Modular Verification and Permissiveness of Distributed Transactions

Will Schultz
Staff Research Engineer @ MongoDB

Agenda

- Distributed Transactions in MongoDB
- Modular Formal Specification
- Checking Isolation & Permissiveness
- Model-based Verification

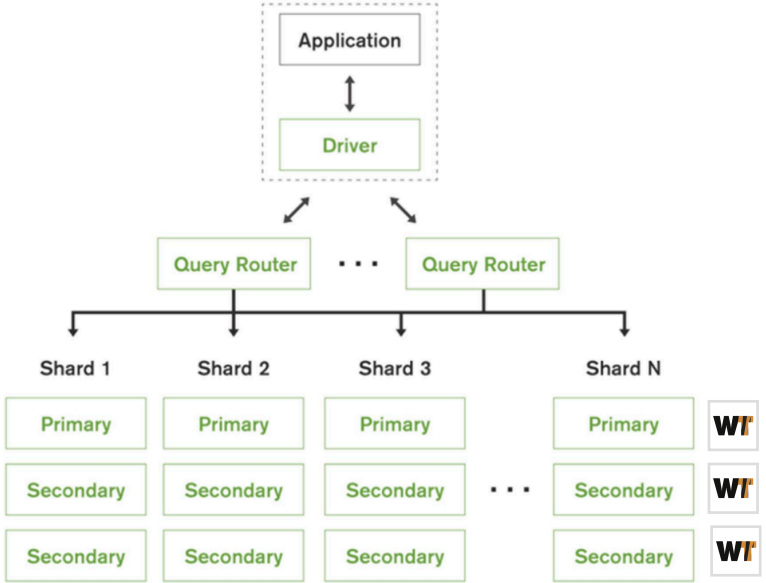
MongoDB Architecture

Database is consists of **collections**, each a set of documents.

Replica sets for fault tolerance.

Sharding for horizontal scalability by logical collection partitioning.

Local WiredTiger instance: snapshot-isolated key-value storage.



Distributed Transactions in MongoDB

Original implementation for sharded clusters in v4.2.

Implements multi-document transactions at a maximum of **snapshot isolation**, using single replica set transaction machinery at each shard.

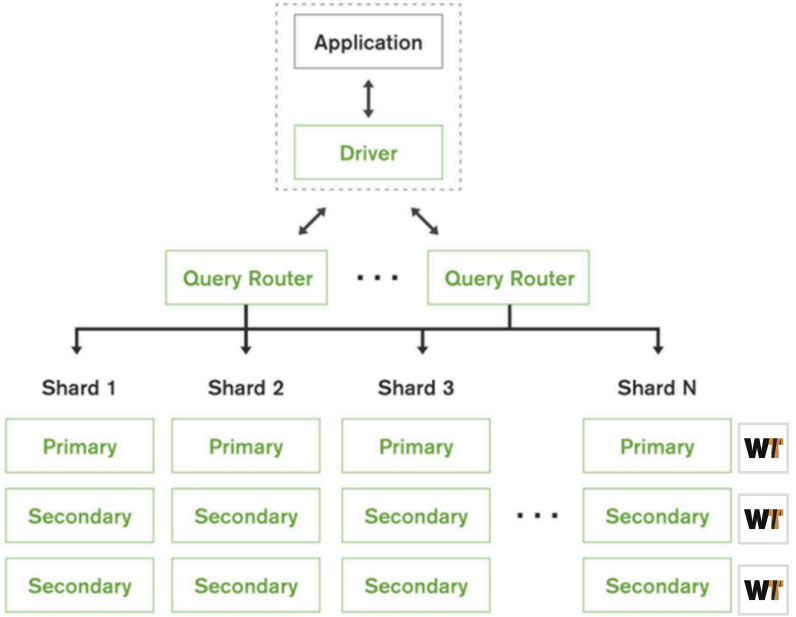
Read concern levels in MongoDB defines durability/consistency guarantees.

- readConcern: “snapshot” (snapshot isolation)
- readConcern: “local” (read committed)

Distributed Transactions in MongoDB

Clients send transaction operations to router, which establishes global **read timestamp**.

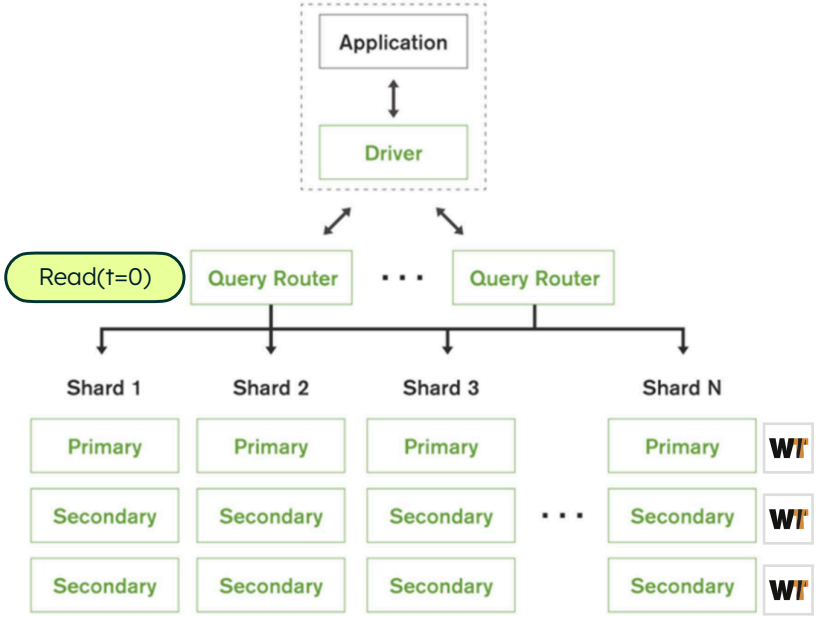
Operations forwarded to shards until commit (or must abort).



Distributed Transactions in MongoDB

Clients send transaction operations to router, which establishes global **read timestamp**.

Operations forwarded to shards until commit (or must abort).



