International Journal of Computer Techniques–IJCT Volume 8 Issue 1, February - 2021

Open Access and Peer Review Journal ISSN 2394-2231 https://ijctjournal.org/

Lakehouse Architecture: Unifying Data Lakes and Data Warehouses

Jeevan Krishna Paruchuri

Independent Researcher
paruchuri.g167@gmail.com

Abstract—Modern data-driven organizations have historically operated two parallel storage systems: a data warehouse for governed analytical workloads and a data lake for raw, semi-structured, and machine-learning-oriented data. This two-system architecture introduces three direct problems: data duplication across stores, brittle extract-transform-load (ETL) pipelines coupling them, and governance discontinuities at the boundary. The cumulative consequence is significant operational overhead. The lakehouse paradigm, an architectural approach that has emerged in academic and industrial literature in the late 2010s and crystallized around 2020, attempts to unify these systems by layering transactional table semantics, schema enforcement, and governance directly onto inexpensive object storage. This paper presents a systematic survey of lakehouse architecture proposals, the open table formats that enable them Delta Lake, Apache Iceberg, and Apache Hudi and early adoption reports from academic and industry literature published through 2020. This survey makes three contributions: (1) a taxonomy of lakehouse design patterns organized along five architectural layers; (2) a structured comparison of the three dominant open table formats (Delta Lake, Apache Iceberg, Apache Hudi) against analytical, ML, and streaming workload requirements; and (3) a set of open research challenges informed by the author's experience deploying a Delta Lake-based lakehouse in a production banking environment. Practitioner observations show that storage-side optimizations dominate cost in real deployments, that small-file fragmentation is the primary query-performance bottleneck for streaming workloads, and that the table format alone does not eliminate governance debt. The paper concludes by arguing that lakehouse research must shift from benchmark-scale experiments toward production-scale governance, compaction, and cost-modeling problems.

Index Terms—lakehouse, data lake, data warehouse, Delta Lake, Apache Iceberg, Apache Hudi, open table format, cloud object storage

I. INTRODUCTION

Enterprise analytical infrastructure has undergone three architectural epochs over the past four decades. The first epoch, dominated by the relational data warehouse, established the practice of consolidating operational data into a centrally governed repository structured around dimensional models [1], [2]. Data warehouses provided strong consistency guarantees, mature SQL tooling, and fine-grained access control, but they imposed rigid schemas, expensive proprietary storage, and a centralized ETL bottleneck that became increasingly difficult to scale as data volumes and source diversity grew.

The second epoch, driven by the economics of commodity storage and the emergence of distributed processing frameworks such as Hadoop and Apache Spark [3], [4], introduced the data lake. Data lakes embraced schema-on-read, separation of storage from compute, and the ability to retain raw data of any format at low cost. They

enabled new categories of workload exploratory analytics, machine learning feature engineering, and large-scale log processing that the warehouse could not economically accommodate. However, the absence of transactional semantics, schema enforcement, and uniform governance produced what practitioners came to describe as the data swamp: large volumes of poorly catalogued, partially trusted data that resisted reliable analytical use [5], [6].

Faced with the limitations of each, most large enterprises adopted both. Data was landed in the lake, partially curated, and then loaded into the warehouse for downstream consumption. The result is a two-system architecture that duplicates data across tiers, fragments governance, and requires substantial engineering effort to keep the two systems in sync. The two-system pattern has been documented across industries, including financial services, where regulatory reporting requirements demand strong governance even on data originally landed in lake storage [5].

The lakehouse paradigm proposes to collapse this two-system architecture into a single tier by adding transactional table semantics, schema enforcement, and governance directly to object storage [9], [13]. The paradigm depends on a set of open table formats Delta Lake, Apache Iceberg, and Apache Hudi that maintain transactional metadata alongside Parquet or ORC data files in cloud object stores such as Amazon S3, Azure Data Lake Storage Gen2, and Google Cloud Storage. These formats provide ACID transactions, schema evolution, time travel, and snapshot isolation on data that remains directly accessible to multiple query engines without proprietary ingestion.

This paper surveys the lakehouse paradigm as it stood at the close of 2020, when the term itself was newly coined and the supporting technologies were undergoing rapid maturation. The survey is organized around three research questions:

RQ1: What architectural patterns define the lakehouse paradigm, and how do they differ from prior data lake and data warehouse designs?

RQ2: How do open table formats (Delta Lake, Apache Iceberg, Apache Hudi) enable lakehouse guarantees, and what are their comparative strengths?

RQ3: What open research challenges remain for lakehouse adoption in enterprise environments?

