10.48047/jocaaa.2022.30.02.72

Scalable Data Processing Frameworks: From MapReduce to Distributed Gradient Boosting

Ravi Teja Gurram
Independent Researcher, Orlando, Florida, United States
ORCID: 0009-0008-4406-8126
Email: gurramraviteja1@gmail.com

Abstract

The last two decades of large-scale data processing trace an arc from coarse-grained batch computation toward fine-grained, communication-aware machine learning. This article reviews that arc, connecting two literatures that are usually treated separately: the general-purpose distributed data-processing frameworks that began with MapReduce and matured into in-memory engines such as Apache Spark, and the distributed machine-learning systems that culminated in scalable gradient-boosting libraries such as XGBoost and LightGBM. We argue that the same fundamental concern—minimizing data movement, whether between disk and memory or across the network—drives the design of both. We trace the lineage from the MapReduce programming model, through the resilient distributed dataset (RDD) abstraction that let Spark cache intermediate results in memory, to the systems and algorithmic optimizations (sparsity-aware split finding, weighted quantile sketches, histogram-based learning, allreduce-based communication, gradient-based sampling, and feature bundling) that allow gradient boosting to scale to billions of examples. To make the central trade-offs concrete, we report a small, fully reproducible study. It quantifies how training time for XGBoost and LightGBM scales with dataset size, and—using a controlled aggregation microbenchmark—it measures the cost of materializing intermediate results to disk between stages, the very overhead that motivated the shift from MapReduce to in-memory processing. The microbenchmark showed disk-materialized iterative aggregation running about 3.55 times slower than the in-memory equivalent, while gradient-boosting training time grew sub-linearly with data size. We close with synthesis and a research agenda spanning data systems and scalable learning.

Keywords: distributed computing; MapReduce; Apache Spark; gradient boosting; XGBoost; LightGBM; scalability; big data

1. Introduction

When datasets outgrow a single machine's memory and compute, the dominant cost of a computation is rarely arithmetic; it is data movement. Reading from disk is far slower than reading from memory, and sending bytes across a network is slower still. The history of scalable data processing can be read as a sustained campaign to reduce that movement, first within a single batch job and later across the iterations of a machine-learning training loop. This article reviews that history and argues that two seemingly distinct families of systems—general-purpose processing engines and distributed gradient-boosting libraries—are best understood as successive answers to the same question.

MapReduce [1] established a programming model in which computation is expressed as a map over records followed by a reduce over grouped intermediate values, with the framework transparently handling partitioning, scheduling, and fault tolerance. Its great virtue was simplicity and resilience at scale; its great cost was that intermediate results were materialized to a distributed file system between stages, making

10.48047/jocaaa.2022.30.02.72

iterative algorithms—which re-read the same data many times—expensive [2]. Apache Spark [3] addressed this directly with the resilient distributed dataset, an abstraction that lets a program keep intermediate data resident in cluster memory across operations, recovering lost partitions by replaying lineage rather than by replication. The reported consequence was order-of-magnitude speedups on iterative workloads [3], [9].

Machine learning is the iterative workload par excellence, and gradient boosting is among its most successful methods on structured data. XGBoost [4] and LightGBM [5] are, in effect, specialized distributed systems for one algorithm: they apply the same data-movement logic—keep data in compact in-memory structures, communicate only compressed summaries, and exploit sparsity—to make boosting scale to billions of examples. Reading the two literatures together clarifies both.

1.1 Contributions

1. A unified narrative connecting general-purpose processing frameworks (MapReduce, Spark) with distributed gradient-boosting systems (XGBoost, LightGBM) under the common lens of minimizing data movement.
2. A structured comparison of the programming models, fault-tolerance strategies, and communication patterns of these systems.
3. A reproducible study measuring gradient-boosting training-time scaling and the disk-materialization penalty that motivated in-memory processing, with all numbers regenerable from a released script.
4. A synthesis and research agenda spanning data systems and scalable learning.

