# ESPBench: The Enterprise Stream Processing Benchmark

Guenter Hesse
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
guenter.hesse@hpi.de

Christoph Matthies
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
christoph.matthies@hpi.de

Michael Perscheid
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
michael.perscheid@hpi.de

Matthias Uflacker
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
matthias.uflacker@hpi.de

Hasso Plattner
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hasso.plattner@hpi.de

## 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.

## CCS CONCEPTS

• **Software and its engineering** → **Software performance**; • **Information systems** → **Stream management**; Database performance evaluation; • **Applied computing** → Enterprise data management.

## KEYWORDS

Performance, Benchmark, Stream Processing, Enterprise Software

**ACM Reference Format:**
Guenter Hesse, Christoph Matthies, Michael Perscheid, Matthias Uflacker, and Hasso Plattner. 2021. ESPBench: The Enterprise Stream Processing Benchmark. In *ICPE '21: ACM/SPEC International Conference on Performance Engineering, April 19–23, 2021, Rennes, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

## 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 [44]. 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 [42]. Manufacturing equipment such as a single saw can generate 50,000 messages or 1.2G B of data on a daily basis [15]. Injection molding machines even produce up to multiple terabytes of sensor data in 24 hours [31]. 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 [24]. 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 [30]
- 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 [22] 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<br>• Queries validated with industry<br>• Design represents real-world settings<br>• Data rates configurable |
| **Portability** | Easy to use on different systems or architectures | • No restrictions on SUT architecture specifics by toolkit<br>• System-independent query definitions |
| **Scalability** | Applicable to both, single node systems and scale-out systems with multiple nodes; benchmark should be scalable to larger systems | • Data rates configurable/scalable<br>• Scale-out scenarios supported by benchmark tools |
| **Simplicity** | Easy to understand and easy to use/implement to ensure result creditbility | • Automation of entire benchmark process<br>• Publication of usable example query implementation<br>• Tool support for essential benchmark functions, e.g., query result validation and data stream generation |

To allow result reproduction and the usage of the benchmark, we published all artifacts [25] and created a public repository for ESPBench[1]. 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 benchmark queries. Section 3 introduces the validation setup for ESPBench, illustrating details on the systems under test and the benchmarking landscape. Subsequently, we discuss the benchmark results in the experimental evaluation section. Sections 5 and 6 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. We start by presenting the overall design objectives associated with ESPBench. Afterwards, 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 performance benchmark is *The Benchmark Handbook for Database and Transaction Processing Systems* by Jim Gray [22]. He defines four criteria a domain-specific benchmark has to meet in order to be considered useful. They 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

Our benchmark scenario is inspired by the 2012 *Grand Challenge* published at the *Distributed and Event-Based Systems (DEBS)* conference [33]. 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 for, 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 [30].

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 [25]. 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* [37], is responsible for storing data and represents the interface to the SUT. ESPBench incorporates a message broker to separate the SUT and the

[1]https://github.com/guenter-hesse/ESPBench

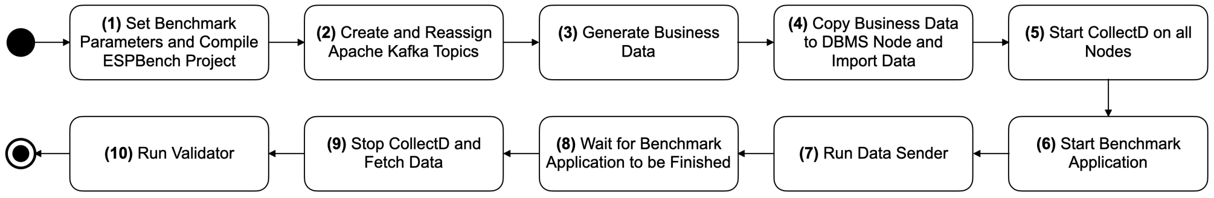

(1) Set Benchmark Parameters and Compile ESPBench Project
(2) Create and Reassign Apache Kafka Topics
(3) Generate Business Data
(4) Copy Business Data to DBMS Node and Import Data
(5) Start CollectD on all Nodes
(6) Start Benchmark Application
(7) Run Data Sender
(8) Wait for Benchmark Application to be Finished
(9) Stop CollectD and Fetch Data
(10) Run Validator

**Figure 1: ESPBench process visualized as Unified Modeling Language (UML) Activity Diagram [21]**

data sender. This separation allows for realistic and objective performance measurements outside of the SUT, which is important as DSPSs have diverse definitions of latency [35]. Additionally, changes to the data sender, e.g., launching multiple instances to increase the input rate, does not require adaptions to query implementations. A potential issue with message brokers identified in [35] 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 [27, 48]. 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 [27]. 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.

