Towards Abstractions for Composable Transactional Isolation

Daniel Pereira*
INESCTEC and U. Minho
Braga, Portugal
daniel.s.pereira@inesctec.pt

Filipe Pereira*
InvisibleLab and U. Minho
Braga, Portugal
filipe.pereira@invisiblelab.dev

Rui Lopes*
INESCTEC and U. Minho
Braga, Portugal
rui.l.martins@inesctec.pt

Nuno Faria
INESCTEC and U. Minho
Braga, Portugal
nuno.f.faria@inesctec.pt

Paulo Sérgio Almeida
INESCTEC and U. Minho
Braga, Portugal
psa@di.uminho.pt

José Pereira
INESCTEC and U. Minho
Braga, Portugal
jop@di.uminho.pt

ABSTRACT

There is a renewed interest in transactional database systems. In addition to large-scale cloud-native OLTP systems, there are novel proposals for HTAP and transactional lakehouse systems. At the same time, research proposals on highly available transactions are emerging, which avoid coordination and are particularly useful in high-concurrency and geo-replicated settings that have yet to see mainstream industrial adoption. However, in contrast to data storage and processing, in which there is an increasing interest in composability and there are widely available and used components that can be combined, transactional guarantees are monolithically built in each system.

In this paper, we conduct a thought experiment on how to design database systems with composable transactional guarantees by examining and combining existing database systems with Cure, a research proposal for highly available transactional causal consistency. As a result, we derive lessons on desirable interfaces and useful building blocks for composable transactional implementations in database systems.

VLDB Workshop Reference Format:
Daniel Pereira, Filipe Pereira, Rui Lopes, Nuno Faria, Paulo Sérgio Almeida, and José Pereira. Towards Abstractions for Composable Transactional Isolation. VLDB 2026 Workshop: Fourth International Workshop on Composable Data Management Systems.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dbr-haslab/composable-transactions.

1 INTRODUCTION

The Composable Data Management System Manifesto [31] advocates a fundamental shift in the way database management systems are designed and evolved. Rather than treating them as monolithic artifacts, it proposes viewing them as interoperable components connected through stable interfaces. Under this vision, innovation no longer requires building an entirely new system. Instead, new functionality can be introduced by replacing, extending, or recombining existing components. This approach has already transformed large parts of the database stack. Query processing [9], storage management [27], indexing [19], and data formats [6] are increasingly provided by reusable modules that can be assembled into specialized systems. Experience illustrates the practical benefits of this model, showing that composability can accelerate development while preserving performance and functionality [30].

Most existing work on composability, however, has focused on data storage and query processing. In contrast, transactional guarantees remain largely embedded within individual database engines. Isolation levels, concurrency-control mechanisms, transaction metadata, and replication protocols are typically implemented as tightly coupled components that cannot be easily reused or exchanged across systems.

Interestingly, an early standard for composition of data-centric services, XA, focuses precisely on transactions [43]. By standardizing protocols for efficient versions of the two-phase commit (2PC) protocol [28], it enabled transactions spanning multiple database systems and external resources, providing an interoperability layer for atomic commitment. While successful in standardizing coordination across heterogeneous systems, XA focused exclusively on atomicity and durability. Richer transactional semantics, such as isolation guarantees and consistency models, remained implementation-specific and deeply integrated into each database engine.

Research has now produced a wide spectrum of transactional models beyond those commonly offered by industrial systems [2, 7, 8, 23–26]. Many of these proposals address challenges arising from modern geo-replicated deployments, typically revolving around the fundamental trade-offs exposed by the CAP theorem [18]. Yet despite their theoretical and practical appeal, they have not seen widespread industrial adoption. One reason is that adopting a new transaction model often requires building a specialized storage engine or substantially modifying an existing one, making experimentation and deployment costly.

Transactional Causal Consistency (TCC) [2] is a representative example that provides the strongest consistency guarantees compatible with high availability under network partitions. AntidoteDB [5] provides TCC but at the cost of fully reimplementing the entire stack, with custom-built storage engines and transaction-processing layers that greatly limit its practicality.

*These authors contributed equally to this paper.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

This observation raises a broader question: if composability has proven successful for query processing and storage management, can it also be extended to transactional guarantees? More specifically, can advanced consistency models such as TCC be realized by composing existing database capabilities with a relatively small amount of external functionality, rather than by building entirely new database systems?

In this paper, we explore these questions through a thought experiment centered on TCC. Using Cure [2] as a reference design, we investigate what would be required to realize TCC on top of widely used relational database systems such as PostgreSQL [33], Citus [12], CockroachDB [41], and YugabyteDB [48]. Our analysis leads to three main observations. First, existing MVCC-based relational databases already provide key mechanisms required to realize TCC. Second, support for Distributed Snapshot Isolation (DSI) only simplifies the composition of transactional guarantees in sharded deployments for the case of a single replication agent in each data center. Third, current database systems expose little control over snapshot assignment and do not provide reusable commit-time timestamps, making these aspects the primary obstacle to implementing advanced transaction protocols on top of them.

