Design and Modular Verification of Distributed Transactions in MongoDB

William Schultz
MongoDB Research
New York, New York
william.schultz@mongodb.com

Murat Demirbas
MongoDB Research
New York, New York
murat.demirbas@mongodb.com

ABSTRACT

MongoDB's distributed multi-document transactions protocol was designed and developed incrementally, building on WiredTiger, an existing single node multi-version storage engine that provided snapshot isolated key-value storage. This layered approach required meticulous management of concurrency control and timestamping mechanisms across system layers, complicated by intricate component interactions and a large evolving codebase. In this paper, we describe our experience using *modular* formal specification techniques to address this challenge. Our approach formally specifies the distributed transactions protocol and its interface with the underlying storage layer, allowing us to verify high level protocol properties while also formalizing the contract between these two components. This modular approach also enables an automated, model-based verification technique for testing conformance of the WiredTiger storage implementation to this interface. We use an explicit state model checker to automatically generate test cases from our storage model, which are then executed against the storage implementation, ensuring the implementation matches the interface relied upon by the transactions protocol. Our work highlights the value of formal modeling not only for verifying high-level protocol correctness but also for precisely defining and validating interactions with lower-level system components in an automated way. Beyond verifying key isolation properties, our specification also enabled us to formally analyze *permissiveness*–how well the protocol maximizes concurrency within a given isolation level–a property not previously examined.

PVLDB Reference Format:
William Schultz and Murat Demirbas. Design and Modular Verification of Distributed Transactions in MongoDB. PVLDB, 18(12): 5045 - 5058, 2025.
doi:10.14778/3750601.3750626

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/mongodb-labs/vldb25-dist-txns.

1 INTRODUCTION

Distributed databases are now standard in modern cloud storage systems and applications [26, 49, 55, 63]. They distribute data across multiple machines, and ensure high availability, scalability, and durability. To support traditional ACID semantics, these systems provide the ability to run distributed transactions at various isolation levels.

Although transaction isolation has been extensively-studied from both theoretical and practical perspectives [3, 6, 22], transaction implementations still vary in their semantics and in the validity of their correctness guarantees [27, 29, 41]. In distributed databases, these complexities are compounded by concurrency, fault tolerance, and other distributed systems concerns [10, 14, 48, 49]. Thus, it has become increasingly valuable to formally characterize and verify the semantics of these protocols for both designers and users of systems relying on these protocols.

MongoDB is a popular document-oriented distributed database that supports fault tolerance via replication and horizontal scalability by sharding [44]. MongoDB evolved its support for atomic, multi-document transactions over several years. At the single node level, it utilizes the WiredTiger storage engine, which provides multiversion, snapshot-isolated transactional key-value storage. This subsequently served as the foundation for initial development of transactions within a single replica set. Distributed transactions then extended this model across shards, requiring synchronization between multiple replica set shards and storage engine instances. This incremental, modular design made the interaction and management of concurrency control mechanisms between layers of the system complicated but crucial to overall protocol correctness.

In this paper, we describe our experience using formal, modular specification techniques to formalize and verify the design of MongoDB's distributed transaction protocol. We developed a formal specification of the protocol in TLA$^+$ [35] and a framework for validating its isolation guarantees, allowing us to gain confidence in the protocol as well as its interaction with underlying system components. We developed our specification using a *modular* technique, which allowed us to formalize the interactions between the distributed transactions protocol and the storage layer at each shard. A key challenge was managing transaction timestamps and concurrency control across system layers, specifically for the complicated yet critical interactions between the protocol and storage layer. To address this, we defined an independent abstract model of the storage layer that operates in synchronous composition with the overall distributed transactions protocol specification. This provides a precise, formal characterization of the storage layer interface relied upon by the distributed transactions protocol.

Leveraging our formal model of the storage layer, we also applied a model-based verification [54] technique to verify conformance between the storage implementation and our abstract storage layer model. We use the TLC explicit-state model checker [59] to generate a set of test case behaviors that exhaustively cover reachable states

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750626

of our model, verifying conformance of the storage engine implementation to these behaviors of the abstract model. Our model-based testing approach demonstrates the benefits of a compact abstract model in providing diverse, formal characterization of the API without the need for a large corpus of manually written test cases.

Beyond correctness, our transactions protocol specification also enabled us to formally analyze *permissiveness* [23], the extent to which a protocol maximizes concurrency within a given isolation level. This analysis revealed opportunities to improve concurrency in both the protocol design and implementation. More broadly, this illustrates how formal specifications can not only verify correctness, but also guide optimizations to enhance system performance.

Our experience suggests that formal methods are essential for verifying transactional correctness, given the inherent complexity of distributed transactions. Yet they are rarely employed in database system development, where correctness often relies on informal intuition and testing. Our formal specification of MongoDB's transactions protocol clarified the contracts between system components (e.g., concurrency control, storage, metadata management) by explicitly defining their invariants. This not only strengthened confidence in correctness and isolation guarantees but also simplified design, development, and testing.

We make our TLA$^+$ specifications and test-case generation code available at [46]. Our repository also provides a link to a web-based tool [43] for interactive exploration of the behaviors of our TLA$^+$ transactions specifications.

To summarize, we make the following contributions:

- We present the design of MongoDB's distributed, multi-document transactions protocol and its various isolation guarantees.
- We present our modular technique for formally specifying the distributed transactions protocol in TLA$^+$, and present results from model checking its isolation and permissiveness properties.
- We present our technique for using modular specification and model-based test-case generation to automatically verify conformance between the storage layer specification (as relied on by distributed transactions) and implementation.

2 BACKGROUND

MongoDB is a document-oriented database that represents data as JSON-like objects, stored in a binary format called BSON. A MongoDB database consists of *collections*, where a collection is a set of documents, and each document is uniquely identified by a primary *key*. MongoDB utilizes the WiredTiger storage engine [56], which is a transactional key-value store that manages access to local durable storage. While MongoDB provides an expressive query language for searching and manipulating documents in a collection, in this paper we primarily treat documents as single objects accessed through their unique keys.

Figure 1 illustrates the architecture of a distributed MongoDB cluster. To provide fault tolerance, MongoDB employs *replica sets*, which are groups of replicated database nodes running a Raft-like consensus protocol [63]. In each replica set, a single *primary* server

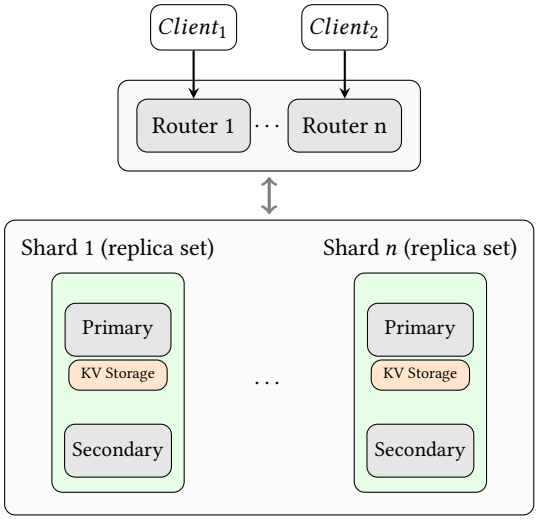

Figure 1: General architecture of a sharded cluster in MongoDB. Data is partitioned horizontally into shards, each of which consists of a replica set, which provides fault tolerance via a Raft-based replication group. Clients can connect to any router, and routers coordinate transaction operations across multiple shards.

handles all write operations and records them in the *oplog*, a sequential log of operations with monotonically increasing timestamps (*optime*). *Secondary* servers replicate the primary's oplog operations and apply them locally to maintain their database state. Secondary servers may optionally process reads from clients, but only primary servers can process writes. Each replica set node interfaces with its local WiredTiger storage engine [56], which provides local snapshot isolated key-value storage.

For horizontal scalability, MongoDB supports *sharding*, which partitions a database across multiple shards, each running as a fault-tolerant replica set. In a sharded cluster, designated *router* processes serve as proxies to handle incoming operations from clients and route these to the appropriate shards.

For all database operations, MongoDB also provides the ability for clients to specify the durability and consistency guarantees of those operations, *read concern* and *write concern*. For standard operations in a replica set, these can determine both durability and recency guarantees of the data read or written, and provide a wide variety of tunable consistency levels [44]. For example, client writes run at *majority* write concern ensure that upon a successful response to the client, these writes will be committed (i.e. durable) in the Raft-based replica set consensus group.

