NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

**FELIPE POUBEL ROSSI DO CARMO**

BSc in Computer Science and Engineering

# CAUSALLY CONSISTENT GEO-REPLICATION FOR LAKEHOUSE SYSTEMS

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING
SPECIALIZATION IN DISTRIBUTED AND PARALLEL SYSTEMS

NOVA University Lisbon
January, 2026

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

# CAUSALLY CONSISTENT GEO-REPLICATION FOR LAKEHOUSE SYSTEMS

**FELIPE POUBEL ROSSI DO CARMO**

BSc in Computer Science and Engineering

**Supervisor**: Nuno Manuel Ribeiro Preguiça
*Full Professor, NOVA University Lisbon*

**Co-supervisor**: João Carlos Antunes Leitão
*Associate Professor, NOVA University Lisbon*

**Examination Committee**

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING
SPECIALIZATION IN DISTRIBUTED AND PARALLEL SYSTEMS

NOVA University Lisbon
January, 2026

# Abstract

The lakehouse is an increasingly popular architecture for maintaining large amounts of data in cheap and scalable cloud storage while providing the ability to run transactions and efficient analytical queries over open file formats. This allows taking advantage of a variety of tools for distributed computing, data analytics, and Machine Learning (ML). However, no existing lakehouse system supports efficient geo-replication, which is found on many cloud-based systems to provide fault tolerance, high availability, and low latency across different regions of the globe. These benefits can be further enhanced by replicating data in a weakly consistent manner, at the cost of allowing replicas to temporarily diverge in state and other concurrency anomalies.

To provide weakly consistent replication for lakehouse systems, we first need to improve their conflict handling methods for higher concurrency and allow diverging histories to converge. We also need to define how the managed metadata might differ across different sites, and how it can be reconciled while keeping it consistent with the data itself. This must be done without sacrificing lakehouses' transactional semantics.

In this document, we argue that lakehouses would benefit from supporting weakly consistent geo-replication, even if it means relaxing their inherent Snapshot Isolation (SI) guarantees. We discuss existing ideas that have been proposed for other types of data stores, and we derive a novel solution adapted for this domain. Our proposed solution will offer Transactional Causal Consistency: a weak consistency model that provides high availability while preserving *happens-before* relations between non-concurrent operations as well as sufficiently strong isolation semantics for many workloads.

**Keywords:** Geo-replication · weak consistency · lakehouse · cloud storage

# Resumo

A *lakehouse* é uma arquitetura emergente para gerir grandes quantidades de dados em armazenamento *cloud* barato e escalável, e permitir que esses dados sejam analisados com acesso direto de forma eficiente. Isto permite aproveitar uma variedade de ferramentas de computação distribuída, análise de dados, e aprendizagem automática. Contudo, não existem sistemas *lakehouse* que tenham suporte para geo-replicação eficiente, que é comum em muitos sistemas *cloud* para providenciar tolerância a falhas, alta disponibilidade, e baixa latência em diferentes partes do mundo. Estes benefícios podem ser melhorados ao replicar os dados de forma fracamente consistente, com o sacrifício de permitir que réplicas divirjam em seus estados, assim como outras anomalias de concorrência.

Para providenciar replicação fracamente consistente em *lakehouses*, primeiro precisamos melhorar seus métodos de resolução de conflitos de forma a permitir maior concorrência e que histórias diferentes convirjam. Também precisamos definir como os metadados geridos possam diferir em réplicas distintas. Estes precisam ser reconciliados de formas que preservem sua consistência em relação aos dados. Isto deve ser feito sem sacrificar as semânticas transacionais das *lakehouses*.

Neste documento, argumentamos a favor de providenciar geo-replicação fracamente consistente para *lakehouses*, mesmo que implique relaxar suas garantias inerentes de isolamento de *snapshots*. Nós discutimos ideias existentes que foram já propostas para outros tipos de sistemas, e derivamos uma nova solução adaptada para este domínio. A nossa solução proposta vai impor consistência causal transacional: um modelo de consistência fraca que providencia disponibilidade alta e preserva relações de precedência entre operações não concorrentes, assim como semânticas de isolamento suficientemente fortes para muitas aplicações.

**Palavras-chave:** Geo-replicação · consistência fraca · *lakehouse* · armazenamento em *cloud*

# Disclosure of Delegation to Generative Artificial Intelligence

I declare that Generative Artificial Intelligence (GenAI) tools were used in the preparation of this work for proofreading, to detect and correct grammatical and structural mistakes. No such tools were used for research.

# Contents

# List of Figures

# Acronyms

**ACID** Atomicity, Consistency, Isolation, and Durability *(pp. 5–7, 14, 16)*

**BI** Business Intelligence *(pp. 1, 2, 15)*

**CoW** Copy-on-Write *(pp. 17, 26)*
**CRDT** Conflict-free Replicated Data Type *(pp. 10, 13)*

**DBMS** Database Management System *(pp. 14, 15, 17, 19, 26, 28)*
**DDL** Data Definition Language *(pp. 27, 31, 32)*

**ETL** Extract, Transform, and Load *(pp. 1, 15)*

**LSM tree** Log Structured Merge tree *(p. 22)*

**ML** Machine Learning *(pp. ii, 1, 15, 16)*
**MoR** Merge-on-Read *(pp. 17, 26)*
**MVCC** multi-versioned concurrency control *(p. 17)*
**MVR** Multi-Valued Register *(p. 10)*

**OCC** Optimistic Concurrency Control *(pp. 6, 13, 17, 19)*
**OLAP** Online Analytical Processing *(pp. 1, 15)*
**OLTP** Online Transaction Processing *(p. 15)*
**OTF** Open Table Format *(pp. 16–23)*

**PSI** Parallel Snapshot Isolation *(pp. 7, 10, 14)*

**RDBMS** Relational Database Management System *(pp. 18, 19, 23)*

**SI** Snapshot Isolation *(pp. ii, 6, 7, 17–19, 31)*

# 1 Introduction

## 1.1 Context

Many companies maintain data pipelines for ingesting data from several operational sources for unified analytics, such as sales or inventory data. Traditionally, such analytics were performed in an SQL Online Analytical Processing (OLAP) database called a *data warehouse*. Warehouses are optimized for doing efficient aggregations over large amounts of relational, historical data, with the purpose of being used for data analytics and Business Intelligence (BI).

As the size of data grew, companies found a need to store ingested data in cheap storage that wasn't coupled with the computing resources as required by most warehouses. With this, they started employing data lakes, which are based on network accessible object storage systems [52], where data is stored in generic open file formats [**ApacheOrc**, 58]. Data lakes reduce data ingestion costs by using cheaper, scalable storage and only transforming and loading to the warehouse a subset of the data after it was curated. The open files also allowed data to be accessed by other analytics engines such as Spark [68], which opened the systems for a wider range of use cases, such as Machine Learning (ML).

This two-tiered data lake and warehouse architecture was the source of some challenges however mainly due to its complexity. Data needed to be Extracted, Transformed, and Loaded (ETLed) into the data lake and then to the warehouse. This introduced more points of failure and duplicated data, as well as inconsistencies or staleness between the data within each location. These systems also didn't reliably track information about the data lake files as would be done over warehouse data. This led to poor data management and inefficiencies in advanced analytics jobs run directly over the data lake.

These limitations gave rise to the lakehouse architecture [14]. It provides warehouse data management and transactional guarantees directly over the data lake. Lakehouse systems do so by maintaining a metadata layer over the data lake, often stored in the data lake itself, which tracks information about schema and which files are part of a table at each snapshot. Although not free from drawbacks, this approach has several benefits: $(i)$ it further decouples compute from storage, allowing easy scalability regarding the amount of data in the system; $(ii)$ it reduces the data staleness and duplication introduced

by the previous two-tiered architecture; (*iii*) data can be efficiently queried by popular tools for different use-cases, both for SQL and non-SQL workloads, such as Spark [68] for distributed computing, or ML and data science tools with DataFrames APIs.

Lakehouses have been shown to be successful by their deployments for maintaining and processing very large amounts of data [13, 23]. They also have frequent developments among the several systems that have been created to implement their architecture [13, 65, 9, 51].

A functionality currently not provided by any lakehouse system is geo-replication. It's provided by many databases to deploy fault-tolerant, highly scalable, and highly available applications with low latency across different parts of the globe. Many systems [24, 61, 69] provide strong consistency for replicated data, making it so the different replicas behave as if they were one and avoiding concurrency anomalies. But in doing so, they must sacrifice availability [35] by requiring coordination across replicas for almost all [61], if not all, transactions. This also compromises latency as the replicas are, by definition, physically distant from each other.

To avoid the costs of strong consistency, many systems [4, 30, 31, 49] settle for providing weaker consistency guarantees such as eventual consistency [66], causal consistency [3], or one of its variants [49, 4, 64, 16]. Causal consistency requires operations to be replicated only after any other operations whose effects were locally visible at the time it was originally issued. This ensures replicated operations exposed to users respect their *happens-before* relations [46], while still allowing them to be replicated asynchronously.

## 1.2 Motivation

While lakehouses' purpose of providing analytics and BI for large evolving datasets should mean they're deployed for companies with global operations [23], work done on lakehouse replication has been very limited, only encompassing simple primary/backup replication [26, 62]. This means current lakehouse replication only provides read-only access at different sites which act as backups, and writes have to be directed to the primary replica. This introduces high write latency for clients in other regions, and means no availability can be guaranteed for writes under a failure of the primary region or network partitions.

Although benefits would be gained from providing synchronous multi-master replication for writes in any region, we argue weaker consistency such as causal consistency is more appropriate. It would allow for highly available geo-replication with low write latencies and higher global concurrency if we also define more sophisticated conflict resolution.

Providing weakly consistent geo-replication where writes can be done in every replica isn't as trivial as simply asynchronously replicating the object store's data and metadata files. Some challenges with this approach would be:

- The lakehouse metadata layer expects a single total ordering of transactions which is

maintained in its metadata files. This is enforced by having clients acquire locks or perform globally atomic operations, requiring coordination. Otherwise, different clients could simultaneously create the same[1] snapshot. This would either generate conflicting metadata files or fork the lakehouse history, making the other invisible.

- Unless the object store replication is itself causally consistent, replicated metadata and data files can refer to other files that were not yet replicated locally. This would make the system's latest snapshots partially unreadable until the missing files were available.

- Lakehouses often make use of catalogs, which would themselves need to be kept consistent with their local object store even in the presence of remotely received data.

With this it becomes apparent that, in order to provide weakly consistent replication, there's a need to treat replication of metadata files and data files differently while keeping them consistent. It's also needed to have convergent conflict resolution policies, as opposed to lakehouses' current behaviour of aborting concurrent conflicting transactions.

## 1.3 Expected contributions

We aim to overcome the challenges presented above in order to provide both eventually and causally consistent geo-replication for lakehouse systems with minimal overhead. Our contributions will be implemented as an extension to an existing system, but ideally be easily adaptable for others. We will also try modifying the lakehouse format specifications and clients as little as possible, providing a solution that is compatible with the current lakehouse ecosystem.

