FUDMA Journal of Engineering and Technology
ISSN: 3092-9385
https://fjet.fudutsinma.edu.ng/index.php/fjet/
Volume 2, Issue 1, 2026, 395-401

Scalability Versus Accuracy Trade-offs in Distributed Big Data Processing Frameworks: A Comparative Evaluation of Apache Spark, Flink, and Dask Using Benchmark Datasets

Isiaq O. ALABI[1*], Hassan T. ABDULAZEEZ[2], Sulaiman AHMAD[3], Yahaya M. SANI[4]

[1*,4]Department of Information Technology, Federal University of Technology, Minna, Nigeria
[2,3]Department of Cyber Security Science, Federal University of Technology, Minna, Nigeria

[1*]isiaq.alabi@futminna.edu.ng, [2]abdulazeezhassant@futminna.edu.ng, [3]ahmads@futminna.edu.ng, [4]yahayasani@futminna.edu.ng

Abstract

The exponential growth in data volume, velocity, and variety has intensified demand for distributed processing frameworks that balance computational scalability with analytical accuracy. Apache Spark, Apache Flink, and Dask represent three dominant open-source ecosystems, yet selecting an appropriate framework requires nuanced understanding of their performance characteristics under diverse workloads. This study presents a systematic comparative evaluation of these frameworks using standardized benchmark datasets (Transactions Processing Performance Council-Decision Support (TPC-DS) at 100 GB scale factor and HiBench version 7.1) across four dimensions: execution time, memory consumption, fault tolerance, and result consistency. Experiments were conducted on Amazon Web Services EC2 infrastructure using identical c5.4xlarge instances (16 vCPUs, 32 GB RAM) configured in standalone cluster mode. Results demonstrate that Spark achieved optimal performance for batch-oriented SQL workloads, completing 92 of 99 TPC-DS queries with the lowest average runtime (18% faster than Flink, 32% faster than Dask). Flink exhibited superior latency characteristics and exactly-once processing semantics, recovering from simulated node failures within 12 seconds compared to Spark's 45 seconds. Dask demonstrated competitive performance for iterative machine learning tasks but exhibited higher memory volatility and occasional floating-point inconsistencies during fault recovery. These findings provide empirical guidance for practitioners designing analytics pipelines in domains requiring both timeliness and computational precision, including cybersecurity threat detection and financial analytics.

Keywords: *Big data processing; Distributed computing; Apache Spark; Apache Flink; Dask; Performance benchmarking; Fault tolerance.*

1.0 Introduction

Contemporary data-intensive applications spanning cybersecurity threat detection, financial fraud analytics, and real-time recommendation systems rely fundamentally on distributed processing frameworks capable of managing petabyte-scale datasets with minimal latency [1],[2]. Conventionally, big data is a massive and rapidly growing datasets that is either structured or unstructured, but surpass the capacity of traditional data processing software. The architectural foundations of these frameworks directly influence both horizontal scalability and the statistical reliability of analytical outputs. Among available solutions, Apache Spark, Apache Flink, and Dask have emerged as the predominant platforms due to their active development communities, extensible application programming interfaces, and compatibility with heterogeneous data sources [3]. Despite widespread adoption across industry and academia, systematic empirical comparisons that simultaneously evaluate performance, resource efficiency, and computational accuracy under controlled experimental conditions remain limited.

Scalability, in this context, refers to a system's capacity to maintain or improve performance as data volume or cluster resources increase without proportional degradation in throughput or latency. Accuracy denotes the correctness and reproducibility of analytical results, particularly under conditions of partial system failure, data skew, or iterative computation where numerical precision may be affected by execution order [4]. Prior evaluations have predominantly emphasized throughput or latency metrics in isolation, frequently neglecting how fundamental architectural decisions influence outcome integrity [5]. Micro-batching versus true streaming paradigms, lineage-based fault recovery mechanisms, and lazy versus eager execution strategies each impose distinct trade-offs that practitioners must navigate when designing production analytics systems.