Cluster-Wide Logical Timestamps. MongoDB sharded clusters implement their own variant of hybrid logical clocks [33, 53], which serve as a timestamping mechanism for causally consistent operations. Each replica set shard of a cluster maintains a local, monotonic timestamp, which aligns with the *optime* used by the oplog. This timestamp is advanced on new writes to the replica set, and is propagated along messages between nodes of the cluster, and

updated on receipt according to standard Lamport timestamp conditions [34]. These timestamps are used both for causally consistent client operations as well as for cluster-wide, consistent snapshot reads.

Snapshot Isolation. We reason about our transactions protocol using the formal, client-centric framework of [14], which formalizes the intuitive notions behind snapshot isolation and other levels precisely. At a high level, snapshot isolation requires that all reads from a transaction observe a consistent snapshot of data, and that concurrent transactions that write to conflicting keys are prevented from both committing. Formally, isolation is defined over a set of committed transactions $\mathcal{T}$. For transactions $\mathcal{T}$ to satisfy a given isolation level, there must exist a totally ordered execution e of $\mathcal{T}$ such that each transaction in $\mathcal{T}$ satisfies an associated *commit test* defined for the isolation level. An execution is simply a total ordering of the transactions in $\mathcal{T}$, associated with a set of *read states*, which are the states the database passes through if it were to execute these transactions in the order prescribed by e. In this model, snapshot isolation is defined by the satisfaction of the Complete and NoConflict isolation commit tests. The former states that every read in a transaction T reads from some unique read state s, and the latter states that no concurrent transaction wrote to a conflicting key in the intervening period after this read state s was created and T committed.

3 DISTRIBUTED TRANSACTIONS PROTOCOL

MongoDB implements a distributed, multi-document transaction protocol that provides snapshot isolation for transactions issued by clients in a conversational manner. The protocol utilizes causally consistent timestamps across the cluster for managing ordering and visibility between transactions, and executes a two-phase commit based protocol for ensuring atomic commit across shards. The distributed transactions protocol is layered on top of snapshot-isolated key-value storage nodes that exists at each replica set shard, to correctly implement distributed transactions that are atomic and appropriately isolated across shards.

We begin by providing an overview of the protocol's behavior and how it operates at snapshot isolation in Section 3.1 and 3.2, followed by a discussion of additional protocol optimizations and details in Sections 3.3, 3.4, and 3.5, and then present a discussion of its isolation guarantees in Section 3.6. There we also discuss how the protocol can operate at weaker isolation levels than snapshot, and how this maps to the *read concern* settings of a transaction.

3.1 Overview

The main logical participants of the protocol consist of *client*, *router*, and *shard* roles. Routers, of which there may be several, handle incoming transaction operations issued to them by clients. Routers forward transaction operations to shards in an interactive fashion, and individual shards are responsible for executing transaction operations as they receive them, reporting their responses back to a router. Note that each shard is a replica set, whose leader receives and processes transaction operations locally. If errors or conflicts occur at local shards (e.g. due to a write conflict), this is also reported back to the router, which can initiate a global abort process.

After issuing the last operation of a transaction, if all operations have completed successfully, a client may issue a *commit* operation, prompting the router to initiate a two-phase commit procedure across all participant shards. The router does this by handing off responsibility to a *coordinator* shard, which coordinates the two-phase commit process across all participant shards. Note that this handoff occurs since we do not provide high-availability at the router i.e. high availability is provided at each replica set shard. In some special cases, e.g. for read-only or single shard transactions, a full two-phase commit procedure can be bypassed, allowing the router to send commit operations directly to each shard (discussed further in Section 3.3).

Once it has assumed control from the router, the coordinator shard then sends *prepare* messages to all participant shards that were involved in the transaction, waits for affirmative responses from all shards, and then makes a decision to commit the transaction, sending out a message indicating this to all shards, which can then individually commit the transaction on that shard. Two-phase commit messages are then exchanged between coordinator and participant shards to drive the transaction to commit.

3.2 Timestamp Semantics

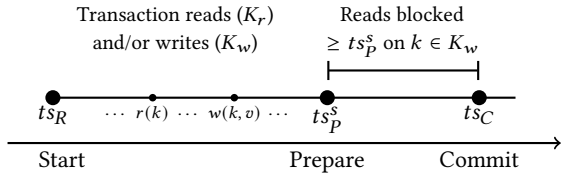

Figure 2: Timeline showing the relationship between transaction timestamps on a shard s. The read timestamp ts_R is chosen when the transaction begins, followed by operations of the transaction, executing reads on keys K_r and writes to keys K_w. The prepare timestamp ts_P^s is then chosen at each shard s, and finally the commit timestamp ts_C which must be greater than all prepare timestamps.

Timestamps are used in the sharded transaction protocol to manage ordering and visibility of transactions across the cluster i.e. for global transaction *read timestamps*, and for *prepare* and *commit* timestamps in the two-phase commit protocol. As part of the global, cluster-wide timestamp mechanism, each replica set shard s maintains a latest timestamp value, $clusterTs_s$, which is incremented on each write operation to the shard, and propagated between shards, routers, and clients.

When a transaction T_i begins on a router, that router selects a *read timestamp* (ts_R) for the transaction, which determines the set of committed transactions that transaction T_i will observe. Routers may select any read timestamp within a range of current timestamp history, though the implementation selects timestamps more strictly e.g. based on the latest known cluster timestamp. Choosing read timestamps too far in the past increases the likelihood of conflicts or lack of retained history at some shards, increasing the likelihood of abort. The read timestamp ts_R chosen by the router is used for all local transaction reads at that shard.

In the lifetime of transaction T_i, the next chosen timestamp is the *prepare timestamp* (ts_P^s), which is chosen separately at each shard s upon receipt and successful processing of a *prepare* command from the transaction coordinator. The *prepare timestamp* ts_P^s chosen at any shard must be $\geq ts_R$, and will also be chosen $> clusterTs_s$ at the time of receiving the *prepare*, based on the standard logical timestamp maintenance rules. The prepare timestamp affects the visibility semantics of ongoing, concurrent transactions once the transaction has entered the *prepared* state. Namely, any transaction T_j that has started at a read timestamp $\geq ts_P^s$ on shard s and attempts to read a key k that was written by T_i will be blocked until T_i has committed or aborted. This is required since the final commit timestamp for T_i is unknown until commit, determining the final visibility of the transaction.

The *commit timestamp* (ts_C) is the final timestamp chosen in the lifetime of a transaction, and is picked by the coordinator after receiving successful responses from all prepared participant shards. The coordinator aggregates the prepare timestamps ts_P^s from all shards $s \in Shard$, and chooses a commit timestamp ts_C such that

$$ts_C \geq max(\{ts_P^s : s \in Shard\})$$

The commit timestamp is the timestamp at which the transaction becomes visible to other transactions, and it is forwarded to all participant shards upon the coordinator's decision to commit. Each shard individually commits, making the transaction's writes visible at this timestamp.

3.2.1 Timestamping Invariants. For correctness of snapshot isolation, there must exist some global, total order on transactions that defines the states of the system at any point in time, and the states which transactions read from. Transaction commit timestamps serve as the mechanism for defining this ordering of transactions in some timestamp history.

It must also be the case, though, that every transaction reads from some state consistent with a unique point in this timestamp ordered history. So, both *read* and *commit* timestamps must satisfy certain key conditions. To guarantee this, we must ensure that such a transaction reading at a timestamp ts_R observes a state consistent with the timestamp history i.e. it must read a state that reflects the effect of commits earlier than ts_R, and no transactions must commit into the past of ts_R in the (real-time) future. In other words, all earlier commit timestamps for transactions that write to these keys are, in some sense, "resolved" i.e. no future transactions will commit at those timestamps.

In MongoDB's distributed transactions protocol, no centralized timestamp assignment mechanism is used, but the above invariants must be upheld. First, it must be ensured that any assigned commit timestamp is greater than any previous read or commit timestamp used in the past (w.r.t dependent transactions). The *prepare* phase of two-phase commit serves as part of this timestamp assignment protocol. That is, each shard is checked during the prepare phase for a timestamp newer than any read or commit timestamp known about on that shard, which is returned as the *prepare timestamp*. Once this timestamp is computed on different shards and returned, we need to pick a commit timestamp consistent with all of these, which we can do by choosing the maximum of all returned prepare timestamps. In addition to this prepare timestamp computation procedure, for correctness we also must ensure that on each shard, concurrently running transactions do not read at timestamps that may end up behind the commit timestamps of transactions that commit in the (real-time) future. In the distributed protocol, we don't know the commit timestamp that will be used for a transaction until the prepare phase completes. So, each *prepare* operation at a timestamp ts_P can be viewed as a reservation of all timestamps $\geq ts_P$ (w.r.t a given key), since we don't know where in the timestamp history a future commit will be placed, though we do know it will be placed somewhere after the prepare timestamp. Thus, future reads at such timestamps must block until the commit timestamp is known.