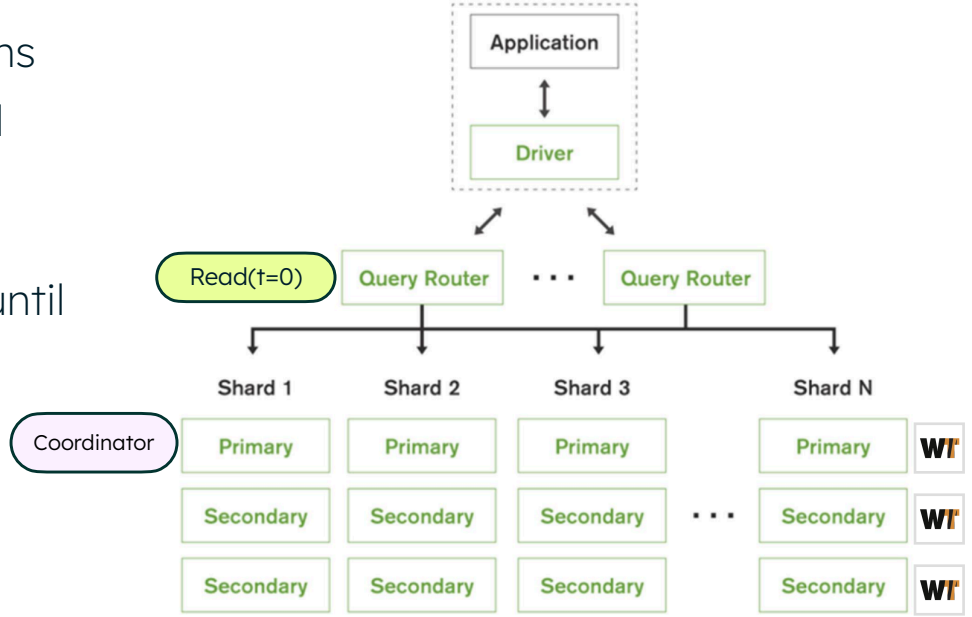
Distributed Transactions in MongoDB

Clients send transaction operations to router, which establishes global **read timestamp**.

Operations forwarded to shards until commit (or must abort).

Coordinator prepares transaction at each shard, receiving prepare timestamp at each.

Prepare timestamps then used using to compute global commit timestamp.



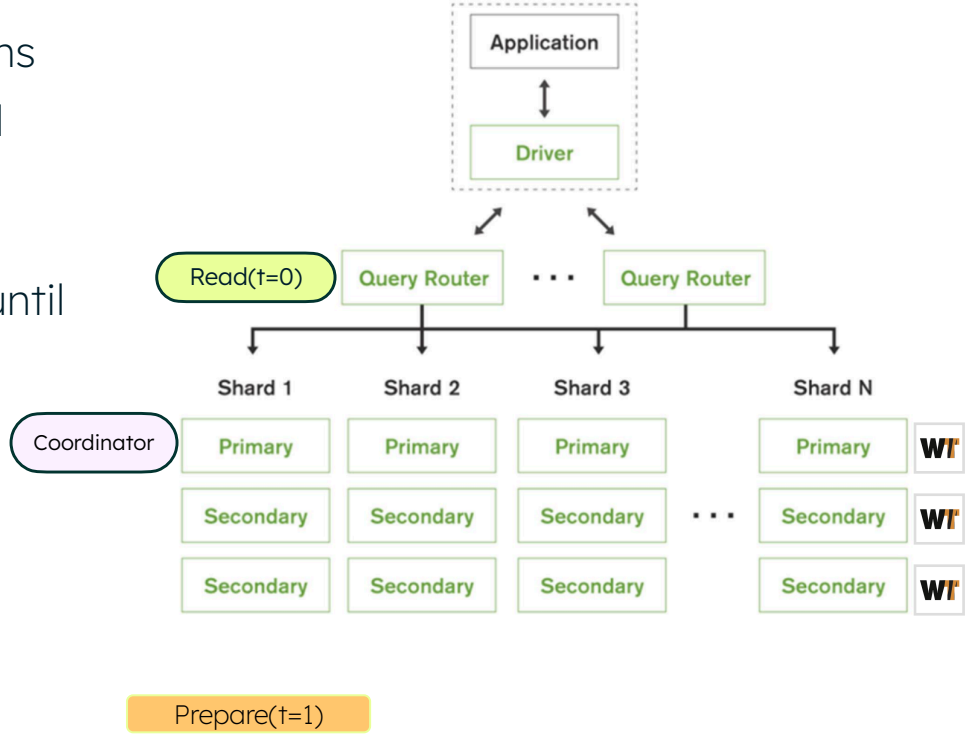
Distributed Transactions in MongoDB

Clients send transaction operations to router, which establishes global **read timestamp**.

Operations forwarded to shards until commit (or must abort).

Coordinator prepares transaction at each shard, receiving prepare timestamp at each.

Prepare timestamps then used using to compute global commit timestamp.



