
# Interoperable ACID Transactions for Open Table Formats

Tobias Götz
Technical University of Munich
Munich, Germany
goetzt@in.tum.de

Daniel Ritter
SAP
Walldorf, Germany
daniel.ritter@sap.com

Muhammad El-Hindi
Technical University of Munich
Munich, Germany
muhammad.el-hindi@tum.de

Jana Giceva
Technical University of Munich
Munich, Germany
jana.giceva@in.tum.de

## ABSTRACT

Open Table Formats (OTFs) enable interoperability across systems by using object stores as a shared storage layer. They support OTF HTAP, in which complex queries execute concurrently with updates on the object store. However, existing OTFs provide transactional guarantees only for single-table transactions. Supporting transactions that access multiple tables typically requires external coordination services, which compromise interoperability and deployment independence. In this paper, we show that this trade-off is unnecessary. We present OTF-native mechanisms for multi-table atomicity, consistency, and isolation that rely solely on object-store primitives. We realize them in LakeVilla, a prototype that provides full ACID transactions while remaining compatible with existing OTF deployments. Our evaluation combines formal verification, extensive benchmarks, and an interoperability case study with Trino, showing that LakeVilla delivers strong transactional guarantees with low overhead, scalable concurrency, and compatibility with existing analytics engines.

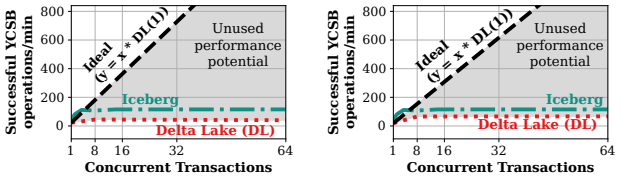
**PVLDB Reference Format:**
Tobias Götz, Daniel Ritter, Muhammad El-Hindi, and Jana Giceva. Interoperable ACID Transactions for Open Table Formats. PVLDB, 19(11): 3216 - 3230, 2026.
doi:10.14778/3836663.3836684

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/goetztj/LakeVilla.

## 1 INTRODUCTION

**Open Table Formats as a portable storage abstraction.** Open table formats (OTFs), such as Delta Lake, Apache Iceberg, and Apache Hudi [81, 83, 88], are widely used in modern cloud analytics [28, 79, 98]. They combine the scalability and low cost of object-storage-backed data lakes with data-warehouse-style features. By storing both data files and metadata in object storage, they establish a shared storage and metadata abstraction that improves interoperability between heterogeneous processing engines [28, 98].


Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836684




**(a) Write-heavy (A, 50% reads)** **(b) Read-heavy (B, 95% reads)**

**Figure 1: Transaction scalability under YCSB A and B for Delta Lake and Iceberg (Spark). The comparison with an idealized no-abort baseline for Delta Lake shows that both systems scale poorly under concurrent workloads.**

OTFs target Hybrid Transactional/Analytical Processing (HTAP) , where analytical queries run concurrently with updates [9]. These workloads require isolation between read and write transactions to avoid inconsistencies and corruption from overlapping updates.

**Per-table design limits transactional guarantees.** However, current OTFs provide limited transactional guarantees under concurrency [9, 20, 21, 40, 41, 44, 57, 80, 85]. They primarily support transactions (Txns) updating one table (single-table transactions) and serialize writers per table [9]. This approach limits modern analytical workloads with a continuous ingestion of updates [35, 95]. As shown in Fig. 1, concurrent YCSB transactions that access and modify the same table scale poorly and exhibit high abort rates. Already with two concurrent transactions, throughput stagnates for both Iceberg and Delta Lake. This behavior stems from per-table metadata coordination in object storage and table-level conflict granularity. Due to this *table-local* design, current OTFs provide no or only limited support for cross-table transactions that update multiple tables [9, 44, 79, 80].

**Existing solutions trade-off interoperability.** Recent proposals such as DuckLake [69] and Project Nessie [67] support concurrent writers and multi-table transactions by using an *external* database or service as a central coordinator. While effective, they weaken interoperability (Table 1) by introducing non-OTF components. As a result, existing integrations (e. g., ingestion pipelines) and enterprise deployments that rely on the common OTF abstractions could break and/or may require substantial migration.

**An OTF-native solution is challenging.** To guarantee interoperability, a portable, OTF-native approach without any external

**Table 1: Properties of OTF solutions. Current solutions trade off interoperability to support cross-table transactions.**

| | (i) Multi-Write Consistency | (ii) Cross-Table Txns | (iii) OTF Inter-operability |
|---|---|---|---|
| Base OTFs [81, 83, 88] | 👎 | 👎 | 👍 |
| DuckLake [69] | 👍 | 👍 | 👎 |
| Project Nessie [67] | 👎 | 👍 | 👎 |
| Hudi MTT [39] | 👍 | 👍 | 👎 |
| LakeVilla (this paper) | 👍 | 👍 | 👍 |

coordination service is required. However, such an approach is non-trivial for three reasons: First, object storage-based append-only commit-log designs limit read-write concurrency and often serialize conflicting operations, leading to the scalability limits in Figure 1. Second, object stores lack multi-object atomicity, requiring careful commit coordination. Third, because OTF metadata is organized per table (separate commit logs and snapshots), cross-table transactions must atomically advance multiple logs and present readers with a consistent cross-table snapshot. Any native, object-storage-based solution must address these challenges without imposing high coordination overhead or degrading throughput.

**Key contribution: Full, OTF-native transactional semantics.** We provide a native solution by adapting established database techniques, such as concurrency control and recovery, to OTFs. For concurrent updates, we introduce (1) OTF files to manage transactions across systems, (2) cross-table dependency graphs to synchronize operations across systems, and (3) a SAGA-style conflict-resolution mechanism [27]. For consistent reads, we propose (4) a global Multi-Version Concurrency Control (MVCC)-inspired layer [15] that tracks table versions to enable consistent reads with low overhead. The key challenge is to realize these mechanisms under OTF and object-store constraints (e. g., no central component, immutable files) while preserving the interoperability between systems.

**LakeVilla.** We validate our design theoretically and experimentally using LakeVilla (LV), a prototype with two specialized configurations. The write-optimized configuration, LV[Write], implements our optimizations for updates (Sect. 4), and supports write transactions spanning multiple tables in the underlying OTF. The read-optimized configuration, LV[Read], provides globally consistent reads (Sect. 5) with minimal overhead. The full LV library combines both configurations to support more complex HTAP (Sect. 6). LV provides strong transactional guarantees while preserving interoperability (Sect. 7). Our evaluation shows that LV achieves competitive performance across a range of OTF workloads and can be further tailored through dynamic reconfiguration (Sect. 8). The results validate the practicability of the OTF-native design and highlight the differences to alternative approaches.

## 2 BACKGROUND AND MOTIVATION

We summarize the key concepts of Open Table Formats and discuss the limitations of their table-local transactional semantics.

**Per-table organization of data and metadata.** OTFs organize storage per table and separate each table into a data layer and a metadata layer. The data layer stores the actual table contents in

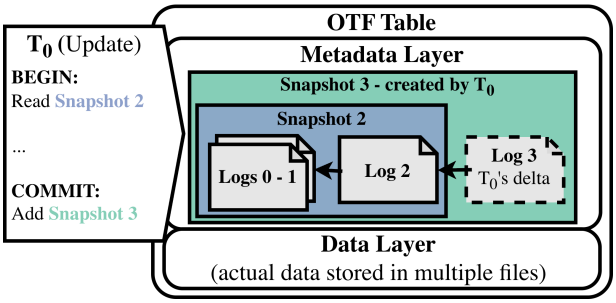


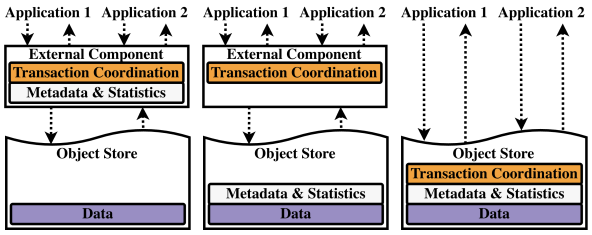

**Figure 2: Simplified metadata layer design of Delta Lake. Data and metadata are organized per OTF table. An ordered log defines snapshots. It causes concurrent writers to contend.**

immutable files, typically using a columnar format such as Parquet [7]. The metadata layer describes which data files belong to a table state and stores additional metadata used for query processing and maintenance. While the concrete metadata design differs across systems, major OTFs share this per-table organization. In the following, we use Delta Lake [9] as a representative OTF. However, our observations apply more broadly to other OTFs (see Sect. 6).

**Table-local versioning and transaction coordination.** Figure 2 illustrates Delta Lake's metadata layer at a high level. For each table, Delta Lake maintains an ordered transaction log, where each committed transaction appends a new log file that records metadata actions such as adding or removing data files. A table state, called a snapshot, is obtained by reading the log up to a specific version. For example, snapshot 2 in Fig. 2 corresponds to the state after applying Logs 0–2. Delta Lake implements table-level ACID guarantees through this per-table log structure. As an example, suppose transaction $T_0$ updates the table based on Snapshot 2. It first reads this snapshot to determine the relevant data files, then stages its update by writing new data files as needed and preparing a new log file with the corresponding metadata changes. If validation succeeds and the commit completes, this log file becomes Log 3 and defines snapshot 3, making the changes visible atomically as the next table version. If the commit fails because a concurrent transaction has already advanced the log, the prepared changes do not become part of the table state. Thus, Delta Lake provides atomic visibility and a serial version history at the granularity of a single table.

**Limitations of table-local transactional support.** Our key observation is that current OTFs use a table as the unit of storage organization and transactional coordination. As a result, their native metadata protocols provide transactional guarantees only at table granularity [20, 40, 85], which leads to three limitations:

- *Write Contention and Starvation*: Writers validate and commit against a table-local version chain. When concurrent transactions attempt to append the next version of the same table, some must abort and retry. Long-running transactions are particularly susceptible to repeated retries (starvation).
- *Lack of Coordinated Cross-Table Writes*: Each table maintains an independent metadata and commit protocol. Consequently, updates spanning multiple tables cannot be committed atomically, and failures during execution can lead to partially committed results.




(a) External (e. g., DuckLake) (b) Gatekeeper (e. g., Nessie) (c) OTF-native (this paper)

**Figure 3: Design paradigms for cross-table transactions. An OTF-native solution avoids external components.**

- *Lack of Cross-Table Snapshot Consistency*: Because tables evolve independently, a transaction accessing multiple tables need not observe states from the same logical point in time. This can lead to inconsistencies across related tables, e. g., violations of referential assumptions.

We propose an OTF-native approach to address these limitations.

## 3 OTF WORKLOADS AND PARADIGMS

Although OTFs have targeted Hybrid Transactional/Analytical Processing (HTAP) [9], the above limitations have emerged with their broader adoption for modern workloads. In the following, we first characterize modern OTF-specific HTAP before discussing how existing approaches address the challenges outlined above.

