

WHITE PAPER · DATA STORAGE ENGINES

Open Table Formats as Storage Layers

Transactions, metadata and interoperability on object storage

White Paper 5 · Data Storage Engines series · Version 1.0 · 26 October 2026

Bassirah Technology — Innovation & R&D

Open Table Formats as Storage Layers: Transactions, Metadata and Interoperability on Object Storage

White Paper 5 · Data Storage Engines series · Bassirah Technology — Innovation & R&D · 26 October 2026

Document control

Field	Value
Series	Data Storage Engines, paper 5 of 6
Author	Bassirah Technology — Innovation & R&D
Version	1.0 · 26 October 2026
Scope	A technology survey. It describes published research and publicly documented systems; it does not report Bassirah benchmarks or customer results.

Contents

1. Abstract.....	4
2. Introduction	4
2.1 From data lakes to lakehouses.....	4
2.2 Table formats as the storage layer	4
2.3 Scope.....	4
2.4 How this paper builds on papers 1 to 4.....	5
3. How a table format works.....	5
3.1 Snapshots, manifests and logs.....	5
3.2 Atomic commits on object storage.....	5
3.3 Optimistic concurrency and isolation	5
3.4 Schema and partition evolution	6
3.5 Time travel, branches and tags.....	6
4. The main formats.....	6
4.1 Apache Iceberg.....	6
4.2 Delta Lake	7
4.3 Apache Hudi.....	7
4.4 Streaming-first design: Apache Paimon.....	7
4.5 Metadata in a database: DuckLake	7
5. Updates, deletes and streaming ingestion	7
5.1 Copy-on-write versus merge-on-read	7
5.2 Delete files and deletion vectors	8
5.3 Streaming writes and change data capture	8
5.4 The small-file problem	8
6. Metadata, catalogs and performance	8
6.1 Metadata scaling and query planning.....	8

6.2 Statistics and data skipping	9
6.3 Catalogs	9
6.4 Object storage as the performance floor	9
7. Table maintenance	9
7.1 Compaction	9
7.2 Clustering and sorting	10
7.3 Snapshot expiry and orphan files	10
7.4 Automating maintenance	10
8. Interoperability and multi-engine access	10
8.1 A shared catalog protocol	10
8.2 Metadata translation: XTable and UniForm	10
8.3 Engine support	11
8.4 Governance across engines	11
9. Comparative analysis	11
10. Open challenges and future opportunities	12
10.1 Metadata scalability and planning latency	12
10.2 Streaming freshness versus small files	12
10.3 Multi-table transactions and catalog-level consistency	12
10.4 Interoperability and format convergence	13
10.5 Governance, access control and privacy deletion	13
10.6 Automated table maintenance and cost	13
10.7 AI, multimodal and vector data	13
10.8 Performance predictability on object storage	14
11. What this means for platform decisions	14
References	14

1. Abstract

Object storage is cheap, durable and shared by many engines, but it offers only a simple interface of whole-object writes, reads and listings. Open table formats add the missing database functions on top of it: they record which immutable data files make up each version of a table, commit changes atomically, and keep the statistics that let engines skip irrelevant data. This paper examines how they do so. It explains snapshots, manifests and transaction logs, atomic commits and optimistic concurrency on object storage, and schema and partition evolution. It then compares Apache Iceberg, Delta Lake and Apache Hudi in depth, with streaming-first Apache Paimon and the database-backed DuckLake as contrasting designs. Later sections cover updates and deletes (copy-on-write, merge-on-read, delete files and deletion vectors), streaming ingestion and the small-file problem, metadata scaling and catalogs, table maintenance, and interoperability across engines. The paper closes with open challenges, framed as research directions and opportunities, and with notes for platform decisions.

2. Introduction

2.1 From data lakes to lakehouses

Data lakes store raw data in its original formats behind a common access interface, in contrast to the schema-on-write approach of data warehouses [1]. Their low cost came with weak management: no transactions, no reliable view of which files formed a dataset, and slow directory listing. Armbrust et al. proposed the lakehouse to close this gap: keep data in a standard file format such as Parquet on low-cost object storage, and add a transactional metadata layer that defines which objects belong to each table version, so that transactions, versioning, indexing and caching can be built above it [2]. Surveys since then have compared lake, warehouse and lakehouse systems and proposed reference architectures for the lakehouse [3].

2.2 Table formats as the storage layer

The metadata layer is now a family of open specifications. Delta Lake was presented as an ACID table storage layer over cloud object stores, motivated by costly object listing and the limited consistency object stores then offered [4]. Apache Iceberg and Apache Hudi followed with different designs. Jain et al. describe the common goal: a storage manager, table metadata and transaction manager delivered as client libraries embedded in many engines, while each engine plans and executes its own queries [5]. Benchmark authors call these systems log-structured tables, because every change appends new files and new metadata rather than modifying data in place [6]. In this sense a table format is a storage engine without a server: its data structures live in object storage, and its logic runs inside whichever engine reads or writes the table.