Our goal is not to present a complete implementation (even though a prototype for the simplest scenario exists). Instead, we use TCC as a concrete and demanding case study to derive insights into the abstractions required for composable transactional isolation. Importantly, the resulting discussion exposes several gaps between existing database interfaces and the needs of advanced transaction protocols.

2 BACKGROUND

2.1 Transactional Isolation Guarantees

Transactional isolation defines which effects of concurrent transactions may be observed by each transaction and what changes can be committed. These guarantees are typically enforced within the Transactional Storage Manager layer [20], which is responsible for coordinating access to stored data. To this end, the TSM relies on concurrency control mechanisms such as two-phase locking (2PL), timestamp ordering (TO), and optimistic (validation-based) concurrency control (OCC) to exclude anomalous executions (i.e., executions that violate the exposed isolation level) and, increasingly often, combines these with multi-version concurrency control (MVCC) to further increase concurrency. We focus mainly on transactional visibility, that is, how the database system controls what each transaction is allowed to observe with read operations.

Read Committed (RC) guarantees that each statement only observes data committed before that statement begins [10]. This prevents dirty reads, where a transaction observes writes from another transaction that has not yet committed. However, RC admits non-repeatable reads, where repeated reads of the same object return different values, and phantom reads, where repeating a predicate-based query returns a different set of matching objects. Snapshot Isolation (SI) strengthens RC by giving each transaction a stable snapshot, so all reads observe a consistent version, while the transaction's own writes remain visible [10]. SI prevents dirty reads, non-repeatable reads, phantom reads, and read skew, where a transaction observes mutually inconsistent values across different objects. However, SI still admits anomalies such as write skew, where concurrent transactions read overlapping data, update disjoint objects, and jointly violate an invariant [1, 10].

Distributed Snapshot Isolation (DSI) extends SI to distributed databases, where a transaction may access data stored across several nodes of a sharded database. As in SI, each transaction must read from a single stable snapshot. However, in a distributed setting this snapshot needs to be consistent across nodes: a transaction should not observe an update at one node while missing another update on which it depends at a different node. Otherwise, the transaction could observe a fractured database version that never existed as a valid global snapshot.

PostgreSQL [33] is a relational database using MVCC. It provides RC and SI, among other guarantees (e.g., Serializable Isolation through Serializable SI [32]). To this end, PostgreSQL assigns an identifier to each transaction when it starts (`xid`) and, to enforce visibility, associates it with an upper bound on previous transactions that have already ended and are visible if committed (`xmin`), as well as a set of identifiers for transactions that have not ended at the time the transaction started and should not be visible (`xip_list`). When a transaction commits, PostgreSQL creates a log record that can be identified by the offset in the log file (i.e., Log Sequence Number [36]). Optionally, PostgreSQL can be configured to generate a (physical) commit timestamp with the `track_commit_timestamp` configuration option [37].

Citus [12] extends PostgreSQL to multiple sharded nodes by using 2PC to ensure atomic updates. However, it keeps the original snapshot mechanism unchanged at each node and thus does not ensure atomic visibility. Consequently, it does not provide DSI: multi-node transactions can observe the partial effects of another transaction that has committed on some nodes but not yet others, an anomaly that SI precludes.

YugabyteDB [48] and CockroachDB [41] are PostgreSQL-compatible distributed databases that enforce transactional isolation, also using MVCC, but on data distributed over multiple nodes. YugabyteDB provides DSI as its default isolation level while CockroachDB provides Distributed Serializable Isolation. In both cases, visibility is enforced with a single timestamp assigned when the transaction commits and used to label all updates, ensuring that each transaction's snapshot is defined simply by the time at which the transaction starts.

When the TSM does not provide these mechanisms, it is possible to directly express transactional logic in SQL. Specifically, TiQuE [17] implements SI by storing both data versions and transaction metadata in regular tables. Much like YugabyteDB, updates are redirected to a temporary area, which in this case is a regular table. Client statements are then intercepted and rewritten against this transactional schema, introducing `WHERE` clauses as needed to filter out invisible versions according to transaction timestamps.