More concretely, we will, in the context of lakehouse systems:

- design a model for handling concurrent conflicting updates so they can converge without coordination;

- design a service to replicate data while tracking causality metadata and enforcing causal ordering;

- implement our solution over an existing lakehouse system;

- validate the functionality of our solution;

- evaluate performance of our implementation compared to a plain lakehouse and other replicated systems. We will measure throughput, latency, and how it scales with additional replicas.

[1] "Same" as having the same sequence number or predecessors, not as representing the same database state.

## 1.4 Document organization

The remainder of this document is organized as follows: in Chapter 2 we study the state-of-the-art regarding both lakehouses and geo-replication along with the related topics of transactions, consistency, and partial replication, explaining any concepts needed to understand them; and in Chapter 3 we describe the work we intend to carry, including how we will conduct evaluation and the work plan.

# 2

# Related work

In this chapter, we discuss the state-of-the-art of the research areas related to the work to be conducted, and introduce fundamental concepts needed to understand the existing problems, our goals, proposed work, and challenges we expect to find.

## 2.1 Transactional semantics

A fundamental aspect of database systems is allowing users to perform transactions over the data they manage. A transaction is composed of a sequence of read or write operations and guarantees that: $(i)$ its results and side effects are semantically consistent with the operations being executed; $(ii)$ the database invariants aren't violated; and $(iii)$ the database can recover from a failure to a consistent state.

These safety guarantees have been formalized as requiring the system to provide some level of Atomicity, Consistency, Isolation, and Durability (ACID) [39] properties regarding transactions. We now explain the meaning of each property, with some being necessary to implement others.

- *Atomicity.* A transaction must be *atomic* in the sense that it either successfully executes in its entirety, or it fails and the database is left at the exact same state as if it had never even started, without the side effects of its intermediate operations.
- *Consistency.* A transaction can only commit results that preserve the database invariants, i.e., its correctness constraints, such as uniqueness of keys and referential integrity.
- *Isolation.* A transaction must not be able to read the effects of other, concurrent transactions, giving the illusion that they had been fully executed one at a time. This is often weakened by some systems to achieve higher concurrency. We explain different isolation levels in Section 2.1.1.
- *Durability.* Once a transaction has been committed, the system must guarantee its results will persist and be recovered even after system failures, so a user can be sure that whatever the system says has happened will always have happened.

### 2.1.1 Isolation levels

While implementing ACID properties gives the system strong safety guarantees, they can be too restrictive and lead to low transaction throughput, so it might be desirable to weaken some of those properties in favor of performance, while allowing some anomalies that may arise. Of particular interest to us, is the different levels of isolation implemented by many systems.

Considering two concurrent transactions $t_1$ and $t_2$, some anomalies that may arise from weakening isolation are:

- *Dirty reads.* $t_1$ sees the effects of the not yet committed $t_2$ which could still fail, and thus violate atomicity.
- *Non-repeatable reads.* $t_1$ reads some data item $x$, $t_2$ writes to $x$ and commits. When $t_1$ reads $x$ again, it sees a different value than returned in the first read.
- *Phantom reads.* $t_1$ reads set of data items $D$ returned from a query $q$, then $t_2$ creates a data item $x$ which satisfies $q$. When $t_1$ executes $q$ again it receives $D \cup \{x\}$, a different set of data items than the one returned in the first query.
- *Read skew.* $t_1$ reads a data item $x$, $t_2$ writes to data items $x$ and $y$, then $t_1$ reads only the new value of $y$, which might be inconsistent when paired with the old value of $x$.
- *Write skew (or short fork).* Two transactions both write to different data items based on the previously read version of some data item later written by the other: $t_1$ reads $v_{x1}$ from $x$ and $t_2$ reads $v_{y1}$ from $y$. Then based on the value $v_{x1}$ read, $t_1$ writes $v_{y2}$ to $y$. Based on the value $v_{y1}$ read, $t_2$ writes $v_{x2}$ to $x$.
- *Long fork.* After a short or long fork happens among transactions $t_1$ and $t_2$, transactions $t_3$ and $t_4$ execute concurrently with one seeing the effects of $t_1$ but not $t_2$, and vice-versa.

Most notably, *read skew* and *write skew* can result in transactions having a view of the database inconsistent with its constraints, permitting consistency violations. [19]

*Serializability* is the strongest isolation level defined by the ISO/IEC SQL standard [41], only allowing executions where the results of all operations of any successful transactions would be the same if they were executed in some total, one at a time, order. [19] Serializable executions don't suffer from any of the anomalies described above.

*Snapshot Isolation (SI)* weakens serializability by allowing for the *write skew* anomaly. It requires that all reads from a transaction return the last committed value of data items before a starting timestamp, having a global view of the database state at that point. Two concurrent transactions are considered to conflict only if they write to some of the same data items, usually with the second committer or writer being rolled back. It's generally implemented with multi-version concurrency control and Optimistic Concurrency Control (OCC), making it highly desirable for low-write scenarios, as read transactions will never

block or need to be rolled back, while write transactions can become more likely to conflict the longer they run for [19]. SI is the default isolation level implemented by lakehouses.

*Parallel Snapshot Isolation (PSI)* [64] weakens SI for better performance in a replicated context. It allows snapshots to be created in parallel in different nodes to minimize coordination and maximize concurrency. Unlike SI, PSI's snapshots don't preserve total ordering. This introduces the *long fork anomaly*. Like with SI, however, if two concurrent transactions have conflicting writes, one of them has to be aborted. Therefore, coordination among replicas is not completely avoidable.

## 2.2 Replication and consistency

Distributed systems often rely on replication of data across a number of servers, designated as *replicas*, to provide a combination of: *fault tolerance and availability*, so data and dependent services can still be accessed even if some replica fails or becomes unreachable; *load balancing*, so the same data can be served from different replicas in order to reduce load on a single replica, enabling horizontal scaling; and *lower latency at a global scale*, by having replicas physically positioned in different parts of the Earth, so clients access data through the geographically closest replica. This does, however, introduce the problem of data inconsistency across replicas, i.e., whenever data is written to a replica, it needs to be propagated to others so it's visible globally, but anomalies might become apparent if we allow writes to be issued and executed on any replica. To understand how we can handle these situations, we need to have concrete notions of consistency[1].

### 2.2.1 Consistency models

Ideally, writes would be propagated immediately to all other replicas so the entire system could easily behave as if there were only a single server, but in reality we have to handle reads and writes that happen concurrently at different replicas which can introduce anomalies if they depend on the same data items.

With replication, accesses to the database are only partially ordered across replicas. Given that an execution is a serialization of a partially ordered set of operations, consistency models can be defined by the set of executions they allow to happen in a system. For example, they can specify what are the allowed results (if any) when: operation $op_1$ in replica $A$ writes the value 1 to data item $x$; operation $op_2$ in replica $B$ writes the value 2 to $x$; $op_1$ is propagated to $B$; then a client reads $x$ in $B$.

A consistency model is considered to be *strictly stronger* than another if it allows only a subset of the executions of the other [16]. The choice of which model to implement in a system is mostly based on a trade-off between consistency, availability, latency, and throughput.

[1]Not to be confused with *consistency* in ACID. In that context it means preserving data correctness invariants, while here it means different nodes having consistent views of replicated state.

Consistency models are mainly divided by providing either *strong* or *weak* consistency, with the former avoiding concurrency anomalies, and only the latter being compatible with availability, as proven by the CAP (Consistency Availability Partition-tolerance) theorem [35]. Availability can be defined as a guarantee requiring that every request successfully received by a node result in a response, which, coupled with partition tolerance, means that a node shouldn't need to contact others to serve any request, which also generally increases latency and throughput. This is why, even if allowing data anomalies to happen and requiring application programmers to deal with them manually, weak consistency is often desired.

#### 2.2.1.1 Strong consistency models

Strong consistency allows the system to behave as reads and writes were being executed one at a time on a single node, avoiding any anomalies caused from inconsistent data across replicas. Strong consistency models can provide either *serializability*, or the stronger *linearizability*.

**Serializability** A serializable data object acts as if all operations done over it, even if concurrent, were executed sequentially, while respecting the sequential specification of the object, as well as the ordering of sequential operations within a single process. More formally: consider two operations $op_1$ and $op_2$, where $op_1$ precedes $op_2$ (denoted $op_1 \prec op_2$) at a process, and two operations $op_3 \prec op_4$ at another process. They are serializable if, and only if, there exists a totally ordered (serial) schedule of those operations where the sequential specification of the object is respected, $op_1 < op_2$, and $op_3 < op_4$. A data object is serializable if, and only if, it only allows serializable executions. [40]

**Linearizability** A linearizable data object is stronger than a serializable one in that it only allows a subset of the serializable executions. It only allows executions that can be serialized to a schedule that preserves real-world order of non-overlapping operations. Therefore it acts as if operations took place atomically at a single instant.

For example, consider two operations $op_1$ and $op_2$ that happen at different replicas, such that $op_1$ finishes before $op_2$ starts. Even if there's a valid serial schedule $op_2 < op_1$, it's not enough to respect linearizability. This is because the linearization point of $op_1$ can't be after it ends, and the linearization point of $op_2$ can't be before it starts. So this execution is only linearizable if there's a valid serial schedule $op_1 < op_2$, that respects real time order.

#### 2.2.1.2 Weak consistency models

Weak consistency allows for availability, but it also allows the state of different replicas to diverge. Therefore, a solution needs to be found for reconciling two divergent states, especially in the presence of conflicts. Conflicts can happen when the same or related data items are modified concurrently at different replicas.

Two popular weak consistency models are *eventual consistency* and the stronger *causal consistency*, each with several variations that provide different guarantees but preserve their main characteristic, as we'll see further in Section 2.2.3.

**Eventual consistency** An eventually consistent system provides no guarantees other than that *eventually*, if no new operations are issued, all replicas will converge to the same state, without any guarantees regarding which anomalies can happen. This means any writes are allowed to be propagated to different replicas at different orders, and a way to conciliate concurrent writes to the same data objects needs to be defined [66].

**Causal consistency** Causal consistency is stronger than eventual consistency as it introduces the notion of *causality*: Consider an operation $op_1$ issued at any replica, and an operation $op_2$ originally issued at replica $A$. $op_1$ causally precedes, or is a causal dependency of, $op_2$ (denoted $op_1 \rightsquigarrow op_2$) if $op_1$'s result was already visible in replica $A$ at the moment $op_2$ was issued. To maintain causal consistency, a node must replicate an operation only after all its casual dependencies have been replicated at that replica. Causality establishes a partial ordering over a system's operations. [46]

A rationale behind choosing causality as a data consistency criterion is the assumption that the observable state at any point is taken into account for the decision of executing a given operation, or determining its result. This means the results of any previous operations could have affected the new operation, even if transitively. Thus, such operation should only be observed at other replicas once the ones that causally precede it have already been observed. Operations that only existed at different replicas at that moment cannot have been locally observable by the new or previous operations. Such operations are considered to be causally unrelated, or *concurrent*, with respect to each other. [46]

This notion of concurrent operations allows causally consistent systems to provide availability even under network partitions, as any operation issued by a client will only be causally dependent on the operations available locally at that moment, allowing the replica to execute the operation immediately and then reply to the client, without needing to coordinate with others.