2.3 Scope

This paper covers the transactional, metadata and maintenance mechanisms of open table formats, and the catalogs and tools that connect them to engines. It concentrates on peer-reviewed work from 2021 to 2026 and on official specifications and documentation, citing earlier work where it introduced an idea still in use. Statements about specifications and products reflect documentation accessed on 26 October 2026. It does not report benchmark

figures. Where a paper reports results, we describe them qualitatively and attribute them to the authors.

2.4 How this paper builds on papers 1 to 4

Paper 1 placed open table formats in Class E of its taxonomy and named network latency and metadata scaling as their bottlenecks. Paper 3 described the files these formats manage (Parquet and ORC encodings, zone maps, clustering keys, object storage as primary storage); this paper works one layer up and does not repeat that detail. Paper 2's LSM-tree reappears in the log-structured way table formats accumulate files, in the compaction that keeps them readable, and directly inside Apache Paimon [7]. Paper 4's vector data returns in Section 10.7.

3. How a table format works

3.1 Snapshots, manifests and logs

Every format separates immutable data files from metadata that lists the live files of each version. In Iceberg, a table metadata file records the schema, partition configuration, properties and snapshots; each snapshot points to a manifest list, which points to manifest files, which list data files with their partition values and column metrics [8]. Manifests are reused across snapshots, so a commit rewrites only the metadata that changed. Delta Lake instead keeps an ordered log of JSON commit files, each recording files logically added and removed, and periodically writes checkpoints that replay the log up to a version in Parquet form [9]. Hudi records every write and table service as an action on a timeline of instants that move from requested to inflight to completed [10].

Jain et al. group these designs into two families. Delta Lake and Hudi use a tabular approach, storing metadata in a log compacted into a checkpoint or an internal metadata table; Iceberg uses a hierarchical approach in which upper levels of the manifest tree act as an index over lower levels [5].

3.2 Atomic commits on object storage

A commit must replace one table version with the next so that readers see all of a change or none of it. Object stores lack cheap atomic renames [4], so each format relies on one small atomic primitive. Iceberg atomically swaps the pointer to the current metadata file, usually through a catalog that performs a check-and-put; the specification deprecates rename-based commits as unsafe in object stores [8]. Delta Lake writers must never overwrite an existing log entry [9]; the original design used put-if-absent where the store supported it and an external coordination service where it did not [4], [5]. Hudi relies on atomic storage operations for timeline transitions [10].

Conditional writes in Amazon S3, announced in 2024, create an object only if its key does not exist, or replace it only if its entity tag is unchanged [11]. Such primitives allow coordination through the object store itself; Hudi, for example, documents a storage-based lock provider that uses conditional writes on a lock file, removing the need for a separate lock service [12].

3.3 Optimistic concurrency and isolation

All three formats use multi-version concurrency control with optimistic validation [5]. A writer reads the current snapshot, writes new data files, and at commit time checks for conflicts with transactions committed in the meantime. In Iceberg, appends can always be applied to a

newer version, while compactions and file-level deletes must confirm that the files they replace are still in the table [8]. Delta Lake documents which operations can conflict, and notes that partitioning by the columns used in concurrent commands keeps their file sets disjoint [13].

The isolation level follows from what is validated. Jain et al. report that Hudi and Iceberg check write-write conflicts by default, giving snapshot isolation, while Delta Lake by default also checks whether rows matching a transaction's read predicates were added concurrently, giving serializability, and can extend the check to reads for strict serializability [5]. They also found that commits coordinated by locks in the Hive Metastore could, in an edge case, allow a dirty write. Hudi adds non-blocking concurrency control, in which writers to the same file group proceed and the reader and compactor resolve conflicts later [12].

3.4 Schema and partition evolution

Iceberg tracks columns by identifier, so columns can be added, dropped, renamed and reordered without rewriting data [8]; Delta Lake's column mapping serves the same goal [9]. Iceberg also makes partitioning a table property. With hidden partitioning, partition values are derived from column values through transforms such as day or bucket, so queries filter on the original columns and the format maps predicates to partitions [14]. A partition spec can evolve: the new spec applies to new data, older files keep their original layout, and planning handles both [8].

3.5 Time travel, branches and tags

Any retained snapshot can be queried. Iceberg adds named references: tags label individual snapshots, and branches are mutable references that receive commits under the same conflict rules as the main table [8]. Hudi records savepoints that protect file versions from cleaning and allow restores [10]. LST-Bench found time-travel queries on the tested formats roughly as efficient as queries on the corresponding latest version [6]. Every retained version keeps its files, which matters for storage cost and deletion obligations (Section 10.5).

4. The main formats

4.1 Apache Iceberg

Iceberg is designed to manage a large, slow-changing collection of files as a table, with goals that include serializable isolation without reader locks, scan planning in a constant number of remote calls by clients rather than a central metadata store, and full schema and partition evolution [8]. Version 1 defined analytic tables on immutable Parquet, Avro and ORC files; version 2 added row-level deletes through delete files; version 3 added new types (nanosecond timestamps, variant, geometry and geography), column default values, multi-argument transforms, row lineage, binary deletion vectors and table encryption keys [8]. Row lineage gives each row a stable identifier and the sequence number of its last change, which supports incremental processing if every writer preserves those fields. The specification describes versions 1 to 3 as complete and adopted, and version 4 as under development; tables may stay on an older version so that engines without the new features can still use them [8].

