ABSTRACT
Growing data volumes and velocities in fields such as Industry 4.0 or the Internet of Things have led to the increased popularity of data stream processing systems. Enterprises can leverage these developments by enriching their core business data and analyses with up-to-date streaming data. Comparing streaming architectures for these complex use cases is challenging, as existing benchmarks do not cover them. ESPBench is a new enterprise stream processing benchmark that fills this gap. We present its architecture, the benchmarking process, and the query workload. We employ ESPBench on three state-of-the-art stream processing systems, Apache Spark, Apache Flink, and Hazelcast Jet, using provided query implementations developed with Apache Beam. Our results highlight the need for the provided ESPBench toolkit that supports benchmark execution, as it enables query result validation and objective latency measures.

1 INTRODUCTION
The need to process growing data volumes has led to the increased importance of data stream processing approaches. While in 2016, the music streaming service Spotify handled 1.5M events/second, this number had dramatically increased to 8M events/second in 2018 [49]. Even businesses less involved with the digital world face high amounts of data, e.g., those from the manufacturing domain. For example, an ultrasonic sensor production plant creates about 170 GB of data per day [47]. Manufacturing equipment such as a single saw can generate 50,000 messages or 1.2 GB of data on a daily basis [16]. Injection molding machines even produce up to multiple terabytes of sensor data in 24 hours [35]. These developments highlight the large data volumes companies are facing today as well as the high rate of growth.

Various new data stream processing systems (DSPSs), which are leveraged for analyzing continuously generated data, have been developed recently [27]. This increased choice has led to more options for DSPS users, which raises the question of how to identify the DSPS that best satisfies current and future demands. Performance benchmarks offer solutions for this challenge as they reveal performance differences between systems or configurations. Currently, there is no satisfying benchmark for DSPSs that comprises:

- Integration of existing, traditional business data, such as production orders or customer information [34]
- Satisfying tool support (data ingestion, query result validation, objective performance result calculation, automation)
- Coverage of the core DSPS functionalities, e.g., windowing and transformation capabilities.

We close this gap with a benchmark for enterprise stream processing architectures. The contributions are as follows:

- We propose ESPBench, an enterprise stream processing benchmark. It includes a toolkit for data ingestion, query result validation, benchmark result calculation, and automation.
- We present an example implementation of the ESPBench queries using Apache Beam [1], an abstraction layer for defining data processing applications, which can be executed on any of the currently more than ten supported DSPSs [2].
- We conduct an experimental evaluation, benchmarking three state-of-the-art stream processing systems with the Apache Beam example implementation with the objective to validate the concepts and tools of ESPBench.
Table 1: Jim Gray’s [25] design criteria for domain-specific database benchmarks as applied to data stream processing systems

| Criteria     | Description                                                                 | Applied to DSPS benchmarks                                                                 |
|--------------|------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|
| Relevance    | Typical operations of the problem domain need to be tested                   | • Queries cover core functionalities of DSPSs                                               |
|              |                                                                              | • Queries validated with industry                                                             |
|              |                                                                              | • Design represents real-world settings                                                       |
|              |                                                                              | • Data rates configurable                                                                    |
| Portability  | Easy to use on different systems or architectures                           | • No restrictions on SUT architecture specifics by toolkit                                   |
| Scalability  | Applicable to both, single node systems and scale-out systems with multiple nodes; benchmark should be scalable to larger systems | • System-independent query definitions                                                       |
| Simplicity   | Easy to understand and easy to use/implement to ensure result credibility      | • Data rates configurable/scalable                                                            |
|              |                                                                              | • Scale-out scenarios supported by benchmark tools                                             |
|              |                                                                              | • Automation of entire benchmark process                                                     |
|              |                                                                              | • Publication of usable example query implementation                                         |
|              |                                                                              | • Tool support for essential benchmark functions, e.g., query result validation and data stream generation |

To allow result reproduction and the use of ESPBench, we published all artifacts [28] and created a public repository1. The remainder of this paper is structured as follows: Section 2 presents ESPBench. We describe the benchmark scenario and its architecture, the benchmark process, the input data, and the queries. Section 3 introduces the validation setup for ESPBench, illustrating the systems under test and the benchmarking landscape. Subsequently, we discuss the benchmark results in the experimental evaluation section, followed by an outline of threats to validity. Sections 6 and 7 elaborate on the lessons learned and highlight related work. The last section concludes and gives an outlook on future work.

2 THE ESPBENCH BENCHMARK

This section introduces the developed ESPBench, a performance benchmark with comprehensive tool support; covering all core functionalities of DSPSs, including the combination of streaming data with structured business data. It builds upon the ideas presented in [30] and [33]. We start by presenting the overall design objectives associated with ESPBench. Afterward, we give an overview of the benchmark scenario, its architecture as well as workflow, and present the employed input data and queries.

2.1 Design Objectives

One of the most influential works on design principles for benchmarking is The Benchmark Handbook for Database and Transaction Processing Systems by Jim Gray [25]. He defines four criteria a domain-specific benchmark has to meet to be considered useful. Although the work is from the early 90s, it shows major overlaps to newer publications of its kind, such as the work by Karl Huppler [36] or v. Kistowski et al. [54]. The four criteria defined by Gray are described in Table 1. We translated these rather general aspects to the requirements of DSPS benchmarks. These applied design principles build the foundation for ESPBench, i.e., they are taken into account for all design decisions and satisfied by ESPBench.

