is another factor that is to be analyzed.

LIST OF FIGURES

LIST OF TABLES

BIBLIOGRAPHY

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[ABB+03a] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[ABB+03b] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. STREAM: The Stanford Stream Data Manager (Demonstration Description). In *ACM International Conference on Management of Data (SIGMOD)*, pages 665–665. ACM, 2003.

[ABC+15] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. In *Proceedings of the VLDB Endowment*, volume 8, pages 1792–1803, 2015.

[Abd15] Mohamed Amine Abdessemed. Real-time Data Integration with Apache Flink & Kafka @Bouygues Telecom. `http://www.slideshare.net/FlinkForward/mohamed-amine-abdessemed-realtime-data-integration-with-apache-flink-kafka`, 2015. Accessed: 2020-12-31.

[ABE+14] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.

[ABW04] Arvind Arasu, Shivnath Babu, and Jennifer Widom. CQL: A Language for Continuous Queries over Streams and Relations. In *International Workshop on Database Programming Languages (DBPL) 2003*, pages 1–19. Springer Berlin Heidelberg, 2004.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[ACÇ+03] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo F. Galvez, David Maier, Anurag Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear Road: A Stream Data Management Benchmark. In *International Conference on Very Large Data Bases (VDLB)*, pages 480–491, 2004.

[ACL18] Tyler Akidau, Slava Chernyak, and Reuven Lax. *Streaming Systems: THE WHAT, WHERE, WHEN, AND HOW OF LARGE-SCALE DATA PROCESSING*. O'Reilly Media, Inc., 2018.

[ADKM21] Mayank Agrawal, Sumit Dutta, Richard Kelly, and Ingrid Millán. COVID-19: An inflection point for Industry 4.0. `https://www.mckinsey.com/business-functions/operations/our-`

insights/covid-19-an-inflection-point-for-industry-40, 2021. Accessed: 2021-03-18.

[AXL+15] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1383–1394. ACM, 2015.

[BCH+18] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *ACM International Conference on Management of Data (SIGMOD)*, pages 221–230, 2018.

[BDD+10] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura M. Haas, Renée J. Miller, and Nesime Tatbul. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. In *Proceedings of the VLDB Endowment*, volume 3, pages 232–243, 2010.

[BGM+20] Maycon Viana Bordin, Dalvan Griebler, Gabriele Mencagli, Cláudio F. R. Geyer, and Luiz Gustavo Leão Fernandes. DSPBench: A Suite of Benchmark Applications for Distributed Data Stream Processing Systems. *IEEE Access*, 8:222900–222917, 2020.

[Bha16] Milind Bhandarkar. AdBench: A Complete Benchmark for Modern Data Pipelines. In *TPC Technology Conference (TPCTC)*, pages 107–120, 2016.

[BKD+14] David Bermbach, Jörn Kuhlenkamp, Akon Dey, Sherif Sakr, and Raghunath Nambiar. Towards an Extensible Middleware for Database Benchmarking. In *TPC Technology Conference (TPCTC)*, volume 8904, pages 82–96. Springer, 2014.

[Bre15] Eric A. Brewer. Kubernetes and the Path to Cloud Native. In *ACM Symposium on Cloud Computing (SoCC)*, page 167, 2015.

[BW01] Shivnath Babu and Jennifer Widom. Continuous Queries over Data Streams. *SIGMOD Rec.*, 30(3):109–120, 2001.

[CDE+16] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum,

Kishorkumar Patil, Boyang Peng, and Paul Poulosky. Benchmarking Streaming Computation Engines: Storm, Flink and Spark Streaming. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshops*, pages 1789–1792, 2016.

[CGH+17] Paris Carbone, Gábor E. Gévay, Gábor Hermann, Asterios Katsifodimos, Juan Soto, Volker Markl, and Seif Haridi. Large-Scale Data Stream Processing Systems. In *Handbook of Big Data Technologies*, pages 219–260. 2017.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[CMN18] Michael Chui, Brett May, and Subu Narayanan. What it takes to get an edge in the Internet of Things. `https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/what-it-takes-to-get-an-edge-in-the-internet-of-things`, September 2018. Accessed: 2020-05-08.