The remainder of this paper is organized as follows. Section 2 provides background on data warehouses, data lakes, and the two-system problem. Section 3 documents the survey methodology, including search strategy and inclusion criteria. Section 4 presents a taxonomy of lakehouse design patterns organized into five architectural layers. Section 5 compares Delta Lake, Apache Iceberg, and Apache Hudi across seven technical dimensions. Section 6 reports practitioner observations from a production banking lakehouse deployment, contributing concrete evidence about cost distribution, file fragmentation, and governance debt. Section 7 identifies open research challenges. Section 8 concludes.

II. BACKGROUND

2.1 Data Warehouse Architecture

The data warehouse, formalized in the work of Inmon and Kimball, organizes business data into subject-oriented, integrated, time-variant, and non-volatile structures intended to support management decision-making [1], [2]. The dominant logical design pattern is the dimensional model, which separates immutable measurements (fact tables) from descriptive context (dimension

tables) and exposes them through star or snowflake schemas optimized for online analytical processing (OLAP) workloads [12]. Physical implementations have included row-store relational systems, columnar OLAP appliances such as Teradata and Vertica, and cloud-native columnar warehouses such as Amazon Redshift, Google BigQuery, and Snowflake [7].

Warehouses excel at governed, SQL-oriented analytical workloads. Schema enforcement is strict; access control is mature; query optimizers are highly tuned; and consistency guarantees are strong. The cost of these guarantees is rigidity. Schema changes are expensive, semi-structured data is awkward to accommodate, and proprietary storage formats restrict the range of compute engines that can read the data. Storage and compute are tightly coupled in many traditional warehouses, although cloud-native systems such as Snowflake have decoupled them while retaining proprietary storage [7].

2.2 Data Lake Architecture

The data lake emerged in response to the growing volume, variety, and velocity of enterprise data. The architecture, popularized by the Hadoop ecosystem and later by cloud object storage, retains raw data in its original form on inexpensive distributed file systems and applies schema only at read time [8], [6]. Storage is decoupled from compute, allowing organizations to scale capacity and processing independently and to retain large historical datasets at marginal cost.

Lakes accommodate workloads that warehouses cannot: ingestion of semi-structured logs and clickstreams, retention of raw machine learning training data, and exploratory analysis where the schema is not known in advance. They have become foundational infrastructure for data science and machine learning pipelines. However, the absence of structural enforcement creates well-documented operational problems. The small-file problem accumulation of millions of small files due to high-frequency ingestion or fine-grained partitioning degrades query performance and increases metadata overhead [5]. Metadata management is fragmented, often relying on the Hive Metastore or custom catalogs that fail to capture lineage, ownership, and freshness reliably. Without ACID semantics, concurrent writers can corrupt data, and failed jobs leave partial results that downstream consumers cannot distinguish from correct outputs [9].

2.3 The Two-System Problem

Most enterprises that adopted data lakes did not retire their data warehouses. Instead, they evolved a two-system architecture in which raw data is landed in the lake, partially processed, and then loaded into the warehouse for governed analytical use. The two systems serve complementary roles the lake for low-cost retention and machine learning, the warehouse for governed business intelligence but their coexistence introduces several problems. Data is duplicated across the tiers, increasing storage cost and creating consistency risk when the two copies diverge. The ETL pipelines that move data from lake to warehouse represent a brittle integration layer subject to schema drift, late-arriving data, and reprocessing complications. Governance is applied inconsistently: warehouses typically enforce access control and lineage, but the lake retains the raw source of truth without comparable controls. Data scientists working from the lake and analysts working from the warehouse often produce divergent metrics for the same underlying business question [5].

2.4 Emergence of the Lakehouse

The lakehouse paradigm proposes to eliminate the two-system architecture by adding warehouse-grade guarantees ACID transactions, schema enforcement, governance, and time travel directly to data lake storage. The term itself was popularized by Databricks in 2020 [13], although the underlying technologies predate the term. Delta Lake was released as an open-source project in 2019 [9]. Apache Iceberg, developed at Netflix to address petabyte-scale table management problems on Amazon S3, was donated to the Apache Software Foundation in 2018 and graduated to a top-level project in 2020. Apache Hudi, developed at Uber for incremental data ingestion, followed a similar trajectory. Each of these technologies provides transactional table semantics on top of immutable file formats (typically Parquet) stored in cloud object storage, but they differ in design choices, target workloads, and ecosystem maturity.

III. METHODOLOGY

This survey was conducted as a structured literature review of academic and industry publications addressing lakehouse architecture and its constituent technologies. Because the term lakehouse itself was only formalized in 2020, the search strategy was designed to capture both the emergent literature using the term explicitly and the foundational work on open table formats, cloud data warehousing, and data lake management that anticipated the paradigm.