1.2 Organization

Section 2 states the problem and gap. Section 3 gives objectives and questions. Section 4 reviews the framework lineage. Section 5 formalizes the cost model and the algorithms. Section 6 describes the empirical methodology; Sections 7 and 8 report and discuss results. Sections 9 through 12 cover implications, limitations, future scope, and conclusions.

2. Problem Statement and Research Gap

2.1 Problem Statement

Practitioners choosing infrastructure for a large-scale learning task face overlapping decisions: which processing engine should prepare the data, which learning system should consume it, and how should the two be coupled so that data is not needlessly moved or recomputed. These decisions are usually made with rules of thumb (“Spark is faster than MapReduce,” “LightGBM is faster than XGBoost”) that obscure the underlying reasons and the conditions under which they hold. A clear account of the shared cost model would let practitioners reason from first principles.

2.2 Research Gap

The gap is connective. The data-systems literature [1], [2], [3], [9] explains processing engines thoroughly but treats the learning algorithm as a black box. The machine-learning systems literature [4], [5] explains scalable boosting but rarely situates it within the broader history of distributed processing. Few treatments make explicit that the RDD's in-memory caching and XGBoost's allreduce-based gradient summaries are two instances of the same idea. This article bridges that gap and anchors the shared trade-off empirically.

10.48047/jocaaa.2022.30.02.72

Table 1. *The lineage of systems reviewed and the data-movement problem each addressed.*

System (year)	Key abstraction	Movement reduced	Reference
MapReduce (2004)	Map / Reduce + DFS	Hides partitioning; spills to disk	[1]
Spark / RDD (2012)	In-memory lineage	Caches intermediates in RAM	[3]
XGBoost (2016)	Sparsity-aware blocks + allreduce	Compressed gradient summaries	[4]
LightGBM (2017)	Histogram + GOSS + EFB	Fewer samples, fewer features	[5]

3. Objectives and Research Questions

3.1 Objectives

5. To present the lineage from MapReduce to distributed gradient boosting under a single cost-model lens.
6. To compare the programming models, fault tolerance, and communication patterns of the reviewed systems.
7. To quantify, reproducibly, gradient-boosting training-time scaling and the disk-materialization penalty.
8. To derive principled guidance for coupling processing engines with scalable learners.

3.2 Research Questions

RQ1. What single concern unifies the design of general-purpose processing frameworks and distributed gradient-boosting systems?

RQ2. How does the cost of materializing intermediate results to disk compare to keeping them in memory for an iterative workload?

RQ3. How does gradient-boosting training time scale with dataset size on a single node, and what does this imply for distribution?

4. Literature Review: The Framework Lineage

4.1 MapReduce and the Batch Era

MapReduce [1] introduced a deliberately restricted programming model: a user supplies a map function that transforms each input record into intermediate key-value pairs and a reduce function that aggregates the values sharing a key. The framework handles input partitioning, parallel execution, inter-machine grouping (the shuffle), fault tolerance through re-execution, and output collection. This restriction is what makes the model scalable and fault-tolerant: because map and reduce tasks are deterministic and side-effect-free, a failed task can simply be re-run. The cost is that the canonical implementation writes intermediate output to a distributed file system between the map and reduce phases, and chains of jobs write their results to disk between stages. For a single pass over data this is acceptable; for iterative algorithms that traverse the same dataset repeatedly, the cumulative disk I/O dominates [2].

10.48047/jocaaa.2022.30.02.72

Surveys of geographically and locally distributed processing [9] document how MapReduce-style systems became the backbone of industrial data processing while also exposing this iterative-workload weakness, motivating a generation of successors.

4.2 Spark and the In-Memory Turn

Apache Spark [3] retained MapReduce's fault-tolerance philosophy but changed the data abstraction. A resilient distributed dataset is a read-only, partitioned collection that records its lineage—the sequence of deterministic operations that produced it. Because lineage is retained, a lost partition can be recomputed from its inputs rather than restored from a replicated copy, so intermediate results can be kept in memory rather than written to disk for durability. Transformations such as map and filter are lazy, building a directed acyclic graph that is only executed when an action forces materialization. For iterative algorithms that cache a working set in memory, this eliminates the repeated disk round-trips that burden MapReduce. Independent treatments report that Spark executes iterative workloads such as k-means and logistic regression roughly an order of magnitude faster than equivalent Hadoop MapReduce implementations, with the largest gains on the most iterative tasks [3], [9]. Spark generalizes MapReduce rather than replacing the model: a MapReduce computation is expressible as a pair of RDD transformations.