2.2 Benchmark Scenario

The benchmark scenario sketches the management of a production plant. It is inspired by the 2012 Grand Challenge published at the Distributed and Event-Based Systems (DEBS) conference [38]. This challenge is about a company that uses high-tech manufacturing machines equipped with sensors. The company aims to improve the monitoring and analytical capabilities of its manufacturing processes by making use of the continuously captured sensor data, e.g., by leveraging stream processing technology. Processing this data enables, e.g., faster reactions to unintended system states and the combination of sensor and structured business data, i.e., vertical integration in the context of Industry 4.0. The integration of these traditionally separate data sources reveals additional product insights and allows for a holistic view of the production process [34].

The employed sensors monitor multiple machine variables, e.g., the power consumption or the state of an additive release valve. Multiple sensors are installed on a single manufacturing machine and their data are collected using an embedded PC. The benchmark scenario includes two such machines sending sensor values of the same structure.

2.3 Benchmark Architecture

Figure 2 shows the architecture of ESPBench. It comprises four major components: input data, the toolkit, the message broker, and the system under test (SUT). The components labeled as toolkit are provided by the benchmark.

The input data is stored in comma-separated values (CSV) format. It comprises business data as well as sensor data stored in three different files. The business data and the production times streaming data are generated by the provided data generator as part of the benchmark process. The other two files can be obtained online, as outlined in the description of ESPBench [28]. Details on the data characteristics and scalability are presented in Section 2.5.

The data sender is part of the toolkit and responsible for two tasks: the import of the business data into the database management system (DBMS) that is part of the SUT as well as the ingestion of streaming data into the message broker. ESPBench defines a configuration file used to configure several parameters, such as message broker information or the data input rate.

The message broker of ESPBench, Apache Kafka [42], is responsible for storing data and represents the interface to the SUT. ESPBench incorporates a message broker to separate the SUT and the data sender. This separation allows for realistic and objective performance measurements outside of the SUT, which is important.

---

1 https://github.com/guenter-hesse/ESPBench
A change in the DBMS only requires minor adaptions to the toolkit, e.g., to the data sender logic for importing business data into the known and widely used DBMS, is the default database of ESPBench.

In current environments, show that these manageable rates are high enough to represent manufacturing companies with revenues of at least a billion EUR. Existing studies on this topic give orientation regarding achievable input rates that Apache Kafka can manage needs to be configured. Experiments in [40] is a change to the partitioning during benchmark runs. We address this point by employing Apache Kafka, which does not perform automated re-partitioning. The combination of Apache Kafka and a DSPS is comparable to other architectures, both within the domains of performance benchmarking as well as data processing [32, 55]. Thus, this architecture is relevant as it represents real-world environments.

It is crucial to ensure that the message broker does not become a bottleneck since the objective of a benchmark is to analyze the SUT and not any of the tooling components. Therefore, an input rate that Apache Kafka can manage needs to be configured. Existing studies on this topic give orientation regarding achievable data input rates for specific settings [32]. Our studies of multiple manufacturing companies with revenues of at least a billion EUR show that these manageable rates are high enough to represent current environments.

The SUT comprises a DSPS and a DBMS in its default setting. It is responsible for answering the defined benchmark queries. The benchmark does not impose any scalability restrictions on the SUT, e.g., regarding the cluster size of the DSPS. PostgreSQL [52], a well-known and widely used DBMS, is the default database of ESPBench. A change in the DBMS only requires minor adaptions to the toolkit, e.g., to the data sender logic for importing business data into the DBMS. Alternatively to the SUT default setup, a single system that is able to store the business data and to answer the benchmark queries can be used, reducing communication and data transfer overhead. While ESPBench provides this flexibility, such a scenario fails to represent most of the current enterprise IT landscapes.

The SUT reads input data from Apache Kafka and writes results back to either Apache Kafka or the DBMS, depending on the query. The validator and result calculator determines the correctness of the query answers and calculates the benchmark results. The tool determines aggregated results, such as the mean latency and percentiles, as well as single latencies for each output record. It writes the output to log and CSV files, which can be used for further analyses, like plotting single latencies. Technically, the tool uses Akka [11], a toolkit for developing distributed applications, also in a streaming fashion. The validator reads the input data, calculates the query results, and compares them to the SUT’s output. Furthermore, it computes result latencies, i.e., timestamp differences. The validator does that by leveraging the Apache Kafka or DBMS timestamps, i.e., the times taken when an input or result record is written to the log of Apache Kafka or the DBMS. This concept allows for the previously mentioned objective performance measurements outside of DSPSs.

### 2.4 Benchmark Process

The activity diagram in Figure 1 shows the process of ESPBench and sums up the Ansible [13] script, which automates the benchmark steps. The process steps are exchangeable and can be extended by the user, e.g., to incorporate additional monitoring tools.

In step (1), the benchmark parameters are set and the project is compiled to a fat Java Archive (JAR), which contains all dependencies. The compilation uses sbt-assembly [15], an assembly plugin for the open-source build tool sbt [14]. The setting of parameters is only relevant in scenarios where multiple runs, i.e., multiple rounds of the benchmarking process, are executed and automated. In this case, the main Ansible benchmarking script is invoked multiple times with the defined parameters for the different runs by an overarching Ansible script. For instance, distinct runs want to use different Apache Kafka topics to be able to distinguish between runs. If there is only a single run, the parameters are read from the configuration files shown in Figure 2, which provide default values.