This paper addresses the identified gap through a controlled comparative evaluation of Spark, Flink, and Dask using two widely-accepted benchmark suites: TPC-DS version *3.2.0* for decision support workloads and HiBench version *7.1* for general big data operations. All experiments were executed on identical virtualized cloud infrastructure to eliminate hardware variability. Three research questions guide this investigation. First, how do execution times and resource utilization patterns vary across frameworks for equivalent SQL and machine learning workloads? Second, to what extent do fault tolerance mechanisms impact result consistency during simulated node

Received: 13/03/2026; Revised: 11/04/2026; Accepted: 17/04/2026; Published: 25/04/2026

failures? Third, what practical trade-offs emerge when deploying these frameworks in resource-constrained or latency-sensitive operational environments?

This study contributes to the literature in three ways. First, it provides a reproducible methodology for cross-framework evaluation using open-source tooling and publicly available datasets, enabling independent verification and extension. Second, it quantifies the scalability-accuracy trade-off through empirical metrics not commonly reported together in existing literature, including result consistency validation via cryptographic hash verification. Third, it derives deployment recommendations tailored to specific application domains where both computational timeliness and analytical precision are critical, including cybersecurity intelligence analysis and financial risk modelling.

2.0 Related Work

Research on distributed big data processing frameworks has expanded considerably over the past decade, though studies vary substantially in scope, methodology, and focus areas. [6] conducted a comparative analysis of Spark and Flink using Yahoo Streaming Benchmarks, demonstrating Flink's superior throughput characteristics for continuous data streams. However, their evaluation focused exclusively on streaming workloads and did not examine batch processing scenarios or assess computational accuracy implications of framework-specific fault tolerance mechanisms. Similarly, the benchmarking framework proposed by [7] for evaluating real-time stream processing engines emphasized latency and throughput metrics while omitting analysis of result determinism under failure conditions.

[8] evaluated Spark, Flink, and Storm for Internet of Things data stream processing, reporting that Flink achieved lower end-to-end latency for windowed aggregations while Spark demonstrated more stable memory utilization patterns. Their experimental design did not include Dask or standardized decision support benchmarks such as TPC-DS, limiting generalizability to SQL-intensive analytical workloads. [9] provided a comprehensive survey of distributed stream processing architectures, synthesizing design patterns and trade-offs across multiple systems, though their analysis relied on reported performance characteristics rather than original empirical measurements using unified workloads.

The TPC-DS benchmark has been employed by [10] to demonstrate Spark SQL optimization capabilities, achieving order-of-magnitude performance improvements through the Catalyst query optimizer. However, comparative baselines against Flink's Table application program interface (API) or Dask's DataFrame operations were not established. HiBench evaluations conducted by [11] incorporated multiple processing engines across diverse workload categories, yet omitted detailed analysis of memory pressure effects on numerical stability during iterative machine learning computations. [12] examined performance characteristics of Spark and Flink for graph processing workloads, identifying that Spark's GraphX library exhibited memory efficiency advantages for large-scale graph analytics while Flink demonstrated superior performance for iterative algorithms requiring incremental state updates.

Recent comparative studies have begun addressing Python-native frameworks for data science workloads. [13] introduced Dask as a flexible parallel computing library designed for interactive analytics, emphasizing its seamless integration with NumPy and Pandas ecosystems. [14] compared Dask with Ray for distributed Python applications, noting Dask's task graph optimization capabilities alongside its limitations in stateful distributed computations. Nevertheless, systematic comparisons positioning Dask against (Java virtual machine) JVM-based frameworks using identical benchmark protocols remain scarce. This study extends prior work by integrating both TPC-DS and HiBench benchmarks, enforcing strict environmental controls through containerized deployment, and introducing accuracy validation mechanisms through checksum verification and result set comparison across all evaluated frameworks.