3.3 2PC Bypass Optimizations

There are some cases where the full two-phase distributed commit protocol can be bypassed. Specifically for

- Read-only transactions.
- Transactions whose writes target only a single shard.

which we refer to as *non-prepared* transactions. If a transaction is read-only, upon initiating commit, the router will send *commit* messages directly to all shards involved. For a read-write transaction that touches only a single shard, the router can similarly send *commit* messages directly to that shard. For a transaction that touches multiple shards but only writes to a single shard, the router first sends *commit* messages to all read-only shards, and then forwards the *commit* message to the single shard that executed the write.

The selection and assignment of timestamps for non-prepared transactions also differs from transactions that go through the *prepare* phase. Specifically, their read timestamps are chosen in the same manner as for prepared transactions, but their commit timestamps are not chosen by the coordinator, and instead are assigned locally by the shard at the time of commit on that shard. Non-prepared transactions write to at most a single shard, so there is no need to coordinate selection of a commit timestamp based on aggregation multiple prepare timestamps. The only requirement for the chosen commit timestamp ts_C at a local shard s is that it is selected as $> clusterTs_s$ at the time of commit, and that $clusterTs_s$ is advanced on that shard to something $\geq ts_C$, so that future transactions are ordered correctly with respect to this committed transaction.

3.4 Speculative Majority and Durability

When transactions begin at a replica set shard in MongoDB, they are always opened at a "local" timestamped snapshot on the node, meaning that the timestamp at which the transaction begins may not represent a majority committed (i.e. durable) timestamp within the replica set for that shard. This is tolerable since, at commit, in order for a transaction to commit successfully, it will require the majority commit of the transaction's read timestamp before returning to the client indicating that a specified isolation guarantee can be provided. We refer to this as *speculative majority* behavior of transactions, as discussed in [44], and is similar to other pipelining optimizations in distributed transactions [49]. This helps prevent potential extra aborts from concurrent transactions that would otherwise need to all start at an earlier, majority committed timestamp that may cause more conflicts based on the read and commit timestamps of these transactions.

Related, throughout the execution of a transaction, the protocol does not require a majority commit of any writes performed at each replica set shard. These writes are not necessarily persisted or replicated on any shard replicas until a transaction is *prepared*, at which time a *prepare* oplog entry is written to the oplog of the primary at each shard, which contains a sequence of all updates performed by the transaction. In order for a *prepare* to complete successfully at a shard, this log entry must be made durable in the shard replica set, so it must be committed at *w:majority*. This is the first inter-replica round trip cost that is forced during the lifecycle of the transaction. Note also that once a transaction has been prepared and majority committed, it is guaranteed to be able to commit.

If a transaction is prepared on some nodes in a replica set but did not complete successfully, the prepare will necessarily fail on that shard, and can either be retried, or that prepared transaction will be cleaned up during recovery process on a new primary.

3.5 Router Recovery and Failover

Throughout their execution, transactions will execute via a single router process, which forward the transaction operations to the appropriate shards. Routers are stateless, however, so if the router handling client operations for a transaction fails, the transaction may be left in-progress on some shards. To account for this case, a recovery procedure is implemented by associating a *recovery shard* with a transaction, which is chosen by the router upon initiation of the transaction and this recovery information is propagated back to clients in transaction operation responses. If the original router has failed or become unreachable, a client can attempt to learn of the commit status of the transaction by sending along a recovery shard token to a new router. This router will then forward a message to the recovery shard in an attempt to join an ongoing commit process by the coordinator, if one is still ongoing.

Recall also that each shard operates as a Raft-based replica set. If the leader crashes or loses its primary status (say due to a network partition or timeout), the replica set will elect a new leader. If this occurs before the transaction is prepared on a shard, then it will be aborted on that shard. Since no updates have been committed, the abort is straightforward and inexpensive. Given that primary failures are rare, occasional transaction aborts due to leader failure are an acceptable cost.

If the leader of a shard fails after sending an affirmative prepare response to the coordinator, that shard's newly elected leader must recover the state of the prepared transaction upon assuming its role as leader, and it may also need to determine the transaction's outcome. It contacts the coordinator shard to learn whether the transaction was committed or aborted and finalizes it accordingly. If the failure occurs at the coordinator shard, the new leader, since the coordinating leader makes prepare phase outcome durable through Raft replication, the overtaking leader can recover the outcome and conclude the transaction.

3.6 Isolation Guarantees

The strongest isolation level provided by the transactions protocol in MongoDB is *snapshot isolation* [9]. Transactions can also be run at a lower isolation level, which provides a variant of *read committed* semantics. These two isolation guarantees are mapped to the transaction *read* and *write concern* settings of a transaction as follows:

- $\{rc : snapshot, w : majority\} \mapsto$ Snapshot Isolation.
- $\{rc : majority, w : majority\} \mapsto$ Read Committed.

In general, for transactions that commit at $w : majority$, isolation is ultimately determined by the behavior on the *read concern* side. That is, on the read side, there are two semantically distinct cases, each determined by whether or not a transaction reads from a globally consistent timestamp across shards.

For *snapshot* read concern, this is the case, and the global transaction timestamp used is chosen by the router. At this read concern, snapshot isolation is then enforce by the combination of local write-write conflict checking locally at each shard, and ensuring that reads block appropriately against prepared writes, to ensure atomic visibility of writes across shards. For weaker read concerns (i.e. $majority$ or $local$ read concern), no uniform read timestamp is used across shards, so the snapshots opened at each shard may not be opened at a consistent global snapshot. For transactions of this type, there is no globally consistent timestamp chosen by the router at the beginning of the transaction. Instead, each shard, when receiving the first operation for a transaction, chooses a local snapshot timestamp based on its local state, meaning that there may be no consistent read timestamp across the cluster for such a transaction. Note that since each transaction executes against a snapshot-isolated store at each shard, the protocol provides some isolation guarantees but weaker than full snapshot isolation. Currently, this is considered as a variant of a *read committed* isolation guarantee, though, in practice, the isolation guarantee may be somewhat stronger than this. We discuss this isolation guarantee in some additional detail in Section 4.5.

4 MODULAR SPECIFICATION

To formalize and verify the properties and guarantees of the distributed transactions protocol, we developed a formal specification of its behavior using a *modular* specification technique. This approach allowed us to check the high level isolation properties of the protocol while also formalizing the boundary between the distributed transactions protocol and the underlying storage engine layer at each shard. Defining an independent abstract model of the storage layer also enabled us to separately apply model-based conformance checking, which we discuss further in Section 5.

Our approach is unique in that it models distributed transactions at a level close to the algorithm while also incorporating interactions with peripheral components at different system layers. By explicitly specifying the boundaries between modules, our technique supports compositional reasoning about different system layers. All of our specifications and associated code can be found at [46].

4.1 TLA$^+$

Our protocol specification is written in TLA$^+$ [35], a formal specification language for modeling concurrent and distributed systems that has been used widely in industry [12, 37, 45]. In TLA$^+$, a system or protocol is formally described as an abstract, non-deterministic state machine, by defining an *initial state* predicate and a *transition*