In step (2), the required Apache Kafka topics are created according to the configuration and the naming schema defined by ESPBench. Topics are the entities in which Apache Kafka organizes and stores data. The created topics are then reassigned to assure an even topic distribution across the Apache Kafka brokers. ESPBench creates topics with one partition, as Apache Kafka only guarantees the correct order of records within a single partition.
In step (3), the Ansible script invokes the data generator. The resulting business data is copied to the DBMS node and imported into the DBMS by the data sender tool (step (4)). The Unix daemon CollectD [3] is started on all nodes in step (5) to gather system data during the benchmark run. This enables further analyses, such as evaluating differences between configurations regarding memory consumption or CPU utilization.

Step (6) starts the benchmark application, i.e., the benchmark query or queries that are to be executed. After a few seconds wait time for the DSPSs to receive and start the application, the script invokes the data sender in step (7), which sends the streaming data to the corresponding Apache Kafka topic(s). The ESPBench naming convention allows identifying the correct topic names based on the configuration parameters, which eases query implementation.

The data sender runs for the configured period of time. Once it completes, the Ansible script waits until the queries have finished data processing, represented as step (8). In step (9), CollectD is stopped and the recorded data is transferred to the main node where the Ansible script is executed. In the last step, step (10), ESPBench invokes the validator and result calculator.

### 2.5 Input Data

The input data of ESPBench is drawn from two domains: business data and sensor data from manufacturing equipment. Both types of data are described in the following.

#### 2.5.1 Sensor Data

There are two types of data streams that ESPBench incorporates. The first one is the data set used at the DEBS Grand Challenge 2012, which contains measurements from multiple sensors combined to single records. There are two machines sending this kind of data. The record’s structure is depicted in Table 2. ESPBench extends the data structure by the column workplace id, which is used for combining sensor and business data. The data includes information from analog as well as binary sensors.

Within the sensor data, the column most relevant for ESPBench is mf01, the electrical power on main phase one. It represents the energy consumption of the machines, which is valuable information, e.g., for identifying irregularities in the production process.

#### 2.5.2 Business Data

The schema of the business data is depicted in Figure 3. It is based on the TPC-C benchmark [43], one of the most known DBMS benchmarks that uses structured business data. Its schema covers core business relations, such as CUSTOMER and ORDER, that are representative of any manufacturing company. We simplified the TPC-C table design without impacting query costs. Inspired by modern business systems, we also added new relations that incorporate industrial manufacturing’s domain character. Specifically, we removed the tables WAREHOUSE, STOCK, DISTRICT, HISTORY, and NEW-ORDER. We
Table 4: ESPBench query set (based on [33])

| #  | 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 `sec` 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 [37] 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_SENSOR AS s1, STREAM_SENSOR2 AS s2, DB_TABLE_1 AS t WHERE (s1.M_ID = t.M_ID AND s1.mf01 < 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 in standby (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.O_NUM = STREAM_TIMES.PT.O.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). |

extend the schema by the tables **PRODUCTION_ORDER**, **PRODUCTION_ORDER_LINE**, and **WORKPLACE**, which are highlighted in green in Figure 3. By default, to have a representative data size, data is generated with a *scale factor* of three, which would equal a TPC-C setting with three warehouses. This configuration parameter has an impact on the overall business data size and can be altered for scaling reasons.

The introduced table **WORKPLACE** contains information about scheduled downtimes. These data allow distinguishing planned downtimes from irregularities that require reactions. The other two added tables contain information 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 concept in Enterprise Resource Planning systems [31, 48].

### 2.6 Benchmark Queries

This section presents the benchmark queries that the SUT is tasked with. Moreover, we introduce the design objectives that were taken into account when developing these queries.

2.6.1 **Relevance of Queries.** When defining benchmark queries, *relevance* and *simplicity* need special consideration. As outlined in Table 1, having easily understandable queries is a crucial requirement for, e.g., credibility reasons. Another aspect influenced by simplicity is the application of the benchmark. A rather straightforward workload is essential for implementing the queries for other streaming architectures at adequate costs, i.e., with implementation efforts for the queries that take a justifiable amount of time. This simplicity also benefits the usage of the benchmark.

To ensure the *relevance* of queries, the proximity of the workload to real-world scenarios and the coverage of important stream processing functionalities need to be considered. We addressed the first aspect by discussing our queries regarding their closeness to real-world use cases with two multi-billion revenue manufacturing companies, which confirmed their applicability.

To guarantee that all essential operations of DSPSs are covered, we first need to identify these. Our definition of these functionalities is based on the core set of operations for event processing systems presented by Mendes [45]. Although this set is defined for event processing systems, it is applicable to stream processing in general [33]. To incorporate the benchmark’s enterprise character, we extend the original list. Particularly, we broaden the included term *pattern detection* by altering it to *machine learning* to better represent current requirements on DSPSs. Furthermore, we add the aspects of transforming data, also included in earlier work by Mendes et al. [46], and of combining streaming with historical data. The latter challenge of integrating stored data is one of the eight requirements of real-time stream processing defined by Stonebraker, Çetintemel, and Zdonik [51].