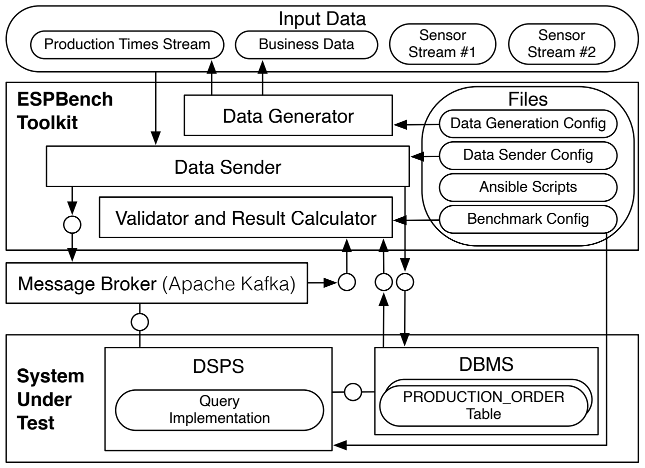


**Figure 2: Architecture of ESPBench as Fundamental Modeling Concepts (FMC) diagram [36]**

The *SUT* as another component of Figure 2 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* [47], 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 [10], 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* [12] script that 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* [14], an assembly plugin for the open-source build tool *sbt* [13]. 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. If there is only a single run, the parameters are read from the configuration files shown in Figure 2, which are provided with default values.

In *step (2)*, the required Apache Kafka topics are created according to the configuration and the naming schema defined by ESPBench. The created topics are then reassigned to assure an even topic distribution across the Apache Kafka brokers.

**Table 2: Data characteristics of the sensor measurement stream (based on [33])**

| # | 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 |
| 6-8 | required fixed32 pc13-pc15 | anode current drop detection cell 1-3 |
| 9-11 | required uint32 pc25-pc27 | anode voltage drop detection cell 1-3 |
| 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 | worklace ID |

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. There are two machines sending this kind of data. The record's structure is depicted in Table 2. ESPBench extends the data structure by a column containing the *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 a valuable information, e.g., for identifying irregularities in the production process.

**Table 3: Data characteristics of the sensor data used for production time determination**

| # | 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 entering / leaving of workplace |

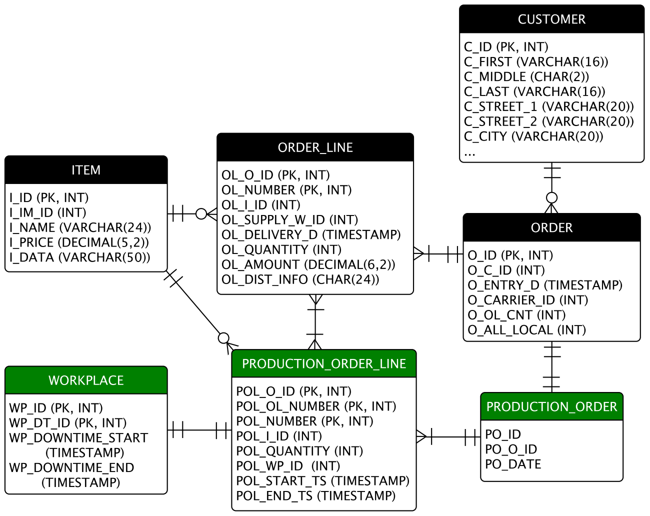


**Figure 3: ESPBench business data in Crow's Foot Notation**

The second type of data stream coming from the manufacturing equipment consists of information about the production times that allow combining sensor and business data. It is not part of the DEBS Grand Challenge, but designed by ESPBench and created by its data generator tool. The data stream's structure is shown in Table 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. The structure of the business data that is visualized in Figure 3 reveals that the first three columns are the primary key of the *PRODUCTION_ORDER_LINE* table and thus, can identify the workplace. Information about when products entered and left a workplace is needed for time-based vertical data integration, e.g., for linking sensor measurements at manufacturing machines to the product currently being worked on. Our conducted industry studies showed that this is a broadly adopted practice enterprise settings [28].