2.2 Transactional Causal Consistency

Under Transactional Causal Consistency (TCC), each transaction reads from a consistent snapshot, similarly to Snapshot Isolation (SI). Unlike SI, TCC does not impose a global commit order or abort

concurrent transactions because of write conflicts, and therefore admits both long fork and lost update anomalies. Essentially, TCC can be understood as the combination of two complementary properties. First, a transaction updating multiple objects must respect atomic visibility, that is, either all updates are made visible simultaneously or none of them are. Note that this guarantee is stronger than failure atomicity [45] alone, which does not account for visibility regarding concurrent readers. Second, every transaction must execute over a causally consistent snapshot of the data store, meaning it is closed under causal dependencies: if it contains the effects of a transaction, it must also contain the effects of every transaction that causally precedes it. A formal definition and comparison to other models can be found in [11].

Cure proposes TCC in a geo-replicated, highly available data store [2]. To circumvent lost updates, Cure stores data as operation-based CRDTs [3], such that the effects of concurrent operations are eventually merged. Cure assumes multiple data centers (DCs), each containing several nodes. The underlying data store is partitioned across the nodes of each DC and fully replicated across DCs using the same partitioning scheme. Nodes maintain multiple versions of each object, tagged with vector clocks derived from loosely synchronized physical clocks, as in Clock-SI [15].

When a transaction starts, its coordinator assigns it a snapshot vector based on the Globally Stable Snapshot (GSS), which summarizes the remote updates known to be available at every local node. Reads return the newest versions included in the snapshot, while writes are buffered until commit. Transactions updating multiple partitions commit atomically through a 2PC within the local DC, with all their writes receiving the same commit vector, given by the maximum of their clocks for all involved nodes.

Updates are propagated asynchronously between DCs, but propagation is decoupled from visibility. A transaction becomes visible only after the local GSS covers its commit vector, thus avoiding causal paradoxes without having to explicitly track dependencies (e.g., as in COPS [24]). Since inter-DC coordination is not required for transaction execution, network partitions between DCs do not prevent clients from making progress, preserving availability.

Cure was implemented in AntidoteDB [5], a key-value store written in Erlang, with Distributed Erlang for intra-DC communication and ZeroMQ [21] for replication across DCs. Interestingly, AntidoteDB can enforce SI locally in each DC and TCC across DCs, allowing local write conflicts to be detected, while concurrent updates originating at different DCs are reconciled through CRDT-based semantics. AQL [4] provides a SQL-like interface over AntidoteDB, but retains its implementation and all its limitations.

3 EXPLORING THE DESIGN SPACE

3.1 General Approach

We consider a geo-replicated system deployed across `D` data centers (DCs). Each DC stores a local replica of the application data and accepts transactions from nearby clients, while committed updates are asynchronously propagated to the remaining DCs. We analyze three scenarios that differ in the number of database nodes and replication agents deployed per DC.

Figure 1 shows a generic architecture for the described scenarios. Clients connect directly to the database, which, depending on the scenario, may need to provide an explicit SQL-based visibility layer. In any case, transparency from the application's perspective is maintained. Agents capture committed updates through a change data capture (CDC) mechanism [13, 34, 35], exchange them with sibling agents in other DCs, and apply remote updates using standard SQL. Depending on the scenario, each DC contains one or more database nodes (and, correspondingly, replication agents).

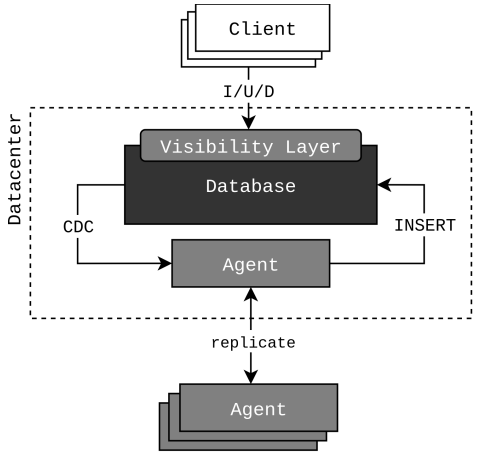

Figure 1: Generic architecture shared by the three scenarios.

Replication agents enforce TCC semantics across DCs. Although relational databases provide point-to-point update propagation mechanisms (e.g., PostgreSQL's logical replication [35]), they are typically unidirectional and provide only a per-publisher total order. When multiple instances act as both sources and destinations, they do not coordinate concurrent updates or provide a modular way to enforce alternative consistency models. Replication agents therefore augment the communication mechanism with the metadata, dependency tracking, and buffering required by TCC.

Moreover, agents exchange committed transactions in batches. Each batch `B` is associated with a version vector (VV) [29] representing its causal dependencies, henceforth referred to as `Deps(B)`. Depending on the scenario, each transaction may also be associated with either a dot [38] or a VV. In the former case, a dot is denoted as `Dot(T) = (d, k)`, where `d` is the origin DC and `k` is the transaction's commit position. In all scenarios, the agent also maintains a persistent `AppliedVV`, stored in agent-managed database tables, where each component increases monotonically and can be either a batch number or a scalar commit timestamp of the last transaction in the batch. The assignment of these timestamps and their required properties is discussed in Section 4.2.