Sources were drawn from four databases: IEEE Xplore, the ACM Digital Library, Scopus, and arXiv (for preprints). Search queries combined the terms lakehouse, Delta Lake, Apache Iceberg, Apache Hudi, ACID data lake, and transactional data lake with related terms including object storage, schema evolution, and time travel. Industry whitepapers from Databricks, Netflix, Uber, and the Apache Software Foundation were included where they presented architectural specifications or production experience reports. The reference cutoff for inclusion was December 2020, consistent with the goal of capturing the lakehouse paradigm at the moment of its emergence rather than its later maturation.

Inclusion criteria required that a publication either present an architectural proposal or experimental evaluation related to lakehouse, transactional data lakes, or open table formats; or provide foundational context on data warehousing, data lake management, or cloud storage architecture relevant to the lakehouse paradigm. Exclusion criteria removed publications addressing only narrow performance benchmarks without architectural discussion, publications focused exclusively on proprietary cloud warehouses without lakehouse implications, and publications whose only mention of relevant terms was incidental.

Because the lakehouse paradigm was emergent at the time of the survey, the corpus is necessarily smaller and more weighted toward foundational and practitioner-oriented sources than would be expected for a mature research area. The survey is best read as a snapshot of the paradigm at its inception, supplemented by direct production experience reported in Section 6.

IV. A TAXONOMY OF LAKEHOUSE DESIGN PATTERNS

Lakehouse architectures can be decomposed into five logical layers: a storage layer, a table format layer, a compute layer, a governance layer, and a serving layer. Each layer corresponds to a distinct set of design decisions and implementation technologies, and the layered structure provides a

framework for comparing alternative lakehouse implementations.

4.1 Storage Layer

The storage layer of a lakehouse is invariably an object store: Amazon S3, Azure Data Lake Storage Gen2, Google Cloud Storage, or an on-premises equivalent such as MinIO. Object storage provides durable, highly available, and inexpensive persistence with effectively unbounded capacity, and its API surface simple PUT, GET, LIST, and DELETE operations accommodates the immutable file model on which lakehouse table formats depend. Critically, object storage decouples persistence from compute: the same data can be read by multiple query engines running on different clusters without the data being copied or migrated. This separation is the economic precondition for the lakehouse paradigm, since it permits storage to scale at the marginal cost of object storage while compute scales independently in response to query load.

Object storage also imposes constraints. List operations on directories with millions of objects can be slow and expensive, motivating the metadata-centric designs of lakehouse table formats that avoid frequent directory listings. Eventual consistency in early object stores complicated transactional designs, although strong read-after-write consistency has since become the norm in major cloud providers. Authentication and access control are typically implemented at the bucket or container level, requiring careful account-level segregation when fine-grained controls are needed across data domains.

4.2 Table Format Layer

The table format layer is the defining innovation of the lakehouse paradigm. A table format is a specification for organizing immutable data files together with transactional metadata such that ACID semantics, schema evolution, snapshot isolation, and time travel can be implemented over object storage. The three dominant open table formats Delta Lake, Apache Iceberg, and Apache Hudi share the basic approach of maintaining a transaction log or manifest that records the set of files comprising each version of a table, but they differ in metadata structure and concurrency control.

Delta Lake represents table state as an ordered transaction log of JSON-encoded actions stored alongside the data files [9]. Each commit appends a new log entry, and readers reconstruct the current snapshot by replaying the log. Apache Iceberg organizes metadata hierarchically into manifest lists and manifest files, enabling efficient pruning and snapshot management at petabyte scale. Apache Hudi distinguishes between copy-on-write tables, in which updates rewrite affected files, and merge-on-read tables, in which updates are appended as delta logs and merged at read time, optimizing for ingestion-heavy workloads. All three formats use Parquet as their default data file format and provide programmatic interfaces for Spark, with varying degrees of support for other engines.

4.3 Compute Layer

The compute layer comprises the query engines that read and write tables stored in the lakehouse. The defining property of the lakehouse paradigm is that the compute layer is not bound to a single engine: because the table format is open and the data files use standard formats such as Parquet, multiple engines can operate on the same data. Apache Spark is the most widely supported engine across all three table formats and is typically used for ETL, machine learning, and large-scale batch processing [4]. Trino and Presto provide low-

latency interactive SQL [10]. Apache Flink supports streaming workloads with stateful processing [11]. Dremio offers a BI-oriented SQL engine designed for direct query against lake storage.

The decoupling of compute from storage enables workload-specific engine selection: a streaming pipeline can use Flink to write to a Hudi table, an ETL pipeline can use Spark to transform a Delta Lake table, and an analyst can query the result through Trino without data movement. This pattern stands in sharp contrast to the proprietary warehouse, in which compute and storage are coupled within a single vendor system.