[Cou90] Transaction Processing Performance Council. TPC benchmark C standard specification, 1990.

[DBE15] Minh Dao-Tran, Harald Beck, and Thomas Eiter. Towards Comparing RDF Stream Processing Semantics. In *Workshop on High-Level Declarative Stream Processing*, volume 1447 of *CEUR Workshop Proceedings*, pages 15–27, 2015.

[DD13] Thomas H. Davenport and Jill Dyché. Big Data in Big Companies. `http://docs.media.bitpipe.com/io_10x/io_102267/item_725049/Big-Data-in-Big-Companies.pdf`, May 2013. Accessed: 2020-12-31.

[DE17] Philippe Dobbelaere and Kyumars Sheykh Esmaili. Industry Paper: Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations. In *ACM International Conference on Distributed and Event-based Systems (DEBS)*, pages 227–238, 2017.

[DF16] T. Dunning and E. Friedman. *Streaming Architecture: New Designs Using Apache Kafka and MapR Streams*. O'Reilly Media, 2016.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Symposium on Operating System Design and Implementation (OSDI)*, pages 137–150, 2004.

[DGKL11] J.N. Druckman, D.P. Greene, J.H. Kuklinski, and A. Lupia. *Cambridge Handbook of Experimental Political Science*. Cambridge University Press, 2011.

[DTC+19] Manh Nguyen Duc, Anh Lê Tuán, Jean-Paul Calbimonte, Manfred Hauswirth, and Danh Le Phuoc. Autonomous RDF Stream Processing for IoT Edge Devices. In Xin Wang, Francesca Alessandra Lisi, Guohui Xiao, and Elena Botoeva, editors, *Semantic Technology - Joint International Conference, JIST*, volume 12032, pages 304–319. Springer, 2019.

[DtH01] Marlon Dumas and Arthur H. M. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *<<UML>> 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 76–90, 2001.

[ES93] Lisa M. Ellram and Sue Perrott Siferd. Purchasing: The Cornerstone of the Total Cost of Ownership Concept. *Journal of Business Logistics*, 14(1):163–184, 1993.

[FAS+12] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. Benchmarking in the Cloud: What It Should, Can, and Cannot Be. In *TPC Technology Conference (TPCTC)*, pages 173–188, 2012.

[FCP+11] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. SAP HANA database: data management for modern business applications. In *ACM International Conference on Management of Data (SIGMOD)*, volume 40, pages 45–51, 2011.

[FM16] Joe Francis and Matteo Merli. Open-sourcing Pulsar, Pub-sub Messaging at Scale. `https://yahooeng.tumblr.com/post/150078336821/open-sourcing-pulsar-pub-sub-messaging-at-scale`, 2016. Accessed: 2020-12-31.

[GKP+10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A Main

Memory Hybrid Storage Engine. In *Proceedings of the VLDB Endowment*, volume 4, pages 105–116, 2010.

[Gra93] Jim Gray. *The Benchmark Handbook - For Database and Transaction Processing Systems*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 1993.

[Gre17] Brendan Gregg. Linux Load Averages: Solving the Mystery. `http://www.brendangregg.com/blog/2017-08-08/linux-load-averages.html`, 2017. Accessed: 2020-12-31.

[HH21] Sören Henning and Wilhelm Hasselbring. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines in Microservice Architectures. *Big Data Research*, 25:2214–5796, 2021.

[HHD+10] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In *Workshop Proceedings of the International Conference on Data Engineering (ICDE)*, pages 41–51, 2010.

[Hit02] Steve Hitchman. The Details of Conceptual Modelling Notations are Important - A Comparison of Relationship Normative Language. *Communications of the Association for Information Systems*, 9:10, 2002.