Importantly, a batch is finalized when a configurable threshold is reached (e.g., elapsed time interval, number of accumulated transactions), or immediately before an incoming batch is applied. The latter is used to prevent false dependencies. Subsequent local transactions are instead accumulated in a new batch under the updated `AppliedVV`.

3.2 Scenarios

Scenario 1 comprises a single agent and a single PostgreSQL instance per DC. This is the simplest deployment and serves as the

baseline for comparison with other scenarios. We implemented a simple prototype of this scenario, made available as an artifact, that demonstrates the feasibility of the general approach.

Scenario 2 comprises a single agent and multiple database nodes per DC. Its main motivation is to support data sharding and transactions that span multiple nodes, thereby increasing storage capacity and local query throughput. The database layer may be implemented using Citus, which distributes PostgreSQL tables and transactions across a coordinator-worker architecture, where clients connect exclusively to the coordinator. Alternatively, we explore PostgreSQL-compatible distributed databases such as CockroachDB and YugabyteDB, which offer native distributed transactional guarantees, as explained in Section 2.1.

Scenario 3 comprises multiple database nodes per DC, with one agent per node, allowing transactions to be extracted, transferred, and applied in parallel. This scenario reproduces the original Cure design [2]. Importantly, Citus is not considered in this setting because its coordinator-based architecture would force the agents to interact through a common coordination point, largely defeating the purpose of deploying multiple agents. CockroachDB and YugabyteDB are better suited here, as their distributed and symmetric designs allow multiple agents to work concurrently.

3.3 Assumptions

In scenarios 2 and 3, transactions whose data resides entirely on one node can use the mechanisms of scenario 1. We therefore focus on the general case, in which a transaction spans multiple nodes and consequently requires distributed coordination.

The messaging middleware does not need to provide any kind of ordering or guarantees (e.g., FIFO, causal delivery). TCC is enforced entirely through the replication agents, the database, and the interactions between these two components.

Although Cure relies on CRDTs to reconcile concurrent updates and avoid lost updates, CRDT support is not intrinsic to TCC itself, whose core requirements are atomic visibility and causally consistent snapshots. We have focused on these requirements. Different CRDT-in-SQL layers [16, 39] may require additional operation-level or object-level metadata or none at all. In any case, there is no one-size-fits-all solution. Assuming a single general approach (e.g., providing a commit VV per transaction) would be inefficient in many cases. We argue that the choice of metadata should be able to be configured at a fine grain, per table. But designing a catalog of choices for metadata is beyond the scope of this paper. Nonetheless, our current prototype (available as an artifact) uses Conflict-free Replicated Data Views (CRDVs) [16], based on metadata in the form of a VV, like Cure.

4 ANALYSIS

4.1 Visibility and Snapshots

This section focuses on the two core challenges underlying TCC, previously introduced in Section 2.2: ensuring atomic visibility and assigning each transaction a causally consistent snapshot. We examine to what extent these properties can be supported by mechanisms already available in relational database systems (those described in Section 2.1 and Section 3.2). The results are summarized in Table 1.

4.1.1 Scenario 1. In this scenario, we approach the visibility challenge by having the agent apply the updates of each remote transaction, or transaction batch, as an atomic transaction.

Atomic batch application alone is insufficient under PostgreSQL's default isolation level (RC), since RC refreshes the reader's snapshot at every statement, as discussed in Section 2.1. Client transactions must therefore execute under SI, which solves the problem by fixing the snapshot when the transaction starts, rendering any batch inserted by the agent after that point invisible.

However, SI can only be directly reused for this purpose if the state installed in PostgreSQL is always closed under causal dependencies: whenever a transaction is present in the database, every transaction that causally precedes it must also be present. In fact, SI preserves a snapshot of the local commit order, but it does not itself determine whether that order respects causal dependencies established across DCs.

The agent can easily ensure this through causal replication, performed at the granularity of batches. Upon receiving a batch `B` from origin `d`, the agent verifies that its dependency vector satisfies the relation `Deps(B)` $\leq$ `AppliedVV` and that `B` is the next unapplied batch from `d`. If either condition is not satisfied, the batch remains buffered. Otherwise, the agent applies the entire batch and updates `AppliedVV` within the same PostgreSQL transaction. In particular, the origin component is advanced to `Deps(B)[d] + 1`, while the remaining components are left unchanged.

> **Takeaway 1.** Given a single replication agent and node per DC, applying each batch as a transaction trivially provides atomic visibility, while agent-enforced causal replication ensures that the installed state is causally closed. PostgreSQL's native SI can therefore directly expose causally consistent snapshots to clients.