3.0 Methodology

All experiments were conducted on Amazon Web Services (AWS) Elastic Compute Cloud (EC^2) using c5.4xlarge instances, each provisioned with 16 virtual CPUs (Intel Xeon Platinum 8000 series, 3.0 GHz base frequency), 32 GB of 4^{th} generation double data rate (DDR4) memory, and Elastic Block Store (EBS) volumes configured with 3,000 provisioned input/output operations per second (IOPS) for consistent storage performance. The operating environment consisted of Ubuntu 20.04 long-term support (LTS) (kernel *5.15.0-1019-aws*) with identical software configurations maintained across all nodes. Each framework was deployed in standalone cluster mode comprising one dedicated master node and three worker nodes, enabling direct comparison without confounding effects from different resource managers such as Yet Another Resource Negotiator (YARN) or Kubernetes.

Runtime environments were standardized using OpenJDK *11.0.16* (AdoptOpenJDK build *11.0.16*+8) for JVM-based frameworks and Python *3.9.13* (CPython) with NumPy *1.23.4*, Pandas *1.5.1*, and PyArrow *10.0.0* for Dask-related executions. Docker containers (version *20.10.21*) ensured environment reproducibility across trials, with all container images archived in a public repository for independent verification.

Received: 13/03/2026; Revised: 11/04/2026; Accepted: 17/04/2026; Published: 25/04/2026

3.2 Benchmark Selection and Configuration

Two complementary benchmark suites were selected to evaluate framework performance across distinct workload categories. TPC-DS version *3.2.0* was configured at scale factor 100, generating approximately 100 GB of structured relational data across 24 tables representing a retail sales decision support schema. This benchmark comprises 99 complex SQL queries incorporating multi-table joins, nested subqueries, and aggregation operations characteristic of business intelligence workloads [15]. HiBench version *7.1* was deployed to assess performance across micro-benchmarks (WordCount, Sort, TeraSort) and machine learning workloads (Naive Bayes classification, *K*-means clustering with k=10 centroids and 1,000 maximum iterations).

3.3 Framework Configuration

Apache Spark *3.3.0* was configured with dynamic resource allocation disabled to ensure deterministic resource assignment across trials. Shuffle partitions were set to 200, the Kryo serializer was enabled for improved serialization performance, and adaptive query execution was activated for SQL workloads. Executor memory was allocated at 24 GB per worker with 4 executor cores, allowing two executors per node.

Apache Flink *1.16.0* employed the RocksDB state backend for large-state operations with checkpoint intervals set to 10 seconds and exactly-once processing semantics enabled. Task parallelism was configured at 12 slots per TaskManager, with 8 GB of managed memory allocated for state operations and network buffer pools.

Dask *2022.12.0* was deployed using the *dask.distributed* scheduler with worker thread counts matching available CPU cores (16 threads per worker) and memory limits enforced at 28 GiB per worker to prevent out-of-memory terminations. The distributed scheduler's work-stealing mechanism was enabled to improve load balancing during heterogeneous task execution.

3.4 Metrics Collection and Validation

Four primary metrics were collected across all experimental runs. Wall-clock execution time was measured from job submission to final output materialization using framework-native instrumentation. Peak memory utilization per node was sampled at 1-second intervals using *cAdvisor* container monitoring, capturing both JVM heap (for Spark and Flink) and Python process memory (for Dask). Network input/output volume was aggregated from container network interface statistics. Result correctness was validated through Message-Digest algorithm-5 (MD5) cryptographic hash comparison of output files against pre-computed golden results derived from single-node reference implementations.

Fault tolerance evaluation proceeded by simulating worker node termination through forced container termination (docker kill) at the midpoint of task execution, measured as 50% progress on the longest-running stage. Recovery time was measured from failure detection to successful job completion, and output deviation was quantified by comparing post-recovery results against non-failure baseline outputs. Each experimental configuration was executed five times, with mean values and standard deviations reported. Statistical significance of performance differences was assessed using paired t-tests with *Bonferroni* correction for multiple comparisons ($\alpha = 0.05$).