4.4 Governance Layer

The governance layer encompasses schema enforcement, access control, lineage tracking, audit logging, and data quality enforcement. In the lakehouse paradigm, some governance functions are implemented within the table format itself (schema enforcement and evolution, snapshot history for limited time-travel-based audit), while others depend on external catalog and policy systems. The Hive Metastore, AWS Glue Data Catalog, and Project Nessie are examples of catalogs used to register lakehouse tables and expose them to multiple engines. Access control is typically delegated to the underlying object storage's identity and access management system, which limits the granularity of policies that can be applied below the table or partition level.

Governance remains the least mature layer of the lakehouse stack. Schema enforcement and evolution are well supported, but lineage tracking, automated data quality, retention enforcement, and column-level access control depend on tooling outside the table format specification. Practitioner experience reported in Section 6 confirms that the table format alone does not solve governance: ownership, freshness, and retention discipline must still be enforced through organizational and operational practice.

4.5 Serving Layer

The serving layer exposes lakehouse data to downstream consumers, including business intelligence tools, dashboards, and operational applications. Because lakehouse tables are queryable by standard SQL engines, BI tools that connect via JDBC or ODBC including Tableau, Power BI, and Looker can issue queries directly against the lakehouse without an intermediate warehouse load. SQL endpoints provided by engines such as Trino, Spark Thrift Server, and Apache Kyuubi expose lakehouse tables as if they were warehouse tables, supporting standard authentication and connection pooling. Federated query patterns allow a single SQL statement to combine lakehouse tables with external sources, although the performance characteristics of such federation vary substantially across implementations.

V. COMPARATIVE ANALYSIS OF OPEN TABLE FORMATS

Delta Lake, Apache Iceberg, and Apache Hudi share the basic architectural commitment of providing ACID semantics over object storage, but they differ in design choices that reflect the distinct production environments in which they originated. Delta Lake originated at Databricks for use with Apache Spark and is closely integrated with the Databricks Runtime [9]. Apache Iceberg originated at Netflix to manage petabyte-scale tables on Amazon S3 with multiple query engines. Apache Hudi originated at Uber to support incremental ingestion of operational database changes into a data lake. This section compares the three formats across seven technical

dimensions, drawing on the design papers, project documentation, and code as published through 2020.

5.1 Transaction Model

Delta Lake implements optimistic concurrency control using a serialized transaction log: each commit attempts to append to the log, and conflicts are detected and resolved by retry [9]. The default storage strategy is copy-on-write, in which updates rewrite affected data files. Apache Iceberg also uses optimistic concurrency control with snapshot isolation, organizing metadata into a hierarchy of manifest lists and manifest files that are atomically swapped on commit. Iceberg's design separates the table state from the underlying directory layout, enabling more efficient handling of partition evolution. Apache Hudi distinguishes copy-on-write tables, which behave similarly to Delta and Iceberg, from merge-on-read tables, which append updates as delta log files and merge them at query time. The merge-on-read mode reduces write amplification for high-frequency update workloads at the cost of read complexity.

5.2 Schema Evolution

All three formats support adding columns, renaming columns, and changing column types under restricted conditions. Iceberg's schema evolution is generally regarded as the most complete because the format assigns a stable identifier to each column independent of its name, allowing safe renaming and reordering without rewriting data. Delta Lake supports add and rename operations through schema mode flags. Hudi supports schema evolution within the constraints of its record key model. None of the formats fully solve the problem of evolving partition columns or changing partition strategies, although Iceberg's hidden partitioning feature provides a partial answer by allowing partition transformations to evolve without requiring rewrites.

5.3 Time Travel and Snapshot Management

Time travel the ability to query a table as of a previous version or timestamp is supported by all three formats. Delta Lake's transaction log naturally supports time travel by replaying the log to a specified version. Iceberg's snapshot model provides time travel as a primary feature, with explicit APIs for managing snapshots. Hudi supports time travel through its incremental query model, which is also used to feed downstream consumers with changed records. Snapshot retention is a critical operational concern: as discussed in Section 6, transaction logs and historical snapshots accumulate quickly in production and must be pruned to control storage costs.

5.4 Compaction and Small File Management

Streaming ingestion produces many small files, which degrade query performance and inflate metadata costs. All three formats provide mechanisms for compaction. Delta Lake exposes an OPTIMIZE command that rewrites small files into larger files targeting a configurable size [9]. Iceberg supports rewrite operations through its actions API. Hudi includes inline and asynchronous compaction processes that can run alongside ingestion. The choice of target file size, the frequency of compaction, and the tradeoff between compaction cost and query benefit are operational decisions that depend on workload characteristics and are discussed at length in Section 6.

5.5 Cross-Engine Compatibility

As of 2020, Delta Lake's most mature integration was with Apache Spark on the Databricks Runtime, with Spark open-source support also available and limited support for other engines emerging. Iceberg was designed from the outset for cross-engine compatibility and supports Spark, Trino, Hive, and Flink, with active development on additional engines. Hudi supports Spark and Hive natively and has integration with Flink for streaming use cases. Cross-engine compatibility is a significant differentiator: organizations adopting a multi-engine strategy face fewer integration challenges with Iceberg, while organizations standardizing on Spark may find Delta Lake's integration depth advantageous.