4.1.2 Scenario 2. In this scenario, replicated transactions may span multiple database nodes. Consequently, the replication layer can no longer rely on the transactional guarantees of a single database instance to provide atomic visibility and causally consistent snapshots. Instead, these properties depend on the guarantees offered by the underlying distributed database.

Databases providing DSI or stronger guarantees (e.g., CockroachDB, YugabyteDB) expose distributed transactions as a single atomic unit while providing a stable distributed snapshot. The agent can therefore apply causal replication as in scenario 1, with the underlying database ensuring both atomic visibility and causally consistent snapshots across all participating nodes.

Citus requires a different mechanism. Although its underlying 2PC protocol [12] guarantees atomic commitment [45], it does not provide a single visibility point or a common snapshot across all participants. Causal replication alone is therefore insufficient: installing a batch only after its dependencies does not prevent its fragments from becoming visible at different moments across nodes.

Instead, as a workaround, visibility must be controlled logically by the agent, within the coordinator. The agent can reuse its current `AppliedVV` as the logical snapshot exposed to application transactions. Application queries must then be transformed, as in TiQuE [17], with an additional `WHERE` clause which ensures that an item

Table 1: Summary of requirements for achieving TCC across three scenarios involving D data centers, covering the database isolation level, inter-agent replication ordering, need for a visibility layer, and the timestamp format (scalar or D-sized vector) used for transaction and for agent metadata.

						Timestamps	
Scenario	**Nodes/DC**	**Agents/DC**	**Isolation**	**Replication**	**Visibility Layer**	Transaction	Agent
S1	1	1	SI	causal	no	none	vector
S2	N	1	DSI	causal	no	none	vector
			row-level	causal	yes	scalar	vector
S3	N	N	row-level	FIFO	yes	vector	vector

written by some T with `Dot(T) = (d, k)` is made visible only if `k` ≤ `AppliedVV[d]`.

Consequently, fragments that have already been physically committed by some 2PC participants remain logically invisible while the transaction is incomplete. Since a dot records only the transaction's origin and scalar commit position, causal replication is still required to ensure that advancing `AppliedVV` cannot expose a transaction before its causal dependencies. Local transactions are made visible immediately, as their causal dependencies are already satisfied: otherwise, read-your-writes (RYW) could be compromised [42]. Importantly, because visibility is enforced through query-time filtering, the database only needs to provide row-level atomicity and nothing else.

PostgreSQL already provides explicit snapshot selection through `SET TRANSACTION SNAPSHOT`. Unfortunately, in Citus, this snapshot is established only for the coordinator's local PostgreSQL transaction and is not propagated to the worker transactions, which therefore continue to determine their snapshots independently.

> **Takeaway 2.** DSI, as available in CockroachDB and YugabyteDB, makes it straightforward to use multiple nodes per DC with a single replication agent. However, DSI would not be necessary if individual nodes in a distributed setup such as Citus allowed a coordinator to explicitly set the snapshot for each node when initiating distributed transactions, thus exposing causally consistent snapshots to clients.

4.1.3 Scenario 3. In this scenario, a single logical transaction may be partitioned across multiple agents during replication, each of which installs its batch fragment through an independent physical transaction. Since the underlying database has no knowledge that these transactions belong to the same logical update, it may expose some fragments before others, violating the atomic visibility required by TCC. This happens even when the underlying database provides DSI, since DSI only guarantees the atomic visibility of each individual physical transaction.

Similarly to scenario 2, atomic visibility must be enforced through a logical visibility layer. The main difference from scenario 2 is that, given N independent replication agents which replicate batches independently, without coordination, it is not possible to ensure that causal dependencies of a batch have reached the DC, possibly at other nodes, before applying a received batch at some node: only per-shard dependencies can be ensured, but not DC-wide causal dependencies. Moreover, attempting to achieve it through intra-DC agent coordination is hindered by the database acting as a hidden channel from the point of view of tracking dependencies by the agents. To decide whether an item is visible, a commit vector tracking dependencies must be stored for the transaction that wrote it, as opposed to only the dot as in scenario 2. Therefore, all fragments of the same replicated transaction carry a common transaction identifier and commit vector.

As in scenario 1, each agent `a` maintains an $\texttt{AppliedVV}_a$, whose components represent the latest contiguous prefixes durably installed at its node. An intra-DC GSS, as in Cure, can then be computed as the component-wise minimum of these vectors. The GSS represents the largest causally consistent prefix that is known to be installed on every node within the DC. As in scenario 2, visibility needs to be controlled with additional `WHERE` clauses inserted in client requests, using this GSS as the logical snapshot. A replicated transaction becomes visible only when its commit vector is covered by the GSS, ensuring readers never observe partial updates. With this strategy, per-shard causal replication ceases to be needed and FIFO inter-DC batch replication is enough. Interestingly, batches no longer need to carry `Deps(B)`, since causal dependencies are encoded in the transaction-level metadata and enforced by the visibility layer rather than through causal batch insertion.