4.2 Delta Lake

Delta Lake's protocol aims at serializable writes, snapshot-isolated reads, scalability to billions of files and a self-describing table that needs no separate metastore [9]. Readers reconstruct table state from the newest checkpoint plus later commits. The protocol has added log compaction files, a checkpoint layout that splits state into sidecar files, and catalog-managed tables, in which the managing catalog decides whether a commit succeeded [9]. Features such as deletion vectors, row tracking, type widening and variant data are enabled per table, and clients that lack a feature are blocked rather than allowed to misread the table [9], [13]. Liquid clustering replaces fixed partitioning and Z-ordering with clustering columns that can be redefined without rewriting existing data, and data is clustered incrementally by OPTIMIZE jobs [13].

4.3 Apache Hudi

Hudi was designed around mutable change streams. Every record has a key, and Hudi maintains a mapping from keys to file groups, which lets an upsert find the files that hold a key without scanning the table [15]. Its timeline is both a commit log and a schedule: compaction, clustering, cleaning, indexing and rollback are actions on it, and its history is itself stored as an LSM tree [10]. Hudi offers two table types. Copy-on-write rewrites base files on update and serves fast reads; merge-on-read writes changes to log files that are merged with base files at query time or by periodic compaction. Merge-on-read tables expose snapshot queries, which see the latest data, and read-optimized queries, which read only compacted base files [10].

4.4 Streaming-first design: Apache Paimon

Apache Paimon applies paper 2's LSM-tree directly to a lake table. A primary-key table is divided into buckets, each bucket holds an LSM tree of sorted runs, reads merge runs according to a configured merge engine, and compaction reduces the number of runs future reads must merge [7]. Snapshots reference manifest lists in a way similar to Iceberg, and changelog producers let streaming readers, typically Apache Flink jobs, consume updates and deletes as a changelog [7].

4.5 Metadata in a database: DuckLake

DuckLake revisits a premise of the other formats. Its authors argue that once a catalog database is needed anyway for atomic commits, it should hold all table metadata, not only a pointer to the current version [16]. DuckLake stores schemas, snapshots, file lists and statistics in tables of any SQL database that supports transactions and primary keys, keeps data in Parquet on object or file storage, supports transactions across multiple tables, and can inline small changes into the metadata database [16]. The DuckDB team announced DuckLake version 1.0 in April 2026 [17], and the specification has since reached version 1.1 [16].

5. Updates, deletes and streaming ingestion

5.1 Copy-on-write versus merge-on-read

Data files are immutable, so a changed row needs either a rewritten file or a separate record of the change. Copy-on-write rewrites every file that contains a changed row: writes are amplified and reads stay fast. Merge-on-read writes the change separately and reconciles it at read time: writes are cheaper and reads pay until compaction [5]. Jain et al. found that merge-on-read made merges faster at the cost of slower subsequent queries, and that format

defaults such as target file sizes affected load and scan performance [5]. LST-Bench, on its own workloads, found copy-on-write configurations ahead of merge-on-read in both latency and I/O [6]. Neither strategy dominates; the choice follows the update-to-read ratio and the freshness required.

5.2 Delete files and deletion vectors

Iceberg version 2 has position deletes, which mark a row by data file and position, and equality deletes, which mark every row matching column values such as an identifier [8]. Equality deletes let a streaming writer delete a key without reading existing data, but readers must apply them to older data files in the partition, and row lineage cannot follow rows updated this way [8]. Hudi merge-on-read stores record-level changes in log files keyed by record [10], [5].

Deletion vectors bound the read cost. A deletion vector is a bitmap of deleted row positions for a single data file. In Iceberg version 3, deletion vectors are stored as Puffin blobs, at most one may exist per data file, and writers must merge new deletes into it; writers may no longer add position delete files [8]. Delta Lake deletion vectors mark rows as removed without rewriting the Parquet file and can be stored inline in the log or in separate files [9], [13]. Paimon also records deletion vectors in its index manifests [7]. The shared trade is that writers do more work at commit time so that readers apply one compact structure per file.

5.3 Streaming writes and change data capture

Streaming ingestion brings frequent small commits and keyed updates from change data capture. Iceberg supports streaming appends and incremental reads from a starting snapshot in Flink [18]. Hudi targets this workload with merge-on-read, record indexes and non-blocking concurrency between writers [10], [15], [12]. Paimon treats it as its primary case, with LSM buckets absorbing upserts and changelog producers feeding downstream streaming jobs [7]. DuckLake can inline small changes into the catalog database instead of writing a small file for each one [16]. Each moves the cost of small writes elsewhere (log files, LSM runs or database rows) that must later be compacted.

5.4 The small-file problem

Every commit adds files and metadata entries. Gruenheid et al. describe how trickle writes and untuned writers accumulate small files, which raise per-object overhead, reduce the effectiveness of columnar compression, enlarge metadata, slow planning and maintenance, and increase request costs on cloud storage [19]. LST-Bench traced progressive slowdown in some engine and format combinations to writers that spread updates across many files, and the slowdown disappeared when another engine wrote the same changes [6]. Small files are a property of the writer as much as of the format.