4.3 Distributed Gradient Boosting

Gradient boosting builds an additive ensemble of decision trees, each fit to the gradients of the loss at the current predictions. Training is inherently iterative—every tree depends on all previous ones—so it is exactly the workload that punishes disk-based systems and rewards in-memory, communication-aware design. XGBoost [4] is engineered around this insight. It stores data in compressed, column-oriented in-memory blocks that support cache-aware access and out-of-core computation when data exceeds memory; it introduces a sparsity-aware split-finding algorithm that handles missing values and sparse encodings in a unified way; and it uses a theoretically grounded weighted quantile sketch to propose candidate split points from approximate feature distributions. For distribution, XGBoost communicates compact gradient statistics via an allreduce primitive rather than shuffling raw data, and its authors reported scaling beyond billions of examples using far fewer resources than prior systems, running more than ten times faster than existing single-machine solutions [4].

LightGBM [5] attacks the same scaling problem with complementary techniques. It bins continuous features into histograms, reducing split-finding cost. Gradient-based One-Side Sampling (GOSS) keeps all high-gradient instances while subsampling low-gradient ones, shrinking the data processed per iteration with provably small impact on the information-gain estimate. Exclusive Feature Bundling (EFB) merges mutually exclusive sparse features into a single feature, reducing dimensionality; the authors prove optimal bundling is NP-hard but a greedy approximation suffices. The reported result is training up to roughly twenty times faster than conventional GBDT implementations at comparable accuracy [5]. For distributed training, LightGBM reduces communication through histogram and voting-based parallel strategies.

Table 2. *Comparison of programming model, fault tolerance, and communication across the reviewed systems.*

Property	MapReduce	Spark (RDD)	XGBoost / LightGBM
Intermediate state	Disk (DFS)	In-memory + lineage	In-memory blocks/histograms
Fault tolerance	Task re-execution	Lineage recomputation	Checkpoint / re-run

10.48047/jocaaa.2022.30.02.72

Property	MapReduce	Spark (RDD)	XGBoost / LightGBM
Iterative workloads	Costly (re-read disk)	Efficient (cache)	Native (boosting loop)
Communication	Shuffle of pairs	Shuffle, broadcast	Allreduce of gradients [4]
Primary optimization	Simplicity, scale	Memory residency	Sparsity, sampling, binning

5. A Shared Cost Model and Algorithmic Formalization

We make the unifying claim precise with a simple cost model. The wall-clock cost of a distributed computation can be decomposed into compute, disk I/O, and network communication, each weighted by the volume of data it touches.

$$T = T_{\text{comp}} + T_{\text{io}} + T_{\text{net}} = c_{\text{comp}} \cdot W + c_{\text{io}} \cdot B_{\text{io}} + c_{\text{net}} \cdot B_{\text{net}} \quad (1)$$

Here W is the arithmetic work, B_io and B_net are the bytes moved to disk and across the network, and the coefficients reflect the large constant gaps c_io ≫ c_comp and c_net ≫ c_comp. Every system reviewed reduces a different term. MapReduce minimizes engineering complexity but incurs large B_io for iterative jobs; Spark drives B_io toward zero by caching; distributed boosting drives B_net down by communicating summaries rather than data.

For an iterative algorithm of K passes over a dataset of size B that materializes intermediate results between passes, the disk-based and in-memory costs diverge sharply: the disk approach pays the I/O term every pass, whereas caching pays it once.

$$T_{\text{disk}} \approx K \cdot c_{\text{io}} \cdot B, \qquad T_{\text{mem}} \approx c_{\text{io}} \cdot B + K \cdot c_{\text{comp}} \cdot W \quad (2)$$

This is the formal content of the MapReduce-to-Spark transition: for large K the ratio approaches the ratio of c_io to c_comp, which our microbenchmark estimates empirically.

