Ouyang HR, Wei HF, Li HX *et al.* Checking causal consistency of mongodb. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 36(6): 1–19 Nov. 2021. DOI 10.1007/s11390-021-1662-0

Checking Causal Consistency of MongoDB

Hong-Rong Ouyang[1], Heng-Feng Wei[1,2,*], *Member, CCF*, Hai-Xiang Li[3,*], *Member, CCF*
An-Qun Pan[3], *Member, CCF*, and Yu Huang[1], *Member, CCF*

[1] *State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China*
[2] *Software Institute, Nanjing University, Nanjing 210093, China*
[3] *TDSQL Team of Technology and Engineering Group (TEG) of Tencent, Tencent Inc., Shenzhen 518054, China*

E-mail: mf20330056@smail.nju.edu.cn; hfwei@nju.edu.cn; blueseali@tencent.com; aaronpan@tencent.com
yuhuang@nju.edu.cn

Received July 15, 2021; accepted October 14, 2021.

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 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 propose a more thorough design of Jepsen testing of causal consistency of MongoDB. Specifically, we fully implement the causal consistency checking algorithms proposed by Bouajjani *et al.* and test MongoDB against three well-known variants of causal consistency, namely CC, CCv, and CM, under various scenarios including node failures, data movement, and network partitions. In addition, we develop formal specifications of causal consistency and their checking algorithms in TLA$^+$, and verify them using the TLC model checker. We also explain how TLA$^+$ specification can be related to Jepsen testing.

Keywords MongoDB, casual consistency, Jepsen, consistency checking, TLA$^+$

1 Introduction

MongoDB is a general-purpose, document-oriented distributed NoSQL database①. 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 [1].

MongoDB achieves scalability by partitioning the data into shards and fault-tolerance by replicating each shard across a set of nodes [2]. 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 Fig.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.

According to the PACELC theorem [3], an extension to the CAP theorem [4, 5], 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,

Regular Paper
Special Section on Software Systems 2021
This work was supported by the CCF-Tencent Open Fund (CCF-Tencent RAGR20200124) and the National Natural Science Foundation of China under Grant Nos. 61702253 and 61772258.
*Corresponding Author
①MongoDB. https://www.mongodb.com/, Oct. 2021.

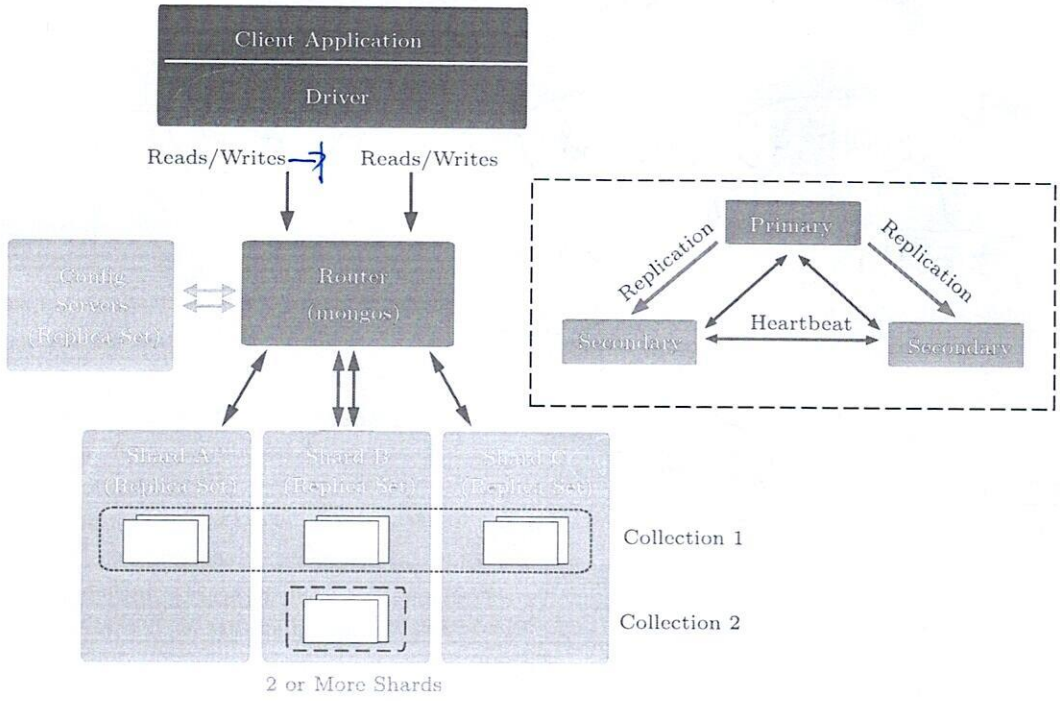


Fig.1. MongoDB deployment as a sharded cluster.