In this case, given how CockroachDB and YugabyteDB implement visibility with a scalar clock, even with explicit interfaces to manipulate it, these mechanisms would not directly map to the requirements for implementing TCC. Interestingly, it seems easier to achieve that with the architecture of Citus, which would allow separate snapshots for each node, each coordinated with the corresponding agent.

> **Takeaway 3.** With multiple replication agents, remote transactions may be materialized as independent physical transactions across nodes, so even native DSI cannot ensure their atomic visibility. Enforcing TCC therefore requires a logical visibility layer that explicitly adds (vectorial) transactional information to data items and that inserts `WHERE` clauses to expose causally consistent snapshots to clients.

4.2 Commit Assignment

In scenarios with a single replication agent and no visibility layer, where causal replication is enforced at the granularity of replication

batches, the batch number can be assigned by the agent itself whenever the batch is finalized. For this reason, in these scenarios, the CDC mechanism must expose transactions directly in commit order. The more challenging problem, given the abstractions exposed by current database systems, is assigning the local component of each transaction's commit timestamp (i.e., its dot) when transaction-level metadata is required.

When a visibility layer is present, a transaction's commit timestamp must be available at, or be atomically associated with, its effective commit point. Otherwise, a locally committed transaction may be excluded by the visibility predicate, even though it has no unsatisfied causal dependencies, thereby violating RYW.

With a single agent, transaction timestamps may still be assigned centrally by the agent. With multiple agents, however, no single agent can assign the transaction commit timestamp, which must be the same for all nodes involved in the transaction. Moreover, the order in which changes are captured by the CDC mechanism does not necessarily provide such a commit timestamp: in a distributed setting, changes may be ordered only within each publisher, without defining a common commit order across all the nodes. Commit timestamps can instead be assigned, for instance, through an adaptation of Clock-SI [15] for logical clocks, or by delegating the ordering decision to the underlying database when it exposes suitable mechanisms: YugabyteDB provides a relevant mechanism through its Virtual WAL [47], which combines changes from multiple nodes and emits them in commit-time order.

> **Takeaway 4.** Batch numbers can be straightforwardly assigned by the replication agent. On the other hand, a commit timestamp can be borrowed from the database only when it exposes a commit-ordering mechanism (e.g., YugabyteDB's Virtual WAL); otherwise, assignment forces coordination across agents. In both cases, because timestamps are assigned only after commit, expressing visibility through them alone breaks RYW. A database abstraction that assigns a reusable commit-time timestamp and atomically exposes it for visibility would close both gaps at once.

5 RELATED WORK

A related line of work explores how to externalize transaction management from storage engines, either to support heterogeneous data stores or to decouple concurrency control from data management. Foundational systems such as Deuteronomy [22] and Cherry Garcia [14] established the idea of separating transaction management from storage. Deuteronomy proposed an explicit decomposition between transactional and storage components, while Cherry Garcia demonstrated how transactional guarantees could be provided across heterogeneous stores using only a small set of common storage primitives. Building on this category of approaches, and extending the Cherry Garcia protocol specifically, ScalarDB [40, 46] provides Strict Serializability across heterogeneous databases through a database-agnostic transaction layer that maintains transactional metadata alongside application data, while introducing abstractions such as atomicity units to exploit native database guarantees and reduce metadata overhead. Polypheny-DB [44] explores composability from a broader systems perspective, combining heterogeneous storage engines behind a unified data-management layer and treating consistency as a configurable concern. Together, these systems demonstrate that transactional functionality can be separated from storage and reused across diverse databases. Our objective is different: rather than providing transactions across heterogeneous systems, we investigate how to realize TCC in a geo-replicated setting by leveraging native transactional mechanisms already available in relational databases, and by identifying the minimal replication-layer functionality and database interfaces required to compose them with TCC.

6 CONCLUSIONS AND FUTURE WORK

This paper explored whether TCC can be realized by composing existing relational database mechanisms with a comparatively small replication layer. The analysis shows that this is feasible in simple deployments: with a single agent and node per DC, PostgreSQL's native SI and causal replication can ensure atomic visibility and provide causally consistent snapshots.

The main difficulties arise when the database guarantees do not align with the logical transactions managed by the replication protocol. Native isolation may provide stable snapshots and atomic visibility for individual physical transactions, yet offer no way to coordinate visibility across independently installed fragments or according to an externally defined causal snapshot. In such cases, TCC requires additional metadata and a logical visibility layer, often duplicating version-management functionality already present in the database (e.g., PostgreSQL's MVCC). The central obstacle to composable transactional isolation is therefore not the absence of transactional mechanisms, but the lack of interfaces through which external components can control and combine them.