6. Metadata, catalogs and performance

6.1 Metadata scaling and query planning

Listing objects is slow, so all three formats keep file lists in metadata that can be read with a few large requests [5]. Jain et al. found that Iceberg's single-node planning, as implemented at the time, did well on small tables, while Delta Lake's distributed planning scaled better on tables with many files, and they suggested a cost model to choose between the two [5].

Describing Google BigQuery, Edara and Pasumansky argue that metadata at very large scale should be managed with the same distributed techniques as data rather than reduced at the expense of query performance [20]. Hudi's metadata table applies this idea in an open format: an internal merge-on-read table in an SSTable-like layout that holds file listings, column statistics, bloom filters and record, secondary and expression indexes, updated transactionally with data commits [15], [21]. The Iceberg REST specification now also defines server-side scan planning [22].

6.2 Statistics and data skipping

Per-file column statistics let engines skip files whose value ranges cannot match a predicate, the table-level counterpart of paper 3's zone maps. Iceberg also keeps partition summaries in manifest lists and optional table and partition statistics files [8]. Skipping works only as well as rows are clustered into files (Section 7.2).

6.3 Catalogs

A catalog maps table names to current metadata and, for Iceberg, performs the atomic commit. The Trino Iceberg connector, for example, accepts a Hive metastore, AWS Glue, a JDBC catalog, a REST catalog, Nessie or a Snowflake catalog [23]. The Iceberg REST catalog specification defines an HTTP interface for namespaces, tables, views and commits, including an operation that commits updates to multiple tables atomically and mechanisms for delegating storage access through vended credentials or remote signing [22]. Apache Polaris, a top-level Apache project since March 2026, is an Iceberg catalog built on this protocol [24]. Unity Catalog, an open-source LF AI & Data Foundation project, catalogs tables, files, functions and models and is compatible with the Hive metastore and Iceberg REST APIs [25]. Project Nessie adds Git-like branches and tags, and its commits can carry operations on several tables [26]. Object-store-native catalogs remove the separate service: MinIO AIStor Tables serves an Iceberg REST catalog from the object store itself, with no external catalog service or metadata database, and documents compatibility with the Iceberg version 3 specification [27].

6.4 Object storage as the performance floor

Every read pays object-storage latency and per-request cost. Durner, Leis and Neumann show that an engine with a well-designed download manager can query object storage directly and perform similarly to cloud warehouses that cache data on local SSDs [28], but per-request latency remains and shapes planning and commit cost. Jain et al. name commit latency on object storage as a limit on write rates under concurrency [5].

7. Table maintenance

7.1 Compaction

Compaction rewrites small files into larger ones and folds delete files or logs into base files. Iceberg documents data-file compaction, manifest rewriting and position-delete compaction [14]; Hudi schedules compaction and log compaction as timeline actions [10]; Paimon compacts LSM runs [7]. LST-Bench found that periodic optimization restored read performance on the formats that accumulated small files, and that default compaction settings could make the operation itself slow without workload-specific tuning [6].

7.2 Clustering and sorting

Clustering co-locates rows with similar values so that statistics prune more files. The Delta Lake paper describes Z-ordering across several columns combined with min/max skipping [4]; liquid clustering later made clustering columns changeable and incremental [13]. Iceberg tables can declare sort orders, and the Trino Iceberg connector's optimize procedure can rewrite files sorted by chosen columns [8], [23]. Research learns layouts from the workload: Qd-tree assigns records to blocks with a tree learned from query predicates [29], and Ding et al. extend instance-optimized layouts beyond a single table [30]. Such ideas are not yet features of the open formats.

7.3 Snapshot expiry and orphan files

Expiring snapshots removes old versions and the files only they referenced; orphan-file removal deletes files left by failed writes. Iceberg warns that removing orphans with too short a retention interval can corrupt a table if an in-progress write's files are deleted [14]. Delta Lake removes unreferenced files with VACUUM after a retention period [9], and Hudi's cleaner removes older file slices [10]. Expiry also ends time travel to the removed versions.

7.4 Automating maintenance

Maintenance must decide which tables need what, and when. AutoComp, from Gruenheid et al., automates compaction by selecting candidate tables and weighing benefits against costs; evaluated on synthetic benchmarks and in production through the OpenHouse control plane, it reduced file counts and improved query performance, according to its authors [19]. Catalogs and storage are taking on this role: AIStor Tables exposes per-warehouse and per-table maintenance configuration through its API [27], and under the Delta catalog-managed tables feature, clients may run cleanup and layout operations only when the managing catalog permits them [9].

8. Interoperability and multi-engine access

8.1 A shared catalog protocol

Engines must agree on the catalog as well as the files. The Iceberg REST catalog specification has become a common contract: a table registered once can be used by any engine that speaks the protocol [22]. Polaris, Unity Catalog, Nessie and AIStor Tables each support it [24], [25], [26], [27]. Compatibility still needs checking engine by engine; the AIStor documentation, for example, gives per-engine connection settings, with some engines signing catalog requests and others authenticating with bearer tokens [27].

