Checking Causal Consistency of MongoDB

Hongrong Ouyang
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
mf20330056@smail.nju.edu.cn

Hengfeng Wei*
State Key Laboratory for Novel Software Technology
Software Institute
Nanjing University
Nanjing, China
hfwei@nju.edu.cn

Yu Huang
State Key Laboratory for Novel Software Technology
Nanjing University
Nanjing, China
yuhuang@nju.edu.cn

ABSTRACT

MongoDB is one of the first commercial distributed databases that support causal consistency. Its implementation of causal consistency combines several research ideas for achieving scalability, fault tolerance, and security. Given its inherent complexity, a natural question arises: *Has MongoDB correctly implemented causal consistency as it claimed?*

To address this concern, the Jepsen team has conducted a black-box testing of MongoDB. However, this Jepsen testing has several drawbacks in terms of specification, test case generation, implementation of causal consistency checking algorithms, and testing scenarios, which undermine the credibility of its reports. In this work, we have proposed a more thorough design of Jepsen testing of the causal consistency protocol of MongoDB. Specifically, we have fully implemented the causal consistency checking algorithms proposed by Bouajjani *et al.* and tested MongoDB under various scenarios against three well-known variants of causal consistency.

CCS CONCEPTS

• **Computer systems organization** → Reliability; • **Software and its engineering** → *Software verification and validation*; • **Information systems** → *Distributed storage*.

KEYWORDS

MongoDB, Casual Consistency, Jepsen, Consistency Checking

ACM Reference Format:
Hongrong Ouyang, Hengfeng Wei, and Yu Huang. 2020. Checking Causal Consistency of MongoDB. In *12th Asia-Pacific Symposium on Internetware (Internetware'20), May 12–14, 2021, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3457913.3457928

*Corresponding Author

Internetware'20, May 12–14, 2021, Singapore, Singapore

ACM ISBN 978-1-4503-8819-1/20/11…$15.00
https://doi.org/10.1145/3457913.3457928

1 INTRODUCTION

Introducing MongoDB. MongoDB is a general-purpose, *document-oriented* distributed NoSQL database [4]. A MongoDB database consists of a set of *collections*, a collection is a set of *documents*, and a document is an ordered set of keys with associated values [30]. [1] MongoDB achieves scalability by partitioning data into shards and fault-tolerance by replicating each shard across a set of nodes [32]. The most general MongoDB deployment is a *sharded cluster*, where each shard is a *replica set* consisting of a *primary* node and several *secondary* nodes; see Figure 1. Client operations are routed to corresponding shards via *routers*, which have access to config servers that are deployed as a replica set to store metadata for deployment. In a replica set, only the primary can accept writes from clients (via drivers), and it will record the writes in its *oplog*. Secondaries can accept reads, and they will replicate the primary's oplog by periodically pulling it from the primary.

Causal Consistency in MongoDB. According to the PACELC theorem [6], an extension to the CAP theorem [13, 18], if there is a network partition (P), a distributed system must trade off availability (A) and consistency (C); else (E), it must trade off latency (L) and consistency (C). For high availability and low latency, MongoDB offers relaxed consistency models. Particularly, in version 3.6 released in November 2017, MongoDB introduced causal consistency [31, 32]. It provides clients with session guarantees including read-your-writes, monotonic-reads, monotonic-writes, and writes-follow-reads [1, 14]. As the Jepsen team [2] denoted, MongoDB is one of the first commercial databases that implement causal consistency [28, 32].

The Official Jepsen Testing of Causal Consistency of MongoDB. Being a production database, MongoDB's implementation of causal consistency requires a multi-dimensional evaluation criteria on performance, scalability, and security [32]. It combines several research ideas, including hybrid logical clocks [22], explicit dependency tracking [8, 16], Raft-based replication consensus protocol [27], and signature-verification mechanism. Given its inherent complexity, a natural question arises: *Has MongoDB correctly implemented causal consistency as it claimed in docs?* To address this concern, the Jepsen team has conducted a black-box testing against MongoDB 3.6.4 and 4.0.0-rc1 [28]. They designed test cases that characterized client operations, ran test cases in various scenarios, collected histories of executions generated by MongoDB, and utilized an adapted version of the causal consistency checking

[1] Roughly speaking, a document is an analog to a row in a relational database, and a collection is to a table.

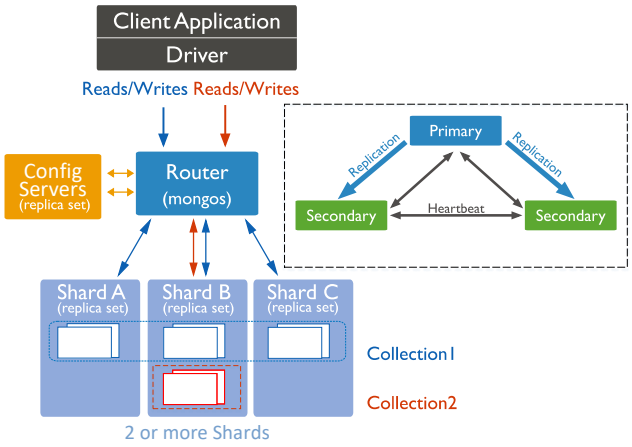


Figure 1: MongoDB Deployment as a Sharded Cluster.

algorithm proposed by Bouajjani *et al.* [12] to check whether these histories satisfy causal consistency.