How causality metadata is tracked and propagated is one of the main implementation decisions causally consistent systems must make. It's often a trade-off between: remote update visibility latency, due to false dependencies introduced by compressed metadata; and transaction throughput, due to bandwidth costs of replicating uncompressed metadata [20].

Some proposed techniques to ensure causal delivery are: using vector clocks [46]; using single scalar clocks [31]; explicitly maintaining a set of dependencies [49]; or propagating messages through a tree dissemination topology [21].

A few variations of causal consistency have been proposed that aim to provide stronger

guarantees without sacrificing availability, e.g.: *Causal+ Consistency*, which requires convergent conflict resolution, i.e., that operations are deterministic, commutative, and associative [49]; *Transactional Causal Consistency*, which applies Causal+ Consistency to PSI to allow fully asynchronous read-write transactions; or *Observable Causal Consistency*, that exposes concurrency between operations, allowing each replica to decide on a result based on several causally unrelated values [16]. All of these have in common that they avoid synchronization to resolve conflicts.

#### 2.2.1.3 Concurrent objects

To ensure state convergence in the presence of concurrent writes, some systems make use of objects that consider concurrency in their specification. Two popular approaches have been proposed for this: using Multi-Valued Registers (MVRs) [16], which expose concurrency to clients by returning the set of all values that were written concurrently, allowing the application to decide how to conciliate the conflicts; or using Conflict-free Replicated Data Types (CRDTs) [63], objects that are guaranteed to reach the same state after receiving all updates, being replicated either by propagating and merging their states with a commutative and associative function, or by propagating operations done over them in a causally consistent order them while requiring that any concurrent operations commute.

A simple way to ensure convergence is by employing the *last-writer-wins* rule. With it, the value with the lower timestamp is considered to have happened earlier and is overwritten the conflicting value with the higher timestamp. As timestamps are propagated alongside the values, every replica will agree on which of two operations should be considered as having happened later. [49, 31] Effectively this this makes resolving concurrent writes deterministic, and hence, convergent.

### 2.2.2 Replication strategies

There are several ways to implement replication, each with different characteristics and which influence what consistency model would ideally be chosen, or vice-versa. We now describe popular replication strategies, which can be combined when designing a system.

#### 2.2.2.1 When writes are propagated

**Eager replication** Writes are propagated synchronously as part of a transaction, and it fails if other replicas fail to confirm it. This always leads to global serialization orders. Eager replication can be implemented by using distributed commit protocols [53], state machine replication protocols with consensus [45], or a centralized coordinator [12]. [38]

**Lazy replication** Writes are propagated asynchronously, and confirmed immediately in the local replica.

#### 2.2.2.2 Where writes can be made

**Primary-copy** In these systems, all writes must be done through a single, primary node, which then has its state replicated to secondary nodes, which can then serve reads, or act as backups of the primary. This makes it trivial to implement strong consistency, as the primary node orders all writes, but concentrates all write load on a single node. To achieve some fault tolerance, secondary nodes may elect a new primary amongst themselves when they suspect the primary has failed.

It's common for systems that use this model to allow writes in multiple replicas by assigning individual data items or partitions to different replicas, as opposed to a single primary for the entire database. In such systems, transactions directed to a replica that is primary to all data items it reads or writes don't need eager replication to provide strong consistency. This approach greatly benefits from high region locality in data access patterns. Coordination or migration of data items is still needed for transactions that access data from different primaries. [38]

**Multi-master** In such systems, every replica can process any operation. This requires eager replication if strong consistency is desired. Weak consistency makes this model compatible with lazy replication.

#### 2.2.2.3 Where writes are replicated

**Full replication** All nodes fully replicate all data items of the database. This allows any replica to respond to any reads.

**Partial replication** To reduce bandwidth and data duplication, some systems have each replica only maintain part of the data. This can also benefit causal replication, as a replica doesn't need to wait for all causal dependencies of an operation to be available locally to replicate it, only requiring those that reference locally replicated data [21]. Like with partitioned primary-copy systems, a difficulty partially replicated systems must overcome is how to handle access to remote data.

### 2.2.3 Existing systems

We now describe a few systems implementing decentralized geo-replication in different contexts, which solve similar problems to ours, and we believe might be relevant to not only understand how we might find a solution, but to understand the problem's nuances.

It's worth noting that it has been shown that linearizability across several co-located nodes can be implemented in a scalable way [5]. Thus, most of the systems described here assume each replica is composed of several nodes which form a linearizable data store.

#### 2.2.3.1 COPS

COPS (Clusters of Order-Preserving Servers) [49] aims to provide, over locally partitioned a key-value store, what they defined as "causal consistency with convergent conflict handling (Causal+ Consistency)" in a scalable manner for geo-replication. *Convergent conflict handling* requires conflicting writes to be handled in the same way in all replicas, guaranteeing that all replicas converge to the same state after receiving the same set of operations. They enforce this by requiring that conflict handling function to be commutative and associative. A simple way to achieve this is by employing the *last-write-wins* rule, in which each operation is completely overwritten if another conflicting one is considered to have happened later, not necessarily respecting real time order.

COPS achieves its goal by having explicit dependency checks for each operation's *nearest dependencies*. This means that before replicating an operation, a node first checks if the operations that only immediately causally precede it have already been replicated. This also guarantees any dependencies that could be generated from transitivity must be replicated before it. As an example, consider the operations $op_1, op_2, op_3$ s.t. $op_1 \rightsquigarrow op_2 \wedge op_1 \rightsquigarrow op_3 \wedge op_2 \rightsquigarrow op_3$. The replication message for $op_3$ sent by the original replica only needs to state that $op_3$ depends on $op_2$, as any replica that has replicated $op_2$ must also have replicated $op_1$. This allows less causality metadata to be transmitted when compared to, e.g., vector clocks.

COPS also has a variant supporting "*get* transactions", which guarantee a causally consistent snapshot of the data store for reading several data items, and requires all direct dependencies of each operation, not just the nearest, to be tracked, so *get* operations for different items on the same transaction can detect an item another depends on has changed since it being first read. This addition doesn't seem to introduce much latency nor throughput overhead.

#### 2.2.3.2 Cure

Cure [4] aims to provide, over a locally partitioned key-value store, what it named *Transactional Causal Consistency*, introduced in Section 2.2.1.2. It does so without compromising availability nor local-area scalability. It minimizes the coordination needed among partitions within a data-center and allowing partitions to take decisions locally. More specifically by: $(i)$ using vector clocks to encode dependencies thus avoiding explicit dependency check messages; $(ii)$ not relying on a central timestamping node; and $(iii)$ having an asynchronous protocol involving partitions to establish the transaction with satisfied dependencies, thus decoupling the propagation of updates to replicas from making the updates visible.

Each partition periodically synchronizes with corresponding partitions in other data-centers, sending a heartbeat if there was no update since the last so the globally stable snapshot of the other can still advance. This approach to dependency verification has the advantage of significantly improving throughput by reducing frequency of data exchange,

but limits remote update visibility latency, as an update is only visible at a replica after all partitions of both data-centers have performed their periodic synchronization. This also prevents replicated updates from being visible even if all data items read by a transaction have already received its causal dependency updates, but some other, in a different partition, hasn't.

Although we won't have to consider intra-data-center partitions in our context, Cure's ideas of asynchronously making updates visible with vector clocks, or allowing convergent forks with CRDTs can still be useful for providing good overall system throughput.

#### 2.2.3.3 TAPIR

As opposed to the previously described systems, TAPIR (Transactional Application Protocol for Inconsistent Replication) [69] aims to provide replication of read-write transactions with strong consistency, while generally preserving low latency and throughput.

TAPIR shows it's possible to achieve linearizable distributed transactions over weakly consistent replication. Instead of performing state machine replication by replicating a globally ordered log, TAPIR's *Inconsistent Replication* protocol replicates an unordered operation set, allowing operations to be replicated in different orders in each replica. To support strong consistency, it allows applications to also issue consensus operations. This approach allows only a few necessary operations to be replicated in a strongly consistent manner, instead of all.

#### 2.2.3.4 Spanner

Spanner [24] is a strongly consistent, synchronously replicated database that heavily relies on data partitioning and locality, providing lock-free reads, atomic schema changes, while avoiding long lived, minutes-long, transactions from likely being aborted as happens with OCC approaches. It reduces the scope of state machine replication by partitioning the data across different Paxos groups. It uses physical timestamps which directly expose clock uncertainty. They use this synchronicity and exposed uncertainty to know when it is safe to Paxos leadership changes and global lock-free snapshot reads without violating consistency.

#### 2.2.3.5 SLOG

SLOG (Serializable, Low-latency, Geo-replicated Transactions) [61] also aims to provide strong consistency but, for workloads with high physical locality on data access, it claims to avoid the latency/throughput trade-off this implies, i.e., workloads where most transactions initiate physically close to the primary region for the accessed data.

SLOG's takes advantage of the fact that if two transactions only access distinct data items, they can be safely commuted without affecting serializability. Thus it maintains, at each replica, a *local log* for linearized local operations which only access locally mastered

data, and a *global log* for operations asynchronously received from remote replicas, which can be safely interleaved among operations from distinct sources.

To achieve linearizability for transactions that access items mastered by multiple replicas, coordination is still needed, and it's achieved by using what they call "lock only transactions" which are linearized at each relevant replica's local log, locking the accessed data items at their master replicas until the transaction finishes.

#### 2.2.3.6 Saturn

Saturn [21] is a causality metadata service aimed to be used on top of data stores for providing different serialization orders at different sites in a way that minimizes visibility latency and respects causality, while supporting *genuine* partial replication, i.e., sites that don't replicate some data, don't need to process nor receive its metadata. It does so by using an auxiliary broadcast tree network to propagate metadata, while the actual data is shared directly in bulk between replicas. Broadcasts trees naturally preserve causality, and Saturn improves this by introducing intentional latency between broadcast nodes so replication metadata ideally reaches other nodes at the same time as the asynchronous bulk data transfer, providing a serialization order which minimizes the amount of time replicas have to wait before making an update visible. It also avoids delivering metadata to sites that don't replicate the data it references.

Although Saturn doesn't support transactions nor data migrations, its goals of providing a causality metadata service on top of a data store with asynchronous propagation to provide different serialization orders at each replica is similar to the assumptions we make for replication in lakehouse systems.

#### 2.2.3.7 Antidote SQL

Antidote SQL [50] defines, for a replicated database on top of Cure [4], sophisticated convergent conflict resolution semantics at row, column, and table levels to support PSI whenever possible while preserving integrity invariants such as primary keys, referential integrity, and numerical bounds. It does so by using CRDTs [63] and escrow techniques [56]. It allows the concurrency policy to be defined for each column in the database schema, also allowing for strong consistency when necessary, which uses shared/exclusive multi-level locks [17] to efficiently allow switching between it and PSI policies.

Although lakehouses don't enforce uniqueness or referential integrity even without replication, Antidote SQL's idea of column level conflict resolution policies can be useful to us to minimize data being overwritten and lost in the presence of conflicts.

## 2.3 Lakehouse systems

Lakehouses are metadata management systems aiming to provide ACID [39] transaction capabilities, typical of Database Management Systems (DBMSs), on top of low-cost cloud

object stores, which are often only eventually consistent and provide fewer guarantees than a file system [52]. They are focused on doing so for analytical payloads over large amounts of data, and have real world deployments that process petabytes of data [13].