8.2 Metadata translation: XTable and UniForm

Because Iceberg, Delta Lake and Hudi all reference Parquet data files, metadata for one format can be generated from another without copying data. Apache XTable (incubating) is a cross-table converter for Hudi, Iceberg and Delta Lake built on a common model of a table [31]. Delta UniForm generates Iceberg metadata asynchronously after each Delta commit, so Iceberg clients can read a Delta table; it requires the IcebergCompatV2 table feature and column mapping [32], [9]. Translation keeps one format authoritative: readers of the translated metadata see the table as of the last conversion, and features without an equivalent cannot be represented.

8.3 Engine support

Support differs by engine, format and format version. Apache Spark was the common engine in the comparative studies cited here and runs Iceberg's documented maintenance actions [5], [6], [14]. Trino reads and writes Iceberg and Delta Lake tables, runs Iceberg maintenance procedures, and reads Hudi tables; its Iceberg version 3 support is documented as experimental, with version 2 as the default for new tables [23]. Flink writes streams to Iceberg and Paimon [18], [7]. ClickHouse reads Iceberg and Delta Lake tables and can write to both, with documented type limitations, and reads but does not write Iceberg deletion vectors [33]. DuckDB reads Iceberg tables directly from storage and needs a REST catalog for writes, uses the Delta Kernel library to read and append to Delta tables, and is the reference implementation of DuckLake [34], [17]. Snowflake can act as an Iceberg catalog, use an external one through a catalog integration, or sync its tables to Snowflake Open Catalog for third-party engines [35]. A table is only as portable as the least capable engine that must use it.

8.4 Governance across engines

Access control can sit in the engine, the catalog or the storage. When each engine enforces its own policies, a shared table needs consistent policies in all of them, and direct file access can bypass them. Catalog-centered designs narrow this gap: the Iceberg REST specification lets the catalog decide what storage access to vend for each table request [22], and Polaris and Unity Catalog present themselves as central points of access governance across engines [24], [25]. Row- and column-level policies are usually evaluated where queries run, so equivalent rules must be configured and tested in each engine.

9. Comparative analysis

Table 1 compares the formats and designs discussed in this paper. The ratings are qualitative and relative to each other. Actual behavior depends on format version, engine, configuration, maintenance and workload.

Format or design	Update model	Streaming support	Metadata approach	Ecosystem and catalog options	Best-fit workloads
Apache Iceberg	Copy-on-write or merge-on-read; position and equality deletes (v2), deletion vectors (v3)	Medium (streaming appends and incremental reads; equality deletes for upserts)	Hierarchical manifest tree; atomic pointer swap in a catalog	High; many engines; Iceberg REST catalog implemented by several catalogs, including object-store-native ones	Multi-engine analytical tables shared across teams and tools
Delta Lake	Copy-on-write; deletion vectors for merge-on-read	Medium (log can be tailed for incremental reads)	Ordered transaction log with checkpoints; optional catalog-managed commits	Medium to High; broad support, with Spark-based stacks as its home ecosystem; Iceberg reads through UniForm	Batch and incremental pipelines centered on one primary writing engine

Format or design	Update model	Streaming support	Metadata approach	Ecosystem and catalog options	Best-fit workloads
Apache Hudi	Copy-on-write or merge-on-read with record-keyed log files and indexes	High (merge-on-read, record indexes, non-blocking concurrency)	Timeline plus internal metadata table with indexes	Medium; read support in several engines; write path centered on Spark and Flink	Change data capture, frequent keyed upserts and incremental pipelines
Apache Paimon	LSM merge per bucket; deletion vectors	High (designed for streaming upserts and changelogs)	Snapshots and manifest lists	Low to Medium; centered on Flink, with growing engine support	Real-time tables fed by streams with continuous updates
DuckLake	Snapshots in SQL; small changes can be inlined in the catalog database	Medium (inlining absorbs small writes)	All metadata in a transactional SQL database; multi-table transactions	Low; DuckDB reference implementation	Small to mid-size lakehouses that already run a reliable SQL database

Table 1: Qualitative comparison of open table formats and related designs.

10. Open challenges and future opportunities

10.1 Metadata scalability and planning latency

Challenge. Planning cost grows with metadata. Single-node planning favors small tables, distributed planning favors large ones, and each adds latency at the other end [5]. **Research direction.** Treating metadata as data, with indexes and distributed processing, is one answer [20], [21]; Iceberg's version 4 draft restructures metadata into an adaptive tree that keeps small commits cheap and stores data files with their deletion vectors to avoid two-phase planning [8]; and catalogs can now plan scans on the server [22]. **Opportunity.** A cost model that chooses between local, distributed and catalog-side planning per query, as Jain et al. suggest, would make planning time predictable across table sizes.

10.2 Streaming freshness versus small files

Challenge. Frequent commits deliver fresh data but multiply files, metadata entries and delete records, and reads degrade until maintenance catches up [19], [6]. **Research direction.** LSM-structured tables, non-blocking writers, and inlining of small changes into a catalog database each absorb small writes differently [7], [12], [16]. **Opportunity.** As paper 2 argued for LSM-trees, exposing freshness and read amplification as explicit, tunable targets, rather than side effects of commit intervals and compaction settings, would let teams trade one for the other deliberately.