Drawbacks of the Official Jepsen Testing. However, the official Jepsen testing has several drawbacks in terms of specification, test case generation, implementation of causal consistency checking algorithms, and testing scenarios, which undermine the credibility of its reports. Specifically,

- There are several variants of causal consistency, including Causal Consistency (CC) [15, 29], Causal Memory (CM) [7], and Causal Convergence (CCv) [29]. Not all of them are comparable [29]. However, the official Jepsen testing did not clearly specify the causal consistency variant against which it tested the MongoDB database.
- In terms of test cases, the official Jepsen testing used *independent* keys. That is, each session accesses only a single key and different sessions access different keys. Concretely, each session performs a sequence of five operations on its key: an initial read, a write of 1, a read, a write of 2, and a final read. However, causal consistency is *not* compositional [26], i.e., the composition of a set of keys satisfying causal consistency may be not causally consistent. Thus, the test cases are too restrictive for causal consistency checking.
- Given the specific test cases above, the official Jepsen testing hard-wired the expected return value of each read operation in its causal consistency checking algorithm. In other words, it has not fully implemented the causal consistency checking algorithms in [12].
- Although the official Jepsen testing has tested the causal consistency of MongoDB under network partitions, it did not cover the scenarios such as node failures and data movement among shards.

Our Contributions. In this work, we propose a more thorough design of Jepsen testing of the causal consistency protocol of MongoDB. [2] Specifically,

- We considered three well-known variants of causal consistency, following the formal specification given in [12].
- We generated the most general operation sequences for clients, without any restrictions on keys.
- We fully implemented the "bad patterns"-based causal consistency checking algorithm proposed by Bouajjani *et al.*;
- We designed more testing scenarios, covering network partitions, node failures, and data movement among shards.

The preliminary experimental results confirmed the claim in MongoDB's documentation [1] that in the presence of node failures or network partitions, causal consistency is guaranteed only for reads with "majority" `readConcern` (explained shortly in Section 2.1) and writes with "majority" `writeConcern`.

The rest of the paper is organized as follows. Section 2 provides preliminaries on causal consistency and the Jepsen testing framework. Section 3 describes the official Jepsen testing of causal consistency of MongoDB and introduces our more thorough design. Section 4 demonstrates our experiments and results. Section 5 discusses related work. Section 6 concludes the paper.

2 PRELIMINARIES

2.1 Causal Consistency in MongoDB

MongoDB enables causal consistency in client sessions. Moreover, MongoDB's causal consistency can be combined with *tunable consistency*, which allows clients to select the trade-offs between consistency and latency, at a per operation level [30]. `writeConcern` specifies the number of replica set members that must acknowledge the write before returning to a client. In particular, `w : ``majority''` requires a write operation to be acknowledged by a majority of the replica set members before being returned to the client. `readConcern` determines what consistency guarantees data returned to a client must satisfy. The default value of `readConcern` is `level : ``local''`, which allows to return the local data in a single replica set member. In contrast, `level : ``majority''` guarantees that the returned data has been written to a majority of the replica set members. As claimed in MongoDB's documentation [1], in the presence of node failure or network partitions, causally consistent sessions can only guarantee causal consistency for reads with "majority" `readConcern` and writes with "majority" `writeConcern`. In a good condition, however, write operations with `w : 1` `writeConcern` can also provide causal consistency.

2.2 Causal Consistency: Formal Specification

We review the formal specification of causal consistency with respect to read-write registers, following [12].

2.2.1 Replicated Objects. We focus on *read/write registers* from $\mathbb{X}$, ranged over by x, y, *etc.* They support a set of methods $\mathbb{M} = \{\mathsf{wr}, \mathsf{rd}\}$ for writing to or reading from a register (i.e., variable), with input or output values from a domain $\mathbb{V}$.

2.2.2 Histories. We model the interactions between clients and a distributed database maintaining replicated read/write registers by *histories*. A history $h = (O, \mathsf{PO}, \ell)$ is the poset (partial-ordered set) (O, PO) labeled by $\mathbb{M} \times \mathbb{V} \times \mathbb{V}$, where

[2] The project can be found at https://github.com/Tsunaou/Checking-Causal-Consistency-of-MongoDB.

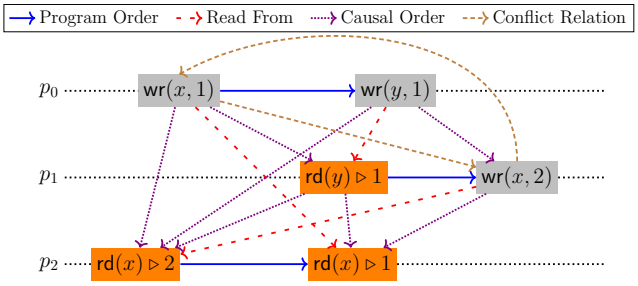


Figure 2: A history h that is *not* CCv. (The arrows for CO that are implied by transitivity are not shown.)