The resulting list of core operations of DSPSs that the queries of ESPBench need to cover to be in line with the design objective of *relevance* is:

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

2.6.2 **Query Definitions.** Table 4 contains the query set of ESPBench. It specifically shows a query id, a brief description of the use case, the tested core functionalities of DSPSs, the exact query definition, and a more detailed explanation of the scenario and motivation. The queries of ESPBench fully cover the previously presented list of core operations. ESPBench defines the queries using a syntax inspired by the Continuous Query Language (CQL) [17]. CQL is based on the popular Structured Query Language (SQL) and extends it by incorporating data streams and corresponding constructs for data stream processing. However, CQL is, unlike SQL, not a language that is supported by a majority of DSPSs.

**Query 1 - Check Sensor Status.** The query *Check Sensors* monitors the attribute `mf01`, i.e., the electrical power in main phase one, to allow operators insights into irregularities as soon as possible by providing useful and up-to-date key performance indicators (KPIs). The query calculates the average, minimum, maximum, and the overall number of sensor values in tumbling windows of one second. The result records contain these calculated KPIs separated by comma.
we benchmark three state-of-the-art DSPSs and PostgreSQL, the de-
window with 500 elements. The query outputs values that are an
Query 2 - Determine Outliers. The second query determines out-
liers based on the input of mf01 and mf02. Its results give hints
irregularities in the manufacturing equipment. We employ the
stochastic outlier selection algorithm [37] on count-based tumbling
winds with 500 elements. The query outputs values that are an
Query 3 - Identify Errors. Query three reports actual errors, which
are defined by unusually high power consumption of a machine, i.e.,
greater than 14,963, the 99.5%tile in main phase one. The output of
Query 4 - Check Machine Power. This query checks if the power
is unexpectedly low, which is a state requiring actions. As input,
the query gets two structurally identical data streams from two
machines and business data. If any of the machines are in an un-
planned phase of being shut-down or on standby, the corresponding
record needs to be logged. This is the case if mf03 falls below the
value of 8,105, the 9%tile, and there is no downtime planned for
the machine. Planned downtimes are persisted in the DBMS that
is part of the SUT. The table WORKPLACE stores the beginning
(WP_DOWNTIME_START) and end (WP_DOWNTIME_END) of
the next scheduled downtime for any machine identifiable by its ID
(WP_ID). This machine or workplace identifier is part of the sensor
data as shown in Table 2.
Query 5 - Persist Processing Times. Query five represents an-
other use case where sensor data and historical business data are
combined, which highlights the enterprise character of ESPBench.
Particularly, this query stores time data in the DBMS, which is
counter to query four that reads business data. Having this
information allows for data integration, i.e., for connecting sensor
data with business data, by using a timestamp-based approach.
Our industry studies revealed that this is a commonly applied technique
and thus, incorporating it strengthens the relevance of ESPBench.
The DBMS relation DB_PRODUCTION_ORDER_LINE, which con-
tains information about the factory’s production orders, needs to
be updated by query five. Data input for this query is the pro-
duction times stream with the structure depicted in Table 3. The
contained data indicates when a product or a part of it entered or
left a workplace. The current timestamp needs to be set on the
beginning PRODUCTION_ORDER_LINE, either in the start or end timestamp
column, depending on the incoming sensor record.

3 ESPBENCH VALIDATION SETUP
This section gives details on the ESPBench validation, particularly
on its concept, the employed DSPSs, and the technical setup.

3.1 Validation Concept
To validate the concepts and functioning of ESPBench with its tools,
we benchmark three state-of-the-art DSPSs and PostgreSQL, the de-
fault DBMS of ESPBench. We developed and published an example
implementation of the benchmark queries using the Apache Beam
SDK in version 2.16.0 for these measurements. This abstraction
layer is employed in academic as well as real-world scenarios, e.g.,
at Lyft [19] and Spotify [44]. Besides, the execution of Apache Beam
applications is not only supported by open-source DSPSs [2], but
also by commercial closed-source DSPSs, such as IBM Streams [9],
and Google Cloud Dataflow [4]. This broad application of Apache
Beam, both from a user perspective and a DSPS perspective, illus-
trates its relevance. Nevertheless, it is important to be aware that
the level of effort put into Apache Beam support by DSPSs is likely
to be reflected in the performance. The performance penalty traded
for the gain in flexibility is analyzed by Hesse et al. [29].