*2.5.2 Business Data.* The schema of the business data is depicted in Figure 3. It is based on the data schema of the TPC-C benchmark [38], 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 for any manufacturing company. We simplified the TPC-C table design without impacting queries 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 [29])**

| # | 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 [32] 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 in 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). |

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 [28, 43].

## 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 [40]. Although this set is defined for event processing systems, it is applicable to stream processing in general [29]. 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. [41], 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 [46].

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) [16]. 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.

*Query 2 - Determine Outliers.* Th second query determines outliers based on the input of *mf01* and *mf02*. Its results give hints on irregularities at the manufacturing equipment. We employ the *stochastic outlier selection* algorithm [32] on count-based tumbling windows with 500 elements. The query outputs values that are an outlier with a probability of $\geqslant$ 50% in order 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 comma.

*Query 3 - Identify Errors.* Query three reports actual errors, which are defined by an unusually high power consumption of a manufacturing machine, i.e., greater than 14,963, the 99.5 percentile in main phase one. The output of query three is the corresponding input sensor record. Benchmark runs with input rates of 1,000 messages/second and 10,000 messages/second output about 4 and 40 errors/second, respectively.

*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 is in an unplanned phase of being shut-down or in standby, the corresponding record needs to be logged. This is the case if *mf03* falls below the value of 8,105, the 9$^{\text{th}}$ percentile, 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 another 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 contrary to query four that reads business data. Having these 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 contains information about the factory's production orders, needs to be updated by query five. Data input for this query is the *production 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 in table *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 general concept, the employed DSPSs and the technical benchmark setup.

### 3.1 Validation Concept

To validate the concepts and functioning of ESPBench with its benchmark tools, we benchmark three state-of-the-art DSPSs and PostgreSQL, the default DBMS of ESPBench. We developed and published an example implementation of the queries defined by ESPBench 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 by companies such as Lyft [18] and Spotify [39]. 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, illustrates 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. [26].

### 3.2 Data Stream Processing Systems

The three analyzed DSPSs are *Hazelcast Jet* [6], *Apache Flink* [20], and *Apache Spark Streaming* [50], which is an extension to the *Apache Spark* [49] 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 follow 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, 24].

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 the 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, 11], the commercial versions *Hazelcast IMDG Enterprise* and *Hazelcast IMDG Enterprise HD* are also offered. The latter one extends Hazelcast IMDG Enterprise 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, Hazelcast 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 [34].

Two deployment modes are supported for both Hazelcast IMDG and Hazelcast Jet: the embedded mode and the client-server mode.

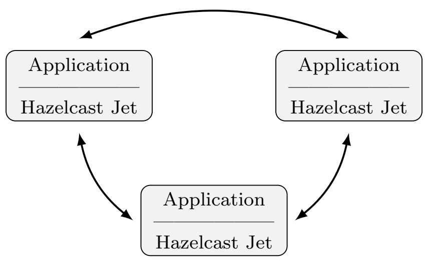


**Figure 4: Architecture of an embedded Hazelcast Jet deployment with three cluster nodes (based on[5, 34])**

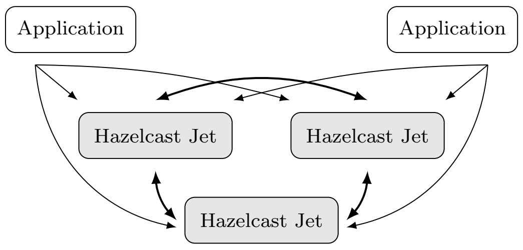


**Figure 5: Architecture of a client-server Hazelcast Jet deployment with three cluster nodes and two applications (based on[5, 34])**

Figures 4 and 5 visualize the embedded and client-server deployments, respectively. In embedded deployment mode, both the application as well as Hazelcast Jet share a single JVM. This design has the advantage of 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 [5, 34]. 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 application and cluster as depicted in Figure 5, but also introduces challenges. For instance, classpath management of both, application and cluster nodes, requires more attention [34]. For validating ESPBench, we employ the client-server deployment, which is is more relevant in corporate contexts, as the provided scalability and flexibility aspects are standard requirements for enterprise IT landscapes.

## 3.3 Server Landscape

The setup used for validation consists of eight virtual machine (VM) nodes, each of which exclusively use 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. [27] indicates that Apache Kafka can easily handle 1,000 as well as 10,000 messages/second. Our conducted industry studies revealed that these are satisfying ingestion rates for the majority of scenarios in the industrial manufacturing sector. Besides, the work shows that input rates in such a range with the used input data characteristics do not saturate the network capacities.