3.5 Reproducibility Protocol

All source code, configuration files, benchmark scripts, and raw experimental logs are archived in a public GitHub repository with a digital object identifier assigned upon publication. Docker images for each framework configuration are available through Docker Hub, and AWS CloudFormation templates enable automated infrastructure provisioning for independent replication.

4.0 Results and Discussion

4.1 TPC-DS Query Performance

Table 1 summarizes aggregate TPC-DS query execution performance across all three frameworks. Spark completed 92 of 99 queries with the fastest execution time, achieving a mean query duration of 47.3 seconds (Standard deviation (SD) = 62.1s). Seven queries failed due to optimizer timeout or resource exhaustion. Flink successfully executed 87 queries with a mean duration of 57.6 seconds (SD = 71.4s), representing an 18% slower aggregate performance relative to Spark ($t(86) = 4.21$, $p < 0.001$). Dask completed 78 queries with a mean duration of 69.8 seconds (SD = 89.2s), demonstrating 32% slower execution compared to Spark ($t(77) = 6.83$, $p < 0.001$). Query 85, involving a 12-way join across the largest fact tables, caused Dask to exceed memory limits, resulting in task failure despite worker memory allocation of 28 GB.

Received: 13/03/2026; Revised: 11/04/2026; Accepted: 17/04/2026; Published: 25/04/2026

Table 1: TPC-DS Query Performance Summary (Scale Factor 100)

Framework	Queries Completed	Mean Time (s)	SD (s)
Apache Spark 3.3.0	92/99	47.3	62.1
Apache Flink 1.16.0	87/99	57.6	71.4
Dask 2022.12.0	78/99	69.8	89.2

Note. Results based on five independent trials per query. SD = standard deviation.

4.2 HiBench Machine Learning Workloads

For iterative *K*-means clustering on a synthetic dataset of 10 million points in 100-dimensional feature space, Flink achieved convergence in a mean of 234.7 seconds (SD = 18.3s), representing 22% faster completion than Spark's 300.2 seconds (SD = 24.1s; $t(4) = 5.47$, $p = 0.005$). This advantage stems from Flink's native support for iterative dataflows with delta iterations, which avoid full dataset re-computation between iterations. Dask completed the same workload in 287.4 seconds (SD = 41.6s), comparable to Spark but with substantially higher variance across trials attributable to work-stealing scheduler dynamics and Python global interpreter lock contention.

Naïve Bayes classification on the Reuters Corpus Volume1 (RCV1) text corpus (804,414 documents, 47,236 features) showed Spark completing training and evaluation in 156.3 seconds (SD = 12.7s), compared to Flink's 168.9 seconds (SD = 15.2s) and Dask's 201.4 seconds (SD = 22.8s). The performance difference between Spark and Flink was not statistically significant ($t(4) = 1.57$, $p = 0.19$), while Dask's slower execution reflected the overhead of cross-process communication in Python's multiprocessing model.

4.3 Resource Utilization Patterns

Memory utilization profiles differed substantially across frameworks. Spark exhibited the highest peak memory consumption, averaging 26.4 GB per worker node (SD = 2.1 GB), attributable to JVM heap overhead, Resilient Distributed Dataset caching behaviour, and broadcast variable memory requirements. Flink maintained more consistent memory utilization at 19.2 GB per node (SD = 1.4 GB), benefiting from its managed memory model that allocates fixed regions for network buffers, state backend operations, and batch sorting. Dask's memory profile proved most volatile, with peak utilization ranging from 14.8 GB to 31.2 GB depending on task graph complexity and materialization timing, averaging 22.6 GB (SD = 5.8 GB).

Network I/O during TPC-DS execution averaged 847 GB for Spark, 912 GB for Flink, and 1,024 GB for Dask across the complete query suite. Flink's higher network utilization relative to Spark reflects its pipelined execution model, which streams intermediate results between operators rather than materializing complete partitions. Dask's elevated network traffic resulted from its fine-grained task scheduling, which generates additional coordination overhead for large task graphs.