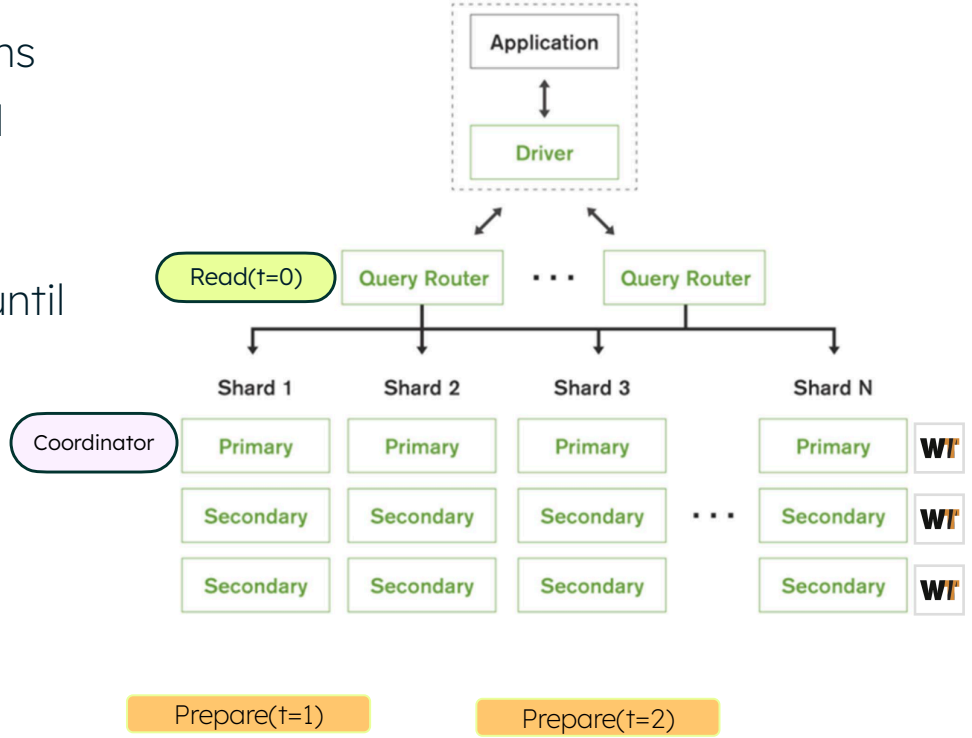
Distributed Transactions in MongoDB

Clients send transaction operations to router, which establishes global **read timestamp**.

Operations forwarded to shards until commit (or must abort).

Coordinator prepares transaction at each shard, receiving prepare timestamp at each.

Prepare timestamps then used using to compute global commit timestamp.



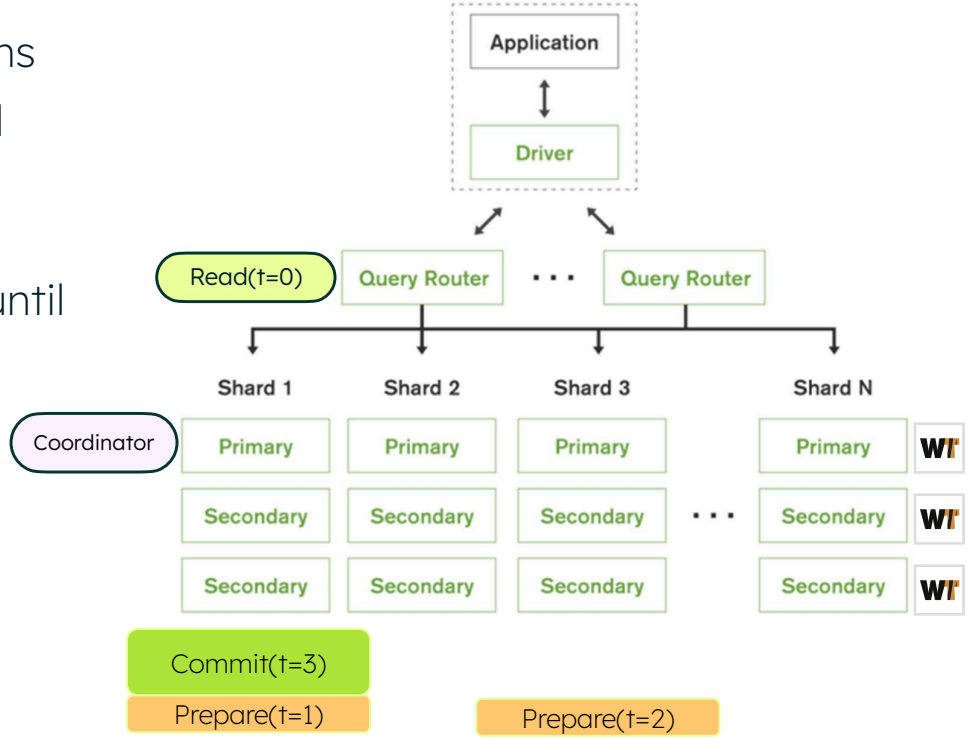
Distributed Transactions in MongoDB

Clients send transaction operations to router, which establishes global **read timestamp**.

Operations forwarded to shards until commit (or must abort).

Coordinator prepares transaction at each shard, receiving prepare timestamp at each.

Prepare timestamps then used using to compute global commit timestamp.



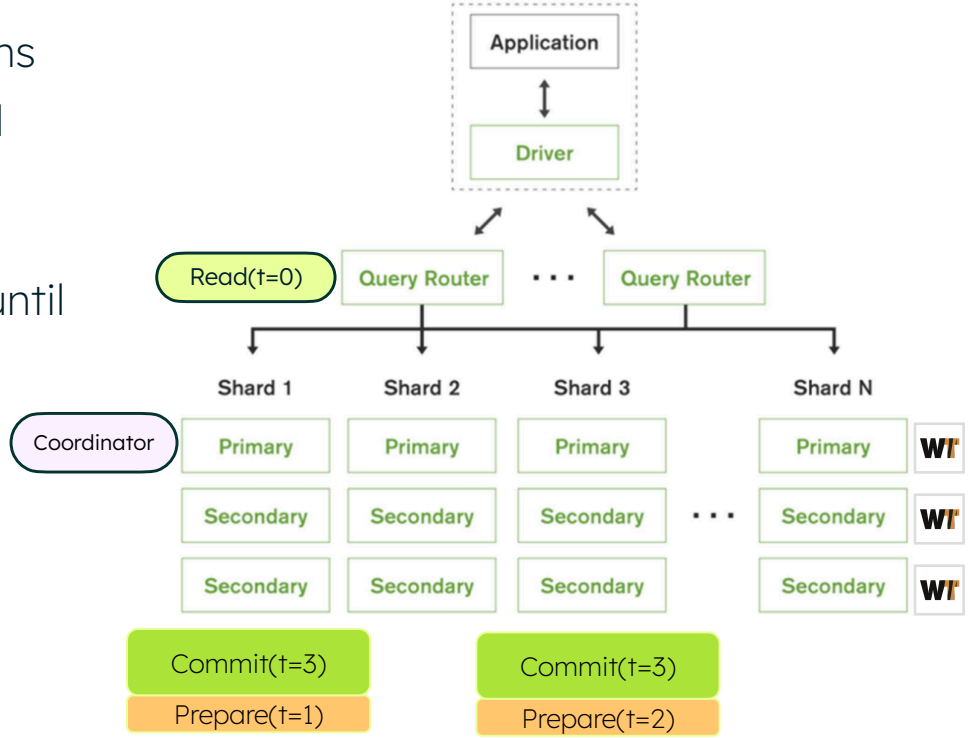
Distributed Transactions in MongoDB

Clients send transaction operations to router, which establishes global **read timestamp**.