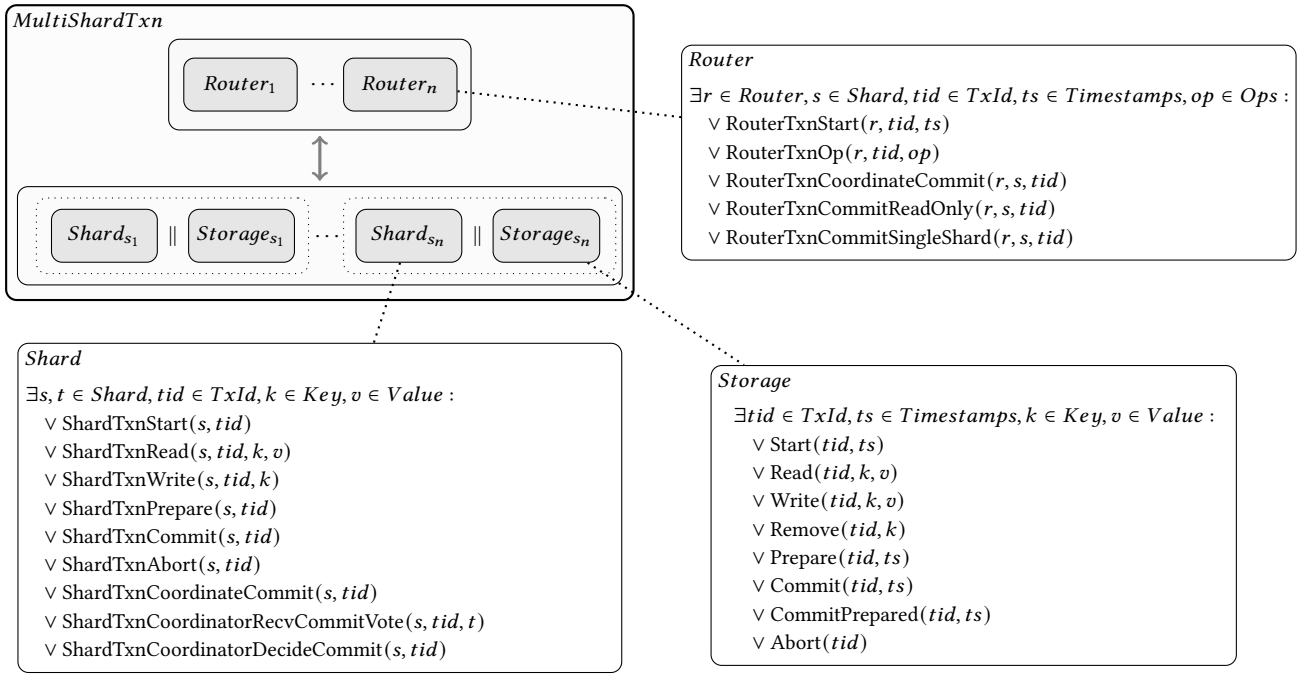

Figure 3: High level overview of our *MultiShardTxn* transactions specification, showing each logical component. Actions of the *Shard* component compose synchronously (indicated by ||) with corresponding, lower level actions of the *Storage* model, while actions of the *Router* and *Shard* interact asynchronously.

relation over a set of *state variables*. The former is a logical predicate that defines the set of possible initial states of the system, and the latter is a logical relation over pairs of system states that defines the set of allowable system state transitions.

Typically, concurrent systems in TLA$^+$ are specified as a logical disjunction of protocol *actions*, each of which describes a set of possible system transitions. An action is then expressed a conjunction of a *guard* and a *update* predicate, which describe, respectively, the logical precondition for an action to be taken and the modification to the system state that occurs as a result of the action. For example, for a state variable x, a simple action such as $(x > 0) \wedge (x' = x + 1)$ expresses an action that increments x by 1 in any state where x is positive (x' denotes the value of x in the state after a transition is taken). A *behavior* of a system is any sequence of states starting from an initial state and where every transition in that behavior satisfies the transition relation. Thus, a system can be semantically viewed as its set of reachable behaviors.

4.2 Specification Overview

Our TLA$^+$ specification models the high-level distributed transactions protocol while also modeling its interface to the underlying storage layer of each shard in a MongoDB cluster. The overall specification, which we refer to as *MultiShardTxn*, consists of approximately 571 lines of TLA$^+$ code, not including the storage engine model (*Storage*), which is discussed in more detail below in Section 4.3. This also does not include the definitions used for the client-centric isolation model, which was originally developed in [47]. An outline of the specification's overall architecture is shown in Figure 3.

The specification defines a set of abstract *Shard* and *Router* identifiers, along with abstract $TxId$, Key, and $Value$ sets, each of which can be instantiated as a finite set of logically unique, opaque identifiers. Also specified is a finite set of $Timestamps$, which defines the set of possible timestamps used by a transaction for reading, preparing, or committing. The protocol specification also defines a global *catalog* mapping $C : Key \rightarrow Shard$, which defines a static function mapping keys to shards. This is used to model the placement of keys in a collection on shards in a sharded cluster, and is used to determine the shard that a transaction will be routed to when it issues an operation on a key. In general, this mapping may change over time (e.g. when chunks of a collection are migrated between shards), but for our model we assumed a fixed catalog mapping to enable initial verification of protocol correctness.

The specification contains a collection of 14 logical actions, 5 of which are associated with actions of a router $r \in Router$, and 9 of which are associated with actions of a shard $s \in Shard$, and which can be seen summarized in Figure 3. There are 25 state variables, 7 of which are dedicated to the *Storage* layer model, and 18 of which are used for the distributed transactions layer of *MultiShardTxn*. The variables of the *MultiShardTxn* specification are logically separated into those maintained at each router process, and those maintained at each shard. Some variables are also used for modeling network communication e.g., a *shardTxnReqs* variable maintains a sequence of incoming transaction operations at each shard, which is used to

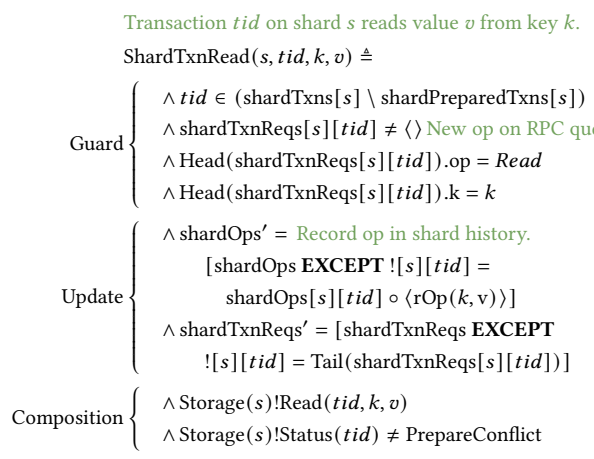

Figure 4: Example of a shard read transaction action, showing its synchronous composition with the underlying storage layer action $Storage(s)!Read$ highlighted. Our model also represent transaction error responses from the API via $Storage(s)!Status(tid)$, which can be checked from the higher level model.

model RPC communication between routers and shards. Each shard maintains details about the status of each ongoing and prepared transaction with *shardTxns* and *shardPreparedTxns* variables, and records the history of transaction operations that have occurred at that shard so far in a *shardOps* array.

Router actions (shown in the *Router* component of Figure 3) are concerned with the starting of a transaction and selection of a read timestamp, and forwarding read or write operations to corresponding shards based on the catalog mapping. Shard actions (shown in the *Shard* component of Figure 3) are related to the processing of operations received from a router, and handling the lifecycle of two-phase commit operations. An example of a shard action for performing a transaction read is shown in Figure 4. As shown there, the action includes a *guard* which defines the logical preconditions for the action to be taken. This includes checking that the transaction is active and not prepared (conjunct 1), and for the existence of a new request on the incoming *shardTxnReqs* queue of the *Read* operation type for key k (conjuncts 2-4). In the action's *update*, it appends a new read operation to the *shardOps* history array and dequeues the incoming operation from *shardTxnReqs*. This is followed by the composition with the underlying *Storage* action, which is explained further in Section 4.3.

Note that we model the storage system at each shard by abstracting it into a single log model, which we discuss further below in Section 4.3. Since the Raft-based replication protocol in a shard is well studied and tested and validated [63], this abstraction helps us keep the state-space size manageable for model checking without loss of interesting behavioral coverage.

4.3 Storage Layer Model

A key aspect of the distributed transactions protocol is its interaction with the underlying storage engine layer at each shard, which is provided by a WiredTiger instance at each MongoDB shard replica. The semantics of this storage engine interface is relied upon by the distributed transactions protocol that runs across shards and is key to the overall correctness of the protocol.

As part of our transactions protocol specification, we formalized this storage layer interface boundary in a way that enables both verification of the high level distributed transactions protocol and use as an independent abstract model of the underlying storage engine behavior, which we examine further in Section 5. The subset of the storage engine API modeled for the distributed transactions protocol is summarized in the *Storage* module in Figure 3, and a logically separate instance of this storage engine module is present on each shard. The full *Storage* specification is approximately 332 lines of TLA$^+$ code, consisting of 8 protocol actions and 7 state variables, related to logging committed transactions (*log*) and tracking data and metadata about actively running transactions (*txnSnapshots*, *txnStatus*, etc.). Generally, the storage layer API relates to the starting, committing, and preparing of transactions at specified timestamps, and reading or writing a given storage engine key.

Developing an abstract reference model of storage API behavior allowed us to capture the semantics of a complex storage implementation in a concise manner e.g., as shown in an action example in Figure 6. This shows the read behavior as modeled in our abstract storage specification, which encodes the essential semantics of a basic, versioned read operation as well as the semantics of prepare conflict blocking as discussed in Section 3.6. In Section 5, we discuss further how we used this model for automated test case generation to verify that the underlying WiredTiger API implementation conformed to the interface represented in our model.