- O is set of operation identifiers, or simply *operations*. We use R and W to denote the set of read and write operations, respectively.
- PO is a union of total orders among operations called *program order*. For $o_1, o_2 \in O$, $o_1 <_{\mathsf{PO}} o_2$ means that o_1 and o_2 were issued by the same client and o_1 occurred before o_2.
- For an operation $o \in O$, its label $\ell(o) = (m, arg, rv) \in \mathbb{M} \times \mathbb{V} \times \mathbb{V}$ indicates that o is an invocation of method m with input argument arg, returning value rv. We sometimes denote $\ell(o)$ by $m(arg) \triangleright rv$.

We use $\mathsf{wr}(x, v) \triangleright \bot$ (or simply $\mathsf{wr}(x, v)$) to denote a write of value $v \in \mathbb{V}$ to register $x \in \mathbb{X}$ returning $\bot \notin \mathbb{V}$, and $\mathsf{rd}(x) \triangleright v$ a read of x returning v. In addition, for an operation o with $\ell(o) = \mathsf{wr}(x, v)$ or $\ell(o) = \mathsf{rd}(x) \triangleright v$, we define $\mathsf{var}(o) = x$ and $\mathsf{val}(o) = v$.

2.2.3 Sequential Semantics. The consistency of replicated read-write registers is defined with respect to its sequential semantics. Intuitively, in any operation sequence on read-write registers, a rd operation returns the value of the latest preceding wr on the same register, or the initial value 0 if there are no such prior writes. Formally, the sequential semantics S_{RW} of read-write registers is the smallest set of sequences labeled by $\mathbb{M} \times \mathbb{V} \times \mathbb{V}$ satisfying

- $\epsilon \in S_{\mathsf{RW}}$, where ϵ is the empty sequence;
- if $\rho \in S_{\mathsf{RW}}$, then $\rho \cdot \mathsf{wr}(x, v) \in S_{\mathsf{RW}}$;
- if $\rho \in S_{\mathsf{RW}}$ contains no writes on x, then $\rho \cdot \mathsf{rd}(x) \triangleright 0 \in S_{\mathsf{RW}}$;
- if $\rho \in S_{\mathsf{RW}}$ and the last write in ρ on register x is $\mathsf{wr}(x, v)$, then $\rho \cdot \mathsf{rd}(x) \triangleright v \in S_{\mathsf{RW}}$.

2.2.4 Causal Consistency. Causal consistency guarantees that all clients agree on the relative ordering of causally related operations [7, 23]. However, operations that are not causally related may be observed in different orders by different clients. In this paper, we take causal convergence (CCv) as an example. [3] Besides the causal order, CCv ensures eventual convergence captured by a total *arbitration order*.

Definition 2.1 (Causal Convergence). A history $h = (O, \mathsf{PO}, \ell)$ is CCv with respect to specification S_{RW} if there exists a strict partial order $co \subseteq O \times O$ called the *causal order* and a strict total order $arb \subseteq O \times O$ called the *arbitration order* such that for each operation $o \in O$, there exists a specification sequence $\rho_o \in S_{\mathsf{RW}}$ such that the following conditions hold:

$$\text{AxCausal} \triangleq \mathsf{PO} \subseteq co,$$
$$\text{AxArb} \triangleq co \subseteq arb,$$
$$\text{AxCausalArb} \triangleq (co^{-1}(o), arb, \ell)\{o\} \preceq \rho_o.$$

Here $co^{-1}(o)$ is the set of operations that precede o in causal order. Formally, $co^{-1}(o) = \{o' \mid o' <_{co} o\}$.

Let $\rho = (O, <, \ell)$ be a $\mathbb{M} \times \mathbb{V} \times \mathbb{V}$ labeled poset and $o \in O$ be an operation. $\rho\{o\}$ is the labeled poset in which only the return value of o is kept. Formally, $\rho\{o\}$ is the $(\mathbb{M} \times \mathbb{V}) \cup (\mathbb{M} \times \mathbb{V} \times \mathbb{V})$ labeled poset $(O, <, \ell')$ where $\ell'(o) = \ell(o)$ and for all $o' \in O$ and $o' \neq o$, $\ell'(o') = (m, arg)$ if $\ell(o') = (m, arg, rv)$.

Let $\rho = (O, <, \ell)$ and $\rho' = (O, <', \ell')$ be two $(\mathbb{M}\times\mathbb{V})\cup(\mathbb{M}\times\mathbb{V}\times\mathbb{V})$ labeled posets. $\rho' \preceq \rho_o$ means that ρ' has less order and label constraints on set O. Formally, $\rho' \preceq \rho$ if $<' \subseteq <$ and for all $o \in O$ and all $m \in \mathbb{M}, arg \in \mathbb{V}, rv \in \mathbb{V}$, $\ell'(o) = \ell(o)$ or $\ell'(o) = (m, arg)$ if $\ell(o) = (m, arg, rv)$.