Given that this work has established what the requirements are on the abstractions needed for composable transactional isolation, our ongoing work investigates the diversity of current implementations and how they can be packaged in interfaces for explicitly controlling the snapshot and for obtaining a durable commit position at the transaction's effective commit point. Such abstractions will eliminate the need to reconstruct visibility through SQL and directly preserve session guarantees such as RYW. A natural next step is to prototype these interfaces in the aforementioned systems, compare them with explicit visibility mechanisms [17], and use them to build and evaluate an end-to-end TCC implementation against specialized state-of-the-art systems.

ACKNOWLEDGMENTS

This work is co-funded by the European Regional Development Fund (ERDF): D. Pereira through the Innovation and Digital Transition program (COMPETE 2030) under Portugal 2030, within project CDMS, reference 17409 (COMPETE2030-FEDER-01193000); R. Lopes through the NORTE 2030 Regional Programme under Portugal 2030, within project Rescueware, reference 21746 (NORTE2030-FEDER-02306900); F. Pereira through the NORTE 2030 Regional Programme under Portugal 2030, within project BCD.S+M, reference 14436 (NORTE2030-FEDER-00584600). This work is co-funded by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia: P. Almeida with support from 10.54499/UID/50014/2025; N. Faria and J. Pereira through the CMU-Portugal project ADAPQO 10.54499/2024.12585.CMU.

REFERENCES

[1] Atul Adya. 1999. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. (1999).

[2] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 405–414. https://doi.org/10.1109/ICDCS.2016.98

[3] Paulo Sérgio Almeida. 2024. Approaches to conflict-free Replicated Data Types. *ACM Comput. Surv.* (Sept. 2024). https://doi.org/10.1145/3695249 arXiv:2310.18220 [cs.DC]

[4] AntidoteDB. 2019. AQL: A SQL-like Interface for the AntidoteDB Data Store. https://github.com/AntidoteDB/antidote_aql.

[5] AntidoteDB Project. 2017. AntidoteDB. https://github.com/AntidoteDB/antidote.

[6] Apache Software Foundation. 2026. *Apache Arrow Overview*. Apache Software Foundation. https://arrow.apache.org/overview Accessed: 2026-06-26.

[7] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2013. Highly available transactions: Virtues and limitations. *Proceedings of the VLDB Endowment* 7, 3 (2013), 181–192.

[8] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3 (July 2016), 1–45. https://doi.org/10.1145/2909870

[9] Edmon Begoli, Jesús Camacho Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data*. Association for Computing Machinery. https://doi.org/10.1145/3183713.3190662

[10] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[11] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory (CONCUR 2015)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 58–71.

[12] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed postgresql for data-intensive applications. In *Proceedings of the 2021 International Conference on Management of Data*. 2490–2502.

[13] Debezium Project. 2026. *Debezium 3.5 Documentation*. Debezium. https://debezium.io/documentation/reference/stable

[14] Akon Samir Dey. 2015. Cherry Garcia: Transactions across Heterogeneous Data Stores. (2015).

[15] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. 173–184. https://doi.org/10.1109/SRDS.2013.26

[16] N. Faria and J. Pereira. 2025. CRDV: Conflict-free Replicated Data Views. *Proc. ACM Manag. Data (SIGMOD)* 3, 1 (2025). https://doi.org/10.1145/3709675

[17] N. Faria, J. Pereira, A. Alonso, R. Vilaça, Y. Koning, and N. Nes. 2023. TiQuE: Improving the Transactional Performance of Analytical Systems for True Hybrid Workloads. *Proc. VLDB Endow.* 16, 9 (May 2023). https://doi.org/10.14778/3598581.3598598

[18] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (June 2002), 51–59. https://doi.org/10.1145/564585.564601

[19] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. 1995. Generalized Search Trees for Database Systems. In *Proceedings of the 21st International Conference on Very Large Data Bases*. Morgan Kaufmann, 562–573.

[20] Joseph M Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a database system. *Foundations and Trends in Databases* 1, 2 (2007), 141–259.

[21] Pieter Hintjens. 2013. *ZeroMQ: messaging for many applications*. O'Reilly Media.