MongoDB introduced causal consistency②. It provides clients with session guarantees including read-your-writes, monotonic-reads, monotonic-writes, and writes-follow-reads [6]③. As the Jepsen team④ denoted, MongoDB is one of the first commercial databases that implement causal consistency [2].

Being a production database, MongoDB's implementation of causal consistency requires multi-dimensional evaluation criteria on performance, scalability, and security [2]. It combines several research ideas, including hybrid logical clocks [7], explicit dependency tracking [8, 9], Raft-based replication consensus protocol [10], 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 black-box testing against MongoDB 3.6.4 and 4.0.0-rc1. The team designed test cases that characterize 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 algorithm proposed by Bouajjani *et al.* [11] to check whether these histories satisfy causal consistency.

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, the drawbacks are as follows.

• There are several variants of causal consistency, including causal consistency (CC) [12, 13], causal memory (CM) [14], and causal convergence (CCv) [13]. Not all of them are comparable [13]. However, the official Jepsen testing did not clearly specify which causal consistency variant it tested against 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 [15], 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

②MongoDB 3.6.0-rc0. https://www.mongodb.com/blog/post/mongodb-360-rc0-is-released, Oct. 2021.

③Causal Consistency. https://docs.mongodb.com/manual/core/causal-consistency-read-write-concerns/, Oct. 2021.

④Jepsen. https://jepsen.io/, Oct. 2021.

Jepsen testing preset 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 [11].

• 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.

In this work, we propose a more thorough design of Jepsen testing of the causal consistency protocol of MongoDB[5]. Specifically, our contributions are as follows.

• We consider three well-known variants of causal consistency, following the formal specification given in [11].

• We generate the most general operation sequences for clients, without any restrictions on keys.

• We fully implement the "bad patterns" based causal consistency checking algorithm proposed by Bouajjani *et al.* in [11].

• We design more testing scenarios, covering network partitions, node failures, and data movement among shards.

Our preliminary experimental results confirm the claim in MongoDB's documentation that in the presence of node failures or network partitions, causal consistency is guaranteed only for reads with `majority readConcern` (explained shortly in Subsection 2.2) and writes with `majority writeConcern`.

This is an extended version of our conference paper[16] of the same title. In this version, we develop the formal specifications of three causal consistency variants, namly CC, CCv, and CM, and the "bad patterns" based checking algorithms in TLA$^+$. We also verify them using the TLC model checker. The model checking results confirm, though on test cases of relatively small scales, the correctness of the checking algorithms. We also explain how TLA$^+$ specification can be further related to Jepsen testing in Subsection 5.5.

The rest of the paper is organized as follows. Section 2 provides preliminaries on causal consistency, the Jepsen testing framework, and TLA$^+$. 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 shows the formal specifications of causal consistency and checking algorithms in TLA$^+$ and the model checking results. Section 6 discusses related work. Section 7 concludes the paper.

2 Preliminaries

2.1 Causal Consistency: Informal Introduction

Causal consistency guarantees that all clients agree on the relative ordering of causally related operations[14,17]. However, operations that are not causally related may be observed in different orders by different clients. We informally explain causal consistency in the classic "Lost-Ring" example[18] (see Fig.2). Alice first posts that she has lost her ring. After a while, she posts that she has found it. Bob sees Alice's posts, and comments "Glad to hear it". We say that there is a read-from dependency from Alice's second post to Bob's get operation, and a session dependency from