10.3 Multi-table transactions and catalog-level consistency

Challenge. The main formats provide transactions within one table, not across tables [5]. Multi-table pipelines can expose partial results, and added coordination services weaken interoperability. **Research direction.** Götz et al. present LakeVilla, which provides multi-table ACID transactions using only object-store primitives while staying compatible with existing

deployments and engines such as Trino [36]. Microsoft's Polaris engine (unrelated to Apache Polaris) adds snapshot-isolated multi-table transactions over immutable files [37]. The Iceberg REST specification defines a multi-table commit [22], and DuckLake and Nessie handle multi-table changes in their catalogs [16], [26]. **Opportunity.** A multi-table transaction model shared across catalogs and engines would let such pipelines stay open.

10.4 Interoperability and format convergence

Challenge. With three major formats, several catalogs and engine support that varies by format version, a table written by one engine may not be fully usable by another [23], [33], and translation layers lag behind the authoritative format [31], [32]. **Research direction.** The formats are converging on shared ideas: bitmap deletion vectors, row lineage or row tracking, and variant types now appear in both the Iceberg and Delta specifications, and both increasingly rely on a catalog to coordinate commits [8], [9]. **Opportunity.** Conformance test suites for readers and writers of each format version, published by the format communities, would let teams check engine compatibility before upgrading a shared table.

10.5 Governance, access control and privacy deletion

Challenge. Deletes in table formats are logical first: a deletion vector or delete file hides a row, but the bytes remain in data files and in retained snapshots until compaction, snapshot expiry and file cleanup run [8], [14], [9]. Shastri et al. showed that the GDPR creates metadata-heavy workloads that existing systems handle poorly [38], and per-engine policies compound the problem (Section 8.4). **Research direction.** Catalog-vended credentials and centralized catalogs move enforcement toward a single point [22], [24], [25], and the table encryption keys tracked in Iceberg version 3 metadata [8] could, in principle, support deletion by destroying keys. **Opportunity.** Formats and catalogs that track physical deletion as a measurable state, from logical delete to the last copy removed, would let teams demonstrate compliance rather than infer it from retention settings.

10.6 Automated table maintenance and cost

Challenge. Maintenance consumes compute and object-store requests, and poorly timed jobs conflict with writers or waste effort [19], [6]. **Research direction.** AutoComp frames compaction as a benefit-cost decision across many tables [19], learned layouts optimize clustering for a workload [29], [30], object-store-native catalogs expose maintenance configuration [27], and Delta lets the managing catalog govern when maintenance may run [9]. **Opportunity.** Workload-aware maintenance that serves several engines and reports its own cost remains largely open, as AutoComp's authors note [19].

10.7 AI, multimodal and vector data

Challenge. AI workloads need random row access, embedding columns, large binary objects and vector indexes; table formats were designed for scans over files written once. **Research direction.** The Lance table format organizes data into fragments whose columns can be extended without rewriting existing files and treats indexes as part of the table lifecycle [39], building on a file format designed for both scans and random access [40]. Iceberg version 3 adds a variant type for semi-structured data, and version 4 drafts a file type [8]. **Opportunity.** Paper 4 argued that vectors should live close to the data and permissions they depend on; table formats that maintain vector indexes transactionally would let lakehouse tables serve retrieval without a separate copy.

10.8 Performance predictability on object storage

Challenge. Table-format benchmark results depend heavily on engine, version and configuration, and published comparisons disagree [6], [5]; object-storage latency adds further variance [28]. **Research direction.** LHBench compares formats under a common engine [5], LST-Bench measures degradation over time and format-engine interactions [6], and Durner et al. derive cost- and performance-optimal configurations for reading from object storage [28]. **Opportunity.** Reporting degradation over time, maintenance cost and request counts alongside query latency would describe production behavior better than single benchmark runs.

11. What this means for platform decisions

Whether format, catalog, engines and maintenance fit together matters more than the format alone. In Bassirah's Open Iceberg Lakehouse blueprint, Apache Iceberg tables live on S3-compatible object storage such as MinIO AIStor, a Strategic Technology Partner, with AIStor Tables as the Iceberg catalog where the selected release supports it; SQL access uses Trino OSS or Starburst, a Strategic Technology Partner; and dbt Core / dbt OSS and DataHub Core provide models and lineage. Format versions, catalog interfaces, engine support and maintenance schedules are validated on representative workloads before adoption, and no outcome is assumed from published results.