[HKZ+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[HL15] Guenter Hesse and Martin Lorenz. Conceptual Survey on Data Stream Processing Systems. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 797–802, 2015.

[HM20] Guenter Hesse and Christoph Matthies. guenter-hesse/ESPBench: Initial ESPBench Release. `https://doi.org/10.5281/zenodo.4322553`, December 2020.

[HMG+19] Guenter Hesse, Christoph Matthies, Kelvin Glass, Johannes Huegle, and Matthias Uflacker. Quantitative Impact Evaluation of an Abstraction Layer for Data Stream Processing Systems. In

IEEE International Conference on Distributed Computing Systems (ICDCS), pages 1381–1392, 2019.

[HMP+21] Guenter Hesse, Christoph Matthies, Michael Perscheid, Matthias Uflacker, and Hasso Plattner. ESPBench: The Enterprise Stream Processing Benchmark. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 201–212, 2021.

[HMRU17] Guenter Hesse, Christoph Matthies, Benjamin Reissaus, and Matthias Uflacker. A New Application Benchmark for Data Stream Processing Architectures in an Enterprise Context: Doctoral Symposium. In *ACM International Conference on Distributed and Event-based Systems (DEBS)*, pages 359–362, 2017.

[HMSU19] Guenter Hesse, Christoph Matthies, Werner Sinzig, and Matthias Uflacker. Adding Value by Combining Business and Sensor Data: An Industry 4.0 Use Case. In *International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 528–532, 2019.

[HMU20] Guenter Hesse, Christoph Matthies, and Matthias Uflacker. How Fast Can We Insert? An Empirical Performance Evaluation of Apache Kafka. In *IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 641–648, 2020.

[HRM+17] Guenter Hesse, Benjamin Reissaus, Christoph Matthies, Martin Lorenz, Milena Kraus, and Matthias Uflacker. Senska – Towards an Enterprise Streaming Benchmark. In *TPC Technology Conference (TPCTC)*, pages 25–40, 2017.

[HSK+19] Razin Farhan Hussain, Mohsen Amini Salehi, Anna Kovalenko, Yin Feng, and Omid Semiari. Federated Edge Computing for Disaster Management in Remote Smart Oil Fields. In *IEEE International Conference on High Performance Computing and Communications (HPCC); IEEE International Conference on Smart City; IEEE International Conference on Data Science and Systems (DSS)*, pages 929–936, 2019.

[HSMU19] Guenter Hesse, Werner Sinzig, Christoph Matthies, and Matthias Uflacker. Application of Data Stream Processing Technologies in Industry 4.0: What is Missing? In *International Conference on*

Data Science, Technology and Applications (DATA), pages 304–310, 2019.

[Hup09] Karl Huppler. The Art of Building a Good Benchmark. In *TPC Technology Conference (TPCTC)*, pages 18–30, 2009.

[HVN16] Marco F. Huber, Martin Voigt, and Axel-Cyrille Ngonga Ngomo. Big data architecture for the semantic analysis of complex events in manufacturing. In *46. Jahrestagung der Gesellschaft für Informatik*, pages 353–360, 2016.

[IAM+19] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana H. Zulkernine, and Shahzad Khan. A Survey of Distributed Data Stream Processing Frameworks. *IEEE Access*, 7:154300–154316, 2019.

[Int14] Intel. A GUIDE TO THE INTERNET OF THINGS. `https://www.intel.de/content/www/de/de/internet-of-things/infographics/guide-to-iot-new.html`, 2014. Accessed: 2020-11-28.

[Jan13] J.H.M. Janssens. *Outlier Selection and One-Class Classification*. PhD thesis, 2013. Series: TiCC Ph.D. Series Volume: 27.

[JHF+12] Zbigniew Jerzak, Thomas Heinze, Matthias Fehr, Daniel Gröber, Raik Hartung, and Nenad Stojanovic. The DEBS 2012 Grand Challenge. In *ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 393–398, 2012.