4.3.1 Compositional Semantics. Note that the *Storage* model behaves in synchronous composition with the distributed transactions protocol at each shard. That is, a shard transaction action within the *MultiShardTxn* specification (as illustrated in Figure 3), it occurs atomically on both the shard process ($Shard_{s_i}$) and the underlying storage instance ($Storage_{s_i}$) synchronously, assuming the preconditions for actions on both state machines are met. For example, the *ShardTxnRead* action definition shown in Figure 4 consists of both the higher level sharded transactions details (i.e. its *guard* and *update* conditions), and its composition with the *Read* action of the lower level *Storage* component (shown in more detail in Figure 6).

Note that TLA$^+$ does not directly provide first class mechanisms for representing this type of composition, so we developed our specifications in a way that enabled expressing this type of compositional structure. More concretely, in the formalism of TLA$^+$, one can consider a system or component as defined by a set of variables and a set of logical actions, where different components can be composed together in various ways [2]. For example, composing two components asynchronously corresponds to a natural model of interleaving concurrency, where any action of any component can be taken independently at any time. Composing two components *synchronously* (which matches the semantics of our *Shard* and *Storage* interactions) essentially requires them to behave in "lockstep" i.e. any system action must execute on both components atomically, satisfying the guard and updates of both, independent

```
Has a distinct, prepared transaction written k behind your timestamp.
PrepareConflict(tid, k) ≜
  ∃t_j ∈ TxId :
    ∧ t_j ≠ tid
    ∧ t_j ∈ ActiveTransactions
    ∧ txnSnapshots[t_j].prepared
    ∧ k ∈ txnSnapshots[t_j].writeSet
    ∧ txnSnapshots[t_j].prepareTs ≤ txnSnapshots[tid].ts
```

Figure 5: Example of prepare conflict semantics modeled in our abstract *Storage* model. Used to determine if a read operation on a key must block against an ongoing prepared transaction that has written to the same key.

```
Transaction tid reads key k.
Read(tid, k, v) ≜
 ∧ tid ∈ ActiveTransactions \ PreparedTransactions
 ∧ ¬txnSnapshots[tid]["aborted"]
 ∧ v = TxnRead(tid, k)
 ∧ ∨ ∧ ¬PrepareConflict(tid, k) Successful read.
     ∧ v ≠ NoValue
     ∧ txnStatus' = [txnStatus EXCEPT ![tid] = Ok])
   ∨ ∧ ¬PrepareConflict(tid, k) Key not found.
     ∧ v = NoValue
     ∧ txnStatus' = [txnStatus EXCEPT ![tid] = NotFound]
   ∨ ∧ PrepareConflict(tid, k) Read blocked on a prepared transaction.
     ∧ txnStatus' = [txnStatus EXCEPT ![tid] = PrepareConflict]
```

Figure 6: Example of an action definition of a read operation on the our abstract storage interface model *Storage* specified in TLA$^+$. The *TxnRead* operator (omitted) performs a snapshot read on the underlying key-value data, and the *PrepareConflict* operator checks if the read is blocked against a prepared transaction, as shown in Figure 5.

components. Modeling systems formally in this compositional manner enables modular reasoning about their independent behavior, and a way formalize the interface contracts between components.

4.4 Checking Isolation

We verify the transaction isolation properties of our *MultiShardTxn* protocol specification using a formalization of the client-centric, state-based isolation model outlined in [14, 47]. The formalization of client-centric isolation in our model defines isolation correctness over a set of committed transactions $\mathcal{T}$. For a given set of committed transactions, satisfaction of an isolation level corresponds to the existence of an execution (i.e. a total order of those transactions), that satisfies a specified *commit test* for each transaction.

We verify these isolation guarantees using the TLC model checker [59], for finite instantiations of the constant parameters of the model. The results of our verification efforts can be seen in Table 1. This type of finite model checking does not provide a formal proof of protocol correctness, but provides guaranteed coverage for small models, which provide confidence in the general correctness of the protocol.

To verify isolation in this model, a history of transaction operations is maintained at each shard, which is added to a global history of committed operations (*ops*) for a transaction once the transaction commits at that shard. This global *ops* structure for checking isolation is simply a map from *TxId* to sequences of transaction read or write operations and the operation result (e.g. for reads). We check varying configurations of the model, verifying both *snapshot isolation* and weaker, *read committed* isolation guarantees. In our models, the constant parameters *Key*, *Shard*, and *TxId* are all instantiated as finite sets, and are declared as *symmetric* [11]. In addition, we also impose an initial state constraint that restricts the catalog mapping to avoid placing all keys on a single shard, as well as a limitation on the maximum number of operations performed per transaction, defined as *MaxOps*.

Although we check small models, we still found these to be effective in their ability to capture deep semantic edge cases of the protocol. For example, when removing a key, necessary detail related to prepared transaction reads that was absent in an initial version of the model (discussed further in Section 5.2), the model checker is able to generate a 27-step counterexample leading to a snapshot isolation violation, with just 2 transactions and 2 operations per transaction, in a few minutes.

4.5 Permissiveness

For the models we check, we also measure *permissiveness*, a finer-grained transaction isolation property [23]. Essentially, permissiveness provides one mechanism to quantify the amount of concurrency allowed by a protocol for a fixed isolation level. For example, some protocols may choose to implement weaker isolation levels by providing a stronger isolation level, but this may be unnecessarily restrictive and a traditional, binary notion of isolation property satisfaction cannot capture these distinctions.

We initially explored permissiveness in order to explore the effect of weakening certain storage layer semantics on the higher level isolation guarantees of the distributed transactions protocol. For example, there are configuration parameters at the storage layer related to whether reads must block on prepared transactions (referred to as *IgnorePrepare*), as discussed in Section 3.2. This *prepare blocking* behavior is a requirement for snapshot isolated transactions in our protocol, but we examined the possibility of weakening this requirement to allow for more permissive implementations of weaker isolation levels (e.g. read committed).

To concretely measure permissiveness for a given model, we consider the full reachable set of states $\mathcal{S}$, and from this compute $\mathcal{S}_{ops}$, which is a projection of $\mathcal{S}$ obtained by projecting away all state variables except *ops*, the global history variable of committed transaction operations as discussed above in Section 4.4. The cardinality $|\mathcal{S}_{ops}|$ then serves as our permissiveness metric, by essentially counting the number of unique, allowed committed schedules over any possible reachable behavior of a system. Note that this metric is computed over the full, finite state space of a system, rather than as a property over a single system behavior.

This provides a formal mechanism to compare two protocol designs at a finer-grained level even if they are expected to satisfy the

Model	IgnorePrepare	States	Depth	Time	$\lvert\mathcal{S}_{ops}\rvert$	Invariant
$Shard = \{s_1, s_2\}$ $Key = \{k_1, k_2\}$ $TxId = \{t_1, t_2\}$	True	2,132,765	35	46s	388	ReadCommitted
$Timestamp = \{1 \ldots 3\}$ $MaxOps = 2$ $RC = majority$	False	1,967,907	35	42s	378	ReadCommitted
$Shard = \{s_1, s_2\}$ $Key = \{k_1, k_2\}$ $TxId = \{t_1, t_2\}$ $Timestamp = \{1 \ldots 3\}$ $MaxOps = 2$ $RC = snapshot$	False	8,454,961	35	3m 20s		SnapshotIsolation

Table 1: Verification results for checking isolation guarantees of our distributed transactions formal specification. All results declare the union of *Shard*, *Key*, and *TxIds* as a symmetry set. Permissiveness metric shown as $\lvert\mathcal{S}_{ops}\rvert$.

same isolation level, and is reflected in the results shown in Table 1. There we report, next to each model of shared isolation level, the permissiveness metric $\lvert\mathcal{S}_{ops}\rvert$. For example, of the two models that both satisfy read committed isolation, when weakening the prepare blocking behavior, this model is more permissive (allows more unique schedules in $\lvert\mathcal{S}_{ops}\rvert$), with 388 unique schedules permitted, versus 378 in the model that does not weaken the prepare blocking behavior. If we had only checked isolation for these models, this finer-grained distinction between variants would not be evident.

We plan to explore this in future as a means to further characterize performance and drive potential protocol optimization opportunities. This is also a unique feature of protocol analysis enabled by formalizing and checking a specification of the transactions protocol.