**Table 5: System characteristics of the Apache Kafka broker nodes**

| 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 ST320004CLAR2000 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 ST320004CLAR2000 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 EXPERIMENTAL EVALUATION

This section presents the results of the experimental evaluation of ESPBench, which are also made available online [2]. 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 writes. It is defined as the number of processes demanding CPU time, specifically processes that are ready to run or waiting for

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

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 [23].

## 4.1 Result Overview

Table 7 shows a summary of the results, i.e., the 90%tile, minimum, maxiumum, 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 1,000 and 10,000 messages/second. A single benchmark run lasts five minutes. Overall, the latency results are diverse with Hazelcast Jet often performing best with respect to the 90%tile and mean values. Next to Table 7 providing an overview with aggregated results, 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 Hazelcast 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, Hazelcast Jet's worst response time is about 700 ms.

Apache Spark Streaming'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. [19] studies the heterogeneity in window semantics that exists in DSPSs.

Figure 6a visualizes the result latencies of query one for the input rate of 1,000 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 to the latency development that we can observe for Hazelcast 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 moreFigure 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 to 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 slightly outperforming Apache Flink. Executing the query implementation for Apache Spark revealed a valuable finding. The systems throws an exception when submitting the Apache Beam program: *java.lang. IllegalStateException: No TransformEvaluator registered for UNBOUNDED transform View.CreatePCollectionView*. This observation indicates that the DSPS exchangeability of Apache Beam applications is limited.

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 is generally lower and both systems create more similar system loads compared to query one. These measurements suggests that the implementation of the stochastic outlier selection has room for improvement regarding its performance, e.g., by making it parallelizable.

## 4.4 Query 3 - Identify Errors

Figure 6d shows the latencies for the input of 1,000 messages/second, specifically every tenth latency for readability reasons. The diagram 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 10,000 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 Streaming 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 10,000 messages/second. The utilization of the Hazelcast 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 1,000 messages/second. Relatively small peaks can be identified for all systems, which might be caused by garbage collection runs.

**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 |
| | 10,000 | 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 |
| 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 |
| | 10,000 | 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 Streaming | n/a | n/a | n/a | n/a |
| 3 - Identify Errors | 1,000 | Apache Flink | 0.011 | 0.001 | 0.045 | 0.005 |
| | | Hazelcast Jet | 0.021 | 0.004 | 0.158 | 0.017 |
| | | Apache Spark Streaming | 0.534 | 0.121 | 1.248 | 0.387 |
| | 10,000 | 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 |
| 4 - Check Machine Power | 1,000 | Apache Flink | 0.717 | 0.003 | 2.792 | 0.251 |
| | | Hazelcast Jet | 0.371 | 0.006 | 4.082 | 0.195 |
| | | Apache Spark Streaming | 1.008 | 0.141 | 1.966 | 0.644 |
| | 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 |
| 5 - Persist Processing Times for Products | 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 |
| | | Hazelcast Jet | 2,129.287 | 6.202 | 2,345.790 | 1,170.944 |
| | | Apache Spark Streaming | 1,863.259 | 1.941 | 2,061.002 | 1,041.930 |

While peaks for Apache Spark Streaming are comparatively high and constant throughout the five minutes period of the benchmark run, the peaks for Apache Flink and Hazelcast Jet are bigger at the beginning with a decreasing trend. For the input rate of 10,000 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 visualizations, this figure includes the DBMS node as the database is incorporated in the queries four and five. Figure 6h draws a similar picture as Figure 6f with the system loads for query three, with the difference of having higher loads for Hazelcast Jet and Apache Spark Streaming. Especially node #1 of the Hazelcast 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.

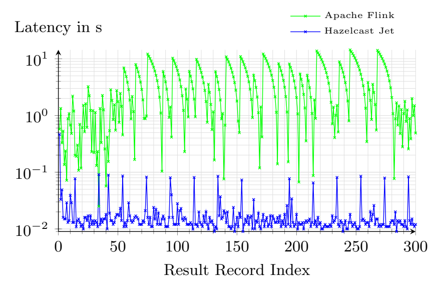
## 4.6 Query 5 - Persist Processing Times

The experimental evaluation shows steadily increasing latencies as for query two. It 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.