Table 1: Definitions of Bad Patterns

Bad Patterns	Description
CyclicCO	$\mathsf{PO} \cup \mathsf{RF}$ is cyclic.
ThinAirRead	$\exists r \in R.\ \mathsf{val}(r) \neq 0 \wedge (\nexists w \in W.\ w <_{\mathsf{RF}} r)$
WriteCOInitRead	$\exists r \in R, w \in W.\ w <_{\mathsf{CO}} r$ $\wedge\ \mathsf{var}(w) = \mathsf{var}(r) \wedge \mathsf{val}(r) = 0$
WriteCORead	$\exists w_1, w_2 \in W, r_1 \in R.\ \mathsf{var}(w_1) = \mathsf{var}(w_2)$ $\wedge\ w_1 <_{\mathsf{CO}} w_2 <_{\mathsf{CO}} r_1 \wedge w_1 <_{\mathsf{RF}} r_1$
CycliCF	$\mathsf{CF} \cup \mathsf{CO}$ is cyclic.

2.3 Causal Consistency Checking

The general decision problem of checking whether a history over read-write registers is causally consistent is NP-complete [12]. However, for *differentiated* histories in which the values written to the same register are distinct, it is polynomial time [12]. Differentiated histories can be achieved by attaching unique timestamps to writes in implementation. We consider only differentiated histories below.

The polynomial-time checking algorithms proposed by Bouajjani *et al.* are based on the notion of "bad patterns". Each causal consistency variant can be precisely characterized by lacking of a set of certain bad patterns. The bad patterns are expressed in terms of program order PO, read-from relation RF, causal order CO, conflict relation CF, and happened-before relation (HB) [4] on operations.

Definition 2.2. The *read-from relation* $\mathsf{RF} \subseteq W \times R$ associates a read with the write from which it obtains the value. Formally,

$$\forall w \in W, r \in R.\ (w, r) \in \mathsf{RF} \iff \mathsf{var}(w) = \mathsf{var}(r) \wedge \mathsf{val}(w) = \mathsf{val}(r).$$

Definition 2.3. The *casual order* $\mathsf{CO} \subseteq O \times O$ is defined as the transitive closure of program order and read-from relation. Formally,

$$\mathsf{CO} = (\mathsf{PO} \cup \mathsf{RF})^+.$$

[3] The interested readers are referred to [12] for the formal definitions and checking algorithms for CC and CM.

[4] HB is used by CM. We omit it in this paper.

Table 2: Comparison between the Official Jepsen Testing and Our Design

	The Official Jepsen Testing	Our Design of Jepsen Testing
Specification	unspecified	three well-known variants: CC, CM, and CCv
Test Case Generation	restricted on keys and operation sequences	general for differentiated histories
Checking Algorithms	ad hoc for restricted test cases	full implementation of [12]
Testing Scenarios	network partition	network partition, data movement, node failure

Table 3: Hardware Configurations

Component	Configuration
Control Node	Intel(R) Core(TM) i5-9500 CPU @ 3.00GHz; 16GB; Ubuntu 20.04
Database Node	Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz; 4GB; Ubuntu 16.04
Checker Server	Intel(R) Core(TM) i9-9900X CPU @ 3.50GHz; 32GB ; Ubuntu 16.04

Definition 2.4. The *conflict relation* $\mathsf{CF} \subseteq W \times W$ orders two writes on the same register according to a third read operation. Formally,

$$\forall w,w' \in W.\ (w,w') \in \mathsf{CF} \iff \exists r' \in R.\ (w',r') \in \mathsf{RF} \wedge \mathsf{var}(w) = \mathsf{var}(r') \wedge (w,r') \in \mathsf{CO}.$$

Example 2.5. Consider the history h of Figure 2. Since $\mathsf{wr}(x,2) <_{\mathsf{RF}} \mathsf{rd}(x) \triangleright 2$ and $\mathsf{rd}(x) \triangleright 2 <_{\mathsf{PO}} \mathsf{rd}(x) \triangleright 1$, we have $\mathsf{wr}(x,2) <_{\mathsf{CO}} \mathsf{rd}(x) \triangleright 1$. In addition, $\mathsf{wr}(x,1) <_{\mathsf{RF}} \mathsf{rd}(x) \triangleright 1$. Thus, $\mathsf{wr}(x,2) <_{\mathsf{CF}} \mathsf{wr}(x,1)$.

The following theorem characterizes CCv in terms of bad patterns defined in Table 1.

Theorem 2.6. *A history h is CCv if and only if h does not exhibit any bad patterns of CyclicCO, WriteCOInitRead, ThinAirRead, WriteCORead, or CyclicHF.*

Example 2.7. Consider the history h of Figure 2. It is *not* CCv. First, since $\mathsf{wr}(x,1) <_{\mathsf{CO}} < \mathsf{wr}(x,2) <_{\mathsf{CO}} \mathsf{rd}(x) \triangleright 1$ and $\mathsf{wr}(x,1) <_{\mathsf{RF}} \mathsf{rd}(x) \triangleright 1$, it exhibits the bad pattern WriteCORead. In addition, there is a cycle in CF: $\mathsf{wr}(x,1) <_{\mathsf{CF}} \mathsf{wr}(x,2) <_{\mathsf{CF}} \mathsf{wr}(x,1)$. Thus, it also exhibits the bad pattern CyclicCF.