Another focus of lakehouses is to achieve its goals while storing data only on open file formats, such as Parquet [58], to allow data to be accessed and queried directly by any program, preventing data access from being locked in to a specific vendor, as is common with DBMSs that use internal, proprietary, data representations. This makes it possible to easily query data stored in a lakehouse directly by any standard data analytics and Machine Learning (ML) tools.

The formats chosen are usually organized in a columnar format typical of Online Analytical Processing (OLAP) DBMSs, to make aggregations over large volumes of data more efficient, while sacrificing performance on Online Transaction Processing (OLTP) workloads, which tend to focus on smaller grained queries and more frequent updates.

### 2.3.1 Lakehouses as successors of data warehouses

Lakehouses are being seen as an evolution over the "data lake + warehouse" architecture, where data from several operational sources would be loaded directly into a data lake, which would hold uncurated data to then be Extracted, Transformed, and Loaded (ETLed) into a warehouse. A warehouse is an OLAP database meant to conduct large analytics over curated historical data, such as Business Intelligence (BI) and data science analysis. A data lake is nothing more than a cheap object storage to serve as a centralized hub where any data worth analyzing is stored in an open file format without much treatment [14], both for data meant to be ETLed into the warehouse and being stored in the data lake again, or processed by other big data jobs such as ML.

This two-tiered architecture introduces a few challenges however, namely: it needs to keep data consistent between the data lake and the warehouse, which can often take days to load new data [14]; data needs to be duplicated and stored both in the data lake and in the warehouse, introducing high storage costs; two ETL steps are needed, to load data from operational sources into the data lake, and then to curate and load data into the warehouse.

Lakehouses solve the issues described by providing warehouse functionality on top of data lakes. They allow all data analysis jobs, including both ML and BI, to be run over the same data source, avoiding data duplication and data staleness, while additionally decoupling of compute and storage. They also provide compatibility with efficient big data tools such as Apache Spark [68], as client libraries analyze the data directly stored in the object store instead of only being readable through some proprietary query engine running in the warehouse servers. Using only open file formats also avoids vendor lock-in of data, meaning migrating from one lakehouse system to another is as simple as translating the metadata from one's format to the other, without needing to perform any changes on the actual data.

### 2.3.2 Querying in Lakehouses

By taking advantage of data lakes' directly accessible network storage, distributed big data tools are ideal for querying lakehouses. After parsing the metadata layer to know which files to consider as part of the table, and which to skip due to statistics, these tools can operate directly on the data files. We now briefly describe the industry trends for distributed data processing commonly used with lakehouses.

Today's model for distributed big data processing was popularized by Google's MapReduce [29], which had several parallel worker nodes read data from remote storage, apply a *map* function, and send the results to fewer worker nodes that would apply a *reduce* function to aggregate the transformed data. This model proved effective for very large volumes of data and Apache Hadoop [7] was created to provide an open implementation, as well as other utilities such as HDFS: a distributed file system optimized for MapReduce-like accesses. Interestingly, HDFS has seen use beyond Hadoop applications and can be used as storage for a data lake [14].

Apache Hive [8] was a precursor to lakehouses, aiming to implement a warehouse on top of HDFS files by maintaining table schemas and other metadata in a separate service, and allowing it to be queried with a SQL-like interface, which would be compiled to Java code for Hadoop MapReduce. Over time, Hive's limitations became apparent: tracking entire folders instead of individual files resulted in long file scans; it had limited version control, schema updates, and ACID compliance. Despite them, the Hive Metastore – the Hive subsystem to track metadata – is still used as a catalog for lakehouses [65].

Despite its success, MapReduce used distributed memory in a very limited way, which penalized workloads that reused intermediate results across computations, such as iterative ML and graph algorithms, or data analysts who ran interactive ad-hoc queries. To overcome this while also providing better fault tolerance, Spark and its Resilient Distributed Datasets abstraction [68] were created, providing considerably better performance than previous paradigms. With its optional SQL and DataFrames interfaces [15], Spark became a very popular query engine for big data analysis and, by using lakehouses' metadata layers, is now able to provide Hive's relational query functionality much more efficiently. Other popular query engines used for querying data lakes and – by extension – lakehouse data are Flink [6], Snowflake [25], BigQuery [47], Presto [60], DuckDB [32], and the Databricks Runtime [28] which modifies Spark with their Photon query engine [18] to producing more efficient query plans considering lakehouse metadata.

### 2.3.3 Lakehouse transactions and Open Table Formats

The major difference between each lakehouse system is their Open Table Format (OTF), which is the specification of how metadata about the tables should be stored and parsed, and how clients should coordinate when committing to a table. In this section we describe the main common characteristics regarding OTFs.

To achieve their goals, lakehouse systems maintain, on the object store, lists of the files that belong to each table, and upon querying a table, a client first reads this list to know which files should be considered in the query, possibly alongside some metadata that allows the client to avoid reading files for which all records would be filtered out [14].

Whenever updating a table by adding rows, a client creates a new file and updates the list of files belonging to that table. For row removals and updates there are two strategies, which are often used together to balance their trade-offs:

1. *Copy-on-Write (CoW)*, where the file containing the old rows is deleted from the table and a new one is created with all the unmodified rows, with the new version of updated rows, and without any deleted row. This does mean, however, that modifying a small number of records within a file requires almost the whole file to be re-uploaded, resulting in a lot of duplicated data.

2. *Merge-on-Read (MoR)*, where the file containing the old rows is kept in the table, and a new *delete file* is created and added to the table, indicating which rows of the other file should be filtered out during a query. This considerably increases the performance of deletes and updates, but results in an also considerable decrease in read performance [42].

Lakehouses use multi-versioned concurrency control (MVCC) and OCC to handle concurrent transactions: at the end of a transaction with writes, the client atomically commits if no other transaction committed since it started, creating a new snapshot with metadata describing the state of the table. If there was some concurrent transaction which modified some of the same data items, the transaction is aborted. If no concurrent transaction modified the same data items, the client tries committing again over the new metadata. How clients coordinate to commit atomically depends on the OTF, as we'll see further in Section 2.3.5.

When coupled with immutable files, this approach gives lakehouses SI guarantees as a client only determines which files it should consider during a transaction once, in the start of the transaction. Since files are immutable, that transaction will always see the database in the same state during its entire execution [14]. Stronger isolation such as serializability can be achieved by also comparing the read set of a committing transaction against the set of modified files of any transaction that committed concurrently, and aborting if they overlap. [22, 28]

OTFs must also track table schemas and allow clients to update the schema in a similar way as writing to a table. An advantage lakehouses have over traditional DBMSs is that when changing the schema, old files don't need to be updated, and the client simply ignores removed columns in old files or considers a default value for new columns in files created before they were defined [14].

As we explain further in Section 2.3.5, different OTFs store this metadata in different ways, be it in a log, a manifest file tree structure, or even an auxiliary DBMS.

### 2.3.4 Limitations of lakehouse systems

Doing everything over an object store and open file formats doesn't come without downsides. We now explain some limitations of current lakehouse systems, some inherent to their design, and others which can be improved.

**Object store latency** Access to data in object stores incurs significantly high latency, although this can be mitigated by combining small files into larger ones, or caching data in faster storage such as an attached SSD [14].

**Large conflict granularity** When detecting conflicts between concurrent transactions, Relational Database Management Systems (RDBMSs) that provide SI only abort a transaction if the same rows were modified concurrently [59]. In contrast, the granularity for lakehouses is often at the file level, which can contain many rows [13, 65, 51]. This leads to transactions that don't logically conflict being aborted, significantly reducing write throughput and leading to wasted file uploads.

This limitation is however addressed to some level by lakehouse clients like the Databricks Runtime, which can detect and solve row-level conflicts, although this feature is not compatible with table partitioning [28].

**Limited isolation options** RDBMSs usually implement, at least, all four isolation levels defined by ISO/IEC SQL standard [41] already described in Section 2.1.1. Lakehouses' architecture and commit protocols limit them to providing at isolation at least as strong as SI. Although Project Nessie [1] proposes clients can also achieve *Read Committed* isolation by fetching table metadata before every operation in a transaction, the protocol they describe seems to results in performance for weaker isolation. Although at least SI is often sufficient, this doesn't allow system administrators to increase concurrency by weakening the isolation level, as would be the case if the *Repeatable Read*, *Read Committed*, or *Read Uncommitted* levels defined in ISO/IEC SQL standard were supported.

It helps that the typical lakehouse workload has few writes, but this also means write transactions are often very long [14], increasing the chance of conflicts between any two transactions, and making rollbacks result in more wasted work, which can be costly in a cloud computing scenario.

**Multi-table transactions** Another feature that is expected of RDBMSs but wasn't generally provided by default in lakehouses until very recently is multi-table transactions. This has two implications: $(i)$ although during a transaction all operations over a table read from a consistent snapshot taken at the time the transaction read the OTF files, there is no such guarantee across different tables within a transaction, meaning a transaction could be seeing two snapshots from two different tables that never existed simultaneously, violating SI; and, more severely, $(ii)$ if a transaction writes to two different tables, it will have to execute the optimistic commit protocol separately for each table, which isn't possible to

do atomically, further weakening isolation as other transactions could read data from a transaction that has not yet fully committed, and violating atomicity as a transaction could fail to commit to a table after it has already committed to another.

This means that, while most lakehouses claim to provide SI or stronger, they can only do so when operating over a single table. Otherwise, the provided isolation level is closer to *Repeatable Read* at the table-level.

Solving this isn't as simple as having a single snapshot log for the entire database. The main argument against this is that the OCC approach of atomically progressing the snapshot would result in too many conflicts even for transactions that are completely unrelated and, as stated earlier, can result in significantly lower throughput.

Only in 2025 were implemented solutions to overcome this limitation, and the trend seems to be using an auxiliary system to take care of this special coordination [27, 1, 2], although it might increase the operational dependencies of deploying a lakehouse. There are also efforts to add this support using only the object stores [37, 10].

**Little support for data consistency constraints** Most lakehouse systems don't provide support for consistency constraints that are required for RDBMSs to be relational and consistent: uniqueness of values within a table and referential integrity. Enforcing them would likely be challenging because: $(i)$ lakehouses indexes are primitive when compared to traditional DBMSs, often consisting of just the prefix in the file names, and being more akin to partitioning, making scans for individual keys costly; $(ii)$ most systems' lack of multi-table transactions make enforcing referential integrity simply impossible; $(iii)$ clients would have to check constraint violation before committing, and again each time some concurrent transaction committed before them.

This is a clear weakness when comparing with warehouses, the type of systems lakehouses aim to replace. As popularized by the Kimball architecture [44], databases running in warehouses tend to rely heavily on primary and foreign keys, so lakehouse databases must find other approaches for data representation.

Although most systems have an approximation of primary keys, as they support or require rows to have uniquely generated IDs to track their lineage across updates, Hudi [9] is currently the only OTF that allows custom generation of these IDs and tracks their existence in a metadata field for each table.