3.2 Data Stream Processing Systems
The three analyzed DSPSs are Hazelcast Jet [6], Apache Flink [21],
and Apache Spark Streaming [57], which is an extension to the
Apache Spark [56] system. All of these systems are mainly written
in a Java Virtual Machine (JVM) language, i.e., Java or Scala. Apache
Spark Streaming processes micro-batches, while the other systems
process records tuple-by-tuple. While Apache Flink and Spark fol-
low a master-worker pattern in their system design, a Hazelcast Jet
cluster only contains nodes of the same kind. All systems are able
to provide exactly-once processing guarantees [8, 27].
As Hazelcast Jet is a less popular system compared to Flink and
Spark, we introduce it in the following paragraph. Hazelcast Jet,
short Jet, is the stream processing engine of the Hazelcast company,
headquartered in Silicon Valley. The streaming engine uses the
Hazelcast distributed in-memory data grid (IMDG), short Hazelcast.
Next to open-source versions of Hazelcast IMDG and Hazelcast
Jet published under the Apache License [7, 12], the commercial
versions Hazelcast IMDG Enterprise and Hazelcast IMDG Enterprise
HD are also offered. The latter one extends Hazelcast IMDG Enter-
prise by, e.g., a high-density (HD) memory store. There is also a
Hazelcast Jet Enterprise component, which extends Hazelcast Jet
by, e.g., security features and a lossless restart functionality [5].
From an architectural perspective, Jet is different from Apache
Spark and Apache Flink due to its masterless design. The oldest
node in the cluster represents the de facto leader and manages data
responsibilities within the cluster. Hazelcast organizes the data
in shards or partitions, which are distributed equally among the
cluster. It keeps data backups at multiple nodes to prevent data loss
in case of a node failure [39].
Two deployment modes are supported for both Hazelcast IMDG
and Jet: the embedded mode and the client-server mode. Figures 4
and 5 visualize the embedded and client-server deployments, re-
spectively. In embedded deployment mode, both the application
as well as Jet share a single JVM. This design has the advantage
of low-latency data access due to the tight coupling of applica-
tion and Jet. A downside of the embedded mode is its inability
to scale the application and data processing engine/data persist-
ence independently [5, 39]. In contrast, a client-server deployment
may create, scale, and manage the cluster independently from any
application executed on it. This setup allows separation of appli-
cation and cluster but also introduces challenges. For instance,
classpath management of both, application and cluster nodes, re-
quires more attention [39]. For validating ESPBench, we employ
the client-server deployment, which is is more relevant in corpo-
rate contexts, as the provided scalability and flexibility aspects are
standard requirements for enterprise IT landscapes.
3.3 Server Landscape

The validation setup consists of eight virtual machines (VMs), each of which exclusively uses their underlying server. This allows running the benchmark components independently of each other.

One of these servers serves as the starting point, from which the Ansible script that automates the benchmark execution is started. The DBMS is deployed on another VM. Three further nodes build the Apache Kafka cluster and the remaining three VMs contain the DSPS. Hazelcast Jet is deployed in the client-server mode using all three nodes. Apache Spark and Apache Flink use two nodes as workers and one node as the master.

The system characteristics of the nodes employed for the experiments are listed in Tables 5 and 6. All system configurations are part of the ESPBench repository. To ensure that Apache Kafka does not become a bottleneck, i.e., to make sure that the SUT is benchmarked as intended, we employ data rates that can provably be handled by Apache Kafka in the described setup. The study presented by Hesse et al. [32] indicates that Apache Kafka can easily handle 1K as well as 10K messages/second. Our conducted industry studies revealed that input rates in such a range with the used input data characteristics do not saturate the network capacities.

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 included figures visualize one-minute-averages of this system load KPI. As we are using servers with an eight-core CPU, described in Section 3.3, it is desirable that no node exceeds a system load of eight to do not over-utilize a machine [26].

4 EXPERIMENTAL EVALUATION

This section presents the results of the experimental evaluation of ESPBench, which are also made available online2. After giving an overview of the measurements, we analyze each query in detail. These individual analyses include a view on the system loads, which ESPBench collects every ten seconds in the applied settings. The system load gives an overview over the CPU and I/O utilization of a server, i.e., also reflecting performance limits regarding disk inputs.

Table 5: System characteristics of the Apache Kafka brokers

| Characteristic                  | Value                                      |
|---------------------------------|--------------------------------------------|
| Operating System                | Ubuntu 18.04 LTS                           |
| CPU                             | Intel(R) Xeon(R) CPU X7560 @ 2.27GHz, 8 cores (2x), Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz, 8 cores (1x) |
| RAM                             | 57GB (2x), 32GB (1x)                       |
| Network                         | 10Gbit via Fujitsu PRIMERGY BX900 S1       |
| Disk                            | 13 Seagate ST320004CLR2000 in RAID 6, access via Fibre Channel with 8Gbit/s |
| Hypervisor                      | VMware ESXi 6.7.0                          |
| Apache Kafka Version            | 2.3.0                                      |
| Java Version                    | OpenJDK 1.8.0_222                          |

Table 6: System characteristics of the SUT nodes

| Characteristic                  | Value                                      |
|---------------------------------|--------------------------------------------|
| Operating System                | Ubuntu 18.04 LTS                           |
| CPU                             | Intel(R) Xeon(R) CPU E5450 @ 3.00GHz, 8 cores |
| RAM                             | 57GB                                       |
| Network                         | 10Gbit via Fujitsu PRIMERGY BX900 S1       |
| Disk                            | 13 Seagate ST320004CLR2000 in RAID 6, access via Fibre Channel with 8Gbit/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                          |

4.1 Result Overview

Table 7 shows a summary of the results, i.e., the 90%tile, minimum, maximum, and mean latencies for the different ESPBench queries 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. We benchmarked data input rates of 1K and 10K messages/second. A single benchmark run lasts five minutes. Overall, the latency results are diverse with Jet often performing best with respect to the 90%tile and mean values. Next to Table 7, Figure 6 visualizes single latencies and system loads for selected queries and settings.

4.2 Query 1 - Check Sensor Status

The results shown in Table 7 reveal that Jet performed significantly better than Apache Flink. Although Apache Flink’s minimum latency is relatively low (49 ms), there are remarkably higher latencies, the maximum being above 18 s and the 90%tile being at more than 10 s. In contrast, Jet’s worst response time is about 700 ms.