4.4 Fault Tolerance and Result Consistency

Table 2 presents fault tolerance evaluation results when one worker node was terminated during mid-execution. Spark recovered from worker failure in a mean of 45.2 seconds (SD = 8.3s), leveraging its lineage-based re-computation mechanism to regenerate lost partitions without full job restart. All post-recovery results matched baseline outputs exactly (100% hash match rate). Flink demonstrated the fastest recovery at 11.8 seconds (SD = 2.1s) by resuming from the most recent distributed snapshot, preserving exactly-once processing guarantees. Result consistency for Flink matched baselines completely across all evaluated queries.

Dask exhibited markedly different recovery behaviour, requiring complete task graph re-computation with a mean recovery time of 103.7 seconds (SD = 24.6s), representing a 2.3× increase over Spark and 8.8× increase over Flink. More concerning for accuracy-critical applications, Dask produced floating-point discrepancies in 3 of 15 *K*-means clustering trials following worker recovery. These inconsistencies manifested as centroid coordinate differences at the 10^{-6} precision level, attributable to non-deterministic task execution order affecting floating-point accumulation sequences. While mathematically minor, such discrepancies may propagate in downstream analytics requiring reproducible results.

Table 2: Fault Tolerance Evaluation: Worker Node Failure Recovery

Framework	Recovery Time (s)	Recovery Mechanism	Result Consistency
Apache Spark	45.2 (8.3)	Lineage re-computation	100% match
Apache Flink	11.8 (2.1)	Checkpoint restoration	100% match
Dask	103.7 (24.6)	Task graph restart	80% match*

*Note: Values in parentheses indicate standard deviation. *Dask exhibited floating-point discrepancies at 10^{-6} precision in 3 of 15 K-means trials.*

Received: 13/03/2026; Revised: 11/04/2026; Accepted: 17/04/2026; Published: 25/04/2026

5.0 Interpretation of Performance Results

The experimental results reveal distinct performance profiles corresponding to each framework's architectural foundations. Spark's superior TPC-DS performance aligns with its mature Catalyst query optimizer, which applies rule-based and cost-based transformations including predicate pushdown, column pruning, and join reordering [16]. The Tungsten execution engine's whole-stage code generation further eliminates virtual function call overhead in tight computational loops. These optimizations have benefited from over a decade of production refinement across diverse enterprise workloads, explaining Spark's advantage in complex analytical SQL queries characteristic of business intelligence applications.

Flink's performance advantages in iterative machine learning workloads reflect fundamental differences in execution model. Unlike Spark's bulk synchronous parallel approach, which materializes intermediate results between stages, Flink's pipelined execution streams records between operators continuously [17]. For iterative algorithms like *K*-means clustering, Flink's delta iteration operators enable computation on only modified data subsets between iterations, avoiding redundant re-computation of unchanged partitions. This architectural decision explains the 22% faster *K*-means convergence observed in our experiments.

Dask's competitive performance for machine learning workloads alongside its SQL execution limitations illustrates the trade-offs inherent in its design philosophy. As a Python-native framework built around NumPy and Pandas semantics, Dask excels when workloads align with these interfaces but incurs overhead when translating relational query semantics into task graph representations. The memory volatility and query failures observed during TPC-DS evaluation suggest that Dask's current SQL support, while improving, remains less optimized than JVM-based alternatives for complex decision support workloads.

5.1 Fault Tolerance Implications

The fault tolerance evaluation results carry significant implications for production deployments where system reliability and result reproducibility are paramount. Flink's rapid checkpoint-based recovery and preserved exactly-once semantics position it as the preferred choice for applications requiring strong consistency guarantees, such as financial transaction processing and regulatory compliance analytics. The 11.8-second mean recovery time enables near-continuous operation even in environments with elevated failure rates.

Spark's lineage-based recovery provides deterministic result regeneration at the cost of longer recovery times, a trade-off appropriate for batch workloads where absolute correctness outweighs latency sensitivity. For cybersecurity forensic analysis, where analytical reproducibility supports legal evidence requirements, Spark's guaranteed result consistency may justify its slower recovery characteristics. The absence of any result deviation across all tested scenarios confirms Spark's suitability for applications where computational reproducibility is a strict requirement.