**No benefits from proprietary file formats** Proprietary warehouse file formats are created and optimized for their respective query engines. By using only open file formats that were created for somewhat general use with several programs, lakehouse vendors lose the ability to make specific optimizations for their specific systems and preferred query engines. However, a client can still benefit from specific optimizations in a limited form when data is cached, by caching it in a proprietary format, since it will only be read by a specific client using a specific engine. For example, the Databricks Engine partially decompresses the Parquet data it loads locally [14].

### 2.3.5 Lakehouse case studies

Now that we have introduced and described general aspects of the lakehouse architecture, we provide concrete examples of lakehouse implementations and the current direction being taken by each.

#### 2.3.5.1 Delta Lake

Delta Lake [13] is an OTF created by Databricks but now under the Linux Foundation [48].

It works by storing a log in the object store that tracks changes in metadata made by transactions such as adding or removing to the set of files that make up table, schema changes, change in version of the Delta Lake protocol being used, adding provenance information, and updating application transaction identifiers, for applications to include their own metadata within logs to, e.g., ensure exactly-once semantics in streaming jobs [13].

Log entries are JSON [43] files with increasing names on increments of 1 (e.g., *000001.json*, *000002.json*, …). Committing clients may checkpoint the log with Parquet files that compress information of all previous logs, such as omitting any file additions that were later removed, and then update the *_last_checkpoint* file to point to the new checkpoint. Whenever a client starts a new transaction, it lists all log files starting from the latest checkpoint and parses them, thus getting a snapshot of the table's state and being able to read its files and understand them. To commit a transaction with writes, it atomically appends to the log a file with the number of the last read log entry plus 1, which is done using the object store's atomic *put-if-absent* operation. If another concurrent transaction committed before, the client fails to create the log file and must try again if no conflicts were detected. It's relevant to note that this approach means the write transaction rate is limited by the latency of the put-if-absent operations.

The transaction log, which can be easily tailed by listing logs entries lexicographically since the last one read, also makes Delta Lake suitable for a message bus system to implement streaming pipelines [13].

Delta Lake provides many optimizations to improve performance when considering the high latency of listing and reading files from cloud object stores. An example of this is the `OPTIMIZE` command, which in a transaction combines many small files into fewer larger ones and computes missing statistics. For this to not cause conflicts with ongoing data streaming queries, a flag is set on the log entry to indicate that although files were changed, the data itself wasn't, i.e., they were just reorganized and the streaming job doesn't need to update its computations.

Logs also contain statistics of added files, such as *null* counts and the minimum and maximum values of a column, to help queries skip reading unnecessary files.

While Delta Lake doesn't directly support multi-table transactions, Databricks is working towards providing it through an integration with the Unity Catalog service [27].

#### 2.3.5.2 Apache Iceberg

Iceberg [65] is an OTF created by Netflix but now under the Apache Software Foundation [11].

Like Delta Lake, it relies on metadata files in the object store to describe the files belonging to a table, schema, and statistics, but it does so by using a hierarchical structure, instead of a linear log, and uses an auxiliary hosted service called a *catalog* to manage concurrency. The metadata layer hierarchy can be described as follows:

1. the **catalog** holds a pointer to each table's current *metadata file*;
2. a **metadata file** tracks the table schema, partitioning configuration, any custom properties, and the snapshots of the table, which in turn contain some information about the transaction and lineage that generated that snapshot, and point to a *manifest list file*;
3. a **manifest list file** points to several *manifest files*, and include metadata for each of them, so queries can avoid reading unnecessary manifest files;
4. a **manifest file** points to several *data files*, alongside statistics, and can be reused by different manifest lists;
5. **data files** are the open-format files (e.g., Parquet, ORC) containing the actual data stored in the lakehouse, and can also refer to *delete files*, which simply describe which rows of other data files don't belong to the table anymore, as the result of a deletion or upsert.

When reading from a table, a client queries the catalog for the latest metadata file for that table, and then parses the files down the hierarchy described above, reading from the latest snapshot.

When committing, a writer creates a new metadata file with snapshot $S + 1$ pointing to a new manifest list pointing to any new or reused manifest files, based on the latest table snapshot $S$ read from another metadata file at the start of the transaction. It then asks the catalog to atomically swap its metadata pointer to the new metadata file with version $S + 1$ if the latest version is still $S$. Otherwise, the commit fails and the writer checks if it can reapply its changes on top of the new version without conflicts and tries again, creating another metadata file based on the newer one.

A powerful Iceberg feature made possible by this approach is branching and tagging which, similarly to Git [36], allows named pointers to several forking histories of the database state to exist simultaneously. Two simple use cases are: $(i)$ tagging historically relevant snapshots to prevent them from being garbage collected for auditing purposes, and for easy posterior access; $(ii)$ performing several transactions on a secondary branch to verify its write effects before fast-forwarding the main branch to match it. This feature alone isn't enough to solve conflicts when merging however. The Nessie catalog [1] uses this feature extensively to, among other things, provide multi-table transactions.

This hierarchical approach introduces downsides however, as a client must separately read three files one at a time to descend the hierarchy, limiting the parallelism allowed for reads, and forcing the client to wait for the object store's latency.

Iceberg also provides an open metadata format for table views, which are essentially queries that can be referenced as tables, and are re-executed every time. Views are implemented by query engines through maintaining metadata about it such as the SQL query that defines it, but this metadata is often in each engine's proprietary format, making it difficult for an engine to read or alter views created through another, even if the information required is largely the same. Iceberg defines an open view metadata format similar to its table metadata format, and can also be modified with optimistic concurrency atomic pointer swaps. This is relevant for us as it might be sufficient to provide weaker consistency or local replication only for view metadata as a special case.

Iceberg tracks the lineage of every row modified on updates, and although not yet supported by the main Iceberg client, the Databricks Runtime Engine uses this to detect and resolve row-level conflicts between concurrent transactions [28]. This is already useful to avoid unnecessary rollbacks across transactions, but can be especially useful in a geo-replication scenario such as in our proposal.

#### 2.3.5.3 Apache Hudi

Hudi [9] is an OTF created by Uber but now under the Apache Software Foundation [11]. It provides similar guarantees to Iceberg, but uses a mix of strategies to improve streaming data ingestion, as opposed to bulk loads, and relies on periodic maintenance jobs to provide efficient queries.

It logs operations done on each table alongside the respective metadata in individual, totally ordered files within a directory. These files are periodically compacted into *Log Structured Merge tree (LSM tree) files*, which contain the contents of several operations, and this is done at several levels. This follows the LSM tree [57] strategy employed by many database systems, which aims to provide fast appends to logs which are stored at several levels. Once a level is full, its contents are appended to the next, bigger and more storage efficient, level. Each level is indeed more efficient than the previous because the larger file sizes avoid the latency of reading many separate object store files, and take better advantage of Parquet data compression. Manifest files are then created to indicate that the new LSM tree files are part of the table metadata.

Similarly to Iceberg, it requires an external coordination service, but since its table versions don't necessarily increase predictably in increments of 1 to also support using real time as timestamps, it can't rely on atomic pointer swaps and instead requires a client to lock the table while it checks for conflicts.

#### 2.3.5.4 DuckLake

DuckLake [51], by MotherDuck and under the DuckDB Foundation [33], is not an OTF and instead implements a lakehouse by replacing the metadata layer which others store in cloud object storage with a traditional RDBMS, while keeping the actual data it references as open-format files in a cloud object store like other lakehouses. They argue this design overcomes the limitations imposed by other systems' strive to reduce operational dependencies and maximize interoperability and scalability by having both data and metadata be fully kept in the object store.

This approach allows DuckLake to avoid some of the limitations described in Section 2.3.4 inherent to most lakehouse systems. It doesn't suffer from the high latency costs of both listing and retrieving different metadata files, as this is efficiently done on the backing database, thus avoiding several round-trips to the object store. It also allows DuckLake to easily provide support for cross-table transactions.

An argument against DuckLake would be the fact that it reintroduces some coupling between compute and storage thus limiting scalability, but metadata is generally much smaller than the data itself. The creators of DuckLake argue a PostgreSQL catalog in a modern machine is able to scale well enough to track metadata referencing hundreds of terabytes of data and to serve thousands of compute nodes [51].

Another interesting use of DuckLake's metadata database is storing small changes to data temporarily, to then flush those changes at once to the object store as new data files. This avoids creating very small data files which can impact performance of table scans on the object stores, although it reduces data freshness by a configurable amount. As exemplified previously, traditional lakehouses mitigate this issue by running periodic data files compaction jobs.

Although already available an open-source, DuckLake is still in pre-release stage, so its specification is subject to changes. Because it's still new, it also doesn't have as many client query engines that support interfacing with it as other lakehouse formats.

#### 2.3.5.5 Comparing lakehouse implementations

A 2023 study [42] compared the performance between Delta Lake, Iceberg, and Hudi. It showed that, with the same query engine, Delta Lake queries tended to be faster than Hudi, and Hudi queries were faster than Iceberg. It concluded this difference mainly came from the difference in throughput from reading the data files. This is because Hudi created more small files, putting emphasis on object store latency. Iceberg, to support column drops and renames, required a custom Parquet reader that was slower than the others. For all three systems, reading metadata was the bottleneck for very small queries.

Importantly, the authors also open-sourced their custom lakehouse benchmark suite, LHBench. So we can use for evaluating our work if appropriate.

#### 2.3.5.6 LakeVilla

LakeVilla [37] is an ongoing research effort to implement both multi-table transactions and partial recovery from aborts without the help of an external service, coordinating only through the object store, all while still allowing for non-LakeVilla clients to read and modify data. Explained briefly, it does so by employing a few techniques: $(i)$ marker files to reserve a snapshot when starting a transaction, possibly marking many tables, acting as commit locks while still allowing data to be written concurrently; $(ii)$ a global vector clock tracking the latest snapshot for each table, which can be incremented independently or atomically for multiple tables; and $(iii)$ having logical logs of a transaction's operations in their marker files, to allow partial rollbacks upon conflict detection.

## 2.4 Summary

To achieve our goals of bringing geo-replication to lakehouses, we need to understand database transactions, replication techniques, replica consistency, cloud computing, and lakehouse systems. In this chapter we presented these concepts and provided examples of recent works that developed them by using techniques we can adopt.

Our study of systems providing replication allows us to adapt their contributions to the context of lakehouses. We took particular interest in lazy multi-master replication with causal consistency. We also explored the different ways these systems handle conflicts and ensure replica convergence.

Having a historical view of the lakehouse architecture helps us understand what problems they aimed to address in the first place, so we don't reintroduce them.

Finally, studying the different lakehouse systems allows us to understand different approaches used to achieve their goals, and what properties we must preserve even in the presence of replication. It's also important to understand what are the overlapping characteristics of these systems, in order to provide a solution that is generally compatible lakehouses.

# 3

# Proposed work

In this chapter, we: present the goals we aim to achieve with our work; describe the existing system on which we intend to implement our solution; describe our initial proposed solution, and how we intend to implement it; how we intend to evaluate the effectiveness of our solution; and, finally, the work plan.

## 3.1 Goals

As presented in Section 1.3, our goals are: $(i)$ enable highly available geo-replication in lakehouse systems, while minimizing performance overhead; $(ii)$ provide sensible convergent conflict resolution; and $(iii)$ implement a usable prototype.