**Characteristics of modern OTF HTAP.** Modern OTF workloads consist of different phases that combine complex analytical reads with concurrent small updates or continuous data ingestion. Such a dynamic workload demands low-latency reads, near-instant write visibility, and low abort rates. OTF workloads also require global data availability across vendors and machines that use OTFs as a shared storage abstraction. The industrial context of OTFs at SAP mirrors these characteristics. For example, the analysis of production data across use cases, like Ariba (procurement) [75] and S/4HANA (ERP) [74], typically consists of an initial load phase (GBs to TBs) followed by a mix of 15% writes, 30% reads, and 55% hybrid transactions. Writes are typically split between 80% inserts and 20% scattered updates [77]. OTF use cases ingest $10^3$ to $10^6$ events per hour, supporting up to $10^3$ concurrent users with data freshness requirements ranging from seconds to a day.

**Cross-table transactions and requirements.** Many modern workloads require guarantees that go beyond a single table. For example, ETL pipelines, maintenance tasks, and application-driven updates may need to modify multiple related tables atomically, while readers may require mutually consistent views across these tables. In the analytical context, such operations can be long-running and may execute concurrently with other writes. Supporting such workloads, therefore, requires coordination beyond a single table.

**OTF design paradigms.** We categorize the approaches that address the current OTF challenges and workload characteristics into three design paradigms, shown in Fig. 3. The *external* design (Fig. 3a), exemplified by DuckLake [69], offloads transactional coordination and metadata to an additional, centralized system. The *gatekeeper* design (Fig. 3b), as used by Project Nessie [67] or active data lakes [30], splits responsibilities between the OTF and an external component. While both paradigms can accelerate workloads by bypassing S3-specific limitations, their reliance on a centralized service contradicts the engine-agnostic interoperability that was the fundamental motivation for OTFs [9, 42]. Under high load, this centralized service can become a performance bottleneck and a single point of failure. A *native* design should extend the OTF without additional external services and preserve full interoperability. It has recently been used in DeepLake [35] to extend OTFs with AI capabilities while preserving full interoperability. However, the OTF-native paradigm remains unexplored for full transaction support, including cross-table transactions.

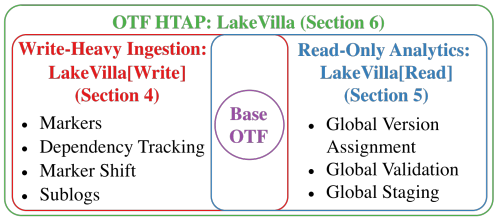



**Figure 4: LakeVilla's mechanisms are encapsulated in two configurations, introducing workload-specific mechanisms.**

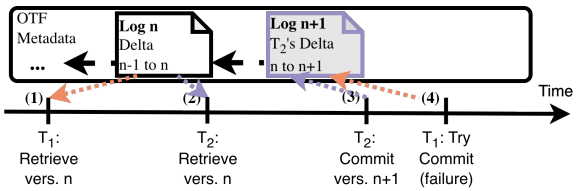
**LakeVilla: OTF-native transactional guarantees.** We close this gap by revisiting existing database techniques and adapting them to be OTF-native and fully compatible with object-store semantics and legacy readers. To handle multifaceted OTF HTAP, LV includes two complementary configurations (Fig. 4): the write-optimized LV[Write] mitigates write contention and coordinates multi-table writes, while the read-optimized LV[Read] ensures cross-table snapshot consistency. LV combines them to provide a unified solution for modern OTF workloads, as we show in our evaluation. LV relies strictly on standard OTF storage assumptions [81, 87, 89] to preserve interoperability, requiring only object-store consistency [4] and conditional/atomic writes [6, 58]. Consequently, it represents a decentralized, non-deterministic coordination approach that uses an object store as a shared storage layer, rather than the global ordering and deterministic nature of *external* or *gatekeeper* designs. In the following, we discuss the design and core concepts to realize this approach.

## 4 OTF-NATIVE WRITE OPTIMIZATIONS

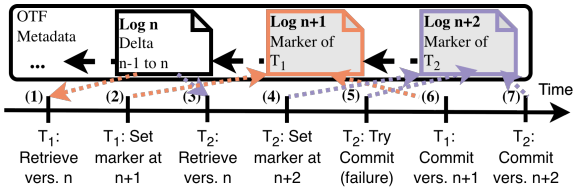
Traditional OTFs usually allow only a single writer per table, leading to *write contention* and preventing *coordinated cross-table writes*, as discussed earlier. We solve these issues by introducing shared write locks called *markers*, and a SAGA-like *sublog* structure per transaction. Markers are an OTF-specific *conflict-resolution* procedure that uses dependency graphs. The following sections discuss these mechanisms in more detail.

### 4.1 The Initial Problem: Starving Transactions

**Why OTFs limit their transactions to a single operation.** As discussed in Sect. 2, OTFs primarily support short transactions (e. g., single reads, appends, or updates) and may cause starvation




**(a) Initial problem: Starving transactions. Transaction $T_1$ (orange) aborts since the short-running $T_2$ (violet) commits and invalidates $T_1$'s state. Repeated retries by $T_1$ lead to starvation.**




**(b) Markers represent shared leases for a certain marker time interval . This avoids starvation of long-running transactions, while bounding the time short-running transactions are blocked.**

**Figure 5: Transaction *starvation* and our *marker* mechanism**

for long-running transactions. This restriction stems from their optimistic, file-based commit protocol [44, 57]. Figure 5a illustrates the problem using Delta Lake. Consider a long-running transaction $T_1$ that begins at snapshot $n$ (Fig. 5a (1)), while a short transaction $T_2$, based on snapshot $n$ (Fig. 5a (2)), commits concurrently and publishes snapshot $n+1$ (Fig. 5a (3)). Since OTFs only validate a Txn at commit time, $T_1$ detects the new snapshot late and must abort because it cannot overwrite log $n+1$ (Fig. 5a (4)). By repeating this scenario endlessly, $T_1$ will never be able to commit, wasting resources and increasing cost, a phenomenon called *starvation*.

**Using markers to address starving transactions.** To remove this OTF limitation and enable arbitrarily sized transactions, we introduce *markers*. Markers are files that seamlessly integrate with the base OTFs and contain dependencies and metadata for cross-system transaction management. They act as file-based shared write locks or leases on specific OTF snapshots and appear on the critical commit path of OTFs. To illustrate how they work, we integrate them in Delta Lake's log chain of the respective metadata layer (Fig. 5b). Since typical OTF commit protocols check for file existence and disallow overwriting most metadata files, a marker prevents other transactions from claiming the same snapshot. Each marker stores the transaction ID, a marker time interval , and auxiliary information required by LV. Only the creating transaction may overwrite its marker within the marker time interval . If the marker time interval expires, other transactions may reclaim the snapshot or invalidate the marker after an additional safety buffer $T_{\text{buffer}}$ .

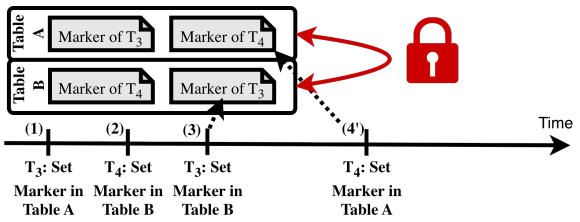
**Marker usage.** Figure 5b shows how markers solve the starvation issue illustrated in Fig. 5a. As before, the long-running transaction $T_1$ retrieves snapshot $n$ (Fig. 5b (1)). However, $T_1$ immediately places a marker for version $n+1$ using an atomic object store request as already required by OTFs [9, 81, 87, 88]. The marker for snapshot $n+1$ also contains an estimation of its execution time, defining the marker time interval (Fig. 5b (2)). Meanwhile, $T_2$ begins (Fig. 5b (3)) and tries placing a marker for snapshot $n+1$ with its own marker time interval . This operation fails due to the used atomic object store requests, and $T_2$ claims n+2 with a marker instead (Fig. 5b (4)). Hence, $T_2$ knows that a concurrent transaction is operating on the same table that might conflict with its own changes. If $T_2$ finishes its execution, it checks if all markers preceding its own were already overwritten or became invalid. During $T_2$'s first commit attempt, this is not the case as $T_1$ still executes and the specified marker time interval has not passed (Fig. 5b (5)). Hence, $T_2$ waits to uphold consistency. When $T_1$ commits, it can replace the marker at snapshot $n+1$ because it has no preceding markers and its marker time interval is valid (Fig. 5b (6)). Now, in $T_2$'s next commit attempt, it checks the newly written log $n+1$ for conflicts and commits or aborts accordingly (Fig. 5b (7)).

**Safety mechanisms for OTFs.** Our marker design avoids write transaction starvation, read overhead, and deadlocks. Because OTF workloads are typically OTF HTAP , markers must appear as empty log entries to minimize read overhead during write synchronization. Markers ensure the consistency of concurrent write transactions by requiring them to wait for the preceding markers within their marker time interval . Since markers are only valid during their marker time interval , they cannot indefinitely lock a table if its transaction crashes. This OTF-native strategy synchronizes writes within a table without introducing any unwanted side effects.

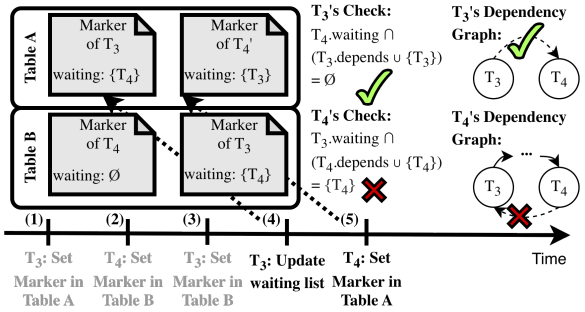
**Handling clock drifts.** Since markers are temporary with respect to their marker time interval across decoupled machines, clock drift poses a potential risk. We analyze the role of time in two dimensions: (I) the local marker time interval held by the creator, and (II) the criteria for other systems to invalidate an expired marker. For (I), marker maintenance is strictly internal to the creating node. Thus, it requires only local monotonic duration tracking, which modern server hardware usually provides. For (II), clock drift could cause a distinct system to prematurely invalidate an active marker. To prevent this safety violation, we introduce a safety buffer ($T_{\text{buffer}}$) derived from standard cloud data center guarantees (e. g., AWS Time Sync ensures 40–100 ms accuracy). Other systems may invalidate a marker only after its marker time interval plus $T_{\text{buffer}}$ has elapsed. Consequently, markers guarantee safety under a bounded clock drift $d$: If $T_{\text{buffer}} > d$, we can purely rely on local duration tracking. In environments lacking a guaranteed time-bound $d$, clock drift can be mitigated using fallback mechanisms already required by OTFs (e. g., ZooKeeper for Delta Lake [9]).