Dask's floating-point inconsistencies following recovery represent a notable concern for accuracy-sensitive applications. While the observed discrepancies at 10^{-6} precision may be acceptable for exploratory data analysis, they could propagate through downstream model training or aggregation pipelines with unpredictable effects. Organizations deploying Dask in production should implement explicit result validation mechanisms or accept that perfect reproducibility across failure events cannot be guaranteed without additional engineering effort.

5.2 Practical Deployment Recommendations

Based on the empirical evidence presented, framework selection should align with specific workload characteristics and operational requirements. For batch-oriented business intelligence and data warehousing applications involving complex SQL queries, Spark represents the optimal choice given its query optimization maturity and consistent performance across diverse query patterns. Organizations with existing investments in the Hadoop ecosystem will additionally benefit from Spark's deep integration with Hadoop Distributed File System (HDFS), Hive Metastore, and Yet Another Resource Negotiator (YARN) resource management.

For real-time stream processing applications, including cybersecurity threat monitoring, fraud detection, and Internet of Things analytics, Flink's low-latency execution and exactly-once processing guarantees provide compelling advantages. The rapid checkpoint recovery enables continuous operation in failure-prone distributed environments while maintaining strong consistency properties. Organizations should evaluate whether workload patterns justify the operational complexity of managing Flink's stateful stream processing semantics.

Dask remains well-suited for Python-centric data science teams conducting exploratory analysis, prototyping machine learning pipelines, or processing workloads that naturally decompose into NumPy and Pandas operations. Its minimal infrastructure requirements and seamless local-to-distributed scaling lower barriers to parallel computing adoption. However, organizations should exercise caution when deploying Dask for production workloads requiring strong fault tolerance guarantees or handling SQL-intensive analytical queries.

Received: 13/03/2026; Revised: 11/04/2026; Accepted: 17/04/2026; Published: 25/04/2026

5.3 Limitations

Several limitations warrant consideration when interpreting these findings. First, the experimental infrastructure used cloud-based virtualized instances, which may exhibit performance variability compared to dedicated bare-metal deployments. Second, framework configurations were selected to represent reasonable production settings but may not reflect optimal tuning for specific workloads. Third, the evaluation focused on batch and micro-batch processing; true continuous streaming scenarios with high-velocity event ingestion were not assessed. Fourth, the three-worker cluster configuration, while sufficient for comparative evaluation, does not capture scaling behaviour at datacenter scale. Finally, this study evaluated only open-source frameworks; commercial platforms such as Databricks, Cloudera, and Google Dataflow offer managed services with potentially different performance characteristics.

6.0 Conclusion

This study provides empirical evidence that distributed big data processing framework selection involves substantive trade-offs between scalability, resource efficiency, fault tolerance, and computational accuracy. No single platform dominates across all evaluated dimensions. Apache Spark demonstrated optimal performance for complex batch SQL workloads with deterministic result reproducibility, completing the most TPC-DS queries with the lowest mean execution time. Apache Flink achieved superior latency and fault recovery characteristics, with exactly-once processing semantics that preserved result consistency during worker failures. Dask offered competitive machine learning performance with minimal infrastructure requirements but exhibited memory volatility and occasional floating-point inconsistencies following fault recovery.

These findings provide actionable guidance for practitioners designing scalable analytics infrastructures. Framework selection should align with specific workload characteristics, consistency requirements, and operational constraints rather than assuming universal superiority of any single platform. For domains requiring both computational timeliness and precision, such as cybersecurity threat intelligence and financial risk analytics, understanding these trade-offs enables informed architectural decisions that balance performance optimization with result integrity requirements.

Future work will extend this evaluation to include serverless deployment models, hybrid batch-streaming workloads, and graph processing scenarios combining network analysis with natural language processing relevant to cybercrime intelligence applications. Additionally, energy consumption metrics will be incorporated to assess sustainability implications of framework selection in resource-constrained computing environments.