Operations forwarded to shards until commit (or must abort).

Coordinator prepares transaction at each shard, receiving prepare timestamp at each.

Prepare timestamps then used using to compute global commit timestamp.



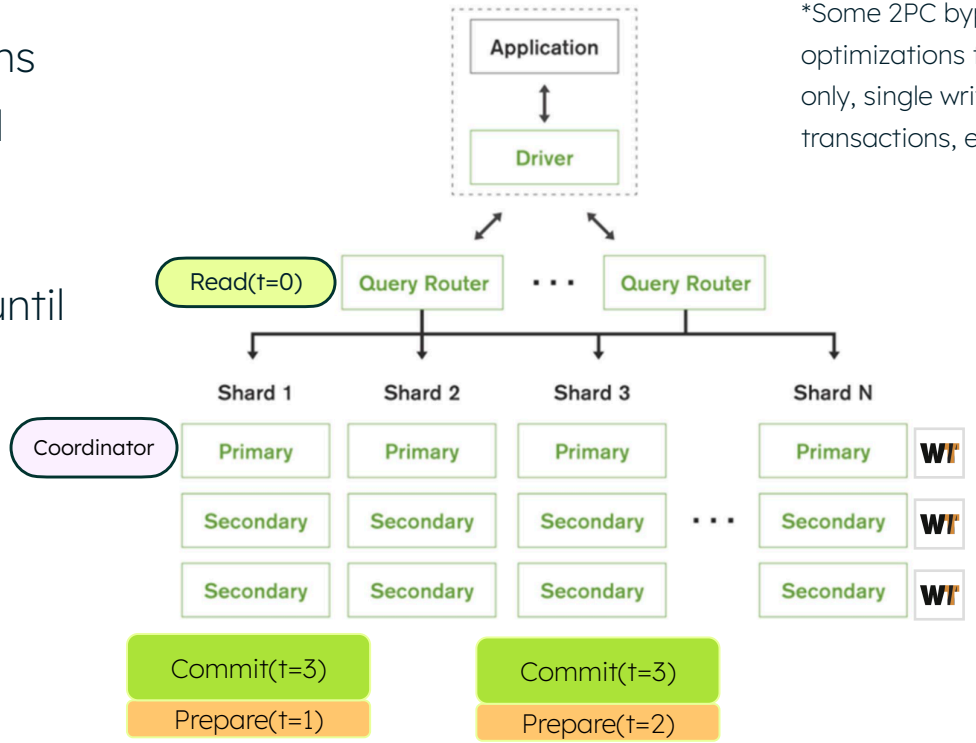
Distributed Transactions in MongoDB

Clients send transaction operations to router, which establishes global **read timestamp**.

Operations forwarded to shards until commit (or must abort).

Coordinator prepares transaction at each shard, receiving prepare timestamp at each.

Prepare timestamps then used using to compute global commit timestamp.



*Some 2PC bypass optimizations for read-only, single write shard transactions, etc.

Modular Protocol Specification

Formal, abstract model of our protocol at a layer above the code.

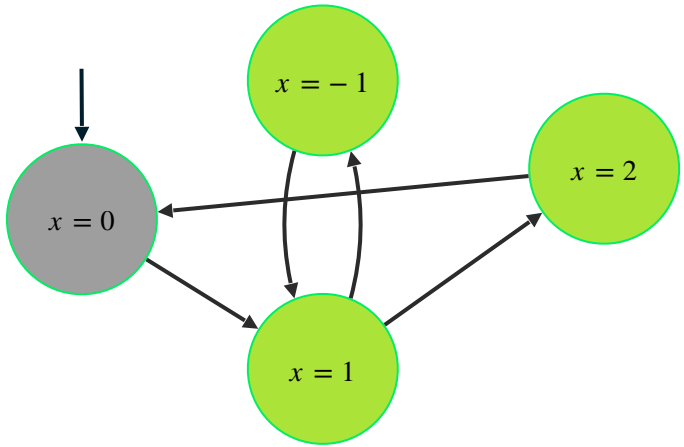
We use **TLA+**, a formal language based on first-order logic for describing any algorithm/system as a discrete transition system.

Modular Protocol Specification

2 basic components for system definition in TLA+:

- What are the set of initial states of your system? (**Init**)
- How can your system transition from one state to another? (**Next**)

$$
\begin{aligned}
&Init \triangleq x = 0 \\
&\\
&Next \triangleq \\
&\quad \lor \land x \geq 0 \\
&\qquad \land x' = (x + 1) \mod 3 \\
&\quad \lor \land x \in \{1, -1\} \\
&\qquad \land x' = -x
\end{aligned}
$$



Modular Protocol Specification

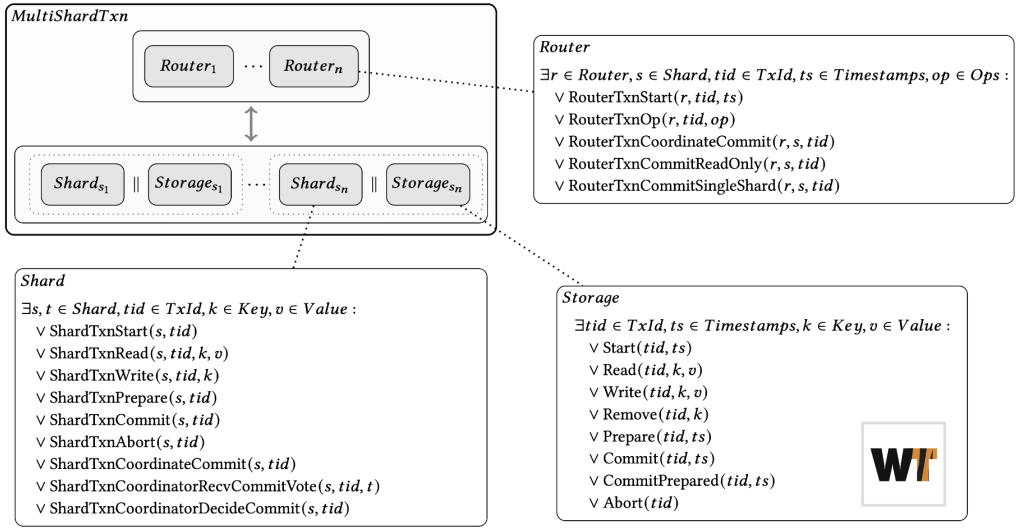
Constant parameters:

- $Shard$: set of shards (e.g. {s1, s2})
- $Router$: set of routers (e.g. {r1})
- Key: Set of possible keys (e.g. {k1, k2})
- $TxId$: set of transaction ids (e.g. {t1,t2})
- RC: global read concern (“local”/”snapshot”).