2https://github.com/guenter-hesse/ESPBenchExperiments
Table 7: Latency result overview of the experimental analysis conducted as part of the validation of ESPBench

| Query                          | Input Rate in messages/second | System              | 90%tile in s | Min in s | Max in s | Mean in s |
|--------------------------------|-------------------------------|---------------------|---------------|----------|----------|-----------|
| 1 - Check Sensors              | 1,000                         | Apache Flink        | 10.659        | 0.049    | 18.591   | 4.269     |
|                                |                               | Hazelcast Jet       | 0.024         | 0.009    | 0.691    | 0.020     |
|                                |                               | Apache Spark Streaming | n/a           | n/a      | n/a      | n/a       |
| 1,000                          |                               | Apache Flink        | 16.492        | 0.048    | 33.423   | 5.767     |
|                                |                               | Hazelcast Jet       | 0.036         | 0.012    | 1.030    | 0.029     |
|                                |                               | Apache Spark Streaming | n/a           | n/a      | n/a      | n/a       |
| 10,000                         | Apache Flink                  | 8,175.784           | 40.446        | 9,147.738| 5,999.666| 4,140.149 |
|                                | Hazelcast Jet                 | 7,425.443           | 24.564        | 8,282.022| 4,140.149| 4,140.149 |
|                                | Apache Spark Streaming        | n/a                 | n/a           | n/a      | n/a      | n/a       |
| 2 - Determine Outliers         | 1,000                         | Apache Flink        | 615.078       | 9.352    | 676.535  | 358.076   |
|                                | Hazelcast Jet                 | 533.177             | 5.353         | 590.170  | 304.689  |
|                                | Apache Spark Streaming        | n/a                 | n/a           | n/a      | n/a      | n/a       |
| 10,000                         | Apache Flink                  | 14.979              | 0.002         | 19.058   | 4.581    |
|                                | Hazelcast Jet                 | 0.011               | 0.001         | 0.045    | 0.005    |
|                                | Apache Spark Streaming        | 0.534               | 0.121         | 1.248    | 0.387    |
| 3 - Identify Errors            | 1,000                         | Apache Flink        | 14.979        | 0.002    | 19.058   | 4.581     |
|                                | Hazelcast Jet                 | 0.016               | 0.005         | 0.795    | 0.014    |
|                                | Apache Spark Streaming        | 1.557               | 0.137         | 5.380    | 0.780    |
| 10,000                         | Apache Flink                  | 470.689             | 1.936         | 517.291  | 275.096  |
|                                | Hazelcast Jet                 | 87.299              | 6.008         | 94.599   | 56.236   |
|                                | Apache Spark Streaming        | 303.432             | 4.255         | 325.951  | 188.158  |
| 4 - Check Machine Power        | 1,000                         | Apache Flink        | 106.892       | 0.506    | 114.750  | 65.261    |
|                                | Hazelcast Jet                 | 88.006              | 1.823         | 96.316   | 51.278   |
|                                | Apache Spark Streaming        | 102.736             | 0.803         | 112.815  | 61.820   |
| 10,000                         | Apache Flink                  | 2,028.137           | 2.274         | 2,211.899| 1,136.910| 1,041.930 |
|                                | Hazelcast Jet                 | 2,129.287           | 6.202         | 2,345.790| 1,170.944| 1,170.944 |
|                                | Apache Spark Streaming        | 1,863.259           | 1.941         | 2,061.002| 1,041.930| 1,041.930 |

Apache Spark’s response times cannot be compared as the query results differ from the expected results. This is the case although we use the same application developed using the Apache Beam SDK for all systems. This valuable finding highlights the importance of having a query result validation for performance benchmarks as published with ESPBench. An explanatory hypothesis for the false results is related to the data architecture of Apache Spark Streaming, i.e., the distinguishing use of micro-batches. Through batching mechanisms, windows might look 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. [20] studies the heterogeneity in window semantics that exists in DSPSs.

Figure 6a visualizes the result latencies of query one for the input rate of 1K messages/second. It becomes visible that there are upward outliers for both systems, while the overall latencies on Apache Flink are significantly higher as identified 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 Jet runs, where the latency, after facing an upward outlier, immediately jumps back to normal, i.e., the value range to which most latencies belong.

The gathered system characteristics reveal that Hazelcast utilizes the cluster more. Figure 6b shows the system load for the SUT nodes while executing the first query. 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 two and six, there is only one Apache Flink node that shows a utilization close to one. The better system utilization is likely to be a reason for the lower latencies that are associated with Hazelcast Jet for query one compared to Apache Flink, cf. Table 7.
4.3 Query 2 - Determine Outliers

The latencies of Table 7 show significantly higher values for query two than for query one, with Hazelcast Jet slightly outperforming Apache Flink. Executing the query implementation on the Apache Spark Streaming system did not reveal any latencies, but another valuable finding. Specifically, the system throws an exception when submitting the Apache Beam program:

```java
java.lang.IllegalStateException: No TransformEvaluator registered for UNBOUNDED transform View.CreatePCollectionView
```

which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.