References

[1] Liang, W., Tadesse, G. A., Ho, D., Fei-Fei, L., Zaharia, M., Zhang, C., & Zou, J. (2022). Advances, challenges and opportunities in creating data for trustworthy AI. *Nature Machine Intelligence*, *4*(8), *669-677*.

[2] Patil, N. V., Rama Krishna, C., & Kumar, K. (2021). Distributed frameworks for detecting distributed denial of service attacks: a comprehensive review, challenges and future directions. *Concurrency and Computation: Practice and Experience*, *33*(10), e6197.

[3] Spillner, J. (2023). Slash: Serverless Apache Spark Hub. In Proceedings of the 17th ACM International Conference on Distributed and Event-based Systems, pp. 195-198.

[4] Ullah, F., Dhingra, S., Xia, X., & Babar, M. A. (2024). Evaluation of distributed data processing frameworks in hybrid clouds. *Journal of Network and Computer Applications*, *224*, 103837.

[5] Van-Dongen, G., & Van-den-Poel, D. (2020). Evaluation of stream processing frameworks. *IEEE Transactions on Parallel and Distributed Systems*, *31*(8), *1845-1858*.

[6] Naayini, P., Kamatala, S., Myakala, P. K., Bura, C., & Jonnalagadda, A. K. (2025). Revolutionizing AI workflows with distributed computing frameworks. In *2025 4th international conference on Advances in Computing, Communication, Embedded and Secure Systems (ACCESS), pp. 115-123*. IEEE.

[7] Henning, S., Vogel, A., Leichtfried, M., Ertl, O., & Rabiser, R. (2024). Shufflebench: A benchmark for large-scale data shuffling operations with distributed stream processing frameworks. *In Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, pp. 2-13*.

[8] Wang, L., Li, M., Zeng, Y., & Liu, Y. (2020). Performance evaluation of stream processing frameworks for IoT applications. *IEEE Internet of Things Journal, 7(9), 8332–8344*.

[9] Gupta, S., & Ravi, S. (2022). A survey on distributed stream processing systems. *ACM Computing Surveys, 55(3), Article 56, 1–36*.

[10] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., & Stoica, I. (2016). Apache Spark: A unified engine for big data processing. *Communications of the ACM, 59(11), 56–65*.

[11] Tardío, R., Maté, A., & Trujillo, J. (2022). Beyond TPC-DS, a benchmark for Big Data OLAP systems (BDOLAP-Bench). *Future Generation Computer Systems*, *132*, *136-151*.

Received: 13/03/2026; Revised: 11/04/2026; Accepted: 17/04/2026; Published: 25/04/2026

[12] Shi, X., Cui, B., Shao, Y., & Tong, Y. (2020). Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the 2016 International Conference on Management of Data, pp. 417–430*.
[13] Karau, H., & Kimmins, M. (2023). *Scaling Python with Dask*. " O'Reilly Media, Inc.
[14] Gueroudji, A., Bigot, J., & Raffin, B. (2021). Deisa: dask-enabled in situ analytics. In *2021 IEEE 28th international conference on High Performance Computing, Data, and Analytics (HiPC),* pp. 11-20. IEEE.
[15] Vines, A. (2024, May). Performance Evaluation of Data Vault and Dimensional Modeling: Insights from TPC-DS Dataset Analysis. In *International Conference on Informatics in Economy, pp. 27-37*. Singapore: Springer Nature Singapore.
[16] Poess, M. (2022). New initiatives in the TPC. In *Technology Conference on Performance Evaluation and Benchmarking, pp. 127-148*. Cham: Springer Nature Switzerland.
[17] Hong, Y., Du, S., & Leng, J. (2022). Evaluating Presto and SparkSQL with TPC-DS. In *International Conference on Database Systems for Advanced Applications, pp. 319-329*. Cham: Springer International Publishing.

Received: 13/03/2026; Revised: 11/04/2026; Accepted: 17/04/2026; Published: 25/04/2026