5 MODEL-BASED VERIFICATION

Decomposing the distributed transactions protocol and storage layer in our specification (as discussed in Section 4.3) helped clarify the formal interface between them, but verifying that our storage layer specification accurately reflected the storage *implementation* remained challenging. There are subtle aspects of how the storage engine managed timestamps and concurrency that directly impact the correctness of the transactions protocol, motivating the need for a precise characterization of storage engine API semantics.

To address this, we coupled our storage layer specification more tightly with the WiredTiger storage layer implementation by using an automated, model-based verification approach [54]. We utilized the TLC model checker [59] to drive exploration of our high level model, and from this automatically generate test cases to check conformance of the storage engine implementation. This technique allowed our specification to serve both as a formalization of the interface between components and as a mechanism for automatically checking that the implementation matches this formally defined interface.

5.1 Test Case Generation

Our storage layer specification, as shown in Figure 3 and discussed in Section 4.3, is a sub-component of the overall *MultiShardTxn* distributed transactions protocol specification, and consists of approximately 332 lines of TLA$^+$ code. The goal of our model-based verification approach was to ensure that the behavior exhibited by the underlying WiredTiger storage implementation [56] conforms to the behavior of this abstract specification.

One general approach is to formulate this as a type of refinement or trace validation problem i.e. by checking that every state transition of the low level model corresponds to some valid state

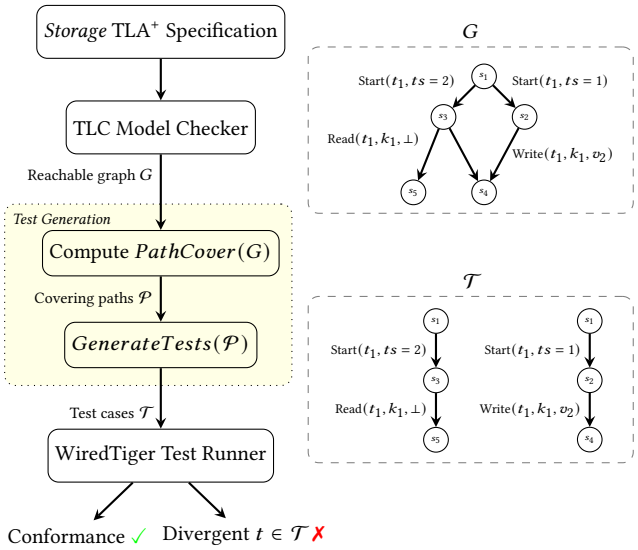

Figure 7: High level summary of test case generation workflow for our storage layer model using the TLC model checker.

transition of the higher level model, possibly under some defined refinement mapping [1, 16, 19]. In our case, we were interested in ensuring that the observable behavior of the storage API was conformant, so we aimed to check something slightly more relaxed. We simply aim to verify that for traces of the model where transitions represent API calls, the observed/returned values match. For example, for all operations, that the return status matches between model and implementation, and any return values match (e.g. for the result of read operations).

Using the TLC explicit state model checker [59] allows us to generate a complete graph of reachable model states for our storage model. TLC can do this efficiently by computing a breadth first exploration of the reachable state space, checking for enabled model actions from any current state. Direct state-based exploration is not possible in general for many program implementations, so, in order to drive implementation testing from a model, we aim to generate some set of traces (executions) for covering reachable states i.e. we use a path-based testing approach using model state coverage to drive test case diversity.

Ideally, given a computed, finite reachable state graph G, we would like to generate a minimal path covering of G which is then converted to a set of test case behaviors. That is, generate a set of

paths $\mathcal{P}$ such that every state in G exists in some path in $\mathcal{P}$, and that $\mathcal{P}$ is minimal in some sense. Computing minimum path coverings of arbitrary directed graphs is in general NP-hard, but for directed acyclic graphs, there exist efficient approaches [38]. For our use case, we implemented an approximate, greedy approach that may be suboptimal but provided adequate performance for our use case.

We also take advantage of symmetry reduction of the state space when possible [11], aided by the native support of this feature in the TLC model checker. Specifically, if we consider the full reachable state graph as G and the graph reduced under symmetry as G_σ, when we select paths to compute a covering, we choose paths from G, but terminate once our set of paths cover all states in G_σ. That is, if we generate a set of paths that cover all states up to symmetry reduction, we consider this as sufficient, which can significantly reduce the number of adequate covering paths needed.

We implement the above functionality in a Python-based tool which makes use of a modified version of the TLC model checker for generating state graphs in the appropriate format. The general workflow of our tool shown in Figure 7, and its implementation code can be found at [46]. TLC model checking is first run for the storage specification and a specified model configuration, generating the reachable state graph G in a structured format. Each edge in this generated graph is annotated with the TLA$^+$ specification action and parameter values of that transition, which are used when converting paths in this graph to concrete test cases. A Python script then processes this generated graph to compute the path covering $\mathcal{P}$ and convert each covering path into a separate unit test case (shown in the *Test Generation* module of Figure 7). Our path covering computation uses a third-party graph processing library [24] to first convert the reachable state graph G to a directed acyclic graph, and then compute paths from the initial state to each node in the graph. We then greedily select paths from this set until we have covered all states in G (or, equivalently, G_σ if we are applying symmetry reduction). Once we have generated each covering path, we convert each edge along the path into its corresponding call on the WiredTiger API (based on the action and parameter values), and generate a unit test code file with appropriate embedded assertions. For example, after each operation, the generated test asserts that the error status for each transaction matches between the model and the implementation, and, for read operations, that the observed value matches the expected value from the model.

5.2 Results and Discussion

The largest model that we checked for conformance and associated metrics are summarized in Table 2. All computation for our experiments is performed on an Amazon EC2 m6g.2xlarge instance with 8 vCPUs and 32 GiB memory, on Ubuntu 22.04. Generation of the underlying state graph is performed by the TLC model checker, using 8 parallel workers, while processing of the state graph to generate test cases is done by a single thread in Python. Execution of the generated tests is then performed against the WiredTiger storage engine using 8 parallel worker threads.

Even for a relatively small model (e.g. 2 transactions), the state space can grow rapidly, and full, path-based coverage can become costly. Generally, this is due to the large number of possible interleavings of transaction operations, with varying parameters

Model	$Keys = \{k1, k2\}$ $TxIds = \{t1, t2\}$ $Timestamp = \{1 \ldots 3\}$
States	490,360
States (symmetry)	132,981
Tests	87,143
Mean Depth	15
State graph generation (TLC)	1m 11s
Test Generation	29m 10s
Test Execution	13m 49s
Conformance	✓

Table 2: Model-based test case generation statistics for largest conformance checking configuration. *Mean Depth* shows the average length over all generated test case behaviors.

(e.g. prepared v.s. non-prepared, different timestamp values, etc.). Symmetry reduction aids in this, but we believe that path-based coverage mechanisms are also somewhat inefficient since many paths may cover redundant states, and only differ by a small number of unique states covered. To exercise increasingly larger model parameters, we believe randomized or approximate sampling of the covering paths is a feasible approach [25, 40]. In general, we can only check conformance between our model and implementation for finite model instances, so these conformance checks do not provide a formal proof of conformance between our model and implementation, but are a strong foundational guarantee.

During initial specification development, we also clarified subtle behaviors of the storage engine semantics that were uncovered by by model-implementation divergences that required subsequent updates to our model. As one notable example, we clarified a particular edge case behavior of transaction semantics involving concurrent, prepared transactions. Specifically, due to the distributed nature of transaction timestamp selection, it is possible for commit timestamps to be chosen behind the read timestamp of an active transaction. Consider the history of events on a shard s that is a participant in concurrent transactions T_1 and T_2 shown in Figure 8. Due to the interleaving of these transactions, it is possible that the commit timestamp of T_1 was chosen before T_2 was started at timestamp 3, but at an earlier timestamp than T_2 reads at. In a non-distributed timestamp assignment mechanism, commits that occur later in real time would be assigned commit timestamps higher than any current read timestamps, but with distributed timestamp assignment this is not always enforced. So, since T_1 commits into the "past" of T_2's history, it must be visible to it, even though it was not present in the snapshot of T_2 when it began in real time. Thus, the read of T_2 of k should return the effect of T_1's write.