While not absolutely necessary, there are two other desirable characteristics that we will aim for: $(i)$ minimize the degree to which current systems would need to be adapted to benefit from our work, so it can be easily adopted by adding a component to the system rather than introducing major modifications; and $(ii)$ have a general enough solution that can be easily adapted to other lakehouse systems by considering their common characteristics.

If we succeed in our main goal with time to spare, we will also look into how we can extend our solution to also provide partial replication.

## 3.2 Base system

We intend to implement our solution on top of DuckLake [51], introduced in Section 2.3.5.4: a lakehouse format that stores its metadata on the catalog's database, as opposed to doing so on the object store, as is the case with other lakehouse formats [13, 65]. We chose DuckLake over the other formats for a few reasons:

- despite being early in development, with the first stable version planned for release on February 2026, it already supports multi-table transactions, which – as explained in Section 2.3.4 – are still ongoing efforts on other systems;

- DuckLake simplifies the lakehouse architecture, moving metadata management to the catalog, which is already required anyway by other formats [65], or is being progressively more used for features like governance, faster metadata access, or multi-table transactions [27, 67];
- although no standardized benchmarks have yet been published to reliably compare DuckLake to other systems, DuckDB Labs claims that even queries over petabytes of data aren't enough to overwhelm a catalog database and that metadata access latency is lower than systems like Iceberg. [54]

Choosing DuckLake doesn't come without drawbacks when compared to other systems. DuckLake still has little industry adoption, so it's yet to be shown whether it provides competitive performance and utility across all lakehouse use cases. Furthermore, if we were to choose Iceberg [65], we would be able to take advantage of its standardized REST catalog API. Having this API would allow us to easily extend any existing catalog with some middleware invisible to clients. This could be used, for example, to notify replicator process of a newly committed transaction. Doing so in DuckLake is more difficult as clients interface directly with the catalog's SQL connector, which can be different depending on the used Database Management System (DBMS).

Some relevant characteristics specific of DuckLake, which should be taken into consideration for replication are:

- the catalog must be a DBMS that supports transactions and primary key constraints as defined by the SQL-92 standard;
- *deletes* always follow a *Merge-on-Read (MoR)* policy, meaning *delete files* are generated referencing a *data file* and its rows;
- the catalog stores, for each data file, the continuous range of row IDs it contains with the *row_id_start* and *record_count* columns;
- transaction conflicts are detected at table granularity, with no conflict resolution, although there seems to be no relevant technical issue to implement row-level conflict detection ourselves at replication time; but
- periodically, clients may execute maintenance jobs, which include merging data files with adjacent row IDs into fewer larger files, or rewriting files with many deleted rows into new files where the non-deleted rows simply don't exist, instead of being referenced by a delete file and ignored while reading, similarly to *Copy-on-Write (CoW)* techniques. This requires us to consider that even without any data updates, rows might get moved to different files due to periodic maintenance;
- DuckLake tracks what order files composing a table should be in when reconstructing a view of that table, but it's unclear whether clients expect this order to be consistent for some reason, or if we can replicate the order differently at different replicas;

- row IDs and schema IDs (for tables, views, and columns) are all sequentially generated, and must be globally unique, but there is no technical requirement for them to be ordered or contiguous, so we can avoid replica coordination by prefixing these IDs with the ID of the replica that generated them;

- snapshot IDs are also sequentially generated and must be unique but, unlike the other IDs, they must both be ordered and contiguous. This is because they define a serialization order of all transactions and serve for detecting concurrency conflicts between transactions that try to commit to the same snapshot ID. We also avoid replica coordination in this case by realising that, given that we aim to support weak consistency, the serialization order of transactions must be different at each replica [21, 64]. Therefore, snapshot IDs *must not* be globally unique, globally ordered, or sequentially increasing across replicas, with these properties being only local. This also means it's insufficient to simply use an existing eventually consistent database as catalog, as we specifically don't want the snapshot order to converge.

Although we focus on a single system for both simplicity and ensuring we have a good specialized solution, it will be interesting to see whether the techniques we develop are compatible or easily adaptable to other lakehouse formats.

To maximize compatibility and ease of adoption, we will try to avoid modifying the DuckLake catalog metadata schema as much as possible. The more changes we introduce to the specification, the more existing systems and clients must be adapted to be compatible with our solution.

## 3.3 Proposed solution

We now present the preliminary design of our proposed solution, specifying the assumptions made.

On a first iteration, we will relax the problem by only considering the replication of data insertion, deletion, and updates, while preserving snapshot reads.

We don't yet plan on allowing concurrent Data Definition Language (DDL) statements (`CREATE`, `ALTER`, `DROP`) which change the schema. Instead we will require executing such operations with full coordination, with protocols such as two-phase commit (2PC) [53]. Allowing concurrency for such operations results in challenging cases for conflict resolution as schema changes are likely to be incompatible among each other. New data respecting the different schemas might also be inserted before any conflict resolution happens, possibly resulting in some of all of such data becoming incompatible with the resulting schema. This, however, sacrifices latency and availability for those operations.

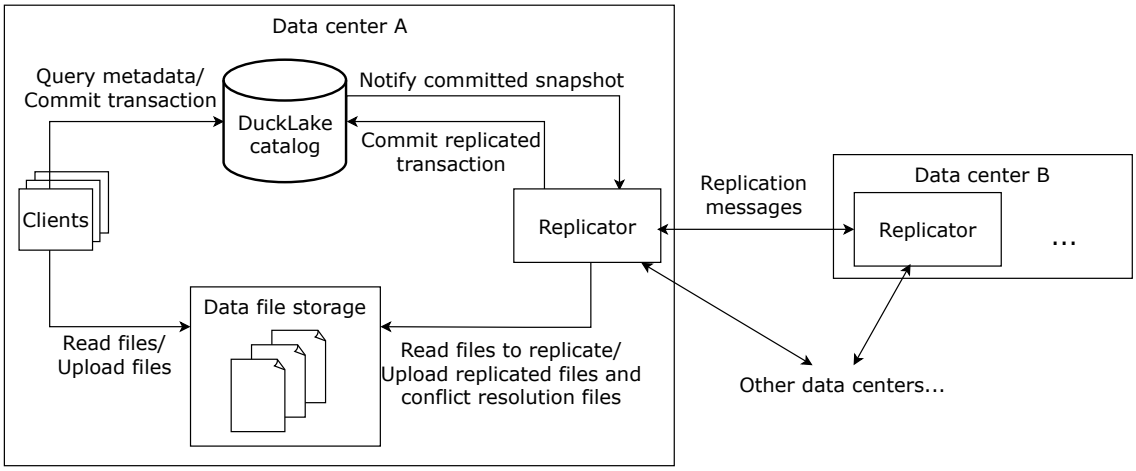
We argue this limitation isn't severe, as such operations are fairly rare in comparison to others.

It's also worth noting that, even without replication, lakehouse systems don't enforce uniqueness constraints or referential integrity, so we will not be concerned with preserving

them in our solution. We intend to leave it as future work, where ideas proposed in Antidote SQL [50] might be adaptable to this context.

### 3.3.1 System components

Figure 3.1: Proposed solution architecture




Given the system characteristics and our goals, we plan to introduce a *replicator* at each replica. Figure 3.1 illustrates a proposal for the system architecture. Arrows originate from the component that starts communication.

The replicator is tasked with intercepting write transactions executed by clients and disseminating them to other replicas. It must also receive transactions from remote replicators and apply them locally, resolving any conflicts. Regarding how it interacts with the base system, we expect to need, at least: $(i)$ a way to notify the replicator that a new transaction was committed locally, which could be done by leveraging some triggered notification functionality on the underlying catalog DBMS, such as PostgreSQL's Asynchronous Notification [59]; and $(ii)$ a way to locally apply writes received from other replicas and resolve conflicts, which can be done by having the replicator use an existing DuckLake client such as DuckDB's DuckLake Extension [51], although considerable modification will likely be required to support sophisticated convergent conflict resolution of concurrent remote writes.

For replicating remote transactions locally, a replicator transactionally creates a new snapshot with the new data and any needed conflict resolution. Although this new snapshot is indistinguishable from any other snapshot resulting from a local transaction it shouldn't be further replicated. Convergence should be achieved by having replicas receive the same updates and using convergent conflict resolution policies. A remote transaction should only be replicated after its referenced data or delete files have already been copied.

We propose that a data or delete file added by a client to a replica's object store is copied to other replicas with the same name and without any internal modifications. This allows us to rely on the fact that data available in a replica will be available in other under

the same name and structure, not only making replication simpler, but also making the system easier to reason about and simpler to maintain or analyze across replicas. This also allows replicas to maintain less information about each other. For example, consider that transaction $t$ deletes row 2 of file $A$ in replica $r_1$ creating delete file $B$. Then replica $r_2$, that already has file $A$, receives the delete file $B$ and metadata of $t$. $r_2$'s replicator can simply create a snapshot that adds $B$ to the table, referencing file $A$. Conflict resolution is explained in Section 3.3.2.

Another aspect that is simplified by not modifying files is data partitioning, which is often used instead of indexing in lakehouses. Filename prefixes are used to partition files according to the values or limits of some columns. If we don't change filenames nor contents, their partitioning is maintained and remains valid after copying.

Although we represent and describe the replicator as a single process, it could be divided into smaller components to reduce the load on a single machine. Data file copying, for example, is simple enough to be done by a separate server process, or even serverless functions. Replicators would then be notified that files have been copied to their local storage.

### 3.3.2 Providing conflict resolution

Making the assumptions we described above, we propose four different policies for handling conflicts of concurrent transactions, and ways to enforce them. In all policies, it is possible that replicated *data* or *delete* files become immediately redundant in the snapshot they are added, and can thus be safely deleted from the data storage and catalog.

We consider two concurrent transactions to conflict if they both modify the same row within a table. Two rows are considered to be the same if they have the same globally unique row ID, which is generated by the replica that first inserted it and maintained by the metadata layer.

Although here we list conflicts between single operations, transactions are often more complex and have several operations of different types, so a combination of the conflict resolutions strategies described for a policy might be needed to replicate a single transaction.

**Update-wins** Update-wins is the policy that results in the least amount changes needed to reconciliate concurrent writes. The conflicts we consider, and how they should be handled are:

- *insert-insert*, *update-insert*, and *delete-insert*: impossible, as row IDs are globally uniquely generated, and the lakehouse doesn't otherwise enforce uniqueness or foreign-key constraints;

- *delete-delete*: nothing should be done, as the same row is deleted and its resulting state is identical after any transaction;

- *delete-update*: nothing should be done, as both transactions will have generated a delete file referencing the same row of the file that originally contained it, except the *update* operation will have also created a file with the new version of the row referencing its ID and added it to the table. Thus, the *delete* operation automatically loses, as it only marks as deleted an old, already replaced, version of the row.

- *update-update*: first, it's necessary to detect whether there were any conflicts by checking whether the IDs of any of the inserted data files overlap. If so, there were conflicts and we can employ a last-writer-wins conflict resolution by creating a local-only delete file referencing the conflicting rows of the conflicting files of the losing transaction, which would be the transaction with the smaller physical clock, replica ID pair. More sophisticated conflict resolution of concurrent updates like what is done per-column in Antidote SQL [50] can be considered, although it would require significantly adapting our *identical file copies* replication assumptions.