Specifying our Transactions Protocol

Specification is composed of 2 modules:

- **MultiShardTxn:** models the sharded transaction protocol ($Shard + Router$)
- **Storage:** models the underlying replication/storage layer at each shard



Specifying our Transactions Protocol

Initial States

Router

```
Init ≜
 ∧ rtxn = [r ∈ Router |-> [t ∈ TxId |-> 0]]
 ∧ rParticipants = [r ∈ Router |-> [t ∈ TxId |-> <<>>]]
 ∧ rTxnReadTs = [r ∈ Router |-> [t ∈ TxId |-> NoValue]]
 ∧ rInCommit = [r ∈ Router |-> [t ∈ TxId |-> FALSE]]
```

Shard

```
 ∧ shardTxnReqs = [s ∈ Shard |-> [t ∈ TxId |->  <<>>]]
 ∧ shardTxns = [s ∈ Shard |-> {}]
 ∧ shardPreparedTxns = [s ∈ Shard |-> {}]
 ∧ aborted = [s ∈ Shard |-> [t ∈ TxId |-> FALSE]]
 ∧ coordInfo = [s ∈ Shard |-> [t ∈ TxId |-> [self |-> FALSE, …]
 ∧ coordCommitVotes = [s ∈ Shard |-> [t ∈ TxId |-> {}]]
 ∧ shardOps = [s ∈ Shard |-> [t ∈ TxId |-> <<>>]]
```

Network

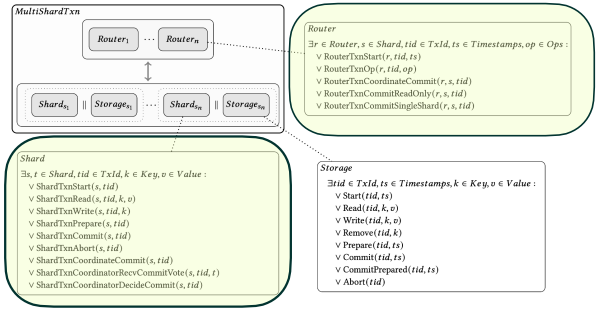
```
 ∧ msgsPrepare = {}
 ∧ msgsVoteCommit = {}
 ∧ msgsAbort = {}
 ∧ msgsCommit = {}
```

Global

```
 ∧ catalog ∈ [Keys -> Shard]
 ∧ ops = [s ∈ TxId |-> <<>>]
```

(Static mapping from keys to shards)

(Stores global transaction histories for isolation checking)

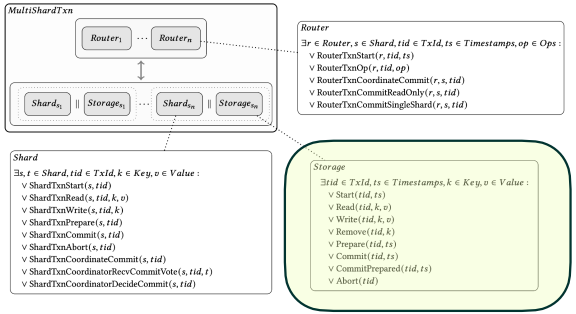


Specifying our Transactions Protocol

State Variables & Initial States

Storage

```
Init ≜
 ∧ log = [s ∈ Shard |-> <<>>]
 ∧ commitIndex = [s ∈ Shard |-> 0]
 ∧ txnSnapshots = [s ∈ Shard |->
                     [t ∈ TxId |-> Nil]]
 ∧ nextTs = [s ∈ Shard |-> 1]
```



Specifying Isolation

At **readConcern**: "snapshot", MongoDB transactions in sharded cluster should provide snapshot isolation.

Check isolation using client-centric isolation model of Crooks [PODC17].

[Adya00] 2000. Generalized Isolation Level Definitions. In Proceedings of the 16th International Conference on Data Engineering (ICDE '00). IEEE Computer Society, USA, 67.

[PODC17] 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 (PODC '17). Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/3087801.3087802

Specifying Isolation

Given a set of transactions T, we say that it satisfies an isolation level if there exists an execution such that every transaction passes a **commit test.**

Commit tests are defined for each isolation level.

Specifying Isolation

Given a set of transactions T, we say that it satisfies an isolation level if there exists an execution such that every transaction passes a **commit test.**

Commit tests are defined for each isolation level.

Set of transactions $\mathcal{T}$

Satisfies isolation level $\mathcal{I}$ iff

$$\exists e : \forall T \in \mathcal{T} : CT_{\mathcal{I}(T,e)}$$

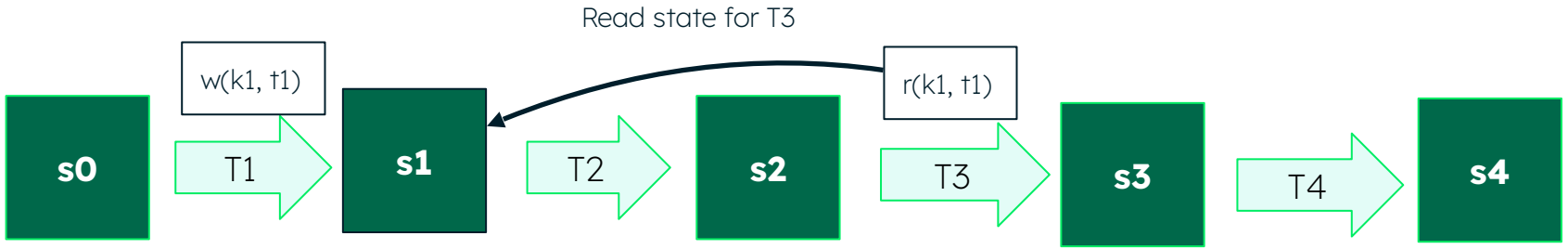
Exists an execution

Commit test holds for every transaction

Specifying Isolation

States of execution

$$S_e = \{s0, s1, s2, s3, s4, s5\}$$



A transaction is a sequence of read/write ops, and an execution is a sequence of transactions along with the states generated by their execution.