5.6 Metadata Scalability

At very large scale tables with hundreds of thousands of partitions or billions of files metadata operations become the dominant performance concern. Iceberg's hierarchical manifest design was specifically engineered for this regime and supports efficient partition pruning and metadata-only query planning. Delta Lake's transaction log is compact and performs well at moderate scale, with checkpointing used to bound log replay cost. Hudi's metadata table, introduced as a feature to address metadata scalability, brings similar benefits to Hudi tables. The differences become significant only at the upper end of the scale spectrum; for most enterprise workloads, all three formats perform adequately.

5.7 Community and Ecosystem Maturity

As of late 2020, Delta Lake had the most visible commercial backing through Databricks and the largest installed base of Spark-centric users. Iceberg had achieved Apache top-level project status and was gaining adoption among organizations seeking engine independence, particularly Netflix, Apple, and Adobe. Hudi had a strong following in organizations with high-velocity ingestion workloads originating in operational databases, anchored by its origins at Uber. All three projects were under active development, and the relative positioning has continued to shift since 2020. Table 1 summarizes the comparison.

Table 1: Comparison of Open Table Formats (as of 2020)

Dimension	Delta Lake	Apache Iceberg	Apache Hudi
Origin	Databricks (2019)	Netflix (2018)	Uber (2017)
Transaction model	Optimistic CC, log-based, copy-on-write	Optimistic CC, snapshot manifests	Copy-on-write or merge-on-read
Schema evolution	Add, rename via flags	Most complete; column IDs	Constrained by record key
Time travel	Log replay to version	Native snapshot API	Incremental query model
Compaction	OPTIMIZE command	Rewrite actions API	Inline / async compaction
Cross-engine support	Spark-centric (2020)	Spark, Trino, Hive, Flink	Spark, Hive, Flink
Metadata scalability	Log + checkpoints	Hierarchical manifests	Metadata table
Primary workload fit	Spark ETL, ML	Multi-engine analytics	High-velocity ingestion
Community (2020)	Databricks-backed	Apache TLP, multi-vendor	Apache TLP, Uber-anchored

VI. PRACTITIONER PERSPECTIVE: PRODUCTION DEPLOYMENT OBSERVATIONS

This section reports observations from a production lakehouse deployment in a financial services environment. The platform serves customer data, financial transactions, and risk and compliance workloads, and was migrated from a legacy Oracle-based batch architecture to a Delta Lake-based lakehouse on Azure Data Lake Storage Gen2 over a period of approximately nine months. The platform team grew from four to twelve engineers during the migration and was structured into domain-aligned squads (customer data, financial transactions, risk and compliance) supported by a platform squad. The compute and ingestion stack comprises Apache Spark, Apache Kafka with Debezium change data capture, Azure Data Factory for orchestration of straightforward pipelines, Apache Airflow for complex orchestration, Azure Kubernetes Service, Apache Kyuubi as a SQL gateway, and Redis and ONNX Runtime for online feature serving and model inference. The observations below are framed as practitioner reports rather than controlled experiments and are intended to motivate the research challenges identified in Section 7.

6.1 The Small File Problem in Streaming Ingestion

The most consequential operational issue encountered in the deployment was small file accumulation. Streaming jobs were initially configured to commit at four-minute intervals, which produced average file sizes of approximately 200 KB per commit. On worst-case tables those with high partition cardinality and modest per-partition write volume the number of files grew to more than 480,000 per table within months. Query latency on these tables degraded substantially, and metadata operations such as table refresh and partition discovery became expensive. The remediation combined two changes: the streaming write interval was increased from four minutes to thirty minutes, raising the average post-write file size, and a Spark OPTIMIZE process was scheduled to compact existing files toward a target size of 256 MB. After compaction, file counts on the worst-case tables fell to under 2,000, average file size rose to approximately 180 MB, and query performance on the same workloads improved by a factor of approximately six. The episode confirms that streaming write cadence and target file size are first-order operational parameters, and that the cost of poor defaults compounds quickly.

6.2 Snapshot Retention and Transaction Log Growth

Delta Lake retains historical snapshots according to a configurable retention policy, and time travel queries depend on the availability of these snapshots. In the deployment, no explicit retention policy had been set, with the result that transaction logs accumulated approximately fourteen months of snapshots before the omission was identified. The accumulated metadata and obsolete data file references contributed materially to storage cost and to the time required for table maintenance operations. Setting an explicit thirty-day retention window and running VACUUM on a scheduled basis reduced both storage consumption and metadata processing time. The episode illustrates that the lakehouse table format introduces operational concerns snapshot retention and metadata maintenance that have no exact analogue in traditional warehouses and that must be explicitly managed.