## 4.2 Enabling Multi-Table Updates

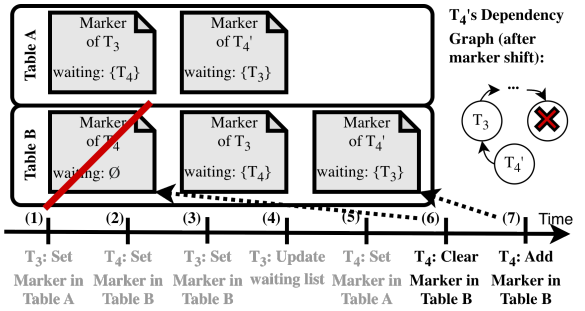
**Why OTFs limit their transactions to a single table.** Markers can prevent transaction starvation when multiple writes concurrently operate on a single OTF table. However, markers do not solve the challenge of *cross-table writes*. Figure 6a illustrates this challenge for two tables, A and B, and two multi-table transactions, $T_3$ and $T_4$. $T_3$ holds a marker in table A (Fig. 6a (1)) and $T_4$ a marker in table B (Fig. 6a (2)). If both transactions continue to operate on the other table (Fig. 6a (3) and (4)), they would depend on each other, forming a deadlock. The marker mechanism cannot solve this problem effectively: Either one transaction aborts or, if we place new markers, we risk indefinite waits. The same applies for original OTFs where the example would cause inconsistencies or partial




**(a) Write contention example: Both transactions $T_3$ and $T_4$ access Table A and B. Markers and their marker time interval alone are not sufficient to avoid deadlocks between transactions.**




**(b) We extend markers with additional *dependency tracking* to enable the detection of conflicts.**




**(c) Transactions use the concept of a *marker shift* to resolve deadlocks and invalidate their own conflicting markers.**

**Figure 6: Example of the *write contention* problem and our dependency check and marker shift mechanisms.**

commits (e. g., $T_3$ can commit in A but fail in B). Hence, OTFs avoid this problem by limiting their transactions to a single table.

**Global dependency tracking and marker shifts.** We introduce dependency tracking and marker shifts to enable cross-table transactions. Dependency tracking is a standard mechanism in database systems for transaction management, whether implemented via locking, commit validation, or dependency graphs [25, 61, 78]. For OTFs that follow the *native* design paradigm, the main challenge is coordinating multiple independent systems that operate concurrently. As a result, no single system on the object store can observe the full concurrency state of an OTF. Instead, each system must adaptively react to changes in the metadata layers to avoid or solve deadlocks and conflicts. Our dependency tracking and marker-shift mechanisms implement this specific behavior.

**An OTF-native dependency tracking mechanism.** To detect conflicts with transactions issued by other systems, potentially from different vendors, a system must observe shared dependency information. We therefore extend markers to detect such concurrent cross-system transactions. When a system encounters an earlier marker with an unknown transaction ID and start time, it infers a global dependency between its transaction and the transaction that created the marker. By placing a marker in every accessed table, as seen in the previous example, a transaction is guaranteed to encounter the markers of all potentially conflicting transactions. During execution, this information allows the system to construct a partial global dependency graph. When a system detects a cycle involving its transactions, it concludes that a deadlock has occurred.

**Building and checking dependencies on the object store.** We extend our marker model as illustrated in Fig. 6, so that individual systems can construct a partial dependency graph despite the high latency of object stores: When a transaction accesses a table, it records its transitive dependencies and maintains the set of transactions known to wait for it in memory. The system publishes the current dependency state by storing it in all relevant markers and overwriting them whenever the dependency graph changes. When another transaction later reads such a marker, it checks whether adding the new dependency would create a deadlock. Specifically, it intersects the marker's dependency set with its own in-memory dependency set, including itself. A non-empty intersection indicates a cycle: the transaction would wait, directly or transitively, for a transaction that already depends on it. An empty intersection indicates that the new dependency is cycle-free. In Fig. 6b, this check succeeds for $T_3$ accessing Table B, so $T_3$ updates its markers to reflect its new dependency on $T_4$ (Fig. 6b (4)). In contrast, in Fig. 6b (5), the check fails for $T_4$ when it accesses Table A, because the intersection is non-empty. From $T_4$'s perspective, adding this dependency would create a cycle in the dependency graph. Thus, the system detects a potential deadlock between $T_3$ and $T_4$.

**Marker shifts on conflict.** When a transaction recognizes a conflict using marker-based dependency tracking, it cannot notify the other conflicting transaction. Otherwise, we would incur additional overhead from object store requests and face compatibility issues across different systems. Thus, the only option to preemptively solve this conflict is for the recognizing transaction itself to give up the problematic snapshot versions using a *marker shift*. A *marker shift* essentially inverts any problematic outgoing edge from the respective transaction node in its partial dependency graph. Figure 6c illustrates this mechanism for our example in Fig. 6. $T_4$ recognizes a conflict with $T_3$. Hence, it performs a *marker shift* and collects any tables where it has not found a preceding marker from $T_3$ (Table B). Then, $T_4$ invalidates its own marker by overwriting it with an empty component of the base OTF (in Delta Lake: empty delta) (Fig. 6c (6)). Next, $T_4$ places a new marker at the latest available snapshot in table B (Fig. 6c (7)). This mechanism guarantees that any conflicting marker in table B is now preceding this marker. $T_4$ confirms this by reading $T_3$'s markers and updating its partial dependency graph. This marker shift effectively breaks the cycle between $T_3$ and $T_4$, guaranteeing that no deadlocks occur. As an additional safety measure to account for the object store latency, the transaction runs an internal, random timeout before it can perform

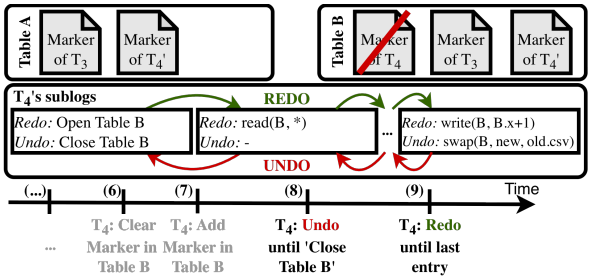



**Figure 7: Simplified recovery procedure after the *marker shift* in Fig. 6. By using *sublogs*, we avoid a full transaction restart.**

another marker shift. This random delay guarantees that no infinite loops of multiple transactions performing marker shifts occur.

## 4.3 Upholding Consistency After Marker Shifts

**Consistency after a marker shift.** Although dependency tracking and marker shifts preserve the global order of conflicting transactions, a marker shift can temporarily invalidate the local view of the shifting transaction. Transactions that do not perform a marker shift remain unaffected because markers isolate them, and each transaction starts from the last committed version. Hence, only transactions performing a marker shift can become inconsistent. Inconsistencies can occur because the shifting transaction does not yet know what the preceding transaction modified. A conservative solution is therefore to abort and restart the shifting transaction from scratch with the new markers already installed, similar to strict two-phase locking. However, restarting is unnecessary if there are no conflicts, or very expensive if the shifting transaction is complex. To avoid this overhead, we introduce a *self-healing mechanism*.

**Self-healing through sublogs.** To let a transaction recover after a marker shift without restarting from scratch, we persist recoverable transaction state in *sublogs* that can be shared across systems. Each transaction stores its sublogs in a private temporary directory on the object store, inspired by SAGAs [27]. The transaction creates this directory during its begin phase and records its object store key in all of its markers so that other systems can locate the sublogs if needed. To guarantee uniqueness, the directory is created with an atomic or conditional request [6, 31, 58] and removed after commit.

**Physical and logical sublogs.** We use two types of sublogs, inspired by Write-Ahead Logging and ARIES [59], to balance durability and tracking overhead. A *physical sublog* records concrete changes to the OTF data layer, such as adding or removing a file. Transactions use physical sublogs when no marker shift can occur, for example, in single-table transactions without preceding markers or in static, conflict-free operations such as blind appends. A *logical sublog*, in contrast, records operations on the data itself, such as updating a row or reading a range of entries. Transactions use logical sublogs for operations whose effects may need to be replayed after a marker shift, such as multi-table updates or primary-key inserts. Together, physical and logical sublogs enable a system to automatically repair a transaction after a marker shift.

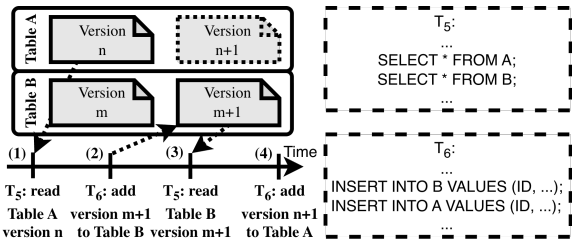
**Recovering consistency with sublogs.** Figure 7 illustrates how both sublog types are used after the marker shift in Fig. 6. Before the shift, $T_4$ has already logged operations (1)–(7) as a sequence of physical and logical sublogs. Without sublogs, $T_4$ would have to restart because its earlier read of the full table B is invalidated by $T_3$'s update. Instead, $T_4$ now performs an UNDO process (Fig. 7 (8)) by sequentially executing the undo information of each ordered sublog until its first access to table B. $T_4$'s first undo instruction swaps its previously written file 'new' with the previous one(s) 'old.csv'. This process is logged as a physical sublog because it is static and explicitly names the original file. After undoing more changes in between, $T_4$ reaches an empty undo instruction representing a NOOP compensating for a read. Lastly, the redo process stops at the records to open/close table B. Now, $T_4$ executes the REDO instructions from this sublog to restore consistency (Fig. 7 (9)). It first rereads table B (`read(B,*)`), now including $T_3$'s changes, and then replays the remaining operations. The final REDO step adds a new data file that increments all tuples in column `x` by 1. Because this step records the transformation to apply (`B.x+1`) rather than naming a specific file, it is logged as a logical sublog. If $T_4$ reaches the newest sublog entry, the transaction has successfully repaired itself. If any REDO or UNDO fails, the transaction aborts or automatically restarts from its sublog.

# 5 OTF-NATIVE READ OPTIMIZATIONS

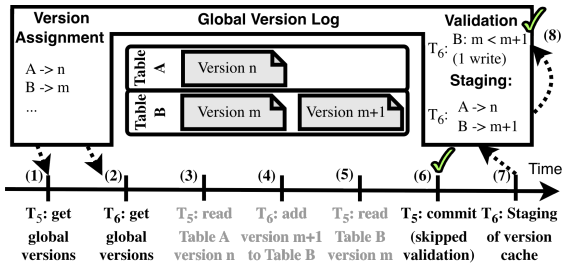
The four mechanisms introduced so far (markers, dependency tracking, marker shifts, and sublogs) coordinate and repair concurrent write transactions. However, OTF workloads also include read-only analytical transactions. For such transactions, write-oriented mechanisms introduce unnecessary overhead because they require systems to write markers or sublogs to the object store, which incurs object store latency. Write transactions can often hide this cost because they already modify object store metadata, but read-only analytical transactions do not. To support multi-table analytics efficiently, we therefore introduce the *global version log*, an OTF-native mechanism inspired by Multi-Version Concurrency Control (MVCC) [14] and adapted for analytical OTF workloads.