2.4 Jepsen

Jepsen is a library for black-box testing of distributed systems [2]. A typical Jepsen testing of a distributed database consists of a deployment of the database and a control node. The control node starts several worker processes called *clients*. A *generator* is responsible for continuously generating operations and dispatching them to clients, according to user-defined rules. Clients interact with the database by issuing operations. The invocations and responses produced are recorded in a *history*. When the test finishes, the history is checked by a *checker* against a desired consistency model.

To test the fault-tolerant capability of the database, special worker processes called *nemesis* continuously inject faults or rare events (such as data movement among shards) into the database deployment.

3 JEPSEN TESTING OF CAUSAL CONSISTENCY OF MONGODB

In this section we first describe the official Jepsen testing of causal consistency of MongoDB 3.6.4 and 4.0.0-rc1 [28], from the perspectives of specification, test case generation, implementation of causal consistency checking algorithms, and testing scenarios. To overcome its drawbacks identified in Section 1, we then design a more thorough Jepsen testing of causal consistency of MongoDB.

3.1 The Official Jepsen Testing

The MongoDB deployment under test consists of two shards, each of which is a replica set of 5 nodes.

3.1.1 Specification. The Jepsen team claimed that they have tested MongoDB against causal consistency. However, they did not clearly specify the variant of causal consistency.

3.1.2 Test Case Generation. Treating a MongoDB collection as a set of read-write registers, the *generator* generates read and write operations for clients. The dispatch rule ensures that each client accesses only a single register and different clients access different registers. Specifically, the operation sequence of each client consists of 5 operations as follows

$$\langle r, w1, r, w2, r \rangle,$$

where r denotes a read of the register that belongs to the client, $w1$ a write of value 1 to the register, and $w2$ a write of value 2 to the register.

3.1.3 Checking Algorithms. Since the test cases are quite restrictive, it suffices for the *checker* to verify whether the three reads of each client return 0, 1, and 2 in order.

3.1.4 Testing Scenarios. The official Jepsen testing has designed a kind of *nemesis* called *partition-random-halves* to trigger network partitions randomly. Specifically, in the 5-node deployment of MongoDB, the network will be split into two disconnected parts: one (denoted P_1) consists of 2 nodes, one of which is the original primary node, and the other (denoted P_2) consists of 3 nodes. Since 3 nodes in P_2 constitute a majority (of 5 nodes), one of them will be

Table 4: Experimental Results of Causal Consistency Checking of MongoDB (✓: satisfaction; ✗: violation; ⊗: unexpected ThinAirRead bad patterns discussed in Section 4.3.)

# of Operations	With *Nemesis*						Without *Nemesis*					
	(majority, majority)			(w1, local)			(majority, majority)			(w1, local)		
	CC	CM	CCv	CC	CM	CCv	CC	CM	CCv	CC	CM	CCv
100	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
200	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
300	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
400	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
500	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
600	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
700	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
800	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
900	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
1000	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1100	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
1200	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
1300	⊗	⊗	⊗	✓	✓	✓	✓	✓	✓	✓	✓	✓
1400	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1500	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1600	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
1700	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
1800	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
1900	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
2000	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
2500	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
3000	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
3500	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
4000	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
4500	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
5000	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓

elected as a new primary. Consequently, there would temporarily be two nodes that consider themselves as the primary *of the cluster*.

After the network recovers, the writes performed on the original primary node during network partition will be rolled back. The Jepsen testing revealed that in the presence of network partitions, causally consistent sessions can only guarantee causal consistency for reads with "majority" `readConcern` and writes with "majority" `writeConcern`.

3.2 Our Design of Jepsen Testing

As shown in Table 2, we have improved the official Jepsen testing in the following aspects.

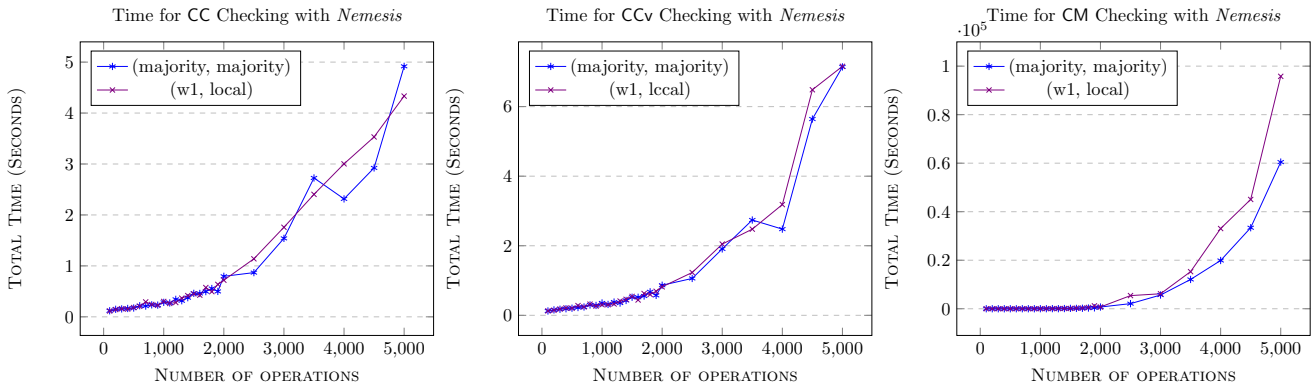
3.2.1 Specification. We test MongoDB against three well-known variants of casual consistency, namely, CC, CM, and CCv. Specifically, we adopt the formal specification given in [12].