[22] Justin Levandoski, David Lomet, and Kevin Keliang Zhao. 2011. Deuteronomy: Transaction Support for Cloud Data. In *Conference on Innovative Data Systems Research (CIDR)*. CIDR. https://www.microsoft.com/en-us/research/publication/deuteronomy-transaction-support-for-cloud-data

[23] Si Liu, Luca Multazzu, Hengfeng Wei, and David A Basin. 2024. NOC-NOC: Towards Performance-optimal Distributed Transactions. *Proc. ACM Manag. Data* 2, 1 (March 2024), 1–25. https://doi.org/10.1145/3639264

[24] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. 23rd ACM Symposium on Operating Systems Principles*. https://www.cs.cmu.edu/~dga/papers/cops-sosp2011.pdf

[25] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 313–328.

[26] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 453–468.

[27] Meta Platforms, Inc. 2026. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. https://github.com/facebook/rocksdb.

[28] C Mohan, B Lindsay, and R Obermarck. 1986. Transaction management in the R* distributed database management system. *ACM Trans. Database Syst.* 11, 4 (Dec. 1986), 378–396. https://doi.org/10.1145/7239.7266

[29] D Stott Parker, Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J Walker, Evelyn Walton, Johanna M Chow, David Edwards, Stephen Kiser, and Charles Kline. 1983. Detection of mutual inconsistency in distributed systems. *IEEE transactions on Software Engineering* SE-9, 3 (1983), 240–247.

[30] Mosha Pasumansky and Benjamin Wagner. 2022. Assembling a query engine from spare parts. In *First International Workshop on Composable Data Management Systems (with VLDB2022)*. https://assets.website-files.com/5e8a264ceaf4870394477fc7/62f141c0ac1aca02e93ae652_firebolt-vldb-cdms-2022.pdf

[31] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The composable data management system manifesto. *Proceedings VLDB Endowment* 16, 10 (June 2023), 2679–2685. https://doi.org/10.14778/3603581.3603604

[32] Dan RK Ports and Kevin Grittner. 2012. Serializable snapshot isolation in PostgreSQL. *arXiv preprint arXiv:1208.4179* (2012).

[33] PostgreSQL Global Development Group. 2026. PostgreSQL. https://www.postgresql.org.

[34] PostgreSQL Global Development Group. 2026. *PostgreSQL 18 Documentation: LISTEN and NOTIFY*. PostgreSQL Global Development Group. https://www.postgresql.org/docs/18/sql-notify.html

[35] PostgreSQL Global Development Group. 2026. *PostgreSQL 18 Documentation: Logical Replication*. PostgreSQL Global Development Group. https://www.postgresql.org/docs/18/logical-replication.html

[36] PostgreSQL Global Development Group. 2026. *PostgreSQL 18 Documentation: pg_lsn Type*. PostgreSQL Global Development Group. https://www.postgresql.org/docs/18/datatype-pg-lsn.html

[37] PostgreSQL Global Development Group. 2026. *PostgreSQL 18 Documentation: track_commit_timestamp*. PostgreSQL Global Development Group. https://www.postgresql.org/docs/18/runtime-config-replication.html#GUC-TRACK-COMMIT-TIMESTAMP

[38] Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. 2010. Dotted version vectors: Logical clocks for optimistic replication. *arXiv preprint arXiv:1011.5808* (2010).

[39] Supabase. 2026. pg_crdt: Conflict-Free Replicated Data Types for PostgreSQL. https://github.com/supabase/pg_crdt.

[40] Toshihiro Suzuki and Hiroyuki Yamada. 2026. Breaking the barriers of database-agnostic transactions. *arXiv [cs.DB]* (Feb. 2026). https://doi.org/10.48550/arXiv.2602.19440 arXiv:2602.19440 [cs.DB]

[41] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[42] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*. IEEE Computer Society, 140–149.

[43] The Open Group. 1991. *Distributed Transaction Processing: The XA Specification*. Technical Report. The Open Group. https://pubs.opengroup.org/onlinepubs/009680699/toc.pdf

[44] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. 2018. Polypheny-DB: Towards a Distributed and Self-Adaptive Polystore. In *2018 IEEE International Conference on Big Data (Big Data)*. 3364–3373. https://doi.org/10.1109/BigData.2018.8622353

[45] Gottfried Vossen. 2009. ACID Properties. In *Encyclopedia of Database Systems*, Ling Liu and M. Tamer Özsu (Eds.). Springer, 19–21. https://doi.org/10.1007/978-0-387-39940-9_831

[46] Hiroyuki Yamada, Toshihiro Suzuki, Yuji Ito, and Jun Nemoto. 2023. ScalarDB: Universal Transaction Manager for Polystores. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3768–3780. https://doi.org/10.14778/3611540.3611563

[47] YugabyteDB, Inc. 2026. *Architecture for CDC Using the PostgreSQL Protocol*. YugabyteDB, Inc. https://docs.yugabyte.com/stable/architecture/docdb-replication/cdc-logical-replication

[48] YugabyteDB, Inc. 2026. YugabyteDB. https://www.yugabyte.com/yugabytedb.