**Why multi-table reads can become inconsistent.** Original OTFs define transactional guarantees only at the level of individual tables. As a result, a transaction that reads multiple tables may observe a cross-table state that was never committed atomically. Figure 8a illustrates this *cross-table snapshot consistency* problem: Assume a read-only transaction $T_5$ accesses tables A and B while an update transaction $T_6$ modifies both tables. $T_5$ first reads snapshot $n$ of A (Fig. 8a (1)) and later observes a newer snapshot $m + 1$ of B (Fig. 8a (2)–(3)). If $T_6$ continues to modify table A in the meantime, then $T_5$ observes a non-atomic cross-table state. For example, it may see foreign key references or related updates in B that correspond to records in A that are not yet visible. The result is a violation of referential integrity and, more generally, of global ordering across tables. Our previous mechanisms do not address this problem, because they coordinate conflicting transactions but do not provide globally atomic visibility across multiple tables. Consequently, consistent multi-table analytics require an additional mechanism for coordinated cross-table snapshot assignment and commit visibility.

**A global version log for cross-table snapshot consistency.** To provide such coordination, we adapt MVCC to the OTF setting through the *global version log*. The key challenge is that classical MVCC assumes a single database system, whereas OTFs enable




**(a) Current OTFs do not guarantee cross-table snapshot consistency. $T_5$ may observe a non-consistent state across table A and B since $T_6$ concurrently modifies table A.**




**(b) The global version log assigns a consistent snapshot across tables, allowing $T_6$ to validate its commit before making new snapshots globally visible.**

**Figure 8: Achieving cross-table snapshot consistency with the global version log.**

multiple independent engines to operate over an object store. We therefore realize MVCC in a file-based form that can be shared across systems, as illustrated in Fig. 8b. This new component maintains globally visible table snapshots and serializes commits across systems to prevent *global inconsistencies*. Overall, the idea of the global version log is simple: At the beginning of a transaction, the system reads the global version log once to obtain a globally consistent set of table snapshots, referred to as *version assignment*. During execution, the transaction may read only snapshots up to the versions assigned in that step. If the transaction modifies a table, it first performs the corresponding table-local commit according to the original OTF protocol. These intermediate table-level changes are not yet globally visible because the earlier version assignment of other transactions constrains them. Afterward, the transaction *stages* and *validates* its changes. If validation succeeds, it atomically advances the globally visible snapshots of all affected tables. In this way, the mechanism ensures that transactions access globally consistent snapshots and prevents *global inconsistencies*.

**Structure and version assignment.** The global version log consists of two files: a *version list* and a *synchronization file*. Version assignment and validation use a version list. It stores all known tables in the OTF, their most recent table snapshot, and the object store key of the synchronization file. A transaction obtains a globally consistent snapshot set by reading the version list once during its begin phase. In Fig. 8b, $T_5$ and $T_6$ read the version list in their begin phases (Fig. 8b (1) and (2)). As a result, both transactions retrieve the same snapshot set, allowing them to access table A up to snapshot $n$ and table B up to snapshot $m$.

**Staging and synchronization.** Write transactions that have completed their local work enter the *staging* phase. This phase uses the synchronization file, which resides at a shared object store key known to all participating engines or obtained from the version list during version assignment. Its purpose is to globally serialize commit attempts across systems. A transaction that wants to commit must acquire temporary exclusive ownership of this file. In Fig. 8b (7), $T_6$ does so by writing a copy of its proposed version list to the shared key. As with markers, it uses non-overwrite flags, conditional write interfaces, or other synchronization mechanisms provided by the object store. If the write succeeds, the transaction proceeds to the validation step. Otherwise, it waits and retries.

**Validation and commit.** Once a transaction has claimed the synchronization file, it validates its execution against the current global state. In the example, $T_6$ successfully acquires the synchronization file and proceeds to validation (Fig. 8b (7)). Classical MVCC expresses validation in terms of read and write sets. We adopt the same logic, but map it to OTF snapshots. A transaction's *read set* contains all tables read but not written during execution, whereas its *write set* contains all tables for which it created a new snapshot. Based on this mapping, validation applies three rules:

R1: $writeset = \emptyset \Rightarrow$ skip R2 and R3 (read-only)
R2: $\forall r \in readset : global(r) = local(r)$ (read set validation)
R3: $\forall w \in writeset : global(w) + \#writes = local(w)$ (write set validation)

**Validation rules.** R1 covers read-only transactions. If a transaction does not modify any table, it can commit immediately after version assignment, because it has already read a globally consistent snapshot set and did not change OTF metadata. Such a transaction can be logically ordered among write transactions by the point at which it performs version assignment. R2 checks for read-write conflicts through read set validation. If another transaction committed an update to a table in the read set after version assignment, then the globally visible snapshot of that table has advanced, and the equality of R2 no longer holds, indicating a read-write conflict. R3 checks for write-write conflicts through write set validation. For each written table, the transaction compares the globally visible snapshot in the version list with the local snapshot it produced. The difference between the two corresponds to the number of snapshots created by the transaction for that table. If another transaction commits a conflicting update in the meantime, this violates their relationship.

**Validation in the running example.** The example in Fig. 8b illustrates the application of these rules. Because $T_5$ is read-only, it satisfies R1, skips staging and validation, and commits immediately (Fig. 8b (6)). In contrast, $T_6$ claims the synchronization file and executes validation (Fig. 8b (7)). Its R2 check succeeds because it read table A at snapshot $n$, which is still the current version in the global version list. Its R3 check also succeeds for table B: $T_6$ created snapshot $m$+1 for B, while the version list still records snapshot $m$; because the transaction produced exactly one new snapshot, the condition holds (Fig. 8b (8)). $T_6$ can therefore commit by replacing the version list with the staged one stored in the synchronization file. This object store operation atomically advances the globally visible snapshots for all tables, guaranteeing global consistency. Finally, $T_6$ removes the synchronization file it created at the shared key, allowing the next transaction to stage its commit.

## 6 LV: SUPPORTING HYBRID WORKLOADS

**A hybrid design for OTF HTAP.** Our write and read optimizations address complementary requirements of transactional OTF workloads. The write optimizations coordinate conflicting updates, whereas the read optimizations provide cross-table snapshot consistency for analytical queries. Since typical OTF workloads combine concurrent analytical reads with transactional writes, LV integrates both optimizations into a single hybrid architecture. It addresses all three core challenges (*write contention*, *coordinated cross-table writes*, and *cross-table snapshot consistency*) within a single OTF-native system. Next, we explain how LV assigns mechanisms to different transaction types and how it synchronizes them.

**Read-only transactions.** LakeVilla applies read optimizations to read-only transactions using the global version log. Thus, analytical transactions obtain a globally consistent snapshot set without writing markers or sublogs. As long as a transaction remains read-only, it can execute exactly as described in Section 5. If such a transaction later decides to modify an OTF table, it must switch to the write path. To do so safely, it scans for markers before writing. If it encounters a marker, it aborts and restarts with a new version assignment to continue using the write-optimized mechanisms.

**Write transactions.** Transactions that modify OTF metadata leverage our write optimizations. They therefore use markers, dependency tracking, marker shifts, and sublogs to coordinate with conflicting updates. To remain visible to read-only transactions, however, write transactions also publish their committed snapshots through the global version log. As a result, each write transaction still passes through the staging and validation steps of the global version log before commit. In LV's unified architecture, however, these validation steps become simpler for write transactions. Once a transaction has successfully acquired all required markers and completed conflict detection, markers already ensure the essential correctness conditions captured by R2 and R3: The transaction observes the most recent relevant snapshots, waits for preceding conflicting transactions when necessary, and repairs its state through self-healing if marker shifts occur. The global version log, therefore, serves primarily to publish the final globally visible snapshot set, while markers continue to manage fine-grained write coordination.

**Influence of workload characteristics** LakeVilla's hybrid approach unifies the strengths of read and write optimizations to provide full ACID transactions for concurrent mixed workloads. The workload composition, however, directly influences the system's behavior. While isolation is always guaranteed, many concurrent long-running write transactions increase the number of marker shifts and redo operations. We deliberately chose to push this overhead to the write-path since read queries dominate OTF workloads. Besides the workload mix, transaction size impacts performance[1]. For write-heavy workloads, large transactions increase contention, raising the cost of the sublog's mechanism and reducing throughput. In contrast, read-heavy and read-only workloads benefit from larger transactions by amortizing metadata access overhead via the global version log while avoiding conflicts for pure reads.

**Hybrid and adaptive configurations.** To address varying workload characteristics more efficiently, LakeVilla allows selectively deactivating optimizations to offer a pure read or write-optimized path. This adaptivity is beneficial for phase-based mix workloads, where write-intensive data ingestion is followed by read-intensive data analytics. During a configuration change to/from LV[Write], LV enforces causal consistency; otherwise, linearizable consistency (cf. Tab. 2). As OTF workloads typically schedule these phases globally, configuration changes apply system-wide at runtime.

**Table 2: Achievable ACID guarantees for OTFs by using LV**

| | Default OTF | LakeVilla [Write] | LakeVilla [Read] | LakeVilla |
|---|---|---|---|---|
| **A**tomicity | Table-level | Table-level | Global | Global |
| Distributed **C**onsistency | Limited | Causal | Linearizable | Linearizable |
| **I**solation | Limited | Serializable | Serializable | Serializable |
| **D**urability | By design | By design | By design | By design |

**Conflict granularity.** Our design intentionally adopts table-level conflict detection to guarantee compatibility with existing OTF solutions. While table-level granularity can introduce coarse-grained conflicts, we mitigate overhead by leveraging our sublog's self-healing mechanism, which minimizes unnecessary transaction aborts. Similar to other OTF solutions, users can optimize performance by partitioning a single logical table into multiple physical OTF tables. While finer-grained conflict detection is a valid alternative, it introduces high metadata overhead and increases tail latency due to frequent interactions with the object store.

**Generalization across OTF backends.** Although our prototype uses Delta Lake to implement LV, the underlying concepts and optimizations are independent of the base OTF. Some implementation details, such as marker placement, depend on the base OTF's commit log or metadata path. For example, Iceberg and Hudi could store markers in manifest files or commit timelines.

## 7 TRANSACTIONAL GUARANTEES AND INTEROPERABILITY OF LAKEVILLA

Our LakeVilla prototype is a library that implements all optimizations (Sect. 6), but also allows using them selectively as LV[Write] (Sect. 4) and LV[Read] (Sect. 5) to support a specific workload. This section examines the ACID guarantees that LV and its specialized configurations provide and how LV preserves OTF interoperability.

### 7.1 ACID Guarantees

**Theoretical overview.** Table 2 summarizes the ACID guarantees of LV and its specialized configurations. Since LV inherits most guarantees from base OTFs, we focus strictly on the upgrades: LV[Read] and LV elevate atomicity to a global guarantee. For distributed consistency, LV[Write] upgrades to causal, and LV[Read]/LV to linearizable. Further, LV elevates Isolation guarantees to serializable [1]. Next, we empirically validate these guarantees using Hermitage [45], comparing LV with Delta Lake, Iceberg, and DuckLake.