3.2.2 Test Case Generation. In our design, the *generator* generates an arbitrary differentiated operation sequence for each client using YCSB [5]. Particularly, we impose no restrictions on keys as the official Jepsen testing does, only controlling the range and distribution of generated keys, and the ratio of read and write operations.

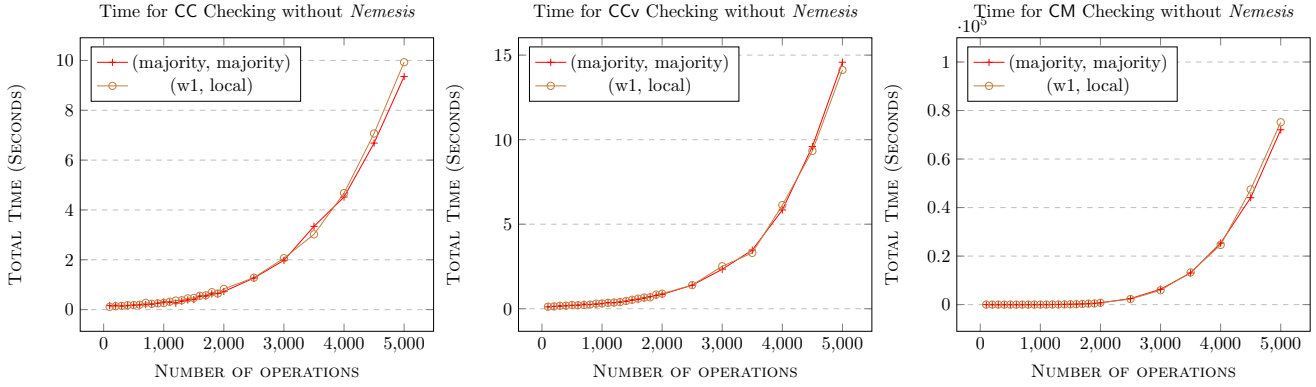
The generated keys follow a uniform distribution. To ensure that all writes on the same register write unique values, the generator attaches values $1, 2, \ldots$ to them in order. We record necessary information about each operation during generation and execution, including its type (i.e., read or write), the value it reads or writes, the client that issues the operation, and the index indicating the order in which the operation is generated.

3.2.3 Checking Algorithms. To check an arbitrary differentiated history against several variants of causal consistency, we fully implement the "bad patterns"-based causal consistency checking algorithms for CC, CM, and CCv [12].

3.2.4 Testing Scenarios. Besides *partition-random-halves* in the official Jepsen testing, we introduce two additional *nemesis* called *node-failure* and *data-mover*. The *node-failure* nemesis randomly selects a database node, suspends it for a while, and then recover



(a) Time for Causal Consistency Checking with *Nemesis*



(b) Time for Causal Consistency Checking without *Nemesis*

Figure 3: Time of Checking Whether Histories Satisfy Causal Consistency.

it. This may trigger leader election. The *data-mover* nemesis periodically moves data among shards. In an execution, *partition-random-halves*, *node-failure*, and *data-mover* are generated and scheduled by the *generator*, according to user-defined rules.

4 PRELIMINARY EVALUATIONS

We have implemented the checking algorithms of [12] and checked histories produced by MongoDB 4.2.3 against CC, CM, and CCv. We use the Jepsen testing framework of version 0.1.17. Table 3 shows the hardware configurations of the control node, the database nodes, and the checker server.

4.1 Experimental Setup

We adopt the same MongoDB deployment as that in the official Jepsen testing: it consists of two shards, each of which is a replica set of 5 nodes.

In each experiment, we fix 100 registers and 10 clients. The *generator* generates read or write operations and appends them into a queue. For each register, the ratio between the number of read operations and that of write operations is 3 : 1. Each client creates a causally consistent session, extracts operations from the operation queue, and issues them to MongoDB servers.

For each experiment, we tune the total number of operations and the `readConcern` and `writeConcern` levels for operations. To handle possible exceptions thrown by MongoDB during write operations, we restart a new causally consistent session in the corresponding client. Moreover, we cover both the scenarios with and without *nemesis*. For each history produced by MongoDB, we check whether it satisfies CC, CM, and CCv.

4.2 Experimental Results

Table 4 shows the experimental results of checking causal consistency of MongoDB.

4.2.1 Causal Consistency Checking. The preliminary experimental results have confirmed the claim in MongoDB's documentation [1] that in the presence of *nemesis* (such as *partition-random-halves*, *node-failure*, and *data-mover*), causally consistent sessions guarantee causal consistency only for reads with "majority" `readConcern` and writes with "majority" `writeConcern`. In contrast, in the presence of *nemesis*, the histories with "local" `readConcern` and "w1" `writeConcern` may violate any of three causal consistency variants. On the other hand, without *nemesis*, MongoDB can provide all three variants of causal consistency even with "local" `readConcern` and "w1" `writeConcern`.