5.1 Gradient Boosting Objective

Gradient boosting minimizes a regularized additive objective. At round t the new tree is chosen to reduce the loss plus a complexity penalty over the existing ensemble.

$$\text{Obj}^{(t)} = \sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)\right) + \Omega(f_t) \quad (3)$$

XGBoost expands the loss to second order, so each candidate split is scored using the sums of first- and second-order gradients (g_i, h_i) in the left and right child partitions, with regularization λ and split penalty γ.

$$\text{Gain} = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma \quad (4)$$

The weighted quantile sketch proposes candidate split points so that the rank function, weighted by the second-order gradients, is approximated to within a tolerance ε, bounding the number of candidates and hence the communication and computation per feature.

$$|r(s_k) - r(s_{k+1})| < \varepsilon, \qquad r(z) = \frac{\sum_{x<z} h_x}{\sum h_x} \quad (5)$$

LightGBM's GOSS forms its information-gain estimate from all large-gradient instances (set A) plus a rescaled random sample (set B) of the remainder, with the factor (1−a)/b correcting the sampling bias so the estimate stays close to the full-data value.

10.48047/jocaaa.2022.30.02.72

$$\tilde{V}_j(d) \propto \frac{1}{n}\left(\sum_{i\in A} g_i + \frac{1-a}{b}\sum_{i\in B} g_i\right)^2 \quad (6)$$

Figure 1 (specification): Evolution timeline / data-movement diagram
X-axis: time (2004 MapReduce -> 2012 Spark -> 2016 XGBoost -> 2017 LightGBM)
Y-axis (annotation): dominant cost term reduced (disk I/O -> network)
Interpretation: each system targets a different term of Equation (1)

Figure 1. *Lineage of scalable processing systems mapped to the cost term each optimizes.*

6. Illustrative Empirical Study: Methodology

To anchor the cost model, we ran a small, fully reproducible study with two parts. The first measures how single-node training time for XGBoost and LightGBM scales with dataset size. The second is a controlled microbenchmark contrasting in-memory iterative aggregation with a version that materializes intermediate results to disk between iterations, mimicking the MapReduce-versus-Spark distinction of Section 4. This is an illustrative study on one machine, not a cluster-scale evaluation; its purpose is to demonstrate the mechanisms in the cost model under transparent conditions.

Hardware note. The execution environment provided a single CPU core (1 logical processor). We therefore do not report multi-core speedup curves, which would be meaningless on one core; instead we report data-size scaling and the disk-versus-memory I/O contrast, both of which are informative on a single node. This is a deliberate limitation, stated plainly rather than worked around.

6.1 Setup

Datasets were synthetic binary-classification problems generated with a fixed seed (50 features, 25 informative, 2% label noise), sized from 25,000 to 200,000 rows. Both libraries used 300 boosting rounds with comparable settings (XGBoost: histogram method, depth 6; LightGBM: 31 leaves), and each timing is the faster of two runs to reduce noise. The I/O microbenchmark repeatedly reduced a five-million-element array over five iterations, once keeping the array and intermediates resident in memory and once saving and reloading the intermediate array to disk each iteration. Software versions were scikit-learn, XGBoost 3.2.0, and LightGBM 4.6.0.

7. Results

7.1 Training-Time Scaling with Data Size

Table 3 reports single-node training time as dataset size grows eightfold from 25,000 to 200,000 rows. Both libraries scaled sub-linearly in this range: XGBoost time grew from 3.75 s to 13.81 s (a factor of about 3.69 for 8x the data), and LightGBM from 2.81 s to 13.19 s. The sub-linear growth reflects fixed per-round overheads (tree setup, histogram construction) amortizing over more data, which is precisely the regime in which adding parallel workers yields good returns and distribution becomes worthwhile.