Read states of a transaction operation are those in its past it could have read from.

Specifying Isolation

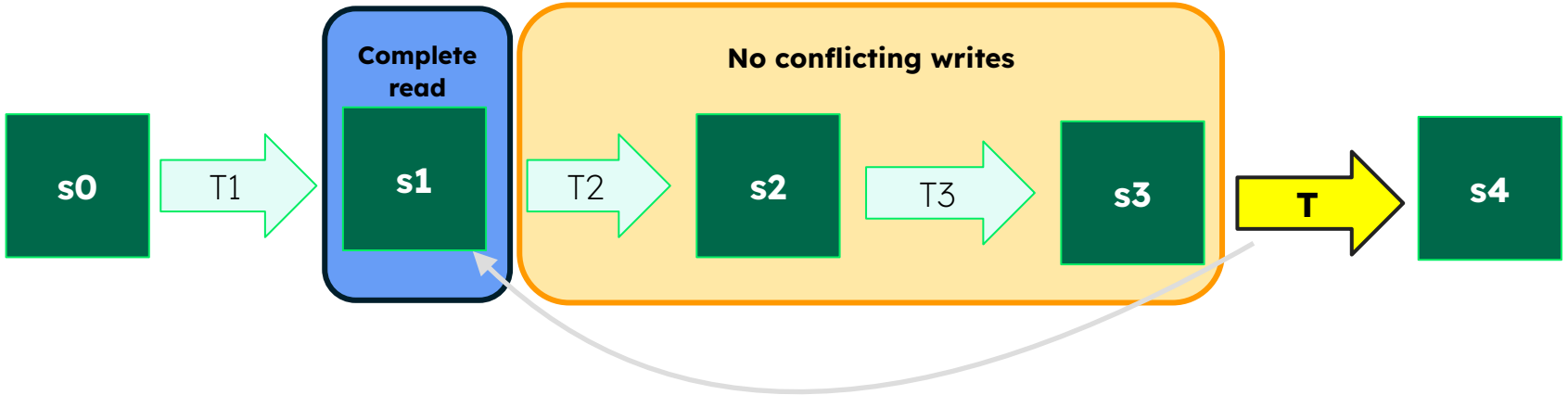
Snapshot isolation commit test:

$$\exists s \in S_e : \text{COMPLETE}_{e,T}(s) \wedge \text{NO-CONFLICT}_T(s)$$

Exists some state in your past.

All reads in T can read from this state.

No intervening transaction wrote a key that T wrote to.



Specifying Isolation

When a transaction commits at a shard, record ops into global history (**ops**).

Then pass **ops** history into state-based isolation checker:

```
ops = (
  t1 :>
      << [op |-> "write", key |-> k1, value |-> t1],
         [op |-> "write", key |-> k2, value |-> t1] >> @@
  t2 :>
      << [op |-> "read", key |-> k1, value |-> t1],
         [op |-> "read", key |-> k2, value |-> NoValue] >> )
```

InitialState == [k \in Keys |-> NoValue]

SnapshotIsolation(InitialState, Range(**ops**))

(No complete state for t2)

Specifying Isolation

When a transaction commits at a shard, record ops into global history (**ops**).

Then pass **ops** history into state-based isolation checker:

```
ops = (
  t1 :>
      << [op |-> "write", key |-> k1, value |-> t1],
         [op |-> "write", key |-> k2, value |-> t1] >> @@
  t2 :>
      << [op |-> "read", key |-> k1, value |-> t1],
         [op |-> "read", key |-> k2, value |-> t1] >> )
```

InitialState == [k \in Keys |-> NoValue]

SnapshotIsolation(InitialState, Range(**ops**)) ✓

Specifying Isolation

When a transaction commits at a shard, record ops into global history (**ops**). Then pass **ops** history into state-based isolation checker:

```
ops = (
  t1 :>
      << [op |-> "read", key |-> k1, value |-> NoValue],
         [op |-> "read", key |-> k2, value |-> NoValue]
         [op |-> "write", key |-> k1, value |-> t1] >> @@
  t2 :>
      << [op |-> "read", key |-> k1, value |-> NoValue],
         [op |-> "read", key |-> k2, value |-> NoValue]
         [op |-> "write", key |-> k1, value |-> t2] >> )
```

InitialState == [k \in Keys |-> NoValue]

SnapshotIsolation(InitialState, Range(**ops**)) ✖

(1 complete state for t1,t2, but with conflict)

Model Checking Isolation Guarantees

Using TLC explicit state model checker: exhaustively explores all reachable behaviors of given protocol specification, while checking safety and/or liveness properties.

Configure model with finite protocol parameters/bounds for termination (e.g. 2 shards, 2 transactions, etc.). **Small models typically effective.**

Model Checking Isolation Guarantees

- $Shard = \{s_1, s_2\}, Router = \{r_1\},$ $TxId = \{t_1, t_2\}, Key = \{k_1, k_2\}$
- **MaxStmts**=2
- **RC** = “snapshot”
- **INVARIANT** SnapshotIsolation

Depth = 35

8,408,701 distinct states (<10 minutes) ✓

- $Shard = \{s_1, s_2\}, Router = \{r_1\},$ $TxId = \{t_1, t_2\}, Key = \{k_1, k_2\}$
- **MaxStmts**=2
- **RC** = “local”
- **INVARIANT** ReadCommitted

Depth = 35

1,950,582 distinct states (<10 minutes) ✓

Permissiveness

Beyond isolation, can also consider a notion of **permissiveness** [1]**.**

For given transactions protocol how much concurrency is allowed within a given isolation level.

$$\mathscr{H}_R \quad / \quad \mathscr{H}_I$$

Histories permitted over all protocol behaviors.