References

1. Hai, R., Koutras, C., Quix, C. and Jarke, M. (2023). Data Lakes: A Survey of Functions and Systems. *IEEE Transactions on Knowledge and Data Engineering*, 35(12), 12571–12590. (<https://doi.org/10.1109/TKDE.2023.3270101>)
2. Armbrust, M., Ghodsi, A., Xin, R. and Zaharia, M. (2021). Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. *CIDR*. (https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf)
3. Harby, A. A. and Zulkernine, F. (2025). Data Lakehouse: A survey and experimental study. *Information Systems*, 127, 102460. (<https://doi.org/10.1016/j.is.2024.102460>)
4. Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., Łuszczak, A., et al. (2020). Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *PVLDB*, 13(12), 3411–3424. (<https://doi.org/10.14778/3415478.3415560>)
5. Jain, P., Kraft, P., Power, C., Das, T., Stoica, I. and Zaharia, M. (2023). Analyzing and Comparing Lakehouse Storage Systems. *CIDR*. (<https://www.vldb.org/cidrdb/papers/2023/p92-jain.pdf>)
6. Camacho-Rodríguez, J., Agrawal, A., Gruenheid, A., Gosalia, A., Petculescu, C., Aguilar-Saborit, J., Floratou, A., Curino, C. and Ramakrishnan, R. (2024). LST-Bench: Benchmarking Log-Structured Tables in the Cloud. *Proceedings of the ACM on Management of Data (SIGMOD)*, 2(1). (<https://doi.org/10.1145/3639314>)
7. Apache Paimon project (2026). Basic Concepts; Primary Key Table. Official documentation, accessed 26 October 2026. (<https://paimon.apache.org/docs/master/concepts/basic-concepts/>; <https://paimon.apache.org/docs/master/primary-key-table/>)

8. Apache Iceberg project (2026). Iceberg Table Spec. Official specification, accessed 26 October 2026. (<https://iceberg.apache.org/spec/>)
9. Delta Lake project (2026). Delta Transaction Log Protocol. Official specification, accessed 26 October 2026. (<https://github.com/delta-io/delta/blob/master/PROTOCOL.md>)
10. Apache Hudi project (2026). Timeline; Table and Query Types. Official documentation, accessed 26 October 2026. (<https://hudi.apache.org/docs/timeline>; https://hudi.apache.org/docs/table_types)
11. Amazon Web Services (2024, 2026). Amazon S3 adds support for conditional writes (product announcement, August 2024); How to prevent object overwrites with conditional writes (Amazon S3 User Guide). Accessed 26 October 2026. (<https://aws.amazon.com/about-aws/whats-new/2024/08/amazon-s3-conditional-writes/>; <https://docs.aws.amazon.com/AmazonS3/latest/userguide/conditional-writes.html>)
12. Apache Hudi project (2026). Concurrency Control. Official documentation, accessed 26 October 2026. (https://hudi.apache.org/docs/concurrency_control)
13. Delta Lake project (2026). Concurrency control; Deletion vectors; Use liquid clustering for Delta tables. Official documentation, accessed 26 October 2026. (<https://docs.delta.io/latest/concurrency-control.html>; <https://docs.delta.io/latest/delta-deletion-vectors.html>; <https://docs.delta.io/latest/delta-clustering.html>)
14. Apache Iceberg project (2026). Partitioning; Maintenance. Official documentation, accessed 26 October 2026. (<https://iceberg.apache.org/docs/latest/partitioning/>; <https://iceberg.apache.org/docs/latest/maintenance/>)
15. Apache Hudi project (2026). Metadata Table; Indexes. Official documentation, accessed 26 October 2026. (<https://hudi.apache.org/docs/metadata>; <https://hudi.apache.org/docs/indexes>)
16. DuckLake project (2026). The DuckLake Manifesto; DuckLake Specification (version 1.1). Official documentation, accessed 26 October 2026. (<https://ducklake.select/manifesto/>; <https://ducklake.select/docs/stable/specification/introduction>)
17. DuckDB Foundation (2026). DuckLake v1.0: The Lakehouse Format Built on SQL Reaches Production-Readiness. DuckDB blog, 13 April 2026, accessed 26 October 2026. (<https://duckdb.org/2026/04/13/ducklake-10.html>)
18. Apache Iceberg project (2026). Flink Getting Started. Official documentation, accessed 26 October 2026. (<https://iceberg.apache.org/docs/latest/flink/>)
19. Gruenheid, A., Camacho-Rodríguez, J., Curino, C., Ramakrishnan, R., Pak, S., Sakdeo, S., Gandhi, L., Singhal, S. K., Nilangekar, P. and Abadi, D. J. (2025). AutoComp: Automated Data Compaction for Log-Structured Tables in Data Lakes. Companion of the 2025 International Conference on Management of Data (SIGMOD), 404–417. (<https://doi.org/10.1145/3722212.3724430>)
20. Edara, P. and Pasumansky, M. (2021). Big Metadata: When Metadata is Big Data. PVLDB, 14(12), 3083–3095. (<https://doi.org/10.14778/3476311.3476385>)
21. Sumit, S., Wason, P. and Narayanan, S. (2026). Metadata-as-Data in Apache Hudi: A Multi-Modal Index Substrate for Lakehouse Tables. Proceedings of the 1st International Workshop on Data FORMATS for Modern Architectures and Workloads, 1–5. (<https://doi.org/10.1145/3802514.3809177>)
22. Apache Iceberg project (2026). Apache Iceberg REST Catalog API (OpenAPI specification). Official specification, accessed 26 October 2026. (<https://github.com/apache/iceberg/blob/main/open-api/rest-catalog-open-api.yaml>)