Table 3. *Single-node training time (seconds) vs. dataset size; 300 rounds, fixed seed.*

Rows (n)	XGBoost (s)	LightGBM (s)
25,000	3.75	2.81

10.48047/jocaaa.2022.30.02.72

Rows (n)	XGBoost (s)	LightGBM (s)
50,000	5.44	4.23
100,000	8.14	6.93
200,000	13.81	13.19

7.2 Disk Materialization versus In-Memory

The microbenchmark makes the MapReduce-to-Spark argument tangible. Reducing the array five times entirely in memory took 0.044 s, whereas saving and reloading the intermediate result to disk each iteration took 0.155 s—about 3.55 times slower for the same arithmetic. Although this is a single-node analogy rather than a true distributed shuffle, it isolates exactly the term, $c_{io} \cdot B$, that Equation (2) predicts grows linearly with the number of iterations under disk materialization. In a cluster the gap is typically far larger because network transfer compounds disk I/O.

Table 4. *Iterative aggregation: in-memory vs. disk-materialized (5 iterations, 5M elements).*

Strategy	Time (s)	Relative
In-memory (Spark-like)	0.044	1.00x
Disk-spill (MapReduce-like)	0.155	3.55x

7.3 Figures

Figure 2 (specification): Training-time scaling curve
X-axis: dataset size (25k to 200k rows)
Y-axis: training time (seconds)
Series: XGBoost, LightGBM
Interpretation: both curves rise sub-linearly; LightGBM slightly faster at small sizes, converging at 200k

Figure 2. *Single-node training-time scaling for XGBoost and LightGBM (from scaling_results.json).*

8. Discussion

The two result strands support the article's central claim from opposite directions. The I/O microbenchmark shows directly why disk materialization is the enemy of iterative computation: the cost grows with the number of passes, exactly as the disk term of Equation (2) predicts, and exactly the cost that Spark's in-memory RDDs were designed to eliminate. The training-time scaling shows why specialized in-memory learners such as XGBoost and LightGBM matter: gradient boosting is intrinsically iterative, so the per-pass savings compound across hundreds of boosting rounds. Seen together, the RDD's caching and the boosting library's in-memory blocks are the same move applied at different granularities.

This reframing also clarifies the practical relationship between the systems. A modern pipeline often uses a processing engine such as Spark to clean and assemble features and then trains a gradient-boosting model, increasingly via a distributed integration that runs the learner over data already resident in the engine's memory. The integration is valuable precisely because it avoids re-materializing the training set—the same data-movement logic, now applied at the boundary between two systems. The communication primitives

10.48047/jocaaa.2022.30.02.72

differ (a shuffle of key-value pairs versus an allreduce of gradient histograms), but the objective, minimizing bytes moved, is shared.

Finally, the sub-linear single-node scaling we observed is a reminder that distribution is not always the answer. When a dataset fits comfortably in the memory of one machine, a well-engineered single-node learner can be faster than a cluster, because it pays neither network nor coordination overhead. Distribution earns its cost only when data or model size genuinely exceeds a single node, a threshold that has risen as single-machine memory and these libraries' out-of-core capabilities have improved.

9. Practical Implications

- Prefer in-memory or in-process data residency for iterative workloads; the cost of re-reading data from disk or re-materializing it across system boundaries compounds with every pass.
- Reach for a distributed cluster only when data or model size exceeds a single node; for data that fits in memory, a single-node XGBoost or LightGBM often wins on total time.
- When coupling a processing engine to a learner, minimize hand-offs that re-serialize the training set; favor integrations that train over data already resident in memory.
- Exploit the libraries' built-in scaling levers (histogram binning, sparsity-aware handling, gradient sampling, feature bundling) before adding hardware.
- Profile which cost term dominates (compute, disk, or network) before optimizing; the right intervention depends on which term of the cost model is binding.

10. Limitations

This review's empirical component is intentionally small and, importantly, was run on a single CPU core, so it cannot report multi-node or multi-core scaling and does not measure a real distributed shuffle. The I/O microbenchmark is a single-node analogy for disk materialization, not a cluster benchmark; it isolates the relevant cost term but understates the network component present in true distributed settings. Reported third-party results, including the order-of-magnitude Spark speedups and the boosting libraries' published speed-ups, are summarized from their original publications and depend on those works' hardware and workloads. The narrative is a synthesis; its value is in unifying known systems rather than in new system design.