**LV Prototype.** Our C++ library[2] exposes a per-transaction API for C, C++, Rust, and Java (via JNI). It implements a baseline Delta Lake client (LV[Delta Lake]), and all LV optimizations (Write, Read).

[1]Formal proofs, TLA+ model, Additional Experiments and Traces available: https://github.com/goetztj/LakeVilla

[2]LakeVilla prototype available: https://github.com/goetztj/LakeVilla.

**Table 3: Empirical evaluation of ACID guarantees using the Hermitage test suite.**

| Anomaly | Spark (OTF) | DuckLake (Postgres) | LV (library) | Trino (OTF) | Trino (LV) |
|---|---|---|---|---|---|
| Write Cycles, Aborted Reads, Interm. Reads, Circ. inform. flow, Obs. Txn. Vanishes, Pred.-M.-Preceders, Lost Update, Read Skew | 👎 | 👍 | 👍 | 👎 | 👍 |
| Write Skew | **N.A.** | 👍 | 👍 | **N.A.** | 👍 |
| Anti-Dep. Cycles | **N.A.** | ✹ | 👍 | **N.A.** | 👍 |
| Atomic Commit | 👎 | 👍 | 👍 | 👎 | 👍 |

prevented: 👍; not prevented: 👎; some tests failed: ✹; not applicable: **N.A.**

The optimizations can be applied dynamically for each transaction, enabling LV to adapt to changing workloads, as shown in Sect. 8.4.

**Baselines.** Throughout the rest of the paper, we use Spark, Trino, and DuckLake. Spark and Trino are distributed systems implementing the base OTF protocols. Neither provides any transactional properties by default. DuckLake is an extension running in DuckDB with a Postgres Catalog as backend. It represents an external design paradigm and provides transactional guarantees similar to LV.

**Empirical evaluation using Hermitage** As shown in Table 3, Delta Lake and Iceberg fail most Hermitage tests because they do not support multi-table transactions. In contrast, LV prevents all tested anomalies across its variants. DuckLake only fails the G2 anomaly test because the anomaly involves overlapping read-write dependencies that its snapshot-isolation semantics do not prevent. We additionally include an *Atomic Commit* test (last row in Tab. 3) to evaluate cross-table commit atomicity and referential integrity. Here, LV and DuckLake succeed, whereas base OTFs expose intermediate table-level states because they commit each table separately. Overall, the empirical results match the theoretical guarantees of the respective LV variants.

## 7.2 Interoperability Case Study

Beyond correctness, we designed LV to preserve the interoperability that makes OTFs attractive. We demonstrate how LV enables interoperability by integrating it in Trino [93] and discussing the interoperability of alternative approaches.

**LV interoperability with Trino.** In this case study, most of the engineering effort is spent mapping LV's Apache Arrow [82] output to Trino's internal Page format[3]. Because LV introduces no external coordination services and relies solely on Arrow as its data interface, it is easy to integrate into existing client implementations. We validated our integration by extending the Hermitage experiment. As shown in Tab. 3, Trino with LV provides the same transactional guarantees as the standalone LV library. This case study demonstrates that existing analytics engines can integrate LV with modest engineering effort.

[3]Proof of Concept Trino implementation here: https://github.com/goetztj/trino

**Interoperability of LV and other solutions.** Our practical evaluation shows that LV is highly interoperable. Other approaches, such as DuckLake [69, 94], also offer some degree of interoperability but make different trade-offs. First, LV's library-based architecture keeps the transactional mechanisms in the middleware layer rather than requiring clients to implement them directly, unlike systems such as DuckLake and Nessie. Second, LV relies solely on object store guarantees that are already widely supported by cloud providers [4, 6, 31, 58], thereby avoiding additional services or centralized infrastructure. Third, LV preserves compatibility with existing OTF deployments and legacy clients because it produces a valid base-OTF state without requiring data migration or client modifications. By contrast, other solutions migrate OTF (meta-)data and expect all clients to adopt their protocol.

# 8 EVALUATION

Having established the transactional guarantees and interoperability of LV, we now evaluate its performance characteristics. Our goal is to understand how LV performs relative to existing OTF solutions and which LV configuration is best suited for a given workload. To this end, we compare LV with DuckLake and Project Nessie using YCSB [18, 70], TPC-DS [92], and TPC-C [91]. We further study LV under low and high contention, with different transaction sizes, and under phase-driven OTF workloads using LSTBench [16].

## 8.1 Experimental Setup

**Hardware.** LV runs on a server with an Intel(R) Xeon(R) Platinum 8260 CPU at 2.40GHz, 2 sockets, 64 cores per socket, and 1048 GB of RAM. Unless otherwise stated, each transaction uses up to 8 cores.

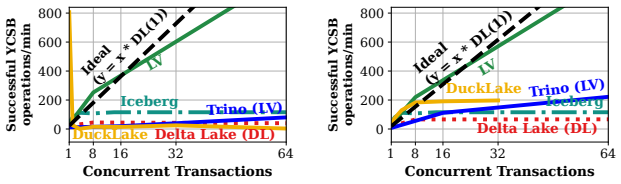
**Baselines.** We compare LV against DuckLake and Spark with Delta Lake, Iceberg, and Project Nessie under a saturation testing model (unlimited input/offered load). DuckDB runs on the same server as LV, whereas the DuckLake catalog uses PostgreSQL on a separate server. Spark 3.5.3 runs on Hadoop 2.12 with Java 11 across 8 machines, with 2 workers (8 cores and 64GB RAM each) per machine. HIVE 3.0.0, its PostgreSQL backend, and Project Nessie catalogs run in two separate Docker containers on a distinct machine. We ensured that all systems have the exact same number of resources available to guarantee fairness. As OTFs, we use Delta Lake 3.0.0 (checkpoint interval 10), Iceberg 1.5.0, and Project Nessie 0.107.1.

**Object Stores.** We use AWS S3 [5] and MinIO (now AIStor Free) [84]. Both provide strict consistency [4, 76] and conditional writes [6].

## 8.2 System Comparison

We begin by comparing the end-to-end overhead of LV with related systems. Because the default use case for OTFs comprises concurrent read and updates, we focus on three aspects: scalability under concurrency, support for more complex transactional workloads, such as TPC-C, and overhead for multi-table analytical reads.

**Concurrency comparison using YCSB.** Concurrency among read and write transactions is one of the key OTF challenges. To isolate the effect of concurrency, we revisit the YCSB experiment from the introduction, in which concurrent clients execute single-operation transactions on a single table. Since OTF HTAP combines reads with scattered updates [9, 16] (see Sect. 3) , we focus on YCSB A (write-heavy) and B (read-heavy). As YCSB operates on a single




(a) Write-heavy (A, 50% reads) (b) Read-heavy (B, 95% reads)

**Figure 9: YCSB with Spark (Delta Lake, Iceberg), DuckLake, and LV. LV scales close to the ideal of its baseline Delta Lake.**

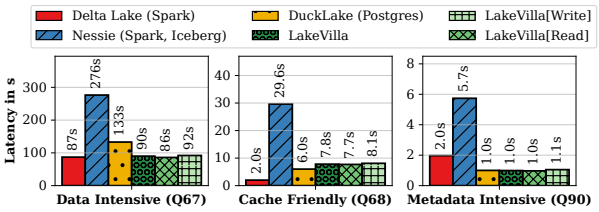



**Figure 10: Data loading latency of selected TPC-DS queries (sf 1000). LV provides competitive performance.**

**Table 4: TPC-C (1 WH, 4 clients) [transactions (txns)/min] on MinIO (1h). LV upholds consistency even for complex txns.**

| System | Transactional Throughput | Abort Rate | #Forced Redos (Timeouts) |
|---|---|---|---|
| Delta Lake (Spark) | 0.00 txns/min | 100.00% | - |
| Nessie (Spark, Iceberg) | 0.00 txns/min | 100.00% | - |
| DuckLake (Postgres) | 0.20 txns/min | 99.98% | - |
| LakeVilla | **2.81** txns/min | **21.92%** | **0** |
| LakeVilla[Write] | 2.79 txns/min | 29.42% | 1 |
| LakeVilla[Read] | 0.62 txns/min | 78.10% | - |

table, this scenario depicts a high contention workload due to the OTF's table-level conflict granularity. For DuckLake and LV, we expect improved scalability with increasing concurrency compared to base OTFs. We omitted Project Nessie in this experiment because it does not automatically resolve transactional conflicts [86]. Trino (OTF) showed the same behavior as Spark using Delta Lake.

**Results.** LV offers consistently low abort rates (< 1%) (data available on LV's Github) and substantially improved throughput (cf. Figure 9). Due to stable p50 latency (7672 ms for multi-op), it scales close to Delta Lake's ideal throughput until conflict resolution and object-store latency become the dominant factors, impacting p99 commit latency (up to 9411 ms). DuckLake achieves higher throughput under low contention by committing in-database metadata faster and reducing object-store interactions through batched updates. However, as contention increases, its throughput drops below LV. We attribute this to DuckLake's table-level conflict detection, which leads to more aborted transactions. This effect is most visible in the write-heavy workload A (Fig. 9a), where conflicts occur more frequently than in workload B (Fig. 9b). Under higher concurrency in YCSB B, DuckLake additionally became unstable and repeatedly crashed once we reached 64 concurrent transactions. In summary, LV's conflict-resolution and self-healing improve scalability and outperform approaches that rely on abort-based conflict handling.

**Complex transactions using TPC-C.** Next, we evaluate LV's overhead with more complex transactional workloads, including multi-table reads and writes, and hence report throughput as txn-s/min. We use TPC-C to stress the conflict-handling and self-healing mechanisms of LV and related systems. To trigger these mechanisms as often as possible, we configure a single-warehouse setup with 4 concurrent clients and run the experiment for one hour. This results in a high contention scenario across multiple tables. We expected LV[Write] and LV to incur more overhead than DuckLake[4] as they perform synchronization, merging, and self-healing on the object store. We omit Nessie and Delta Lake because they do not automatically merge such conflicts in a meaningful way [67, 89].

**Results.** Table 4 displays our results. LV[Write] and LV heal transactions on conflict, but occasionally hit a timeout or retry limit, which lowers throughput and raises their abort rate to ca. 20%. Contrary to our expectations, DuckLake performs worse than LV, which we attribute to the large number of conflicts. This suggests that DuckLake could benefit from a native self-healing mechanism. The experiment highlights the costs of conflict handling in complex OLTP workloads and the benefit of LV's self-healing.

**Multi-table analytics with TPC-DS.** The previous section established that LV can provide cross-table consistency in mixed workloads. This ability, however, comes with overhead for pure analytical workloads due to additional coordination. To quantify the overhead, we evaluate TPC-DS at scale factor 1000 under no contention caused by concurrent transactions using representative queries (Q67, Q68, Q90), following prior OTF comparisons [42]. Since LV is a storage layer rather than a query engine, we focus on data-loading rather than query execution time. For better introspection and experiment statistics, we use MinIO instead of S3.