**Delete-wins** Delete-wins is similar to *update-wins* in all cases but for *delete-update* conflicts, where active conflict resolution becomes needed. All *delete files* added by the *delete* operation need to be scanned to know whether it deleted any row modified by the *update* operation. Then the replicator creates a new *delete file* referencing any rows that were concurrently deleted and updated, in the respective *data files* added by the *update*.

**Last-writer-wins** Last-writer-wins is similar to *delete-wins*, but only deleting rows updated by the transaction with the smaller timestamp.

### 3.3.3 Providing transactional causal consistency

To provide Transactional Causal Consistency we can apply the conflict resolution described above, while only allowing remote transactions to be replicated once its causal dependencies are locally met. As shown by the systems introduced in Section 2.2.3, there are several ways to do this, often with trade-offs between causality metadata size and remote update visibility latency.

Our scenario is somewhat different than the one presented by those systems, and has to be adapted. They often assume local partitions within a replica [4, 49, 21, 34], while the object store's interface gives us the illusion of a single partition and allows us not to worry about visibility of operations across local partitions. On the other hand, we have the additional challenge of having to keep the catalog metadata consistent with the data itself after replication.

Although Saturn [21] shows good performance results and has a similar proposal as ours for replicating data and metadata separately, its additional operational dependencies and maintenance cost from requiring a separate broadcast tree topology to disseminate metadata is undesirable and complicates the easy scalability that motivates the lakehouse architecture.

We propose to follow a design similar to COPS [49] for metadata tracking, replicating, alongside each transaction, its *immediate* dependencies. We don't have to consider additional metadata introduced by COPS's "get transactions" variant as it essentially provides local Snapshot Isolation (SI) read guarantees, which are already provided by the lakehouse design.

### 3.3.4 Supporting all lakehouse functionality

Before we can use our solution in a real system, we need to address any simplifications we made on the assumptions of the available lakehouse functionality. More specifically, we need to support: $(i)$ DDL statements, which might not only require coordination, but that clients interface directly with the replicator; $(ii)$ maintenance jobs, which result in files being largely rewritten. This not only introduces conflicts for no change in actual data, but also alter previous snapshot metadata; $(iii)$ keeping statistics consistent with data after replication.

### 3.3.5 Partial replication

After providing causal consistency under full replication, we will try to explore solutions compatible with partial replication if time allows, which is likely to be a challenge and require a more intrusive solution. We expect the need to modify clients to be able to execute queries where not all required data is available at the local replica. The lakehouse metadata will also likely need to be modified in order to track and tell clients where non-replicated data is.

## 3.4 Evaluation

We plan to evaluate our work by comparing its throughput and latency against a non-replicated DuckLake system with clients across regions. We will consider different amounts of physically distant replicas in our solution to measure how the system and causality metadata scales.

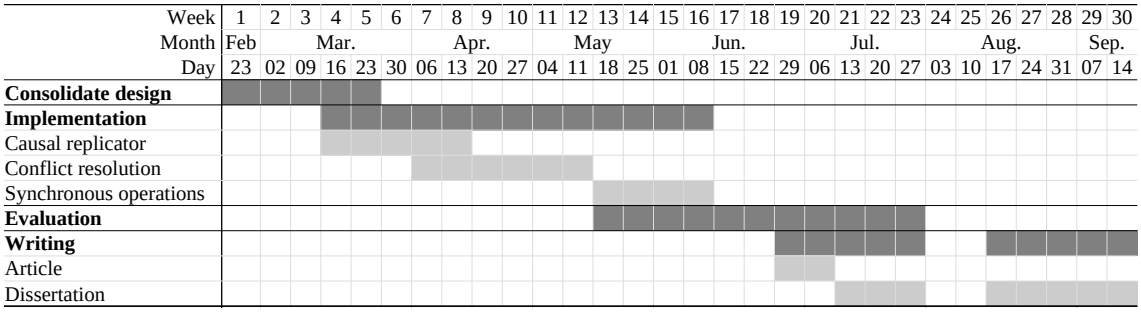
We will also study any extra resource costs introduced by our solution. We expect overhead to be introduced by conflict resolution files, data transfer bandwidth, and replicated data. Additionally, we will measure how any coordination introduced for DDL operations affects throughput.

Finally, if we succeed in implementing partial replication, we will compare this version against the causally consistent, fully replicated, one.

An interesting option for conducting the benchmarks is with the LHBench benchmark [42] made specifically for lakehouses. It extends the TPC-DS data warehouse benchmark [55] with tests aimed to expose differences of *update* implementations and metadata handling. TPC-DS models a decision support system, with a star schema and workloads such as business reports, iterative queries, and data mining.

## 3.5 Work plan

Figure 3.2: Proposed schedule Gantt chart




In this section we present the work plan leading up to the dissertation delivery. As illustrated by Figure 3.2, we plan to take the following steps within their expected schedules, which may overlap:

1. **Consolidate design.** Feb. 23 – Mar. 27 *(5 weeks)*

   First, we will improve the design proposed in Section 3.3, polishing it and making sure it is adequate to support all lakehouse features as described in Section 3.3.4.

2. **Implement solution.** Mar. 4 – Jun. 14 *(13 weeks)*

   Then we will implement, over DuckLake, the different components of our solution. We will begin by implementing a replicator that enforces causal delivery, following the architecture described in Section 3.3.1 and the techniques proposed in Section 3.3.3. Then we will implement convergent conflict resolution of concurrent updates as described in Section 3.3.2. Finally, we will add coordination for synchronous operations, such as DDL statements.

3. **Evaluate solution.** May 18 – Aug. 2 *(11 weeks)*

   After we're able to provide replication with Transactional Causal Consistency, we will begin evaluating our solution as described in Section 3.4.

4. **Write article.** Jun. 29 – Jul. 12 *(2 weeks)*

   We will try to submit a preliminary version of our work to INForum, possibly without the complete evaluation.

5. **Write the dissertation.** Jul. 13 – Aug. 2, Aug. 17 – Sep. 20 *(8 weeks)*

   We will begin writing the dissertation as soon as we finish the article. We plan on taking a 2 week break before resuming and finishing the work.

Partial replication will be considered if we manage to implement our solution considerably ahead of schedule.

## 3.6 Summary

In this chapter, we presented the work we intend to conduct in order to address the problem presented in Chapter 1. We stated our goals, described the system on top of which we pretend to implement our solution, presented a proposal of the solution design, and described how we intend to evaluate its effectiveness.

# Bibliography


[1] *About Nessie - Project Nessie: Transactional Catalog for Data Lakes with Git-like Semantics*. URL: https://projectnessie.org/guides/about/ (visited on 2025-10-23) (cit. on pp. 18, 19, 21).

[2] *Add Multi-Table Transaction API · Issue #10617 · Apache/Iceberg*. GitHub. URL: https://github.com/apache/iceberg/issues/10617 (visited on 2025-11-20) (cit. on p. 19).

[3] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. "Causal Memory: Definitions, Implementation, and Programming". In: *Distributed Computing* 9.1 (1995-03-01), pp. 37–49. ISSN: 1432-0452. DOI: 10.1007/BF01784241 (cit. on p. 2).

[4] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguica, and M. Shapiro. "Cure: Strong Semantics Meets High Availability and Low Latency". In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS). Nara, Japan: IEEE, 2016-06, pp. 405–414. DOI: 10.1109/icdcs.2016.98 (cit. on pp. 2, 12, 14, 30).

[5] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. "FAWN: A Fast Array of Wimpy Nodes". In: *Commun. ACM* 54.7 (2011-07-01), pp. 101–109. ISSN: 0001-0782. DOI: 10.1145/1965724.1965747 (cit. on p. 11).

[6] *Apache Flink®*. URL: https://flink.apache.org/ (visited on 2026-02-05) (cit. on p. 16).

[7] *Apache Hadoop*. URL: https://hadoop.apache.org/ (visited on 2026-02-05) (cit. on p. 16).

[8] *Apache Hive*. URL: https://hive.apache.org/ (visited on 2026-02-05) (cit. on p. 16).

[9] *Apache Hudi Technical Specification 1.0 | Apache Hudi*. URL: https://hudi.apache.org/learn/tech-specs-1point0 (visited on 2026-01-06) (cit. on pp. 2, 19, 22).

[10] Apache Iceberg, director. *Multi-Table Multi-Statement Transaction with Apache Iceberg*. 2025-04-30. URL: https://www.youtube.com/watch?v=Dsao9M6zfwM (visited on 2025-11-19) (cit. on p. 19).

[11] *Apache Software Foundation*. 2025-12. URL: https://www.apache.org (cit. on pp. 21, 22).

[12] *Apache ZooKeeper*. URL: https://zookeeper.apache.org/ (visited on 2026-02-08) (cit. on p. 10).

[13] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. Van Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, and M. Zaharia. "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores". In: *Proceedings of the VLDB Endowment* 13.12 (2020-08), pp. 3411–3424. ISSN: 2150-8097. DOI: 10.14778/3415478.3415560 (cit. on pp. 2, 15, 18, 20, 25).

[14] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia. "Lakehouse: A New Generation of Open Platforms That Unify Data Warehousing and Advanced Analytics". In: 11th Annual Conference on Innovative Data Systems Research (CIDR '21). Online, 2021-01. URL: https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf (cit. on pp. 1, 15–19).

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. "Spark SQL: Relational Data Processing in Spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD/PODS'15: International Conference on Management of Data. Melbourne Victoria Australia: ACM, 2015-05-27, pp. 1383–1394. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742797 (cit. on p. 16).

[16] H. Attiya, F. Ellen, and A. Morrison. "Limitations of Highly-Available Eventually-Consistent Data Stores". In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. PODC '15. New York, NY, USA: Association for Computing Machinery, 2015-07-21, pp. 385–394. ISBN: 978-1-4503-3617-8. DOI: 10.1145/2767386.2767419 (cit. on pp. 2, 7, 10).

[17] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. "Putting Consistency Back into Eventual Consistency". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. New York, NY, USA: Association for Computing Machinery, 2015-04-17, pp. 1–16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741972 (cit. on p. 14).

[18] A. Behm, S. Palkar, U. Agarwal, T. Armstrong, D. Cashman, A. Dave, T. Greenstein, S. Hovsepian, R. Johnson, A. Sai Krishnan, P. Leventis, A. Luszczak, P. Menon, M. Mokhtar, G. Pang, S. Paranjpye, G. Rahn, B. Samwel, T. Van Bussel, H. Van Hovell, M. Xue, R. Xin, and M. Zaharia. "Photon: A Fast Query Engine for Lakehouse Systems". In: SIGMOD/PODS '22: International Conference on Management of Data. Philadelphia PA USA: ACM, 2022-06-10, pp. 2326–2339. ISBN: 978-1-4503-9249-5. DOI: 10.1145/3514221.3526054 (cit. on p. 16).

[19] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. "A Critique of ANSI SQL Isolation Levels". In: *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. SIGMOD '95. New York, NY, USA: Association for Computing Machinery, 1995-05-22, pp. 1–10. ISBN: 978-0-89791-731-5. DOI: 10.1145/223784.223785 (cit. on pp. 6, 7).