Figure 6c visualizes the system loads for both systems. The load of the ESPBench feature to output single record latencies. The latencies of both systems show a steadily growing latency, which indicates queuing in the system. That means the outlier detection cannot be performed fast enough for the configured input rate. This queuing does not become visible when looking at the overall numbers shown in Table 7, which highlights the importance of the ESPBench feature to output single record latencies.
4.4 Query 3 - Identify Errors

Figure 6d shows the latencies for the input of 1K messages/second, specifically every tenth latency for readability reasons. It illustrates that there are ups and downs that stay in system-specific ranges. Apache Flink having the lowest latencies, followed by Hazelcast Jet and Apache Spark Streaming. Apache Spark Streaming further shows a pattern-like trend, which can be due to the use of micro-batches.

Figure 6e shows every 100th latency for 10K messages/second. Hazelcast Jet and Apache Spark Streaming again show ups and downs within a comparatively small range. The previously observable pattern for Apache Spark is not present anymore. Apache Flink presents a 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 this new level, with relatively high swings though. This unique progress can be observed in all Apache Flink runs for this input rate and query. It again highlights the importance of having and studying single latencies to get full insights.

Figure 6f visualizes the system load of query three for the input rate of 10K messages/second. The utilization of the Jet runs is similar to the one observed in Figure 6b, i.e., a higher utilization on all three nodes. Moreover, the utilization of each node is greater than the highest system loads of both, Apache Flink and Apache Spark Streaming runs.

4.5 Query 4 - Check Machine Power

Figure 6g visualizes the latencies for a run with an input rate of 1K messages/second. Relatively small peaks can be identified for all systems, which might be caused by garbage collection runs. While peaks for Apache Spark are comparatively high and constant throughout the benchmark run, the peaks for Apache Flink and Jet are bigger at the beginning with a decreasing trend. For the input rate of 10K messages/second, steadily increasing latencies can be observed, similar to the latency trend for query two. It again indicates that records were queuing up, i.e., the SUTs cannot handle the higher data input rate in a sustainable fashion.

Figure 6h presents the system loads for query four. Contrary to the other system load charts, this figure includes the DBMS node as the database is incorporated in the queries four and five. Figure 6b draws a similar picture as Figure 6f with the system loads for query three, with the difference of having higher loads for Jet and Apache Spark. Especially node #1 of the Jet cluster shows the overall highest measured load of about 6.5 for a short period. The PostgreSQL node shows a very low system load of less than one for all settings.

4.6 Query 5 - Persist Processing Times

The evaluation shows steadily increasing latencies as for query two. It again indicates that the SUT cannot handle the load properly. The gathered results reveal that the DBMS is the bottleneck for this write-heavy query with about 300K required updates for the five minute runs with 1K messages/second as data input rate.

Figure 6i visualizes the system loads with an overall maximum below two. Hazelcast Jet again created the highest system loads compared to the other two DSPSs. The loads on the PostgreSQL VM are higher than for the previous query, with Hazelcast Jet causing the highest system load on the server where the DBMS is installed.

5 THREATS TO VALIDITY

One threat to validity is related to the applicability of results produced by ESPBench to domains different from manufacturing. The workload of ESPBench belongs to the manufacturing domain and is validated with companies from this industry sector. Depending on the target domain, the employed combination of data and queries might have a varying value. However, as the queries cover all core functionalities of DSPSs, results of ESPBench give at least a general hint on a system’s performance.

Another threat to validity is the limited number of metrics employed by ESPBench. However, it benefits the simplicity of the benchmark. Regarding the results of the experimental evaluation, the amount of three executions for each benchmark run is another threat to validity, even though variances between runs are low.

6 LESSONS LEARNED

We learned three main lessons from developing and evaluating ESPBench, one of them being the importance of result validation, which unfortunately lacks proper tool support in existing benchmarks. We highlight this importance by pointing to differing results for the same application executed on different DSPSs. Depending on the scenario, correctness might be more or less important to users of DSPSs. However, it is crucial to be aware of a system’s behavior.

Secondly, the portability of Apache Beam applications is not always given, i.e., it is not guaranteed that the paradigm ‘write once, execute anywhere’ holds. In particular, we logged an exception after successfully executing it on Apache Flink and Hazelcast Jet.

Thirdly, we learned that having single response times is of paramount importance. Aggregated KPIs often do not allow to fully understand the system’s behavior or are even misleading. That is the case, e.g., when 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 production deployment, i.e., when the application is running permanently.

7 RELATED WORK

Though there is an abundance of benchmarks for DBMSs, only a few focus on stream processing architectures. Linear Road by Arasu et al. [18] is one of the most popular benchmarks for DSPSs. It includes a toolkit comprising a data generator, a result validator, and a data sender stub, i.e., an incomplete implementation of a data sender that is designed to be completed by a benchmark user. Providing only a stub introduces the danger of different implementations among benchmark users, resulting in less comparable benchmark results. Furthermore, the benchmark creators cannot assure that the data sender does not become a bottleneck while benchmarking. An additional challenge of the Linear Road benchmark is the lack of supporting tools for automation. The scenario of Linear Road is a variable tolling system for a metropolitan area with multiple expressways. The accumulated tolls vary and depend on the city’s traffic situation. Linear Road defines four queries. However, in the implementations described in the benchmark paper, one of them is skipped due to complexity [18]. Besides streaming data, historical data is sparsely incorporated as tolling history for some queries.
Second, the majority of current stream processing benchmarks lack account in all major DSPS benchmarks. We believe that this is a consequence of several reasons. First, historical data is not or only barely taken into account in the benchmark use cases. This absence of proper tooling complicates the application of these benchmarks and retrieving objective and credible results. Finally, the workloads of existing benchmarks fail to cover the core DSPS functionalities, which leads to limited meaningfulness of results.