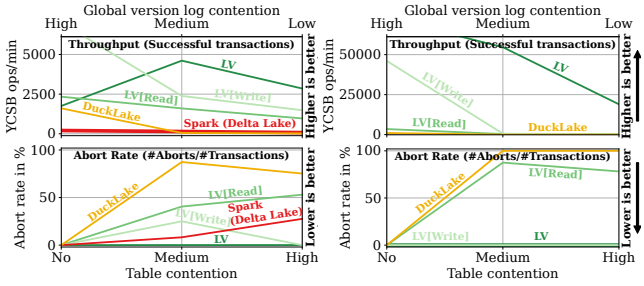
**Results.** We report data-loading latency in Fig. 10, as throughput metrics mask the observable differences. For data-intensive queries (Q67), all systems perform similarly because they must fetch all data from the object store. For cache-friendly queries such as Q68, LV suffers from the lack of caching compared to the baselines with built-in caches. LV[Write] shows the highest overhead because it must place markers even for reads. For metadata-intensive queries (Q90), LV and DuckLake perform comparably, while Delta Lake is slower. Spark with Nessie shows higher absolute latency due to using Iceberg. In summary, even without caching, LV remains competitive for analytical workloads while providing stronger transactional guarantees than traditional OTFs.

## 8.3 Workload Sensitivity

The preceding experiments demonstrate that LV and OTF approaches are highly workload-sensitive. Hence, we isolate the primary factors driving this variance: table-level contention, the number of accessed tables, and the effect of data freshness on latency.

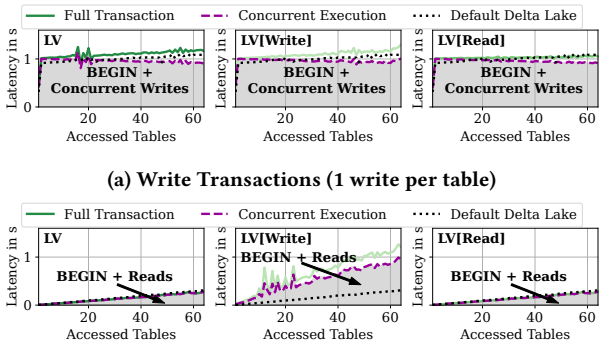
**Table-level contention.** We evaluate the effects of contention using YCSB A, as the closest to OTF HTAP, with 32 clients by scaling the concurrently accessed tables across three scenarios: *No* (32

[4]Benchbase DuckLake integration: https://github.com/goetztj/benchbase




(a) Single-Op Transaction (b) Multi-Op Transaction

**Figure 11: YCSB table-contention with 32 MinIO clients and unlimited arrival rate. LV maintains competitive throughput and low abort rates under both high and low contention.**




(a) Write Transactions (1 write per table)

(b) Read-only Transactions/Analytics (1 read per table)

**Figure 12: Table Scaling (read and write transactions) using LV, LV[Write], and LV[Read]. For all configurations and transaction types, we observe a linear increase in commit latency with the number of accessed tables.**

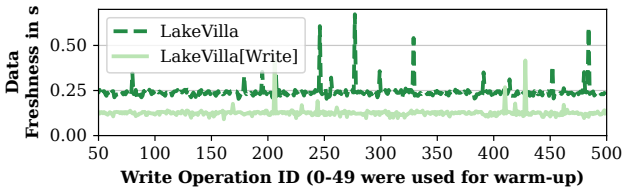



**Figure 13: Data freshness for new data using LV. LV guarantees consistency at the cost of data visibility.**

clients on 32 tables), *Medium* (8 clients per table, 4 tables), and *High* (all clients on 1 table) contention. Since all approaches use table-level conflict granularity, adjusting the table distribution modulates contention between clients. The expectation is that higher contention decreases throughput across all approaches.

**Results.** The results in Fig. 11 show that LV and LV[Write] maintain consistently low abort rates (0.0%–0.8%) due to sublog self-healing, whereas LV[Read], DuckLake, and Spark exhibit increasing abort rates as contention rises. We also observe a performance crossover between LV and LV[Write], and a higher abort rate for LV[Write] under medium contention with single-query transactions (Fig. 11a). This effect stems from the interaction between the global version log and table-level contention: with zero table-level contention (32 clients on 32 tables), LV[Write] avoids global synchronization for disjoint table accesses and therefore outperforms LV, while LV still enforces global serialization. Under medium and high contention, however, LV benefits from coordinating through both the global version log and table-level markers, which reduces redos and aborts compared to LV[Read]'s purely local conflict resolution. The crossover is mainly visible for short transactions because they incur frequent commit and synchronization costs. With longer transactions (Fig. 11b), these costs are amortized, and LV benefits from its read optimizations compared to the single-query setting (Fig. 11a). Overall, the experiment highlights the tradeoffs between LV configurations and explains why OTF approaches perform well for low-contention workloads (e. g., TPC-DS) but become more sensitive under high contention (e. g., YCSB, TPC-C).

**Scalability with number of tables.** The number of tables accessed by a transaction can inherently impact LV and table-based OTF formats. To evaluate the scalability of our solution, we design a synthetic, contention-free experiment that scales the number of accessed tables from 1 to 64. All tables are derived from the TPC-H (SF1000) customer table, and transactions execute point reads or point writes of a single row per table. We compare to vanilla Delta Lake, executing each table-access as a single transaction.

**Results.** We report the end-to-end latency of each transaction using a single client, since the overhead of multi-table transactions primarily manifests as increased latency. Our results in Fig. 12 show that commit latency for writes scales linearly with the number of tables, due to object-store write overhead. The highest latency increase occurs between 1 and 2 tables, where a transaction transitions to a multi-table context, triggering parallel cross-table dependency tracking (LV, LV[Write]) and metadata parsing (logs, JSON, Parquet). Hence, write transactions (Fig. 12a) for LV and LV[Write] have a flat penalty of ~100 ms across all table counts. For the read-only workloads (Fig. 12b), LV and LV[Read] benefit from the read optimizations, allowing them to slightly outperform vanilla Delta Lake, while LV[Write] (middle) exhibits higher overhead due to marker maintenance. These results confirm the observations from our previous experiments: YCSB achieves high throughput by executing simple operations on a single table, whereas TPC-C transactions incur a multi-table penalty due to marker and dependency tracking.

**Data freshness requirements.** OTF workloads come with different data freshness requirements. LV's configurations also expose tradeoffs between freshness and coordination overhead. We quantify freshness as the time until a write becomes visible to concurrent transactions. We compare LV, which provides global consistency, with LV[Write], which provides causal consistency (see Sect. 7). Our synthetic microbenchmark uses two concurrent transactions: One inserts a timestamped row, while the other repeatedly attempts to read it and measures the elapsed time until it becomes visible. This experiment simulates a low contention scenario with a single reader and a single writer accessing the same table.

**Results.** Figure 13 shows that LV exhibits a higher freshness delay than LV[Write] due to its synchronization via the global version log. The average delay is 117 ms, with a maximum of about 600 ms.

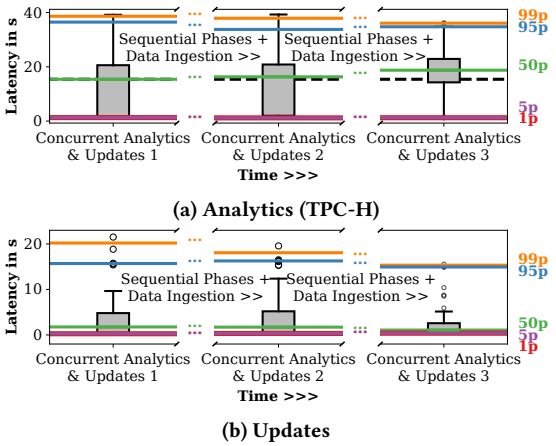



**Figure 14: LSTBench (scale factor 100, W3): Concurrent phases of a multi-phase OTF workload. LV does not impact the stability of its base OTF.**

This overhead corresponds to two to three S3 write latencies [24]. These writes lie on the global version log's critical path for staging and publishing synchronized commits. Thus, LV provides global consistency at the cost of a predictable delay in data visibility.

## 8.4 Long-Running Workloads

**Stability.** A key question for executing phase-driven workloads (see Sect. 3) remains: whether it would preserve the effectiveness of the OTF metadata layer over long execution periods. To study this behavior, we use LSTBench [16], which models changing ingestion, maintenance, analytical, and OTF HTAP phases inspired by industrial OTF workloads [16]. Each phase defines its own scenario, concurrency, contention , and duration.

**LSTBench Benchmark.** To test whether LV decreases the effectiveness of the OTF metadata layer, we use LSTBench's W3 workload with TPC-H sf 1000 as a baseline. This particular workload consists of sequential ingestion and OTF HTAP phases during which LV becomes active. We expect that LV does not degrade the stability of the OTF metadata layer, because its coordination mechanisms primarily create temporary coordination objects.

**Results.** Figure 14 shows the latency distributions for reads and updates across the three OTF HTAP phases. Despite the sequential phases that increase data volume and throughput, 50p, 99p, and 1p latency remain relatively stable with rare aborts (< 2%). The latency of analytical queries in Fig. 14a grows only sublinearly with data size. This result indicates that the OTF metadata layer continues to prune data effectively even when LV manages concurrent scattered transactions. Interestingly, the third concurrent phase shows a lower 99p latency despite the larger data volume. This effect is caused by the increased spread of the data across more files, making the OTF metadata layer more effective. The same effect appears for updates in Fig. 14b: increased data distribution reduces worst-case rewrite costs and lowers tail latency . These results demonstrate that LV preserves the long-term stability of the underlying OTF while introducing predictable overhead under concurrency, making it suitable for sustained phase-driven OTF workloads.

# 9 RELATED WORK

**Distributed transactions and theory.** Prior work studied distributed transactions in geodistributed systems [38], key-value stores [22, 43], or other cloud systems [13, 55, 65, 68, 99]. In this context, transactional guarantees [12, 37] and how systems can offer them have been key research questions. LV takes inspiration from these techniques seen in these systems, such as multi-level locks [15, 33], leases [32], synchronization [19, 52], dependency graphs [1], SAGAs [27], MVCC [14, 26, 62], and OCC [47]. LV adapts these techniques to work without global coordination in an OTF-native setup and demonstrates its effectiveness via TLA+ [63].

**Data Lakes, OTFs, and Lakehouses.** With the rise of data lakes, research includes studies on improving efficiency through optimized file formats, columnar encodings, and elastic cloud execution [2, 24, 29, 36, 46, 48, 49, 51, 53, 60, 66, 71–73, 96, 97, 100]. Previous research has focused on the operational efficiency of data lakes, such as improving latency when accessing files by introducing strategies to encode relational data differently or to improve file encodings [2, 29, 48]. Others focused on efficient parquet readers [71, 72], explored elastic computing, or additional structures to reduce cost and data lake traffic [24, 46, 49, 60, 66, 73, 96, 97]. These approaches either do not target the OTF domain or focus on performance and scalability, often relying on the external design paradigms [9, 64, 98]. LV complements these efforts by strengthening transactional guarantees while remaining object-store-native. Several systems extend OTFs, which took inspiration from database storage techniques [54] , with redesigned metadata layers, specialized optimizations, or domain-specific features and services [10, 17, 23, 34, 35, 50, 56, 90]. Notable examples include Photon [11], XTable [3, 8], DeepLake [35], and Active Data Lakes [30].