Histories contained in isolation level definition.

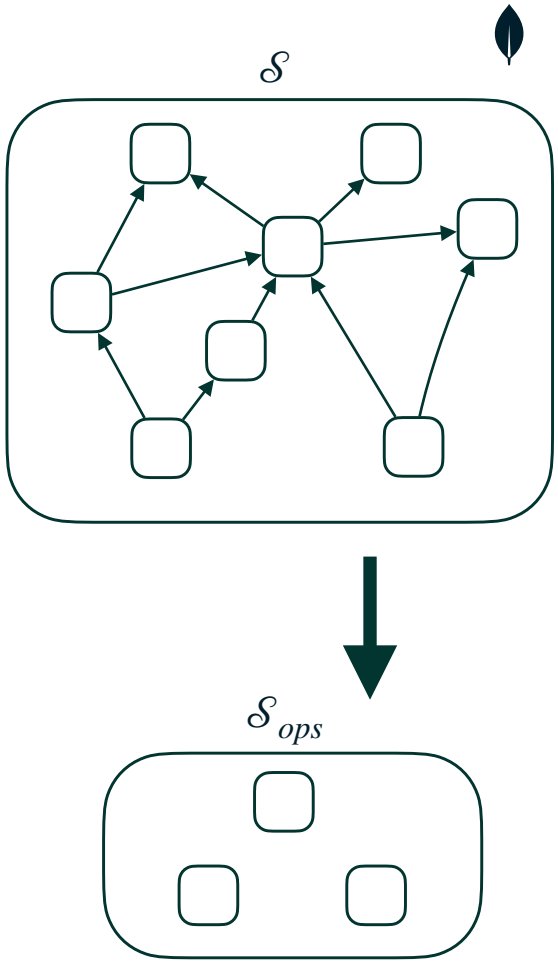
[1] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. 2008. Permissiveness in Transactional Memories. In Proceedings of the 22nd international symposium on Distributed Computing (DISC '08). Springer-Verlag, Berlin, Heidelberg, 305–319. https://doi.org/10.1007/978-3-540-87779-0_21

Permissiveness

Checking permissiveness for a given model M.

Consider its set of reachable states $\mathcal{S}$, and the projection of all possible schedules over any reachable behavior, $\mathcal{S}_{ops}$.

Can compute this with finite model checker and compare permissiveness between protocols.



Permissiveness

- $Shard = \{s_1, s_2\}, Router = \{r_1\},$ $TxId = \{t_1, t_2\}, Key = \{k_1, k_2\}$
- **MaxStmts**=2

Read Concern	Write Conflicts	Prepare Conflicts	Permissiveness
SI definition	-	-	1.0
“snapshot”	Yes	Yes	0.81

Snapshot Isolation

Read Concern	Write Conflicts	Prepare Conflicts	Permissiveness
RC definition	-	-	1.0
“local”	No	No	0.792
“local”	Yes	No	0.790
“local”	Yes	Yes	0.76

Read Committed

MongoDB default

Permissiveness

- $Shard = \{s_1, s_2\}, Router = \{r_1\},$ $TxId = \{t_1, t_2\}, Key = \{k_1, k_2\}$
- **MaxStmts**=2

Read Concern	Write Conflicts	Prepare Conflicts	Permissiveness
SI definition	-	-	1.0
“snapshot”	Yes	Yes	0.81

Snapshot Isolation

Read Concern	Write Conflicts	Prepare Conflicts	Permissiveness
RC definition	-	-	1.0
“local”	No	No	0.792
“local”	Yes	No	0.790
“local”	Yes	Yes	0.76

Read Committed

MongoDB default

Permissiveness

Schedule prevented by prepare conflict blocking, permitted under read committed.

```
t1 : << [op |-> "read", key |-> k2, value |-> NoValue],
        [op |-> "read", key |-> k2, value |-> t2]>> @@

t2 : << [op |-> "write", key |-> k2, value |-> t2]>>
```

(Non-repeatable read)

Read Concern	Write Conflicts	Prepare Conflicts	Permissiveness
RC definition	-	-	1.0
“local”	No	No	0.792
“local”	Yes	No	0.790
“local”	Yes	Yes	0.76

Read Committed

MongoDB default

Model-Based Verification

Formally connect our high level protocol specification to lower level storage layer.

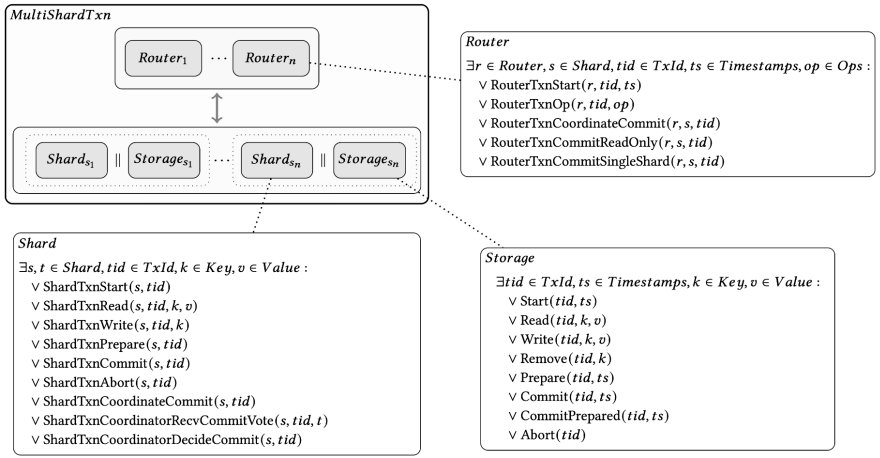


Figure 3: High level overview of our *MultiShardTxn* transactions specification, showing each logical component. Actions of the *Shard* component compose synchronously (indicated by ||) with corresponding, lower level actions of the *Storage* model, while actions of the *Router* and *Shard* interact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

Model-Based Verification

Formally connect our high level protocol specification to lower level storage layer.