6.3 Cost Distribution: Storage Versus Compute

A common assumption in lakehouse adoption is that compute will dominate cost, given the elasticity of cloud query engines and the relatively low unit cost of object storage. Observations from

this deployment contradicted that assumption. After approximately three months of operation, a cost audit attributed roughly 40 percent of total platform cost to storage and roughly 60 percent to compute, but more importantly, the dominant compute cost was attributable to queries scanning fragmented storage rather than to inherently expensive workloads. Storage-side optimizations compaction, snapshot cleanup, and streaming file size adjustment delivered an aggregate 40 percent cloud cost reduction over the same period, decomposed approximately as 15 percent from compaction and query efficiency improvements, 12 percent from snapshot cleanup, 8 percent from the streaming file size fix, and 5 percent from miscellaneous optimizations. The observation that storage-side optimizations dominated cost reduction was counterintuitive to the team's initial assumption that compute tuning would be the primary lever, and motivates the research challenge of cost modeling discussed in Section 7.

6.4 Migration from Oracle Batch to Streaming Lakehouse

The migration from a legacy Oracle batch architecture, originally orchestrated through SQL Server Integration Services, to the Kafka-fed lakehouse required approximately nine months of engineering effort. The initial snapshot of source tables, totaling approximately 800 million rows, took roughly fourteen hours to complete. Ongoing change data capture used Oracle LogMiner with supplemental logging enabled, which added an estimated three to four percent to redo log volume on the source database a non-trivial overhead that had to be negotiated with the database administration team. Once steady-state operation was achieved, end-to-end latency from source commit to lakehouse availability improved from approximately twenty-four hours under the legacy batch architecture to under two minutes. The latency improvement was the principal user-visible benefit of the migration and enabled new categories of near-real-time analytical and operational use cases.

6.5 Orchestration and the Limits of a Single Tool

Azure Data Factory was initially selected as the primary orchestration tool because of its tight integration with the Azure ecosystem and its low barrier to entry for less specialized engineers. After approximately 143 pipelines had been built, the team encountered substantial difficulty managing complex inter-pipeline dependencies, conditional execution, and dynamic parameterization. Apache Airflow was added alongside Data Factory to handle the more complex orchestration logic, while Data Factory was retained for simpler ingestion patterns. The episode illustrates a common pattern in lakehouse adoption: tools that are excellent for the median pipeline can become limiting for the long tail of complex pipelines, and a hybrid orchestration strategy may be preferable to forcing all workloads into a single tool.

6.6 Persistent Governance Debt

A storage audit conducted approximately one year into operation found that 23 percent of stored data had not been read in twelve or more months, and that 31 percent of registered tables had no documented owner. Both findings reflect governance debt that the lakehouse table format does not solve. Schema enforcement and ACID semantics ensure that data is internally consistent, but they do not ensure that data is needed, owned, or maintained. The governance gap is not a failure of the lakehouse paradigm but a reminder that the paradigm addresses one set of problems storage and compute unification while leaving another set organizational governance discipline as work for

human and process design. This observation motivates the research challenges of governance-native table formats and lifecycle automation discussed in Section 7.

VII. OPEN RESEARCH CHALLENGES

The practitioner observations in Section 6 and the comparative analysis in Section 5 together suggest several open research challenges that the lakehouse paradigm must address as it matures from emerging architecture to mainstream enterprise infrastructure.

7.1 Automated and Adaptive Compaction

Compaction is currently a manual or scheduled operation in all three major table formats. Practitioners must choose target file sizes, schedule compaction jobs, and monitor outcomes. Research is needed on adaptive compaction strategies that observe query patterns and write velocities and automatically adjust target file sizes, partition layouts, and compaction frequency. The optimization is non-trivial because the cost of compaction must be weighed against its query-performance benefit, and the optimal policy depends on workload characteristics that may evolve over time.

7.2 Cross-Format Interoperability

Many enterprises adopt multiple table formats, either through deliberate engine specialization or through acquisition and integration of teams with different tooling preferences. Migrating data between Delta Lake, Iceberg, and Hudi currently requires explicit conversion. Research is needed on catalog federation approaches that present heterogeneous tables uniformly to query engines, on intermediate representations that allow zero-copy interoperability, and on the semantics of cross-format transactions where they are required.

7.3 Real-Time Lakehouse Without the Small File Penalty

The small file problem is the principal architectural tension between streaming ingestion and lakehouse query performance. Reducing commit frequency improves file size but increases latency; increasing commit frequency improves latency but degrades query performance. Research is needed on table format designs and ingestion architectures that decouple write latency from query-time file granularity, perhaps through tiered storage layouts that buffer recent writes in a query-efficient structure before committing them to long-term storage.