**OTF Transactions.** Multi-table ACID transaction support is a heavily requested feature for OTFs [9, 41, 44, 57, 69, 85, 86]. Project Nessie, Apache Hudi's Multi-Table Transaction Manager, and Delta Lake 4.0 move commit coordination to external services [39, 41, 67], while DuckLake relocates metadata to DuckDB [69]. In contrast, as seen in previous sections, LV provides serializable multi-table transactions without external dependencies, preserving interoperability and compatibility with existing OTF deployments.

# 10 CONCLUSION

This paper presents native mechanisms to provide full ACID transactions for Open Table Formats without relying on external coordination. By adapting database techniques to OTFs' object-store design, we enable serializability while preserving interoperability. Our prototype, LakeVilla, implements all techniques, requires minimal integration effort into existing systems, and gives users full control over their data. LakeVilla strengthens the OTF ecosystem by providing interoperable, full OTF HTAP and offering insights for future OTF designs and integrations. Finer-grained conflict granularity, advanced interoperability, and self-healing are exciting directions for future work on LV and OTF transactions.


# ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers, the TUM Database Chair, all colleagues in the HANA Cloud team, and the SAP&TUM Industry-University-Collaboration for their feedback and support.

# REFERENCES


[1] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, David B. Lomet and Gerhard Weikum (Eds.). IEEE Computer Society, 67–78. https://doi.org/10.1109/ICDE.2000.839388

[2] Azim Afroozeh and Peter A. Boncz. 2023. The FastLanes Compression Layout: Decoding >100 Billion Integers per Second with Scalar Code. *Proc. VLDB Endow.* 16, 9 (2023), 2132–2144. https://doi.org/10.14778/3598581.3598587

[3] Ashvin Agrawal, Tim Brown, Anoop Johnson, Jesús Camacho-Rodríguez, Kyle Weller, Carlo Curino, and Raghu Ramakrishnan. 2024. XTable in Action: Seamless Interoperability in Data Lakes. *CoRR* abs/2401.09621 (2024). https://doi.org/10.48550/ARXIV.2401.09621 arXiv:2401.09621

[4] Amazon Web Services. 2024. Amazon S3 Strong Consistency. https://aws.amazon.com/s3/consistency/. Accessed: 2024-07-24.

[5] Amazon Web Services. 2024. Cloud Object Storage - Amazon S3 - AWS. https://aws.amazon.com/s3/. Accessed: 2024-04-30.

[6] Amazon Web Services, Inc. or its affiliates. 2014. Amazon S3 now supports conditional writes. https://aws.amazon.com/about-aws/whats-new/2024/08/amazon-s3-conditional-writes/. Accessed: 2024-12-10.

[7] Apache Parquet. 2024. Parquet. https://parquet.apache.org/. Accessed: 2024-04-30.

[8] Apache XTable. 2024. Apache XTable (Incubating). https://xtable.apache.org/. Accessed: 2024-05-20.

[9] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424. https://doi.org/10.14778/3415478.3415560

[10] Edmon Begoli, Ian Goethert, and Kathryn Knight. 2021. A Lakehouse Architecture for the Management and Analysis of Heterogeneous Data for Biomedical Research and Mega-biobanks. In *2021 IEEE International Conference on Big Data (Big Data), Orlando, FL, USA, December 15-18, 2021*, Yixin Chen, Heiko Ludwig, Yicheng Tu, Usama M. Fayyad, Xingquan Zhu, Xiaohua Hu, Suren Byna, Xiong Liu, Jianping Zhang, Shirui Pan, Vagelis Papalexakis, Jianwu Wang, Alfredo Cuzzocrea, and Carlos Ordonez (Eds.). IEEE, 4643–4651. https://doi.org/10.1109/BIGDATA52589.2021.9671534

[11] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2326–2339. https://doi.org/10.1145/3514221.3526054

[12] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, Michael J. Carey and Donovan A. Schneider (Eds.). ACM Press, 1–10. https://doi.org/10.1145/223784.223785

[13] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. 2015. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1295–1309. https://doi.org/10.1145/2723372.2737788

[14] Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control - Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (1983), 465–483. https://doi.org/10.1145/319996.319998

[15] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley. http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx

[16] Jesús Camacho-Rodríguez, Ashvin Agrawal, Anja Gruenheid, Ashit Gosalia, Cristian Petculescu, Josep Aguilar-Saborit, Avrilia Floratou, Carlo Curino, and Raghu Ramakrishnan. 2024. LST-Bench: Benchmarking Log-Structured Tables in the Cloud. *Proc. ACM Manag. Data* 2, 1 (2024). https://doi.org/10.1145/3639314

[17] Zhe Chen, Hangyu Shao, Yuping Li, Hongru Lu, and Jiahui Jin. 2022. Policy-Based Access Control System for Delta Lake. In *Tenth International Conference on Advanced Cloud and Big Data, CBD 2022, Guilin, China, November 4-5, 2022*. IEEE, 60–65. https://doi.org/10.1109/CBD58033.2022.00020

[18] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[19] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8. https://doi.org/10.1145/2491245

[20] Databricks. 2024. Delta Lake limitations on S3 | Databricks on AWS. https://docs.databricks.com/en/delta/s3-limitations.html. Accessed: 2024-06-10.

[21] Databricks. 2024. Isolation levels and write conflicts on Databricks. https://docs.databricks.com/en/optimizations/isolation-level.html. Accessed: 2024-12-04.

[22] Niv Dayan and Stratos Idreos. 2019. The Log-Structured Merge-Bush & the Wacky Continuum. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 449–466. https://doi.org/10.1145/3299869.3319903

[23] Akon Dey, Alan D. Fekete, and Uwe Röhm. 2015. Scalable distributed transactions across heterogeneous stores. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman (Eds.). IEEE Computer Society, 125–136. https://doi.org/10.1109/ICDE.2015.7113278

[24] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782. https://doi.org/10.14778/3611479.3611486

[25] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA database: data management for modern business applications. *SIGMOD Rec.* 40, 4 (2011), 45–51. https://doi.org/10.1145/2094114.2094126

[26] Alan D. Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2 (2005), 492–528. https://doi.org/10.1145/1071610.1071615

[27] Hector Garcia-Molina and Kenneth Salem. 1987. Sagas. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*, Umeshwar Dayal and Irving L. Traiger (Eds.). ACM Press, 249–259. https://doi.org/10.1145/38713.38742

[28] Gartner Inc. 2023. Gartner Says Cloud Will Become a Business Necessity by 2028. https://www.gartner.com/en/newsroom/press-releases/2023-11-29-gartner-says-cloud-will-become-a-business-necessity-by-2028. Accessed: 2024-07-17.

[29] Bogdan Ghita, Diego G. Tomé, and Peter A. Boncz. 2020. White-box Compression: Learning and Exploiting Compact Table Representations. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p4-ghita-cidr20.pdf

[30] Pascal Ginter and Viktor Leis. 2026. Active Data Lakes: Regaining Physical Data Independence Without Losing Interoperability. *Proc. VLDB Endow. (Preprint)* (2026). https://www.cs.cit.tum.de/fileadmin/w00cfj/dis/papers/ActiveDataLakes.pdf

[31] Google Cloud. 2024. Cloud Storage | Google Cloud. https://cloud.google.com/storage. Accessed: 2024-05-05.

[32] Cary G. Gray and David R. Cheriton. 1989. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles, SOSP 1989, The Wigwam, Litchfield Park, Arizona, USA, December 3-6, 1989*, Gregory R. Andrews (Ed.). ACM, 202–210. https://doi.org/10.1145/74850.74870

[33] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. 1976. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, G. M. Nijssen (Ed.). North-Holland, 365–394.

[34] Anja Gruenheid, Jesús Camacho-Rodríguez, Carlo Curino, Raghu Ramakrishnan, Stanislav Pak, Sumedh Sakdeo, Lenisha Gandhi, Sandeep K. Singhal, Pooja Nilangekar, and Daniel J. Abadi. 2025. AutoComp: Automated Data Compaction for Log-Structured Tables in Data Lakes. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 404–417. https://doi.org/10.1145/3722212.3724430

[35] Sasun Hambardzumyan. 2023. Deep Lake: A Lakehouse for Deep Learning. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p69-buniatyan.pdf

[36] Yeye He, Jie Song, Yue Wang, Surajit Chaudhuri, Vishal Anil, Blake Lassiter, Yaron Goland, and Gaurav Malhotra. 2021. Auto-Tag: Tagging-Data-By-Example

in Data Lakes. *CoRR* abs/2112.06049 (2021). arXiv:2112.06049 https://arxiv.org/abs/2112.06049

[37] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[38] Joshua Hildred, Michael Abebe, and Khuzaima Daudjee. 2023. Caerus: Low-Latency Distributed Transactions for Geo-Replicated Systems. *Proc. VLDB Endow.* 17, 3 (2023), 469–482. https://doi.org/10.14778/3632093.3632109

[39] Hudi Developers and Sivabalan Narayanan. 2025. GitHub Issue: Enhance TransactionManager to support multi-table transaction #16198. https://github.com/apache/hudi/issues/16198. Accessed: 2026-05-13.

[40] Prakhar Jain. 2024. [Protocol Change Request] Delta Coordinated Commits #2598. https://github.com/delta-io/delta/issues/2598. Accessed: 2024-11-25.

[41] Prakhar Jain and Databricks. 2024. TOWARDS MULTI-TABLE TRANSACTIONS IN DELTA LAKE. https://www.databricks.com/dataaisummit/session/towards-multi-table-transactions-delta-lake. Accessed: 2024-07-10.

[42] Paras Jain, Peter Kraft, Conor Power, Tathagata Das, Ion Stoica, and Matei Zaharia. 2023. Analyzing and Comparing Lakehouse Storage Systems. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p92-jain.pdf

[43] Konstantinos Kanellis, Badrish Chandramouli, Ted Hart, and Shivaram Venkataraman. 2025. From FASTER to F2: Evolving Concurrent Key-Value Store Designs for Large Skewed Workloads. *Proc. VLDB Endow.* 18, 12 (2025), 4910–4923. https://doi.org/10.14778/3750601.3750615

[44] Oz Katz. 2025. Hudi vs Iceberg vs Delta Lake: Data Lake Table Formats Compared. https://lakefs.io/blog/hudi-iceberg-and-delta-lake-data-lake-table-formats-compared. Accessed: 2026-02-28.

[45] Martin Kleppmann. 2014. Isolation levels and write conflicts on Databricks. https://github.com/ept/hermitage. Accessed: 2024-12-07.