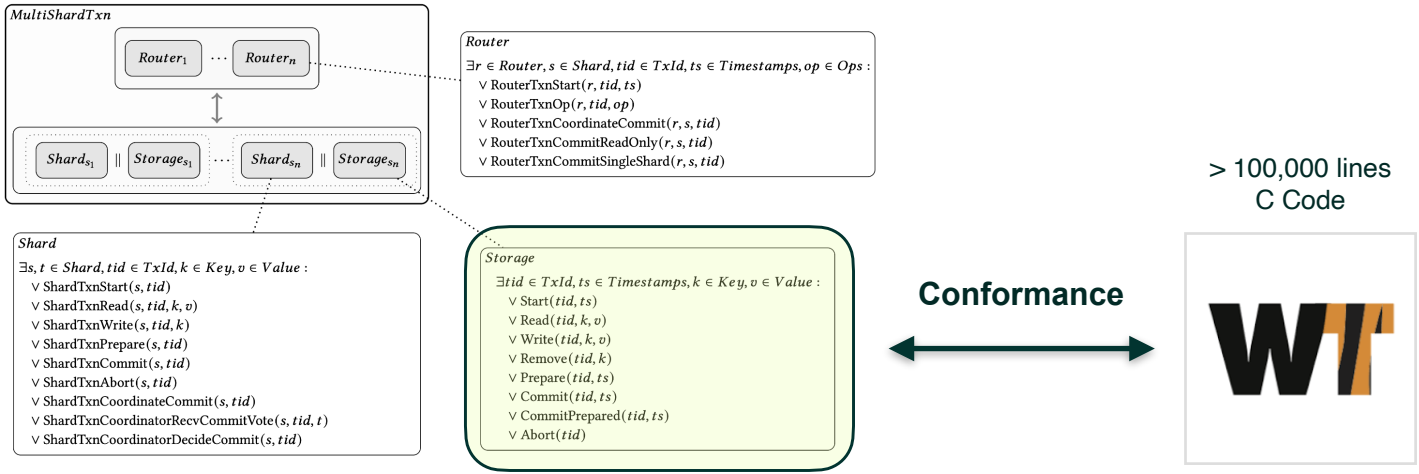
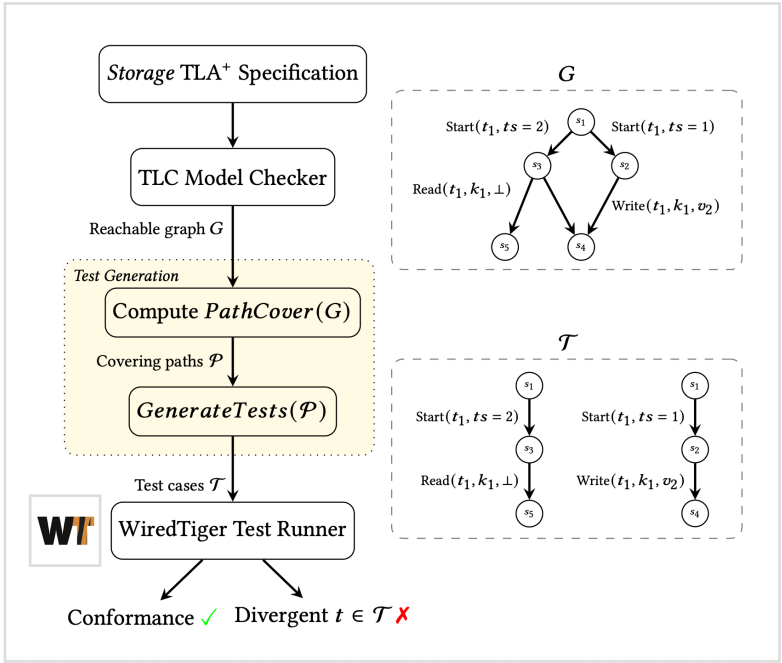


Figure 3: High level overview of our *MultiShardTxn* transactions specification, showing each logical component. Actions of the *Shard* component compose synchronously (indicated by ||) with corresponding, lower level actions of the *Storage* model, while actions of the *Router* and *Shard* interact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

Modular Specification

Use model to automatically generate test cases for checking conformance of WiredTiger implementation to abstraction to model.



Model-Based Verification

Behavior produced from TLC model checker (e.g. path in state graph).

```
[Action 1]: StartTransaction(readTs=1, rc="snapshot", ignorePrepare="false", n=n, tid=t1) res:OK
[Action 2]: StartTransaction(readTs=3, rc="snapshot", ignorePrepare="false", n=n, tid=t2) res:OK
[Action 3]: TransactionWrite(v=t2, k=k1, n=n, tid=t2) res:OK
[Action 4]: TransactionWrite(v=t2, k=k2, n=n, tid=t2) res:OK
[Action 5]: TransactionRead(v=t2, k=k2, n=n, tid=t2) res:OK
[Action 6]: AbortTransaction(n=n, tid=t2) res:OK
[Action 7]: TransactionWrite(v=t1, k=k1, n=n, tid=t1) res:OK
[Action 8]: TransactionWrite(v=t1, k=k2, n=n, tid=t1) res:OK
[Action 9]: TransactionRead(v=t1, k=k1, n=n, tid=t1) res:OK
[Action 10]: TransactionRead(v=t1, k=k2, n=n, tid=t1) res:OK
[Action 11]: TransactionRemove(k=k2, n=n, tid=t1) res:OK
[Action 12]: PrepareTransaction(prepareTs=3, n=n, tid=t1) res:OK
[Action 13]: CommitPreparedTransaction(durableTs=3, n=n, tid=t1, commitTs=3) res:OK
```

Model-Based Verification

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	✓

Model-Based Verification

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	✓

Generate and run test cases against WiredTiger

Takeaways & Future Work

- Leverage modular formal models for both high level algorithm correctness and formalizing and checking lower level module interfaces
- Permissiveness checking of transactions enables finer-grained comparison of concurrency allowance for a given protocol: guide towards optimization opportunities for a transactions protocol

Future Work

- Analyze permissiveness of broader range of existing protocols from literature
- Cutoff bounds for model checking transactional protocols
- Automatic checking of application level consistency/isolation guarantees

Find the transactions model here:

github.com/muratdem/MDBTLA/tree/main/MultiShardTxn

Thanks!