[20] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. E. Rodrigues. "On the Use of Clocks to Enforce Consistency in the Cloud." In: *IEEE Data Eng. Bull.* 38 (2015), pp. 18–31. URL: https://scholar.tecnico.ulisboa.pt/records/aVjPlQnpB-LjlsGsfBCihPgI_5PuSYEfwQIf?lang=en (visited on 2026-02-04) (cit. on p. 9).

[21] M. Bravo, L. Rodrigues, and P. Van Roy. "Saturn: A Distributed Metadata Service for Causal Consistency". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17: Twelfth EuroSys Conference 2017. Belgrade Serbia: ACM, 2017-04-23, pp. 111–126. DOI: 10.1145/3064176.3064210 (cit. on pp. 9, 11, 14, 27, 30).

[22] M. J. Cahill, U. Röhm, and A. D. Fekete. "Serializable Isolation for Snapshot Databases". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. New York, NY, USA: Association for Computing Machinery, 2008-06-09, pp. 729–738. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376690 (cit. on p. 17).

[23] B. Chattopadhyay, P. Pedreira, S. Agarwal, S. Vakharia, P. Li, W. Liu, and S. Narayanan. "Shared Foundations: Modernizing Meta's Data Lakehouse". In: (2023) (cit. on p. 2).

[24] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. "Spanner: Google's Globally-Distributed Database". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. USA: USENIX Association, 2012-10-08, pp. 251–264. ISBN: 978-1-931971-96-6 (cit. on pp. 2, 13).

[25] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. "The Snowflake Elastic Data Warehouse". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016-06-14, pp. 215–226. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2903741 (cit. on p. 16).

[26] Databricks, director. *Incremental Iceberg Table Replication at Scale*. 2025-07-07. URL: https://www.youtube.com/watch?v=2ojNGYn6tt8 (visited on 2026-01-31) (cit. on p. 2).

[27] Databricks, director. *Multi-Format, Multi-Table, Multi-Statement Transactions on Unity Catalog*. 2025-07-07. URL: https://www.youtube.com/watch?v=I7bPyNP1OCs (visited on 2026-01-24) (cit. on pp. 19, 20, 26).

[28] *Databricks Documentation*. 2025-08-15. URL: https://docs.databricks.com/aws/ (visited on 2026-01-04) (cit. on pp. 16–18, 22).

[29] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (2008-01), pp. 107–113. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/1327452.1327492 (cit. on p. 16).

[30] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-Value Store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (2007-10-14), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281 (cit. on p. 2).

[31] J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel. "GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014-11-03, pp. 1–13. ISBN: 978-1-4503-3252-1. DOI: 10.1145/2670979.2670983 (cit. on pp. 2, 9, 10).

[32] *DuckDB – An in-Process SQL OLAP Database Management System*. DuckDB. URL: https://duckdb.org/ (visited on 2026-02-05) (cit. on p. 16).

[33] *DuckDB Foundation*. DuckDB. URL: https://duckdb.org/foundation/ (visited on 2026-01-06) (cit. on p. 23).

[34] P. Fouto, J. Leitão, and N. Preguiça. "Practical and Fast Causal Consistent Partial Geo-Replication". In: *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*. 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA). 2018-11, pp. 1–10. DOI: 10.1109/NCA.2018.8548067 (cit. on p. 30).

[35] S. Gilbert and N. Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *SIGACT News* 33.2 (2002-06-01), pp. 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601 (cit. on pp. 2, 8).

[36] *Git*. URL: https://git-scm.com (cit. on p. 21).

[37] T. Götz, D. Ritter, and J. Giceva. *LakeVilla: Multi-Table Transactions for Lakehouses*. 2025-04-29. DOI: 10.48550/arXiv.2504.20768. Pre-published (cit. on pp. 19, 24).

[38] J. Gray, P. Helland, P. O'Neil, and D. Shasha. "The Dangers of Replication and a Solution". In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96. New York, NY, USA: Association for Computing Machinery, 1996-06-01, pp. 173–182. ISBN: 978-0-89791-794-0. DOI: 10.1145/233269.233330 (cit. on pp. 10, 11).

[39] T. Haerder and A. Reuter. "Principles of Transaction-Oriented Database Recovery". In: *ACM Comput. Surv.* 15.4 (1983-12-02), pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291 (cit. on pp. 5, 14).

[40] M. P. Herlihy and J. M. Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990-07-01), pp. 463–492. ISSN: 0164-0925. DOI: 10.1145/78969.78972 (cit. on p. 8).

[41] *ISO/IEC 9075-1:2023*. ISO. URL: https://www.iso.org/standard/76583.html (visited on 2026-01-13) (cit. on pp. 6, 18).

[42] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, and M. Zaharia. "Analyzing and Comparing Lakehouse Storage Systems". In: (2023) (cit. on pp. 17, 23, 31).

[43] *JSON*. URL: https://www.json.org/ (visited on 2026-01-04) (cit. on p. 20).

[44] R. Kimbal and M. Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling, 3rd Edition*. 3rd ed. Wiley, 2013-07. 608 pp. ISBN: 978-1-118-53080-1. URL: https://www.wiley.com/en-us/The+Data+Warehouse+Toolkit%3A+The+Definitive+Guide+to+Dimensional+Modeling%2C+3rd+Edition-p-9781118530801 (visited on 2026-02-04) (cit. on p. 19).

[45] L. Lamport. "Paxos Made Simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001-12-15), pp. 51–58. URL: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/ (visited on 2026-02-02) (cit. on p. 10).

[46] L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978-07-01), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563 (cit. on pp. 2, 9).

[47] J. Levandoski, G. Casto, M. Deng, R. Desai, P. Edara, T. Hottelier, A. Hormati, A. Johnson, J. Johnson, D. Kurzyniec, S. McVeety, P. Ramanathan, G. Saxena, V. Shanmugan, and Y. Volobuev. "BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse". In: *Companion of the 2024 International Conference on Management of Data*. SIGMOD/PODS '24: International Conference on Management of Data. Santiago AA Chile: ACM, 2024-06-09, pp. 334–346. ISBN: 979-8-4007-0422-2. DOI: 10.1145/3626246.3653388 (cit. on p. 16).

[48] *Linux Foundation*. 2025-12. URL: https://www.linuxfoundation.org (cit. on p. 20).

[49] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11: ACM SIGOPS 23nd Symposium on Operating Systems Principles. Cascais Portugal: ACM, 2011-10-23, pp. 401–416. DOI: 10.1145/2043556.2043593 (cit. on pp. 2, 9, 10, 12, 30, 31).

[50] P. Lopes, J. Sousa, V. Balegas, C. Ferreira, S. Duarte, A. Bieniusa, R. Rodrigues, and N. Preguiça. *Antidote SQL: Relaxed When Possible, Strict When Necessary*. 2019-02-10. DOI: 10.48550/arXiv.1902.03576. Pre-published (cit. on pp. 14, 28, 30).

[51] H. M. Mark Raasveldt. *The DuckLake Manifesto: SQL as a Lakehouse Format*. DuckLake. URL: https://ducklake.select/manifesto/ (visited on 2025-10-23) (cit. on pp. 2, 18, 23, 25, 28).

[52] M. Mesnier, G. Ganger, and E. Riedel. "Object-Based Storage". In: *IEEE Communications Magazine* 41.8 (2003-08), pp. 84–90. ISSN: 1558-1896. DOI: 10.1109/MCOM.2003.1222722 (cit. on pp. 1, 15).

[53] C. Mohan and B. Lindsay. "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions". In: *SIGOPS Oper. Syst. Rev.* 19.2 (1985-04-01), pp. 40–52. ISSN: 0163-5980. DOI: 10.1145/850770.850772 (cit. on pp. 10, 27).

[54] H. Mühleisen. *DuckLake - The SQL-Powered Lakehouse Format for the Rest of Us by Prof. Hannes Mühleisen*. director TigerBeetle. 2025-04-08. URL: https://www.youtube.com/watch?v=YQEUkFWa69o (visited on 2026-01-07) (cit. on p. 26).

[55] R. O. Nambiar and M. Poess. "The Making of TPC-DS". In: *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006-09-01, pp. 1049–1058. URL: https://dl.acm.org/doi/10.5555/1182635.1164217 (visited on 2026-02-07) (cit. on p. 31).

[56] P. E. O'Neil. "The Escrow Transactional Method". In: *ACM Trans. Database Syst.* 11.4 (1986-12-01), pp. 405–430. ISSN: 0362-5915. DOI: 10.1145/7239.7265 (cit. on p. 14).

[57] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. "The Log-Structured Merge-Tree (LSM-tree)". In: *Acta Informatica* 33.4 (1996-06-01), pp. 351–385. ISSN: 1432-0525. DOI: 10.1007/s002360050048 (cit. on p. 22).

[58] *Parquet*. Apache Parquet. URL: https://parquet.apache.org/ (visited on 2025-09-25) (cit. on pp. 1, 15).

[59] PostgreSQL Global Development Group. *PostgreSQL*. PostgreSQL. 2026-01-23. URL: https://www.postgresql.org/ (visited on 2026-01-23) (cit. on pp. 18, 28).

[60] *Presto: Free, Open-Source SQL Query Engine for Any Data*. PrestoDB. URL: https://prestodb.io/ (visited on 2026-02-05) (cit. on p. 16).

[61] K. Ren, D. Li, and D. J. Abadi. "SLOG: Serializable, Low-Latency, Geo-Replicated Transactions". In: *Proceedings of the VLDB Endowment* 12.11 (2019-07), pp. 1747–1761. ISSN: 2150-8097. DOI: 10.14778/3342263.3342647 (cit. on pp. 2, 13).

[62] *Replicating S3 Tables - Amazon Simple Storage Service*. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-tables-replication-tables.html (visited on 2026-01-31) (cit. on p. 2).

[63] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. "Conflict-Free Replicated Data Types". In: *Lecture Notes in Computer Science*. Ed. by X. Défago, F. Petit, and V. Villain. Vol. 6976. Lecture Notes in Computer Science. Grenoble, France: Springer, 2011-10, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29 (cit. on pp. 10, 14).

[64] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. "Transactional Storage for Geo-Replicated Systems". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011-10-23, pp. 385–400. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043592 (cit. on pp. 2, 7, 27).

[65] *Spec - Apache Iceberg™*. URL: https://iceberg.apache.org/spec/ (visited on 2025-11-17) (cit. on pp. 2, 16, 18, 21, 25, 26).

[66] W. Vogels. "Eventually Consistent". In: *Commun. ACM* 52.1 (2009-01-01), pp. 40–44. ISSN: 0001-0782. DOI: 10.1145/1435417.1435432 (cit. on pp. 2, 9).

[67] *What Is Unity Catalog? | Databricks on AWS*. 2025-11-14. URL: https://docs.databricks.com/aws/en/data-governance/unity-catalog/ (visited on 2025-11-17) (cit. on p. 26).

[68] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: () (cit. on pp. 1, 2, 15, 16).

[69] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. "Building Consistent Transactions with Inconsistent Replication". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. New York, NY, USA: Association for Computing Machinery, 2015-10-04, pp. 263–278. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815404 (cit. on pp. 2, 13).