Table 5: Snippet of a History that Exhibits an Unexpected ThinAirRead Bad Pattern

No.	Operation	Exception	`readConcern`
1128	wr(85, 5)	`MongoWriteException`	majority
1129	wr(20, 5)	`MongoWriteException`	majority
1149	rd(20) ▷ 5	⟍	majority
1266	rd(85) ▷ 5	⟍	majority
1336	rd(20) ▷ 5	⟍	majority
3756	rd(20) ▷ 5	⟍	majority

4.2.2 Performance. Figure 3 demonstrates the performance of checking whether histories satisfy causal consistency. According to [12], it takes $O(n^3)$ to check a differentiated history with n operations against CC or CCv. In contrast, it takes $O(n^5)$ against CM. The experimental results in Figure 3 have exhibited such a substantial performance gap.

4.3 Unexpected ThinAirRead Bad Patterns

We have observed some unexpected ThinAirRead bad patterns in our preliminary evaluations, marked ⊗ in Table 4. They appear in some histories that are produced without *nemesis* and consist of reads with "majority" `readConcern` and writes with "majority" `writeConcern`. Table 5 shows a snippet of such a history. Note that the write operation wr(85, 5) of No. 1128 incurs a runtime exception called `com.mongodb.MongoWriteException`. Since the causal consistency checking algorithms in [12] implicitly assume that all write operations are successful, this write operation is considered failed and discarded from the history. However, a later read operation rd(85) of No. 1266 obtains the value 5 from key 85, indicating that the write operation wr(85, 5) has actually written its value to the database. This gives rise to a ThinAirRead bad pattern during checking.

We remark that the unexpected ThinAirRead bad patterns above do not necessarily imply bugs in the causal consistency protocols of MongoDB. However, to better explain such unexpected results, it needs to design checking algorithms for histories which may contain failed write operations.

5 RELATED WORK

The Jepsen Testing of MongoDB. The Jepsen team has tested MongoDB concerning its consistency models several times in recent years.

- In 2013, they tested the election and data replication protocol of MongoDB 2.4.3 [9]. It showed that acknowledged writes may be lost under network partitions at all consistency levels.
- In 2015, they tested the single-document consistency of MongoDB 2.6.7 [10]. It showed that "strictly consistent" reads may see stale versions of documents, and worse still they may return garbage data that has never been written before.
- In 2017, they tested the v0 and v1 replication protocols of MongoDB 3.4.0-rc3 [20]. It showed that the v0 replication protocol may lose the majority-committed documents. The new v1 replication protocol also contained bugs, allowing data loss in all versions up to MongoDB 3.2.11 and 3.4.0-rc4.
- In 2018, they tested the causal consistency protocol of MongoDB 3.6.4 [28]. It showed that in the presence of node failures or network partitions, causal consistency is guaranteed only for reads with "majority" `readConcern` and writes with "majority" `writeConcern`. In this paper, we have identified several drawbacks of this testing in terms of specification, test case generation, implementation of causal consistency checking algorithms, and testing scenarios. We have also proposed a more thorough design of Jepsen testing of the causal consistency protocol of MongoDB.
- In 2020, they tested the transactional consistency models of MongoDB 4.2.6 [21]. It showed that MongoDB failed to preserve snapshot isolation, even for reads with "majority" `readConcern` and writes with "majority" `writeConcern`.

The Consistency Checking Problem. Much work has been devoted to the problem of checking whether a given history satisfies a desirable consistency model. Gibbons *et al.* [17] have systematically studied the complexity of the checking problem against strong consistency models, including linearizability [19] and sequential consistency [11]. Regarding weak consistency models, Wei *et al.* [33] have addressed the problem of checking PRAM consistency [25] over histories of read/write registers. They first proved that for non-differentiated histories, the decision problem is NP-complete, and then proposed a polynomial-time checking algorithm for differentiated histories. Recently, Bouajjani *et al.* have addressed the problem of checking causal consistency [12]. They considered three well-known variants of causal consistency, namely CC, CM, and CCv. They proved that checking whether a general history of arbitrary replicated objects satisfies CC, CM, or CCv is NP-hard, and that it is NP-complete for histories of read/write registers. Moreover, they proposed polynomial-time algorithms for differentiated histories of read/write registers. In this paper, we have fully implemented these efficient checking algorithms and utilized them to test the causal consistency protocol of MongoDB.

6 CONCLUSION AND FUTURE WORK

We have proposed a thorough design of Jepsen testing of the causal consistency protocol of MongoDB. It has strengthened the official Jepsen testing in 2018 [28] in terms of specification, test case generation, implementation of causal consistency checking algorithms, and testing scenarios. We have conducted a preliminary evaluation of our design and more intensive experiments are needed.