7.4 Governance-Native Table Formats

Current table formats provide schema enforcement and limited audit capability through snapshot history, but they do not natively encode ownership, retention policy, lineage, or column-level access control. Embedding these properties into the table format specification rather than relying on external catalog systems could reduce governance debt of the kind documented in Section 6.6. Research is needed on the design of governance-native table formats and on the tradeoffs between in-format governance and external policy systems.

7.5 Cost Modeling for Lakehouse Architectures

No standard framework exists for predicting the total cost of ownership of a lakehouse architecture relative to a traditional warehouse or two-system architecture. The observation in Section 6.3 that storage-side optimizations dominated cost reduction in one production deployment runs counter to common assumptions and suggests that practitioner intuition about cost drivers is poorly

calibrated. Research is needed on empirical cost models that account for storage fragmentation, snapshot retention, query patterns, and engine selection, and on tooling that can estimate the cost impact of architectural decisions before they are deployed at scale.

7.6 Multi-Engine Query Optimization

The lakehouse paradigm permits multiple query engines to operate on shared data, but each engine optimizes queries independently. Research is needed on query optimization strategies that account for the presence of multiple engines, on shared statistics and metadata caches, and on workload routing policies that match queries to the most appropriate engine based on latency, cost, and resource availability.

VIII. CONCLUSION

The lakehouse paradigm represents a genuine architectural shift in enterprise data management, enabled by open table formats that bring transactional semantics, schema enforcement, and time travel to inexpensive object storage. This survey has organized the paradigm into five architectural layers, compared the three dominant open table formats across seven technical dimensions, and reported practitioner observations from a production banking deployment that document concrete benefits and persistent challenges.

The principal finding of the survey is that the lakehouse paradigm solves a specific and important problem the storage and compute unification problem that motivated the two-system architecture but it does not solve adjacent problems that organizations often hope to address through architectural change. Governance debt, operational discipline around compaction and snapshot retention, and cost predictability remain open challenges that the table format alone cannot address. The practitioner observations reported in Section 6 show that storage-side optimizations dominated cost reduction in one production environment, that small file fragmentation was the principal query-performance bottleneck for streaming workloads, and that governance gaps persisted even after the lakehouse migration was complete.

These findings suggest that lakehouse research should shift its emphasis from benchmark-scale experiments comparing query latencies on synthetic workloads toward production-scale challenges in adaptive compaction, governance-native table format design, cross-format interoperability, and empirical cost modeling. The lakehouse is no longer a speculative architecture; it is being deployed at scale in regulated industries, and the research questions that matter most are increasingly the ones that can only be observed at production scale. The contributions of this survey the layered taxonomy, the structured comparison, and the practitioner observations are intended to support that shift in research orientation.

REFERENCES

[1] Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., Łuszczak, A., Switakowski, M., Szafrański, M., Li, X., Ueshin, T., Mokhtar, M., Boncz, P., Ghodsi, A., Paranjpye, S., Senster, P., Xin, R., and Zaharia, M. (2020). Delta Lake: reliable transactional storage layer for cloud data platforms. Proceedings of the VLDB Endowment, 13(12), 3411–3424.

[2] Carbone, P., Katsifodimos, A., Ewen, S., Markl, V., Haridi, S., and Tzoumas, K. (2015). Apache Flink: Stream and batch processing in a single engine. Bulletin of the IEEE

Computer Society Technical Committee on Data Engineering, 36(4).

[3] Chaudhuri, S., and Dayal, U. (1997). An overview of data warehousing and OLAP technology. ACM SIGMOD Record, 26(1), 65–74.

[4] Chaudhuri, S., Dayal, U., and Narasayya, V. (2011). An overview of business intelligence technology. Communications of the ACM, 54(8), 88–98.

[5] Dageville, B., Cruanes, T., Zukowski, M., Antonov, V., Avanes, A., Bock, J., Claybaugh, J., Engovatov, D., Hentschel, M., Huang, J., et al. (2016). The Snowflake elastic data warehouse. Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, 215–226.

[6] Dean, J., and Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. Communications of the ACM, 51(1), 107–113.

[7] Dixon, J. (2010). Pentaho, Hadoop, and data lakes. James Dixon's Blog.

[8] Fang, H. (2015). Managing data lakes in big data era: What's a data lake and why has it became popular in data management ecosystem. IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), 820–824.

[9] Ghemawat, S., Gobioff, H., and Leung, S. T. (2003). The Google file system. Proceedings of the 19th ACM Symposium on Operating Systems Principles, 29–43.

[10] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., and Pirahesh, H. (1997). Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data Mining and Knowledge Discovery, 1(1), 29–53.