23. Trino Software Foundation (2026). Iceberg connector; Delta Lake connector; Hudi connector. Official documentation, accessed 26 October 2026.
(<https://trino.io/docs/current/connector/iceberg.html>;
<https://trino.io/docs/current/connector/delta-lake.html>;
<https://trino.io/docs/current/connector/hudi.html>)
24. Apache Polaris project (2026). Apache Polaris documentation and repository; The Apache Software Foundation, Apache Gluten and Apache Polaris Become Top-Level Projects, 5 March 2026. Accessed 26 October 2026. (<https://polaris.apache.org/>;
<https://github.com/apache/polaris>; <https://news.apache.org/foundation/entry/the-apache-software-foundation-graduates-two-open-source-projects-from-incubator>)
25. Unity Catalog project (2026). Unity Catalog repository README. Accessed 26 October 2026. (<https://github.com/unitycatalog/unitycatalog>)
26. Project Nessie (2026). Project Nessie: Transactional Catalog for Data Lakes with Git-like semantics; Nessie Specification. Official documentation, accessed 26 October 2026. (<https://projectnessie.org/>; <https://projectnessie.org/develop/spec/>)
27. MinIO (2026). AIStor Tables; Connecting query engines to AIStor Tables; AIStor Tables API Reference. Official documentation, accessed 26 October 2026.
(<https://docs.min.io/aistor/administration/aistor-tables/>;
<https://docs.min.io/aistor/developers/aistor-tables/connect-query-engines/>;
<https://docs.min.io/aistor/developers/aistor-tables/aistor-tables-api/>)
28. Durner, D., Leis, V. and Neumann, T. (2023). Exploiting Cloud Object Storage for High-Performance Analytics. PVLDB, 16(11), 2769–2782.
(<https://doi.org/10.14778/3611479.3611486>)
29. Yang, Z., Chandramouli, B., Wang, C., Gehrke, J., Li, Y., Minhas, U. F., Larson, P.-Å., Kossmann, D. and Acharya, R. (2020). Qd-tree: Learning Data Layouts for Big Data Analytics. SIGMOD, 193–208. (<https://doi.org/10.1145/3318464.3389770>)
30. Ding, J., Minhas, U. F., Chandramouli, B., Wang, C., Li, Y., Li, Y., Kossmann, D., Gehrke, J. and Kraska, T. (2021). Instance-Optimized Data Layouts for Cloud Analytics Workloads. SIGMOD, 418–431. (<https://doi.org/10.1145/3448016.3457270>)
31. Apache XTable (incubating) project (2026). Apache XTable repository README. Accessed 26 October 2026. (<https://github.com/apache/incubator-xtable>)
32. Delta Lake project (2026). Universal Format (UniForm). Official documentation, accessed 26 October 2026. (<https://docs.delta.io/latest/delta-uniform.html>)
33. ClickHouse (2026). Iceberg table engine; DeltaLake table engine. Official documentation, accessed 26 October 2026. (<https://clickhouse.com/docs/en/table-engines/integrations/iceberg>; <https://clickhouse.com/docs/en/table-engines/integrations/deltalake>)
34. DuckDB Foundation (2026). Iceberg extension; Delta extension. Official documentation, accessed 26 October 2026.
(https://duckdb.org/docs/stable/core_extensions/iceberg/overview;
https://duckdb.org/docs/stable/core_extensions/delta)
35. Snowflake (2026). Apache Iceberg tables. Official documentation, accessed 26 October 2026. (<https://docs.snowflake.com/en/user-guide/tables-iceberg>)
36. Götz, T., Ritter, D., El-Hindi, M. and Giceva, J. (2026). Interoperable ACID Transactions for Open Table Formats. PVLDB, 19(11), 3216–3230. (<https://doi.org/10.14778/3836663.3836684>)

37. Aguilar Saborit, J., Ramakrishnan, R., Bocksrocker, K., Halverson, A., Kosinsky, K., O'Connor, R., Poliakova, N., Shafiei, M., Ansari, H. M., Crivat, B., Cunningham, C. and Kim, T. (2024). Extending Polaris to Support Transactions. Companion of the 2024 International Conference on Management of Data (SIGMOD), 321–333. (<https://doi.org/10.1145/3626246.3653392>)
38. Shastri, S., Banakar, V., Wasserman, M., Kumar, A. and Chidambaram, V. (2020). Understanding and Benchmarking the Impact of GDPR on Database Systems. PVLDB, 13(7), 1064–1077. (<https://doi.org/10.14778/3384345.3384354>)
39. Lance project (2026). Lance Table Format. Official specification, accessed 26 October 2026. (<https://lance.org/format/table/>)
40. Pace, W., She, C., Xu, L., Jones, W., Lockett, A., Wang, J. and Shah, R. (2025). Lance: Efficient Random Access in Columnar Storage through Adaptive Structural Encodings. arXiv preprint 2504.15247. (<https://arxiv.org/abs/2504.15247>)

Product and project names are trademarks of their respective owners. Their mention describes publicly documented technology and does not imply a business relationship, except where Bassirah names a Strategic Technology Partner.