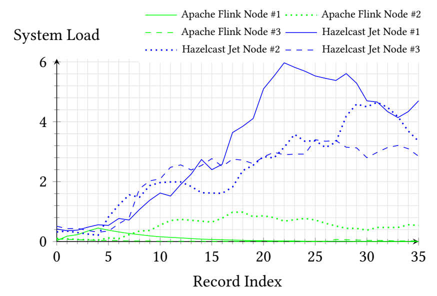
Figure 6i visualizes the system loads with an overall maximum below two. Hazelcast Jet again created the highest system loads compared to the other two DSPSs. The loads on the PostgreSQL VM are higher than for the previous query, with Hazelcast Jet causing the highest system load on the server where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

## 5 LESSONS LEARNED

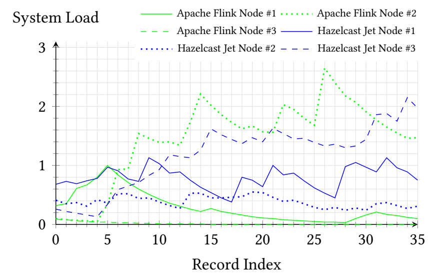
We learned three main lessons from developing and evaluating ESPBench, one of them being the importance of result validation, which unfortunately lacks proper tool support in existing benchmarks.



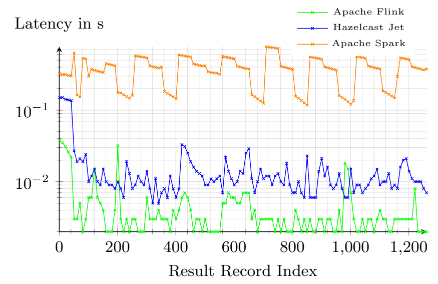
**(a) Latencies query 1 for a data input rate of 1,000 messages/second**



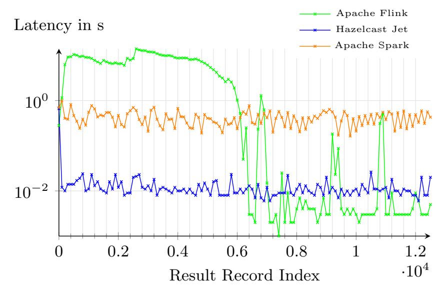
**(b) One-minute-average of short-term system load for query 1 for a data input rate of 1,000 messages/second**



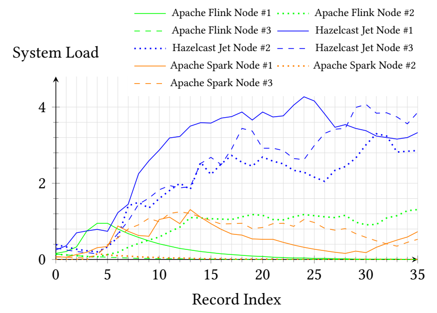
**(c) One-minute-average of short-term system load for query 2 for a data input rate of 1,000 messages/second**



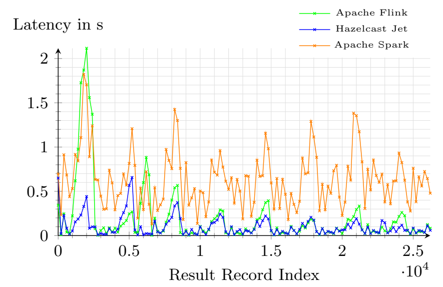
**(d) Latencies query 3 for a data input rate of 1,000 messages/second**



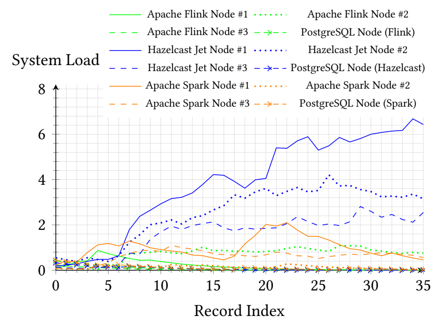
**(e) Latencies query 3 for a data input rate of 10,000 messages/second**



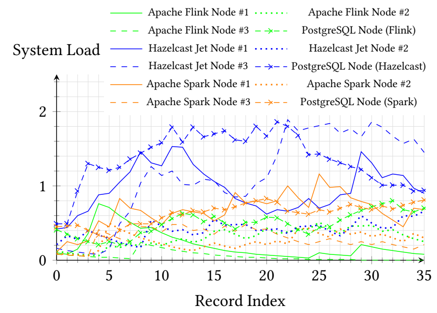
**(f) One-minute-average of short-term system load for query 3 for a data input rate of 10,000 messages/second**



**(g) Latencies query 4 for a data input rate of 1,000 messages/second**



**(h) One-minute-average of short-term system load for query 4 for a data input rate of 1,000 messages/second**



**(i) One-minute-average of short-term system load for query 5 for a data input rate of 1,000 messages/second**

**Figure 6: Graphs visualizing selected latencies and system loads measured during the experimental analysis conducted as part of the validation of ESPBench**

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.