[JMS+08] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stanley B. Zdonik. Towards a Streaming SQL Standard. In *Proceedings of the VLDB Endowment*, volume 1, pages 1379–1390, 2008.

[Joh15] M. Johns. *Getting Started with Hazelcast - Second Edition*. Packt Publishing, 2015.

[Jos65] E. O. Joslin. Application Benchmarks: The Key to Meaningful Computer Evaluations. In *Proceedings of the National Conference*, pages 27–37, 1965.

[JS19] Bartosz Janota and Robert Stephenson. Spotify's Event Delivery – Life in the Cloud. `https://engineering.atspotify.com/2019/`

11/12/spotifys-event-delivery-life-in-the-cloud/, 2019. Accessed: 2020-11-28.

[KGT05] Andreas Knöpfel, Bernhard Gröne, and Peter Tabeling. Fundamental Modeling Concepts. *Effective Communication of IT Systems*, page 51, 2005.

[KKG+11] Jens Krüger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast Updates on Read-Optimized Databases Using Multi-Core CPUs. In *Proceedings of the VLDB Endowment*, volume 5, pages 61–72, 2011.

[KKLLC15] Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, and Yves Caniou. Parallel and Distributed Stream Processing: Systems Classification and Specific Issues. 2015.

[KNR11] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a Distributed Messaging System for Log Processing. In *International Workshop on Networking Meets Databases (NetDB)*, pages 1–7, 2011.

[Kos16] Joel Koshy. Kafka Ecosystem at LinkedIn. `https://engineering.linkedin.com/blog/2016/04/kafka-ecosystem-at-linkedin`, 2016. Accessed: 2020-12-31.

[KRK+18] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking Distributed Stream Data Processing Systems. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1507–1518, 2018.

[KW19] Rakesh Kumar and Thomas Weise. The magic behind your Lyft ride prices - A case study on machine learning and streaming. `https://conferences.oreilly.com/strata/strata-ca-2019/cdn.oreillystatic.com/en/assets/1/event/290/The%20magic%20behind%20your%20Lyft%20ride%20prices_%20A%20case%20study%20on%20machine%20learning%20and%20streaming%20Presentation.pdf`, 2019. Accessed: 2020-12-28.

[KYA17] Ziya Karakaya, Ali Yazici, and Mohammed Alayyoub. A Comparison of Stream Processing Frameworks. In *International Conference on Computer and Applications (ICCA)*, pages 1–12, 2017.

[LC85] Byron C. Lewis and Albert E. Crews. The Evolution of Benchmarking as a Computer Performance Evaluation Technique. volume 9, pages 7–16. Management Information Systems Research Center, University of Minnesota, 1985.

[LD93] Scott T. Leutenegger and Daniel M. Dias. A Modeling Study of the TPC-C Benchmark. In *ACM International Conference on Management of Data (SIGMOD)*, pages 22–31, 1993.

[LGM+18] Shen Li, Paul Gerver, John Macmillan, Daniel Debrunner, William Marshall, and Kun-Lung Wu. Challenges and Experiences in Building an Efficient Apache Beam Runner For IBM Streams. In *Proceedings of the VLDB Endowment*, volume 11, pages 1742–1754, 2018.

[Li17] Neville Li. Big Data Processing at Spotify: The Road to Scio (Part 1). `https://labs.spotify.com/2017/10/16/big-data-processing-at-spotify-the-road-to-scio-part-1/`, 2017. Accessed: 2020-12-28.

[LKT18] Thomas Lindemann, Jonas Kauke, and Jens Teubner. Efficient Stream Processing of Scientific Data. In *IEEE International Conference on Data Engineering (ICDE) Workshops*, pages 140–145, 2018.

[LLD16] Martin Andreoni Lopez, Antonio Gonzalez Pastana Lobato, and Otto Carlos M. B. Duarte. A Performance Comparison of Open-Source Stream Processing Platforms. In *IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2016.