Table 8: Comparison of data stream processing benchmarks

| Benchmark type | Linear Road [18] | StreamBench [50] | RIoTBench [50] | YSB [23] | OSPBench [53] | ESPBench |
|----------------|------------------|------------------|----------------|----------|---------------|----------|
| Historical data | application briefly; one file with historical tolls | micro | mixed | application briefly; key-value store with advt. data | application | application comprehensively; business data, based on TPC-C |
| DSPS functionalities | partially covered | partially covered | partially covered | partially covered | partially covered | fully covered |
| Data sender | yes | - | - | yes | yes | yes |
| Result validator | - | - | - | yes | yes | yes |
| Automation | - | - | - | yes | yes | yes |
| Query implementations published | - | - | yes | yes | yes | yes |

StreamBench [50] is a micro benchmark aimed at distributed DSPSs. It defines a group of seven queries, which contains queries with a single as well as with multiple computational steps. The employed queries further differ regarding their requirements on keeping state. Although the queries cover a variety of functionalities, typical streaming operations, such as window functions, are not incorporated. StreamBench uses two real-world data sets as seeds for data generation. Contrary to Linear Road and similarly to ESPBench, StreamBench employs Apache Kafka for decoupling data generation and consumption. However, a benchmark tool for data ingestion is neither published nor described by the authors of StreamBench. StreamBench uses multiple real-world metrics, including latency and throughput. Additionally, StreamBench presents: a durability index (uptime), a throughput penalty factor (assessing throughput change in a node failure scenario), and a latency penalty factor (assessing latency change in a node failure scenario). The benchmark does not offer a tool for query result validation.

RIoTBench [50] defines micro benchmark as well as application benchmark use cases. These cover Extract, Transform, and Load (ETL) processes, statistics generation, model training, and predictive analytics scenarios. RIoTBench uses scaled real-world data sets from Internet of Things (IoT) domains, comprising smart city, smart energy, and health. Neither a data sender tool nor an application for query result validation is provided by the benchmark. RIoTBench measures latency, throughput, CPU, and memory utilization, as well as jitter. The latter expresses the difference between the expected and actual output rate during a certain period.

Yahoo! published a DSPS benchmark, known as Yahoo! Streaming Benchmark (YSB), in 2015 [22, 23]. YSB distinguishes it from the other benchmarks as it originates from the industry. However, it contains only a single query in the domain of advertisement. Van Dongen and Van den Poel [53] contributed measurements of the relatively new client library Kafka Streams [10] using their own benchmark. However, the authors do not incorporate historical data in the proposed Open Stream Processing Benchmark (OSPBench).

Table 8 compares the five benchmarks with ESPBench. In summary, we see the need for a new stream processing benchmark for several reasons. First, historical data is not or only barely taken into account in all major DSPS benchmarks. We believe that this is a crucial aspect for enterprises, because, 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. Almost none of the benchmarks supports full automation, which is essential for the simplicity of a stream processing benchmark. This absence of proper tooling complicates the application of these benchmarks and retrieving objective and credible results. Finally, the workloads of existing benchmarks fail to cover the core DSPS functionalities, which leads to limited meaningfulness of results.

8 CONCLUSION AND FUTURE WORK

This paper presents ESPBench, a benchmark for data stream processing architectures in the enterprise context, where streaming data is combined with structured business data. The ESPBench workload covers all core functionalities of DSPSs. As part of ESPBench, we provide an example implementation using Apache Beam and a comprehensive toolkit, which simplifies the benchmark use. The toolkit allows an objective result calculation, i.e., ESPBench does not rely on the different and differently measured performance metrics several systems provide. We validated ESPBench using the example query implementation in an experimental evaluation, benchmarking three state-of-the-art DSPSs along with a modern DBMS. The benchmark results that incorporate the impact of Apache Beam reveal that no system outperforms the others for all scenarios.

To include additional use cases for ESPBench in the future, the validator can be extended to allow ignoring the query result order, focusing only on the existence of results. This additional feature allows for extended scaling concerning the Apache Kafka topic partitions. In the current setting, we topics with a single partition to ensure the correct order of records. However, there might be use cases where the order result is not crucial that justify having this validator option. By scaling via Apache Kafka topic partitions, the data input rate manageable by the message broker can be increased.

Future work further includes query extensions, e.g., other kinds of windowing scenarios, and extended evaluations. Topics of interest include analyses of the scalability characteristics of the DSPSs and the impact of altered DBMSs or their configurations on the latencies. In this work, we have focused on open-source DSPSs. Results of comparisons with commercial systems using ESPBench will lead to a more complete overview of the DSPS landscape. Regarding the benchmark results, the metrics calculated by the validator and result calculator tool can be extended by, e.g., throughput.

ESPBench represents an easy-to-use yet meaningful benchmark that produces objectively determined results. We are confident that ESPBench improves performance benchmarking in the area of DSPSs and invite others to apply it and to propose improvements.