[11] Gupta, A., Agarwal, D., Tan, D., Kulesza, J., Pathak, R., Stefani, S., and Srinivasan, V. (2015). Amazon Redshift and the case for simpler data warehouses. Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, 1917–1923.

[12] Hai, R., Geisler, S., and Quix, C. (2016). Constance: An intelligent data lake system. Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, 2097–2100.

[13] Halevy, A., Korn, F., Noy, N. F., Olston, C., Polyzotis, N., Roy, S., and Whang, S. E. (2016). Goods: Organizing Google's datasets. Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, 795–806.

[14] Inmon, W. H. (1992). Building the Data Warehouse. QED Technical Publishing Group.

[15] Inmon, W. H., and Linstedt, D. (2014). Data Architecture: A Primer for the Data Scientist. Morgan Kaufmann.

[16] Khine, P. P., and Wang, Z. S. (2018). Data lake: A new ideology in big data era. ITM Web of Conferences, 17, 03025.

[17] Kimball, R., and Ross, M. (2013). The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling (3rd ed.). Wiley.

[18] Kleppmann, M. (2017). Designing Data-Intensive Applications. O'Reilly Media.

[19] Kreps, J., Narkhede, N., and Rao, J. (2011). Kafka: A distributed messaging system for log processing. Proceedings of the NetDB Workshop.

[20] Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., and Bear, C. (2012). The Vertica analytic database: C-Store 7 years later. Proceedings of the VLDB Endowment, 5(12), 1790–1801.

[21] Madden, S. (2012). From databases to big data. IEEE Internet Computing, 16(3), 4–6.

[22] Maier, D., Megler, V. M., and Tufte, K. (2014). Challenges for dataset search. Database Systems for Advanced Applications, 1–15.

[23] Melnik, S., Gubarev, A., Long, J. J., Romer, G., Shivakumar, S., Tolton, M., and Vassilakis, T. (2010). Dremel: Interactive analysis of web-scale datasets. Proceedings of the VLDB Endowment, 3(1–2), 330–339.

[24] Nargesian, F., Zhu, E., Miller, R. J., Pu, K. Q., and Arocena, P. C. (2019). Data lake management: Challenges and opportunities. Proceedings of the VLDB Endowment, 12(12), 1986–1989.

[25] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. (2008). Pig Latin: A not-so-foreign language for data processing. Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, 1099–1110.

[26] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. (1996). The log-structured merge-tree (LSM-tree). Acta Informatica, 33(4), 351–385.

[27] Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., and Stonebraker, M. (2009). A comparison of approaches to large-scale data analysis. Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, 165–178.

[28] Quix, C., Hai, R., and Vatov, I. (2016). Metadata extraction and management in data lakes with GEMMS. CAiSE Forum, 28–35.

[29] Russom, P. (2017). Data lakes: Purposes, practices, patterns, and platforms. TDWI Best Practices Report.

[30] Sethi, R., Traverso, M., Sundstrom, D., Phillips, D., Xie, W., Sun, Y., Yegitbasi, N., Jin, H., Hwang, E., Shingte, N., and Berner, C. (2019). Presto: SQL on everything. Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE), 1802–1813.

[31] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The Hadoop distributed file system. IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 1–10.

[32] Stonebraker, M. (2010). SQL databases v. NoSQL databases. Communications of the ACM, 53(4), 10–11.

[33] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., et al. (2005). C-Store: A column-oriented DBMS. Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), 553–564.

[34] Terrizzano, I. G., Schwarz, P. M., Roth, M., and Colino, J. E. (2015). Data wrangling: The challenging journey from the wild to the lake. Conference on Innovative Data Systems Research (CIDR).

[35] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. (2009). Hive: A warehousing solution over a map-reduce framework. Proceedings of the VLDB Endowment, 2(2), 1626–1629.

[36] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al. (2013). Apache Hadoop YARN: Yet another resource negotiator. Proceedings of the 4th Annual Symposium on Cloud Computing, 1–16.

[37] Verbitski, A., Gupta, A., Saha, D., Brahmadesam, M., Gupta, K., Mittal, R., Krishnamurthy, S., Maurice, S., Kharatishvili, T., and Bao, X. (2017). Amazon Aurora: Design considerations for high throughput

cloud-native relational databases. Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data, 1041–1052.

[38] Walker, C., and Alrehamy, H. (2015). Personal data lake with data gravity pull. IEEE Fifth International Conference on Big Data and Cloud Computing, 160–167.

[39] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud).

[40] 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., and Stoica, I. (2016). Apache Spark: A unified engine for big data processing. Communications of the ACM, 59(11), 56–65.

[41] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. (2013). Discretized streams: Fault-tolerant streaming computation at scale. Proceedings of the 24th ACM Symposium on Operating Systems Principles, 423–438.