[Ltd19] Autonomous Manufacturing Ltd. Industry 4.0: 7 Real-World Examples of Digital Manufacturing in Action. `https://amfg.ai/2019/03/28/industry-4-0-7-real-world-examples-of-digital-manufacturing-in-action/`, 2019. Accessed: 2020-05-12.

[LTW+17] Min Li, Jian Tan, Yandong Wang, Li Zhang, and Valentina Salapura. SparkBench: a spark benchmarking suite characterizing large-scale in-memory data analytics. *Cluster Computing*, 20(3):2575–2589, 2017.

[Luc15] Greg Luck. Hazelcast's Business Model, Open Source, Open Standards & Community. https://hazelcast.com/blog/business-model-open-source-community/, 2015. Accessed: 2020-02-25.

[LWI+14] Xiaoyi Lu, Md. Wasi-ur-Rahman, Nusrat S. Islam, Dipti Shankar, and Dhabaleswar K. Panda. Accelerating Spark with RDMA for Big Data Processing: Early Experiences. In *IEEE Annual Symposium on High-Performance Interconnects (HOTI)*, pages 9–16, 2014.

[LWXH14] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. StreamBench: Towards Benchmarking Modern Distributed Stream Computing Frameworks. In *IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pages 69–78, 2014.

[LZZ+18] Yu Liu, Hantian Zhang, Luyuan Zeng, Wentao Wu, and Ce Zhang. MLBench: Benchmarking Machine Learning Services Against Human Experts. In *Proceedings of the VLDB Endowment*, volume 11, pages 1220–1232, 2018.

[MBM09] Marcelo R. N. Mendes, Pedro Bizarro, and Paulo Marques. A Performance Study of Event Processing Systems. In *TPC Technology Conference (TPCTC)*, pages 221–236, 2009.

[MCAP16] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, and María S. Pérez-Hernández. Spark versus Flink: Understanding Performance in Big Data Analytics Frameworks. In *IEEE International Conference on Cluster Computing (CLUSTER)*, pages 433–442, 2016.

[MCB+15] James Manyika, Michael Chui, Peter Bisson, Jonathan Woetzel, Richard Dobbs, Jacques Bughin, and Dan Aharon. THE INTERNET OF THINGS: MAPPING THE VALUE BEYOND THE HYPE. http://www.mckinsey.com/~/media/McKinsey/Business%20Functions/McKinsey%20Digital/Our%20Insights/The%20Internet%20of%20Things%20The%20value%20of%20digitizing%20the%20physical%20world/The-Internet-of-things-Mapping-the-value-beyond-the-hype.ashx, June 2015. Accessed: 2020-12-31.

[McK15] McKinsey & Company. Industry 4.0 - How to navigate digitization of the manufacturing sector.

http://www.forschungsnetzwerk.at/downloadpub/mck_industry_40_report.pdf, 2015. Accessed: 2020-12-31.

[McK16] McKinsey & Company. Industry 4.0 after the initial hype - Where manufacturers are finding value and how they can best capture it. https://www.mckinsey.com/~/media/mckinsey/business%20functions/mckinsey%20digital/our%20insights/getting%20the%20most%20out%20of%20industry%204%200/mckinsey_industry_40_2016.ashx, 2016. Accessed: 2020-12-31.

[Men14] Marcelo Rodrigues Nunes Mendes. *Performance Evaluation and Benchmarking of Event Processing Systems*. PhD thesis, 2014.

[MWBA10] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the Graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[NCB17] Paul Le Noac'h, Alexandru Costan, and Luc Bougé. A Performance Evaluation of Apache Kafka in Support of Big Data Streaming Applications. In *IEEE International Conference on Big Data*, pages 4803–4806, 2017.

[NLW+09] Raghunath Othayoth Nambiar, Matthew Lanken, Nicholas Wakou, Forrest Carman, and Michael Majdalany. Transaction Processing Performance Council (TPC): Twenty Years Later - A Look Back, a Look Ahead. In *TPC Technology Conference (TPCTC)*, pages 1–10, 2009.