In an initial version of our model we assumed it was sufficient for each transaction to store a static snapshot of data which is recorded at the time the transaction starts, and which is used for all of the transaction's reads as it executes. This scenario breaks that assumption, however, since a transaction's snapshot at a given timestamp may, in fact, be updated "after" (in real time) the transaction started. So, this needs to be taken into account in the read behavior for transactions in the model. Clarifying this subtle behavior was valuable since it is a case that does not directly occur in single instance

Transaction T_1	Transaction T_2
$start(ts_R = 0)$	
$write(k, v = T_1)$	
$prepare(ts_P = 1)$	
	$start(ts_R = 3)$
$commit(ts_C = 2)$	
	$read(k)$

Figure 8: Example history of events on a shard s that is a participant in concurrent transactions T_1 and T_2, where T_1 commits in real time at a timestamp behind T_2's read timestamp.

or replica set deployments of MongoDB. This case was unique to the use of prepared, distributed transactions and their mechanism for timestamp selection.

Overall, we found that formalizing a model of the storage API was useful for both verification and conformance testing via test generation, but also for precisely characterizing and understanding the semantics of the storage layer for external consumers of the API, the distributed transactions protocol being the notable example. The set of behaviors that are allowed or disallowed in the API and their semantics is often a source of discussion and misunderstanding in development. Ad hoc unit tests can be written to verify a given behavior or edge case, but these are often not a comprehensive representation of API behavior, and can only be understood in isolation.

We also explored extensions to our storage model that included a wider array of API operations, including details related to setting durability related checkpoint timestamps, rolling back the storage engine to a timestamped checkpoint, and checking other more fine-grained, timestamp related properties. Although they are not directly related to the interface relied upon by distributed transactions, they are valuable aspects of the API to formalize and test further. Similarly, we do not test concurrently executing clients in our model-based testing approach, but we view this as a valuable and feasible extension. For example, we could consider running several, isolated parallel threads that execute traces generated from our model, on different transactions and different keys. Moreover, we could incorporate linearizability checking for concurrent, conflicting workloads using our models as a sequential reference spec [4].

6 RELATED WORK

Distributed Transactions. There has been a wide range of work on building and optimizing transactions in distributed database systems in recent decades. Spanner [13] was one of the earliest modern systems to provide an implementation of distributed, ACID transactions. It provided strict serializability for one-shot transactions, coupling this with the use of tightly synchronized atomic clocks. Percolator [42] and Megastore [5] were also precursors to this work. More recently, there have been several open source variants largely derived from the architecture of Spanner, including CockroachDB [49], Yugabyte [60] and TiDB [26], all which implement various forms of transactions with differing sets of optimizations or modifications. Relying on timestamping for concurrency control, these distributed SQL databases must ensure consistent timestamp management across layers within a single node, including transaction coordination, concurrency control, and storage. If not managed well, transactions can see anomalies or experience unnecessary delays, aborts and retries.

Calvin [52] introduced a *deterministic* approach to scheduling distributed transactions, eliminating the need and cost of two-phase commit by preordering transactions to avoid conflicts at runtime. However, this requires advanced knowledge of transaction read-/write sets, restricting the system to non-conversational (one-shot) transactions. These ideas have been productionized into the FaunaDB distributed database [21]. More recently, Chardonnay [20] builds on similar principles to optimize transaction processing for single datacenter deployments with fast two-phase commit.

An alternative line of research focused on optimizing transactions by refining the underlying consensus protocols within shards, tailoring them to the transaction protocols running on top. Notable examples include MDCC [32], which explores multi-datacenter consistency, ROCOCO [36], which reduces coordination overhead for distributed transactions, and TAPIR [61], which eliminates the need for traditional consensus and ordering in some cases to improve performance.

Isolation. Isolation semantics in distributed databases are subtle and intricate, balancing consistency, concurrency, and performance. MongoDB on a single node provides transactions with strong snapshot isolation, which ensures transactions observe all transactions whose commits precede its start in real time, thanks to WiredTiger. Distributed MongoDB transactions instead satisfy session snapshot isolation, which only requires a transaction to observe all transactions that precede it within its session. As another relaxation, Walter [48] introduced *Parallel Snapshot Isolation (PSI)*, which ensures consistent snapshots and transaction ordering within a site while enforcing causal ordering across sites and preventing write-write conflicts. By allowing different sites to have different commit orderings, PSI relaxes the strict total ordering requirements of strong snapshot isolation, allowing for improved performance in distributed deployments.

There have been a variety of attempts to formally define transaction isolation over many years, going back to an attempt to improve on standard ANSI SQL definitions [3, 7]. Recent attempts try to formulate isolation in a *client-centric* manner, e.g. [14], and also in an abstract framework that takes advantage of the *atomic visibility* property of most widely used isolation levels [10]. The conformance between isolation definitions in these various formalisms and the semantics provided by real systems can still be opaque and difficult to understand, though, with a variety of anomalies discovered in well-known and mature database systems [8, 15, 27, 28].

Also, the notion of *permissiveness* of a transaction system appears to have been previously explored in the context of transactional memory systems [23], but we are not aware of prior work that explicitly explored this issue in the context of distributed database transaction protocols.

Testing and Verification. There have been several tools developed in recent years to aid in automatic and online verification of database isolation and consistency levels. The Elle isolation checker [30] was developed as a part of the Jepsen testing suite, and the Viper snapshot isolation checker [62], and also Cobra [50] also provide new efficient algorithms for black-box isolation checking. Hermitage [31] was also an attempt at providing standard test suites to characterize the isolation levels provided by various database systems. Recent work [39] reviewed the transactional consistency protocols at MongoDB and argued that they satisfy snapshot isolation. In our work, we provide formal TLA$^+$ specification for the transaction protocols, and use model checking and test-case generation to validate the correctness of the protocols.

The domain of model-based testing has also been well explored, applied in various domains and styles. For example, previous efforts have used a similar approach to ours to directly use a model checker to drive test case generation for hardware verification [58]. This ranges from online, dynamic approaches to testing to static test case generation [16, 51]. The P language [17, 18] has also been applied in similar domains, for both modeling systems at a high level and generating executable systems code from these models. The MODIST tool [57] is another example of an attempt at combining model checking concepts with running distributed systems implementations, but implements its approaches directly at the system implementation layer, with no explicit abstract model.

We found our approach unique in its application to the distributed transactions domain, its use of a compositional approach for both high level protocol verification and low level conformance checking.

7 CONCLUSIONS AND FUTURE WORK

In this work, we presented a formal, modular specification of MongoDB's distributed transactions protocol and used it to verify isolation properties. We also leveraged the modular structure of our specification in a model-based verification strategy to ensure conformance between the storage layer model and the underlying storage engine implementation.

Based on our experience, we argue that using formal methods is essential to verify transactional correctness. Distributed transactions are inherently complex due to handling concurrency, version control, and cross-layer interactions. Yet, the database community often builds these protocols without a formal model, relying on informal reasoning and ad hoc testing. We find that a formal specification clarifies the contracts between system components–query processing, concurrency control, storage, and metadata management–by explicitly stating their invariants. This clarity simplifies both design and development and supports automated test-case generation.

Formal modeling helped us gain confidence in the protocol's correctness, but we also see it as a foundation for further exploration. A promising direction is using verification and modeling techniques to assess whether an application behaves correctly under a given isolation level. This could reduce the burden on users, who often struggle with the opaque and intricate relationship between isolation level semantics and application-level correctness.

In future work we aim to develop better formal characterizations of weaker, non-snapshot isolation levels in our system and explore how they can be leveraged for performance and correctness tradeoffs. Additionally, we plan to extend our model to capture interactions with dynamic catalog operations, such as key redistribution across shards concurrent with running transactions.

REFERENCES

[1] Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (May 1991), 253–284. https://doi.org/10.1016/0304-3975(91)90224-P

[2] Martín Abadi and Leslie Lamport. 1995. Conjoining specifications. *ACM Trans. Program. Lang. Syst.* 17, 3 (May 1995), 507–535. https://doi.org/10.1145/203095.201069

[3] A. Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions.* Technical Report. USA.

[4] Anish Athalye. 2017. Porcupine: A fast linearizability checker in Go. https://github.com/anishathalye/porcupine.

[5] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Conference on Innovative Data Systems Research (CIDR)* (Asilomar, California). 223–234.

[6] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA) *(SIGMOD '95)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/223784.223785

[7] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *SIGMOD Rec.* 24, 2 (May 1995), 1–10. https://doi.org/10.1145/568271.223785

[8] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021. MonkeyDB: Effectively Testing Correctness under Weak Isolation Levels. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–27.

[9] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2009. Serializable Isolation for Snapshot Databases. *ACM Trans. Database Syst.* 34, 4, Article 20 (Dec. 2009), 42 pages. https://doi.org/10.1145/1620585.1620587

[10] 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) (Leibniz International Proceedings in Informatics (LIPIcs))*, Luca Aceto and David de Frutos Escrig (Eds.), Vol. 42. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

[11] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. 1998. Symmetry reductions in model checking. In *Computer Aided Verification*, Alan J. Hu and Moshe Y. Vardi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 147–158.