[46] Peter Kraft, Qian Li, Xinjing Zhou, Peter Bailis, Michael Stonebraker, Xiangyao Yu, and Matei Zaharia. 2023. Epoxy: ACID Transactions Across Diverse Data Stores. *Proc. VLDB Endow.* 16, 11 (2023), 2742–2754. https://doi.org/10.14778/3611479.3611484

[47] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (1981), 213–226. https://doi.org/10.1145/319566.319567

[48] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26. https://doi.org/10.1145/3589263

[49] Donghun Lee, Thomas Willhalm, Minseon Ahn, Suprasad Mutalik Desai, Daniel Booss, Navneet Singh, Daniel Ritter, Jungmin Kim, and Oliver Rebholz. 2023. Elastic Use of Far Memory for In-Memory Database Management Systems. In *Proceedings of the 19th International Workshop on Data Management on New Hardware, DaMoN 2023, Seattle, WA, USA, June 18-23, 2023*, Norman May and Nesime Tatbul (Eds.). ACM, 35–43. https://doi.org/10.1145/3592980.3595311

[50] Justin J. Levandoski, Garrett Casto, Mingge Deng, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugan, and Yuri Volobuev. 2024. BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 334–346. https://doi.org/10.1145/3626246.3653388

[51] Tianyu Li, Badrish Chandramouli, Philip A. Bernstein, and Samuel Madden. 2024. Distributed Speculative Execution for Resilient Cloud Applications. *CoRR* abs/2412.13314 (2024). https://doi.org/10.48550/ARXIV.2412.13314 arXiv:2412.13314

[52] Barbara Liskov. 1991. Practical Uses of Synchronized Clocks in Distributed Systems. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 19-21, 1991*, Luigi Logrippo (Ed.). ACM, 1–9. https://doi.org/10.1145/112600.112601

[53] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2025. Data formats in analytical DBMSs: performance trade-offs and future directions. *VLDB J.* 34, 3 (2025), 30. https://doi.org/10.1007/S00778-025-00911-1

[54] Raymond A. Lorie. 1977. Physical Integrity in a Large Segmented Database. *ACM Trans. Database Syst.* 2, 1 (1977), 91–104. https://doi.org/10.1145/320521.320540

[55] Hanzheng Lyu, Shaokang Xie, Jianyu Niu, Ivan Beschastnikh, Yinqian Zhang, Mohammad Sadoghi, and Chen Feng. 2025. Orthrus: Accelerating Multi-BFT Consensus Through Concurrent Partial Ordering of Transactions. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19-23, 2025*. IEEE, 2615–2627. https://doi.org/10.1109/ICDE65448.2025.00197

[56] Samuel Madden, Jialin Ding, Tim Kraska, Sivaprasad Sudhir, David E. Cohen, Timothy G. Mattson, and Nesime Tatbul. 2022. Self-Organizing Data Containers. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org. https://www.cidrdb.org/cidr2022/papers/p44-madden.pdf

[57] Dipankar Mazumdar. 2025. Concurrency Control in Open Data Lakehouse. https://hudi.apache.org/blog/2025/01/28/concurrency-control. Accessed: 2026-02-28.

[58] Microsoft Azure. 2024. Azure Blob Storage | Microsoft Azure. https://azure.microsoft.com/en-us/products/storage/blobs. Accessed: 2024-05-05.

[59] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162. https://doi.org/10.1145/128765.128770

[60] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 115–130. https://doi.org/10.1145/3318464.3389758

[61] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[62] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 677–689. https://doi.org/10.1145/2723372.2749436

[63] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (2015), 66–73. https://doi.org/10.1145/2699417

[64] Keonwoo Oh, Pooja Nilangekar, and Amol Deshpande. 2025. TreeCat: Standalone Catalog Engine for Large Data Systems. *Proc. VLDB Endow.* 18, 11 (2025), 4323–4336. https://doi.org/10.14778/3749646.3749696

[65] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX Association, 251–264. http://www.usenix.org/events/osdi10/tech/full_papers/Peng.pdf

[66] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 131–141. https://doi.org/10.1145/3318464.3380609

[67] Project Nessie. 2026. Project Nessie: Transactional Catalog for Data Lakes with Git-like semantics. https://projectnessie.org/. Accessed: 2026-05-31.

[68] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. 2020. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication. In *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, Angela Bonifati, Yongluan Zhou, Marcos Antonio Vaz Salles, Alexander Böhm, Dan Olteanu, George H. L. Fletcher, Arijit Khan, and Bin Yang (Eds.). OpenProceedings.org, 73–84. https://doi.org/10.5441/002/EDBT.2020.08

[69] Mark Raasveldt and Hannes Mühleisen. 2025. DuckLake: SQL as a Lakehouse Format. https://duckdb.org/2025/05/27/ducklake. Accessed: 2025-05-30.

[70] Jinglei Ren, Chris Kjellqvist, and Long Deng. 2024. GitHub - basicthinker/YCSB-C: Yahoo! Cloud Serving Benchmark in C++, a C++ version of YCSB. https://github.com/basicthinker/YCSB-C. Accessed: 2024-05-15.

[71] Alice Rey. 2024. Bridging the Gap Between Data Lakes and RDBMSs: Efficient Query Processing with Parquet. In *Proceedings of the Workshops of the EDBT/ICDT 2024 Joint Conference co-located with the EDBT/ICDT 2024 Joint Conference, Paestum, Italy, March 25, 2024 (CEUR Workshop Proceedings)*, Themis Palpanas and H. V. Jagadish (Eds.), Vol. 3651. CEUR-WS.org. https://ceur-ws.org/Vol-3651/PhDW-3.pdf

[72] Alice Rey, Michael Freitag, and Thomas Neumann. 2023. Seamless Integration of Parquet Files into Data Processing. In *Datenbanksysteme für Business, Technologie und Web (BTW 2023), 20. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 06.-10, März 2023, Dresden, Germany, Proceedings (LNI)*, Birgitta König-Ries, Stefanie Scherzinger, Wolfgang Lehner, and Gottfried Vossen (Eds.), Vol. P-331. Gesellschaft für Informatik e.V., 235–258. https://doi.org/10.18420/BTW2023-12

[73] Daniel Ritter. 2022. Cost-aware process modeling in multiclouds. *Inf. Syst.* 108 (2022), 101969. https://doi.org/10.1016/J.IS.2021.101969

[74] SAP SE. 2024. SAP S4/HANA - About. https://api.sap.com/products/SAPS4HANA/overview. Accessed: 2024-07-27.

[75] SAP SE. 2024. What is Ariba? https://www.sap.com/products/acquired-brands/what-is-ariba.html. Accessed: 2024-07-27.

[76] Matt Sarrel. 2024. Strict Consistency is a Hard Requirement for Primary Storage. https://blog.min.io/strict-consistency-hard-requirement-for-primary-storage/. Accessed: 2024-07-24.

[77] Tobias Schmidt, Andreas Kipf, Dominik Horn, Gaurav Saxena, and Tim Kraska. 2024. Predicate Caching: Query-Driven Secondary Indexing for Cloud Data Warehouses. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 347–359. https://doi.org/10.1145/3626246.3653395

[78] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[79] Riza Suminto. 2023. 12 Times Faster Query Planning With Iceberg Manifest Caching in Impala. https://blog.cloudera.com/12-times-faster-query-planning-with-iceberg-manifest-caching-in-impala/. Accessed: 2024-04-23.

[80] The Apache Software Foundation. 2021. Concurrency Control | Apache Hudi. https://hudi.apache.org/docs/concurrency_control. Accessed: 2024-04-30.

[81] The Apache Software Foundation. 2021. Hello from Apache Hudi | Apache Hudi. https://hudi.apache.org/. Accessed: 2024-04-30.

[82] The Apache Software Foundation. 2024. Apache Arrow | Apache Arrow. https://arrow.apache.org/. Accessed: 2024-05-15.

[83] The Apache Software Foundation. 2024. Apache Iceberg - Apache Iceberg. https://iceberg.apache.org/. Accessed: 2024-04-23.

[84] The Apache Software Foundation. 2024. MinIO. https://min.io/. Accessed: 2024-01-10.

[85] The Apache Software Foundation. 2024. Multi-Table Transaction support . GitHub. https://github.com/apache/iceberg/projects/30. Accessed: 2024-06-25.

[86] The Apache Software Foundation. 2024. Nessie - Apache Iceberg. https://iceberg.apache.org/docs/nightly/nessie/?h=transaction. Accessed: 2024-06-25.

[87] The Apache Software Foundation. 2024. Spec - Apache Iceberg. https://iceberg.apache.org/spec/. Accessed: 2024-05-06.

[88] The Linux Foundation Projects. 2024. Home | Delta Lake. https://delta.io/. Accessed: 2024-04-23.

[89] The Linux Foundation Projects. 2024. Introduction - Delta Lake Documentation. https://docs.delta.io/latest/delta-intro.html. Accessed: 2024-05-05.

[90] Yuanyuan Tian, Jesús Camacho-Rodríguez, Carlo Curino, Cesar A Galindo-Legaria, Ashit Gosalia, Brian Kroth, Sergiy Matusevych, Nicolas Bruno, Ashvin Agrawal, Stefan Grafberger, Beysim Sezgin, Milan Potocnik, Mahesh Behera, Milind Joshi, and Xiaoyu Li. 2025. Towards Query Optimizer as a Service (QOaaS) in a Unified LakeHouse Platform: Can One QO Rule Them All?. In *15th Conference on Innovative Data Systems Research, CIDR 2025, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org. https://vldb.org/cidrdb/2025/towards-query-optimizer-as-a-service-qoaas-in-a-unified-lakehouse-ecosystem-can-one-qo-rule.html

[91] TPC. 2025. TPC-C Homepage. https://www.tpc.org/tpcc/. Accessed: 2025-06-11.

[92] TPC. 2026. TPC-DS Homepage. https://www.tpc.org/tpcds/. Accessed: 2026-02-20.

[93] Trino Software Foundation. 2024. Trino | Distributed SQL query engine for big data. https://trino.io/. Accessed: 2024-04-30.

[94] Thomas F McGeehan V. 2025. GitHub: DuckTales: DuckLake Demo Suite. https://github.com/TFMV/DuckTales/tree/main. Accessed: 2026-02-05.

[95] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (2023), 1413–1425. https://doi.org/10.14778/3583140.3583156

[96] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[97] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. On-Demand State Separation for Cloud Data Warehousing. *Proc. VLDB Endow.* 15, 11 (2022), 2966–2979. https://doi.org/10.14778/3551793.3551845

[98] Matei Zaharia, Ali Ghodsi, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf

[99] Jianqiu Zhang, Kaisong Huang, Tianzheng Wang, and King Lv. 2023. Efficiently Making Cross-Engine Transactions Consistent. *SIGMOD Rec.* 52, 1 (2023), 27–34. https://doi.org/10.1145/3604437.3604444

[100] Jiatang Zhou, Kaisong Huang, Zhuoyue Zhao, Dong Xie, and Tianzheng Wang. 2025. Analytics Are Heavy. The DBMS Is Busy. When Will My Mission-Critical Transaction Start Running? *Proc. VLDB Endow.* 18, 12 (2025), 5299–5302. https://doi.org/10.14778/3750601.3750656