The second lesson 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 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.

**Table 8: Comparison of data stream processing benchmarks**

| | Linear Road [17] | StreamBench [45] | RIoTBench [45] | **ESPBench** |
|---|---|---|---|---|
| Benchmark type | application | micro | mixed | application |
| Historical data | rudimentary; single file w/ historical tolls | - | - | business data, schema based on TPC-C |
| DSPS functionalities | partially covered | partially covered | partially covered | fully covered |
| Data sender | only stub provided | - | - | provided |
| Result validator | provided | - | - | provided |
| Automation | - | - | - | provided |
| Query implementations published | - | - | yes | yes |

## 6 RELATED WORK

In comparison to the number of benchmarks for DBMSs, there are only a few ones focussing on data stream processing architectures. *Linear Road* by Arasu et al. [17] 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 supposed to be finished by a benchmark user. Providing only a stub introduces the danger of different implementations among benchmark users, which results in a reduction of compatibility regarding benchmark results. Moreover, it cannot be assured by the benchmark creators that the data sender does not become a bottleneck while benchmarking. With respect to benchmark automation, there are no tools provided by Linear Road. The scenario of Linear Road is a variable tolling system for a metropolitan area with multiple expressways. The amount of accumulated tolls varies depending on the city's traffic situation. Linear Road defines four queries, one of them being skipped in the implementations described in the original benchmark paper [17] due to its complexity. Besides streaming data, historical data is frugally incorporated as tolling history for some queries.

*StreamBench* [45] is a micro benchmark aimed at distributed DSPSs. It defines seven queries, which include queries with single and multiple computational steps. Some queries require to keep a state to produce correct results while others do not. 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 like 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 result 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* [45] focuses on distributed DSPSs and 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.

Table 8 compares the three 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. None of the benchmarks supports 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 query workloads of existing performance benchmarks fail to cover the core DSPS functionalities, which leads to a limited meaningfulness of their results.

## 7 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 only use Apache Kafka topics with a single partition, as Apache Kafka only guarantees the correct order of records within a single partition. However, there might be use cases where the result order 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 or aggregation scenarios, and extended evaluations. Topics of interest include analyses of the scalability characteristics of the DSPSs and the impact of altered DBMSs or its 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 evaluate different systems and to propose improvements to ESPBench.

## REFERENCES

[1] [n.d.]. Apache Beam Overview. https://beam.apache.org/get-started/beam-overview/. Accessed: 2020-10-06.

[2] [n.d.]. Beam Capability Matrix. https://beam.apache.org/documentation/runners/capability-matrix/. Accessed: 2020-10-06.

[3] [n.d.]. collectd - The system statistics collection daemon. https://collectd.org. Accessed: 2020-10-06.

[4] [n.d.]. Dataflow. https://cloud.google.com/dataflow/. Accessed: 2020-10-06.

[5] [n.d.]. Hazelcast IMDG Reference Manual. https://docs.hazelcast.org/docs/3.12.4/manual/html-single/index.html. Accessed: 2020-10-06.

[6] [n.d.]. Hazelcast Jet. https://jet-start.sh. Accessed: 2020-10-06.

[7] [n.d.]. Hazelcast Jet - Distributed Stream and Batch Processing. https://github.com/hazelcast/hazelcast-jet. Accessed: 2020-10-06.

[8] [n.d.]. Hazelcast Jet FAQ. https://jet.hazelcast.org/faq/. Accessed: 2020-10-06.

[9] [n.d.]. IBM Streaming Analytics for IBM Cloud. https://www.ibm.com/de-en/marketplace/stream-computing. Accessed: 2020-10-06.

[10] [n.d.]. Lightbend - akka. https://akka.io. Accessed: 2020-10-06.