[NOE+18] Judit Nagy, Judit Oláh, Edina Erdei, Domician Máté, and József Popp. The Role and Impact of Industry 4.0 and the Internet of Things on the Business Strategy of the Value Chain—The Case of Hungary. *Sustainability*, 10(10):3491, 2018.

[NPP+17] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful Scalable Stream Processing at LinkedIn. In *Proceedings of the VLDB Endowment*, volume 10, pages 1634–1645, 2017.

[NWZ+19] Thanh Tam Nguyen, Matthias Weidlich, Bolong Zheng, Hongzhi Yin, Quoc Viet Hung Nguyen, and Bela Stantic. From Anomaly Detection to Rumour Detection using Data Streams of Social Platforms. In *Proceedings of the VLDB Endowment*, volume 12, pages 1016–1029, 2019.

[PHPP16] Milinda Pathirage, Julian Hyde, Yi Pan, and Beth Plale. SamzaSQL: Scalable Fast Data Management with Streaming SQL. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS) Workshops*, pages 1627–1636, 2016.

[Pla09] Hasso Plattner. A Common Database Approach for OLTP and OLAP Using an In-Memory Column Database. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1–2, 2009.

[Rab17] Tilmann Rabl. Big Data Stream Processing. `http://www.bbdc.berlin/fileadmin/news/photos/BD_SummerSchool-17-18/StreamProcessing-TilmannRabl.pdf`, 2017. Accessed: 2020-12-31.

[RFD+15] Tilmann Rabl, Michael Frank, Manuel Danisch, Hans-Arno Jacobsen, and Bhaskar Gowda. The Vision of BigBench 2.0. In *Workshop on Data analytics in the Cloud*, pages 1–4, 2015.

[RS94] Arnon Rosenthal and Leonard J. Seligman. Data Integration in the Large: The Challenge of Reuse. In *Proceedings of the VLDB Endowment*, pages 669–675, 1994.

[RT18] Paul Ralph and Ewan D. Tempero. Construct Validity in Software Engineering Research and Software Metrics. In Austen Rainer, Stephen G. MacDonell, and Jacky W. Keung, editors, *International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 13–23. ACM, 2018.

[Rus19] John Russell. Alibaba acquires German big data startup Data Artisans for $103M. `https://techcrunch.com/2019/01/08/alibaba-data-artisans/`, 2019. Accessed: 2020-02-21.

[SAS19] Elkhan Shahverdi, Ahmed Awad, and Sherif Sakr. Big Stream Processing Systems: An Experimental Evaluation. In *IEEE International Conference on Data Engineering (ICDE) Workshops*, pages 53–60, 2019.

[SCS17] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. RIoTBench: An IoT benchmark for distributed stream processing systems. *Concurrency and Computation: Practice and Experience*, 29(21):1–22, 2017.

[SÇZ05] Michael Stonebraker, Ugur Çetintemel, and Stanley B. Zdonik. The 8 Requirements of Real-Time Stream Processing. *SIGMOD Record*, 34(4):42–47, 2005.

[SDJ01] Marion K Slack and Jolaine R Draugalis Jr. Establishing the internal and external validity of experimental studies. *American Journal of Health-System Pharmacy*, 58(22):2173–2181, 2001.

[Ser19] Sergei Sokolenko. Spotify: 500 billion events per day as of nov 2019. `https://twitter.com/datancoffee/status/1197505299168604162`, 2019. Accessed: 2020-11-07.

[Smi10] Ryan Smith. Intel Settles With the FTC. `https://www.anandtech.com/show/3839/intel-settles-with-the-ftc`, 2010. Accessed: 2020-02-27.

[Sou17] Prashanth Harish Southekal. *Data for Business Performance: The Goal-Question-Metric (GQM) Model to Transform Business Data into an Enterprise Asset*. Technics Publications, 2017.

[SR86] Michael Stonebraker and Lawrence A. Rowe. THE DESIGN OF POSTGRES. In *ACM International Conference on Management of Data (SIGMOD)*, pages 340–355, 1986.