In the future, we plan to improve the official Jepsen testing of the transaction protocols of MongoDB 4.2.6 [21]. On the other hand, we are also interested in applying formal methods to MongoDB's protocols. Specifically, we will formally specify these protocols in TLA$^+$ (Temporal Logic of Action) [24], verify them using the TLC model checker [34], and develop mechanical correctness proofs for them using TLAPS (TLA Proof System) [3].

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China under Grant No. 61702253 and No. 61772258.

REFERENCES

[1] [n.d.]. *Causal Consistency and Read and Write Concerns.* https://docs.mongodb.com/manual/core/causal-consistency-read-write-concerns/

[2] [n.d.]. *Jepsen.* https://github.com/jepsen-io/jepsen

[3] [n.d.]. Microsoft Research – Inria Joint Centre: TLA$^+$ Proof System (TLAPS). https://tla.msr-inria.inria.fr/tlaps/content/Home.html. [Accessed: Jan. 21, 2019].

[4] [n.d.]. *MongoDB.* https://www.mongodb.com/

[5] [n.d.]. *YCSB.* https://github.com/brianfrankcooper/YCSB

[6] Daniel Abadi. 2012. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *IEEE Computer* 45, 2 (2012), 37–42. https://doi.org/10.1109/MC.2012.33

[7] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. 1995. Causal memory: definitions, implementation, and programming. *Distributed Computing* 9, 1 (1995), 37–49.

[8] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. [n.d.]. Cure: Strong semantics meets high availability and low latency. In *Proceedings of the 36th International Conference on Distributed Computing Systems (ICDCS'2016).* 405–414.

[9] Aphyr. 2013. *Jepsen: MongoDB.* https://aphyr.com/posts/284-call-me-maybe-mongodb

[10] Aphyr. 2015. *Jepsen: MongoDB stale reads.* https://aphyr.com/posts/322-jepsen-mongodb-stale-reads

[11] Hagit Attiya and Jennifer L. Welch. 1994. Sequential Consistency versus Linearizability. *ACM Trans. Comput. Syst.* 12, 2 (May 1994), 91–122. https://doi.org/10.1145/176575.176576

[12] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. [n.d.]. On Verifying Causal Consistency. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL'2017).* 626–638.

[13] Eric A. Brewer. 2000. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '00).* ACM, New York, NY, USA, 7. https://doi.org/10.1145/343477.343502

[14] Jerzy Brzezinski, C. Sobaniec, and Dariusz Wawrzyniak. 2004. From session causality to causal consistency. In *Proceedings of the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing (EUROMICRO-PDP '00).* 152– 158. https://doi.org/10.1109/EMPDP.2004.1271440

[15] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1-2 (Oct. 2014), 1–150. https://doi.org/10.1561/2500000011

[16] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. 2014. GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14).* Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2670983

[17] Phillip Gibbons and Ephraim Korach. 1999. Testing Shared Memories. *SIAM J. Comput.* 26 (10 1999).

[18] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News* 33, 2 (2002), 51.

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

[20] Kyle Kingsbury. 2017. *Jepsen Testing of MongoDB 3.4.0-rc3.* https://jepsen.io/analyses/mongodb-3-4-0-rc3

[21] Kyle Kingsbury. 2020. *Jepsen Testing of MongoDB 4.2.6.* https://jepsen.io/analyses/mongodb-4.2.6

[22] Sandeep S Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical physical clocks. In *International Conference on Principles of Distributed Systems.* 17–32.

[23] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.

[24] Leslie Lamport. 2002. *Specifying Systems: The TLA$^+$ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[25] Richard J Lipton and Jonathan S Sandberg. 1988. *PRAM: A scalable shared memory.* Technical Report.

[26] Nancy A. Lynch. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[27] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14).* USENIX Association, 305–320.

[28] Kit Patella. 2018. *Jepsen Testing of MongoDB 3.6.4.* https://jepsen.io/analyses/mongodb-3-6-4

[29] Matthieu Perrin, Achour Mostefaoui, and Claude Jard. [n.d.]. Causal Consistency: Beyond Memory. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '16).* Article 26. https://doi.org/10.1145/2851141.2851170

[30] William Schultz, Tess Avitabile, and Alyson Cabral. 2019. Tunable Consistency in MongoDB. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2071–2081.

[31] The MongoDB Team. 2017. *MongoDB 3.6.0-rc0 is released.* https://www.mongodb.com/blog/post/mongodb-360-rc0-is-released

[32] Misha Tyulenev, Andy Schwerin, Asya Kamsky, Randolph Tan, Alyson Cabral, and Jack Mulrow. [n.d.]. Implementation of Cluster-Wide Logical Clock and Causal Consistency in MongoDB. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD'2019).* 636–650.

[33] Hengfeng Wei, Yu Huang, Jiannong Cao, Xiaoxing Ma, and Jian Lu. 2013. Verifying Pipelined-RAM Consistency over Read/Write Traces of Data Replicas. *IEEE Transactions on Parallel and Distributed Systems* 27 (02 2013). https://doi.org/10.1109/TPDS.2015.2453985

[34] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model Checking TLA$^+$ Specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME '99).* 54–66. https://doi.org/10.1007/3-540-48153-2_6