[11] [n.d.]. Open Source In-Memory Data Grid https://www.hazelcast.org. https://github.com/hazelcast/hazelcast. Accessed: 2020-10-06.

[12] [n.d.]. Red Hat Ansible. https://www.ansible.com. Accessed: 2020-10-06.

[13] [n.d.]. sbt. https://www.scala-sbt.org. Accessed: 2020-10-06.

[14] [n.d.]. sbt-assembly. https://github.com/sbt/sbt-assembly. Accessed: 2020-10-06.

[15] [n.d.]. Senseforce.io - Real-time edge-cloud data warehouse for industrial IoT. https://crate.io/customers/senseforce-iot/. Accessed: 2020-10-06.

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

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

[18] Akshay Balwally and Thomas Weise. [n.d.]. Streaming your Lyft Ride Prices. https://www.slideshare.net/FlinkForward/flink-forward-san-francisco-2019-streaming-your-lyft-ride-prices-thomas-weise-akshay-balwally. Accessed: 2020-10-06.

[19] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura M. Haas, Renée J. Miller, and Nesime Tatbul. 2010. SECRET: A Model for Analysis of the Execution Semantics of Stream Processing Systems. *Proceedings of the VLDB Endowment* 3, 1 (2010), 232–243. https://doi.org/10.14778/1920841.1920874

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

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

[22] Jim Gray (Ed.). 1993. *The Benchmark Handbook for Database and Transaction Processing Systems (2nd Edition)*. Morgan Kaufmann.

[23] Brendan Gregg. 2017. Linux Load Averages: Solving the Mystery. http://www.brendangregg.com/blog/2017-08-08/linux-load-averages.html. Accessed: 2020-10-06.

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

[25] Guenter Hesse and Christoph Matthies. 2020. *guenter-hesse/ESPBench: Initial ESPBench Release*. https://doi.org/10.5281/zenodo.4010167

[26] Guenter Hesse, Christoph Matthies, Kelvin Glass, Johannes Huegle, and Matthias Uflacker. 2019. Quantitative Impact Evaluation of an Abstraction Layer for Data Stream Processing Systems. In *IEEE International Conference on Distributed Computing Systems, ICDCS*. 1381–1392.

[27] Guenter Hesse, Christoph Matthies, Tilmann Rabl, and Matthias Uflacker. 2020. How Fast Can We Insert? A Performance Study of Apache Kafka. *CoRR* abs/2003.06452 (2020). arXiv:2003.06452 https://arxiv.org/abs/2003.06452

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

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

[30] Guenter Hesse, Werner Sinzig, Christoph Matthies, and Matthias Uflacker. 2019. Application of Data Stream Processing Technologies in Industry 4.0: What is Missing?. In *International Conference on Data Science, Technology and Applications (DATA)*. 304–310.

[31] Marco F. Huber, Martin Voigt, and Axel-Cyrille Ngonga Ngomo. 2016. Big Data Architecture for the Semantic Analysis of Complex Events in Manufacturing. In *46. Jahrestagung der Gesellschaft für Informatik*. 353–360. http://subs.emis.de/LNI/Proceedings/Proceedings259/article173.html

[32] J.H.M. Janssens. 2013. *Outlier Selection and One-Class Classification*. Ph.D. Dissertation. Series: TiCC Ph.D. Series Volume: 27.

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

[34] M. Johns. 2015. *Getting Started with Hazelcast - Second Edition*. Packt Publishing. https://books.google.de/books?id=r8cIswEACAAJ

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

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

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

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

[39] Neville Li. [n.d.]. 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/. Accessed: 2020-10-06.

[40] Marcelo Rodrigues Nunes Mendes. 2014. *Performance Evaluation and Benchmarking of Event Processing Systems*. Ph.D. Dissertation.

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

[42] Judit Nagy, Judit Oláh, Edina Erdei, Domicián Máté, and József Popp. 2018. 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 (2018), 3491.

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

[44] Sergei Sokolenko. [n.d.]. Spotify: 500 billion events per day as of nov 2019. https://twitter.com/datancoffee/status/1197505299168604162. Accessed: 2020-10-06.

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

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

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

[48] Alex Woodie. 2019. What's Behind Lyft's Choices in Big Data Tech. https://www.datanami.com/2019/05/10/whats-behind-lyfts-choices-in-big-data-tech/. Accessed: 2020-10-06.

[49] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *USENIX Workshop on Hot Topics in Cloud Computing, HotCloud*.

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