[SS15] Abdul Ghaffar Shoro and Tariq Rahim Soomro. Big data analysis: Apache spark perspective. *Global Journal of Computer Science and Technology*, 15(1), 2015.

[SSS+15] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun C. Murthy, and Carlo Curino. Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1357–1369, 2015.

[TTPM02] Pete Tucker, Kristin Tufte, Vassilis Papadimos, and David Maier. NEXMark – A Benchmark for Queries over Data Streams DRAFT. `http://datalab.cs.pdx.edu/niagara/pstream/nexmark.pdf`, 2002. Accessed: 2020-12-31.

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh

Mittal, and Dmitriy V. Ryaboy. Storm @Twitter. In *ACM International Conference on Management of Data (SIGMOD)*, pages 147–156, 2014.

[Tzo19] Kostas Tzoumas. ververica - Introducing our new name! `https://www.ververica.com/blog/introducing-our-new-name`, 2019. Accessed: 2020-02-21.

[Ung16] Gordon Mah Ung. AMD accuses BAPCo and Intel of cheating with Sysmark benchmarks. `https://www.pcworld.com/article/3023373/amd-accuses-bapco-and-intel-of-cheating-with-sysmark-benchmarks.html`, 2016. Accessed: 2020-02-27.

[vDdP20] Giselle van Dongen and Dirk Van den Poel. Evaluation of Stream Processing Frameworks. *IEEE Transactions on Parallel and Distributed Systems*, 31(8):1845–1858, 2020.

[vKAH+15] Jóakim von Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. How to Build a Benchmark. In *ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 333–336, 2015.

[VL16] Mihail Vieru and Javier López. Flink in Zalando's World of Microservices. `http://www.slideshare.net/ZalandoTech/flink-in-zalandos-world-of-microservices-62376341`, 2016. Accessed: 2020-12-31.

[VMD+13] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *ACM Symposium on Cloud Computing (SOCC)*, pages 5:1–5:16, 2013.

[VMSK12] Marco Vieira, Henrique Madeira, Kai Sachs, and Samuel Kounev. Resilience benchmarking. In Katinka Wolter, Alberto Avritzer, Marco Vieira, and Aad P. A. van Moorsel, editors, *Resilience Assessment and Evaluation of Computing Systems*, pages 283–301. Springer, 2012.

[VW14] Pamela Vagata and Kevin Wilfong. Scaling the Facebook data warehouse to 300 PB. `https://code.facebook.com/posts/`

229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/, April 2014. Accessed: 2020-12-31.

[WL14] Steven Weiner and David Line. Manufacturing and the data conundrum – Too much? Too little? Or just right? *The Economist Intelligence Unit*, 2014.

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, 2012.

[ZDL+13] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 423–438, 2013.

[ZL19] Jinfeng Zhuang and Yu Liu. PinText: A Multitask Text Embedding System in Pinterest. In *ACM International Conference on Knowledge Discovery & Data Mining (SIGKDD)*, pages 2653–2661, 2019.

[ZWL+17] Mingming Zhang, Tianyu Wo, Xuelian Lin, Tao Xie, and Yaxiao Liu. CarStream: An Industrial System of Big Data Processing for Internet-of-Vehicles. In *Proceedings of the VLDB Endowment*, volume 10, pages 1766–1777, 2017.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A Unified Engine for Big Data Processing. *Communications of the ACM*, 59(11):56–65, 2016.

EIGENSTÄNDIGKEITSERKLÄRUNG (DECLARATION OF AUTHORSHIP)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die anderen Werken im Wortlaut oder dem Sinn nach entnommen sind, sind durch Angaben und Quellen kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen. Ich erkläre weiterhin, dass die vorliegende Arbeit bisher an keiner anderen Hochschule oder Prüfungsbehörde eingereicht worden ist, weder in dieser noch in ähnlicher Form.

Potsdam, November 2021

Guenter Hesse