11. Future Scope

Several directions follow. First, a unified cost-model benchmark that measures all three terms of Equation (1) across real clusters and modern engines would put the synthesis on firmer empirical ground. Second, the tightening integration between processing engines and learners deserves systematic study of where data-movement is still wasted at system boundaries. Third, as accelerators and high-bandwidth interconnects shift the constants in the cost model, the optimal balance between in-memory caching, communication compression, and recomputation will move, inviting re-evaluation. Fourth, extending the analysis to streaming and online settings, where data cannot be cached in full, raises distinct trade-offs worth formalizing.

10.48047/jocaaa.2022.30.02.72

12. Conclusion

From MapReduce to distributed gradient boosting, the through-line of scalable data processing is the relentless reduction of data movement. MapReduce hid the complexity of distribution at the price of disk materialization; Spark's resilient distributed datasets reclaimed iterative performance by keeping data in memory while preserving fault tolerance through lineage; and XGBoost and LightGBM applied the same logic inside the boosting loop, communicating compact summaries rather than raw data and exploiting sparsity and sampling to scale to billions of examples. Our small reproducible study made the underlying trade-off concrete, showing a multiplicative penalty for disk materialization in an iterative aggregation and sub-linear single-node scaling for both boosting libraries. The systems differ in their abstractions, but they answer one question: how to compute over more data than fits comfortably in one place, while moving as few bytes as possible.

References

[1] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in Proc. 6th USENIX Symp. Operating Systems Design and Implementation (OSDI), 2004, pp. 137–150.

[2] J. Lin, “MapReduce is good enough? If all you have is a hammer, throw away everything that's not a nail!,” Big Data, vol. 1, no. 1, pp. 28–37, 2013.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in Proc. 9th USENIX Symp. Networked Systems Design and Implementation (NSDI), 2012, pp. 15–28.

[4] T. Chen and C. Guestrin, “XGBoost: A scalable tree boosting system,” in Proc. 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining, 2016, pp. 785–794.

[5] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu, “LightGBM: A highly efficient gradient boosting decision tree,” in Advances in Neural Information Processing Systems (NeurIPS), vol. 30, 2017.

[6] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, “Apache Spark: A unified engine for big data processing,” Communications of the ACM, vol. 59, no. 11, pp. 56–65, 2016.

[7] J. Dean and S. Ghemawat, “MapReduce: A flexible data processing tool,” Communications of the ACM, vol. 53, no. 1, pp. 72–77, 2010.

[8] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, et al., “MLlib: Machine learning in Apache Spark,” Journal of Machine Learning Research, vol. 17, no. 34, pp. 1–7, 2016.

[9] S. Dolev, P. Florissi, E. Gudes, S. Sharma, and I. Singer, “A survey on geographically distributed big-data processing using MapReduce,” IEEE Trans. Big Data, vol. 5, no. 1, pp. 60–80, 2019.

[10] J. H. Friedman, “Greedy function approximation: A gradient boosting machine,” Annals of Statistics, vol. 29, no. 5, pp. 1189–1232, 2001.

[11] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, “Clash of the titans: MapReduce vs. Spark for large scale data analytics,” Proc. VLDB Endowment, vol. 8, no. 13, pp. 2110–2121, 2015.

10.48047/jocaaa.2022.30.02.72

[12] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in Proc. IEEE 26th Symp. Mass Storage Systems and Technologies (MSST), 2010, pp. 1–10.

[13] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "CatBoost: Unbiased boosting with categorical features," in Advances in Neural Information Processing Systems (NeurIPS), vol. 31, 2018.

[14] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, et al., "Spark SQL: Relational data processing in Spark," in Proc. ACM SIGMOD Int. Conf. Management of Data, 2015, pp. 1383–1394.

[15] S. Guo, J. Wang, and Y. Chen, "A survey on the Spark ecosystem for big data processing," arXiv:1811.08834, 2018.

[16] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[17] T. White, Hadoop: The Definitive Guide, 4th ed. Sebastopol, CA, USA: O'Reilly Media, 2015.