[12] Confluent. 2018. Hardening Kafka Replication. Online. https://www.confluent.io/kafka-summit-sf18/hardening-kafka-replication/ Accessed: 2025-01-14.

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 251–264.

[14] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC '17)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/3087801.3087802

[15] Ziyu Cui, Wensheng Dou, Qianwang Dai, Jiansen Song, Wei Wang, Jun Wei, and Dan Ye. 2022. Differentially testing database transactions for fun and profit. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[16] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. 2020. eXtreme Modelling in Practice. *Proc. VLDB Endow.* 13, 9 (May 2020), 1346–1358. https://doi.org/10.14778/3397230.3397233

[17] Ankush Desai, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. *SIGPLAN Not.* 48, 6 (June 2013), 321–332. https://doi.org/10.1145/2499370.2462184

[18] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional programming and testing of dynamic distributed systems. *Proc.*

ACM Program. Lang. 2, OOPSLA, Article 159 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276529

[19] Star Dorminey. 2020. Kayfabe: Model-based Program Testing with TLC. In *TLA+ Conference*. https://conf.tlapl.us/2020/11-Star_Dorminey-Kayfabe_Model_based_program_testing_with_TLC.pdf

[20] Tamer Eldeeb, Xincheng Xie, Philip A. Bernstein, Asaf Cidon, and Junfeng Yang. 2023. Chardonnay: Fast and General Datacenter Transactions for On-Disk Databases. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 343–360. https://www.usenix.org/conference/osdi23/presentation/eldeeb

[21] Fauna. 2025. Fauna: Distributed Serverless Database. https://fauna.com/ Accessed: 2025-03-11.

[22] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. 2004. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.* 33, 3 (Sept. 2004), 12–14. https://doi.org/10.1145/1031570.1031573

[23] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. 2008. Permissiveness in Transactional Memories. In *Distributed Computing*, Gadi Taubenfeld (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 305–319.

[24] Aric Hagberg, Pieter Swart, and Daniel S Chult. 2008. *Exploring network structure, dynamics, and function using NetworkX*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).

[25] Pei Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula, Robert Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. 2000. Smart simulation using collaborative formal and simulation engines. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-Aided Design* (San Jose, California) *(ICCAD '00)*. IEEE Press, 120–126.

[26] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: a Raft-based HTAP database. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3072–3084. https://doi.org/10.14778/3415478.3415535

[27] Jepsen. 2024. Jepsen: MySQL 8.0.34. https://jepsen.io/analyses/mysql-8.0.34 Accessed: 2024-11-15.

[28] Kyle Kingsbury. 2019. Jepsen: FaunaDB 2.5.4. https://jepsen.io/analyses/faunadb-2.5.4 Accessed: 2024-02-04.

[29] Kyle Kingsbury. 2024. Jepsen 15: What Even Are Transactions? https://www.youtube.com/watch?v=ecZp6cWhDjg Accessed: 2025-02-04.

[30] Kyle Kingsbury and Peter Alvaro. 2020. Elle: inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 268–280. https://doi.org/10.14778/3430915.3430918

[31] Martin Kleppmann. 2014. Hermitage: Testing the I in ACID. https://martin.kleppmann.com/2014/11/25/hermitage-testing-the-i-in-acid.html Accessed: 2025-02-04.

[32] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 113–126. https://doi.org/10.1145/2465351.2465363

[33] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical Physical Clocks. In *Principles of Distributed Systems*, Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro (Eds.). Springer International Publishing, Cham, 17–32.

[34] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[35] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. 2002. Specifying and verifying systems with TLA+. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop* (Saint-Emilion, France) *(EW 10)*. Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/1133373.1133382

[36] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 479–494. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/mu

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

[38] S.C. Ntafos and S.L. Hakimi. 1979. On Path Cover Problems in Digraphs and Applications to Program Testing. *IEEE Transactions on Software Engineering* SE-5, 5 (1979), 520–529. https://doi.org/10.1109/TSE.1979.234213

[39] Hongrong Ouyang, Hengfeng Wei, Yu Huang, Haixiang Li, and Anqun Pan. 2021. Verifying transactional consistency of mongodb. *arXiv preprint arXiv:2111.14946* (2021).

[40] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. 2019. FuzzFactory: domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 174 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360600

[41] Andrew Pavlo. 2017. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 3. https://doi.org/10.1145/3035918.3056096

[42] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 251–264.

[43] William Schultz. 2024. Spectacle: Interactive, web-based tool for exploring, visualizing, and sharing formal specifications in TLA+. https://github.com/will62794/spectacle.

[44] William Schultz, Tess Avitabile, and Alyson Cabral. 2019. Tunable consistency in MongoDB. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2071–2081. https://doi.org/10.14778/3352063.3352125

[45] William Schultz, Ian Dardik, and Stavros Tripakis. 2022. Formal verification of a distributed dynamic reconfiguration protocol. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 143–152. https://doi.org/10.1145/3497775.3503688

[46] William Schultz and Murat Demirbas. 2025. MongoDB distributed transactions protocol specifications. https://github.com/mongodb-labs/vldb25-dist-txns.

[47] Tim Soethout, Tijs van der Storm, and Jurgen J. Vinju. 2021. Automated Validation of State-Based Client-Centric Isolation with TLA +. In *Software Engineering and Formal Methods. SEFM 2020 Collocated Workshops - ASYDE, CIFMA, and CoSim-CPS, 2020, Revised Selected Papers (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics))*, Loek Cleophas and Mieke Massink (Eds.), Vol. Cham. Springer Science and Business Media Deutschland GmbH, 43–57. https://doi.org/10.1007/978-3-030-67220-1_4 Publisher Copyright: © 2021, Springer Nature Switzerland AG.; 2nd International Workshop on Automated and Verifiable Software System Development, ASYDE 2020, 2nd International Workshop on Cognition: Interdisciplinary Foundations, Models and Applications, CIFMA 2020 and 4th International Workshop on Formal Co-Simulation of Cyber-Physical Systems, CoSim-CPS 2020 collocated with the 18th International Conference on Software Engineering and Formal Methods, SEFM 2020 ; Conference date: 14-09-2020 Through 15-09-2020.

[48] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 385–400. https://doi.org/10.1145/2043556.2043592

[49] 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* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[50] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 63–80. https://www.usenix.org/conference/osdi20/presentation/tan

[51] Ruize Tang, Xudong Sun, Yu Huang, Yuyang Wei, Lingzhi Ouyang, and Xiaoxing Ma. 2024. SandTable: Scalable Distributed System Model Checking with Specification-Level State Exploration. In *Proceedings of the Nineteenth European Conference on Computer Systems* (Athens, Greece) *(EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 736–753. https://doi.org/10.1145/3627703.3650077

[52] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[53] Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, and Jack Mulrow. 2019. Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 636–650. https://doi.org/10.1145/3299869.3314049

[54] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability* 22, 5 (2012), 297–312. https://doi.org/10.1002/stvr.456 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.456

[55] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, and Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding

Distributed Consensus for I/Os, Commits, and Membership Changes. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 789–796. https://doi.org/10.1145/3183713.3196937

[56] WiredTiger. 2024. WiredTiger. https://github.com/wiredtiger/wiredtiger

[57] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 09)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/nsdi-09/modist-transparent-model-checking-unmodified-distributed-systems

[58] Yuan Yu. 2002. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *Proceedings of the 3rd IEEE International Workshop on Microprocessor Test and Verification (MTV '02)* (proceedings of the 3rd ieee international workshop on microprocessor test and verification (mtv '02) ed.). Institute of Electrical and Electronics Engineers, Inc. https://www.microsoft.com/en-us/research/publication/using-formal-specifications-to-monitor-and-guide-simulation-verifying-the-cache-coherence-engine-of-the-alpha-21364-microprocessor/

[59] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA+ Specifications. In *Correct Hardware Design and Verification Methods*, Laurence Pierre and Thomas Kropf (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 54–66.

[60] Yugabyte. 2025. YugabyteDB - Distributed SQL Database. https://github.com/yugabyte/yugabyte-db Accessed: 2025-03-13.

[61] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 263–278. https://doi.org/10.1145/2815400.2815404

[62] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 654–671. https://doi.org/10.1145/3552326.3567492

[63] Siyuan Zhou and Shuai Mu. 2021. Fault-Tolerant Replication with Pull-Based Consensus in MongoDB. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 687–703. https://www.usenix.org/conference/nsdi21/presentation/zhou