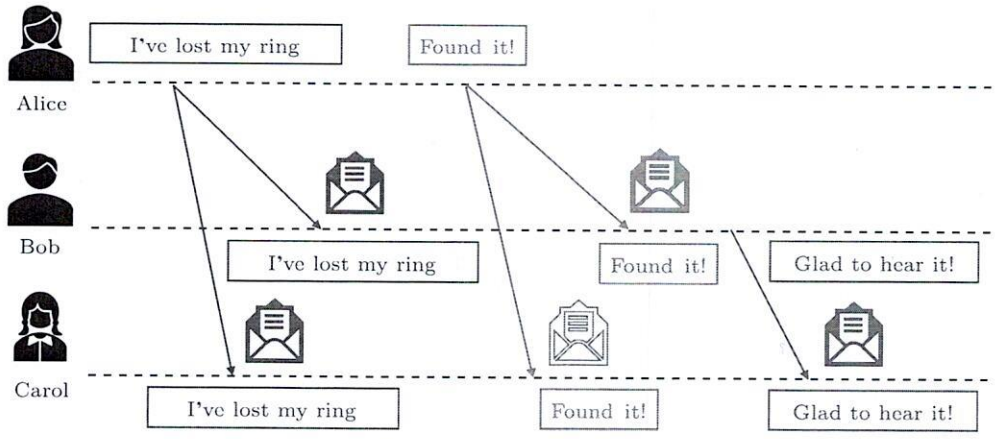


Fig.2. "Lost-Ring" example for causal consistency.

[5]The project can be found at https://github.com/Tsunaou/Checking-Causal-Consistency-of-MongoDB, Oct. 2021.

Bob's get operation to his own comment. By transitivity, Bob's comment causally depends on Alice's second post. Thus, when Carol sees Bob's comment, she should also see Alice's second post. Otherwise, she would be quite confused, mistakenly thinking that Bob is glad to hear that Alice has lost her ring.

2.2 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 [1]. `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, 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 `w1 writeConcern` can also provide causal consistency.

2.3 Causal Consistency: Formal Specification

We review the formal specification of causal consistency with respect to read-write registers, following [11].

2.3.1 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., key), with input or output values from $\mathbb{V}$.

2.3.2 Histories

We model the interactions between clients and a distributed database maintaining replicated read/write registers by histories.

Definition 1 (Histories). *A history $h = (O, \mathtt{PO}, \ell)$ is the poset (partial-ordered set) $(O, \mathtt{PO})$ labeled by $\mathbb{M} \times \mathbb{V} \times \mathbb{V}$, where*

- *O is set of operation identifiers, or simply operations; we use R and W to denote the set of read and write operations, respectively;*
- *$\mathtt{PO}$ is a union of total orders among operations called program order; for $o_1, o_2 \in O$, $o_1 <_{\mathtt{PO}} o_2$ 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 $\mathtt{var}(o) = x$ and $\mathtt{val}(o) = v$.

Let $\rho = (O, <, \ell)$ be an $\mathbb{M} \times \mathbb{V} \times \mathbb{V}$ labeled poset and $O' \subseteq O$ be a set. $\rho\{O'\}$ is the labeled poset in which only the return values of the operations in O' are 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 for all $o \in O'$, $\ell'(o) = \ell(o)$, and for all $o' \in O \setminus O'$, $\ell'(o') = (m, arg)$ if $\ell(o) = (m, arg, rv)$. We denote $\rho\{O'\}$ by $\rho\{o\}$ if $O' = \{o\}$.

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 the set O. Formally, $\rho' \preceq \rho$ if $<' \subseteq <$ and for all $o \in O$, $\ell'(o) = \ell(o)$ or $\ell'(o) = (m, arg)$ if $\ell(o) = (m, arg, rv)$.

2.3.3 Sequential Semantics

The consistency of replicated read-write registers is defined with respect to the sequential semantics of read-write registers. Intuitively, in any operation sequence on read-write registers, an 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 $S_{\mathtt{RW}}$ of read-write registers is the smallest set of sequences labeled by $\mathbb{M} \times \mathbb{V} \times \mathbb{V}$ satisfying

- $\epsilon \in S_{\mathtt{RW}}$, where ϵ is the empty sequence;
- if $\rho \in S_{\mathtt{RW}}$, then $\rho \cdot \mathsf{wr}(x, v) \in S_{\mathtt{RW}}$[6];
- if $\rho \in S_{\mathtt{RW}}$ contains no writes on x, then $\rho \cdot \mathsf{rd}(x) \triangleright 0 \in S_{\mathtt{RW}}$;
- if $\rho \in S_{\mathtt{RW}}$ and the last write in ρ on register x is $\mathsf{wr}(x, v)$, then $\rho \cdot \mathsf{rd}(x) \triangleright v \in S_{\mathtt{RW}}$.

[6] The symbol $\cdot$ means the connection between operations.

2.3.4 Causal Consistency

Following [11], we consider three well-known variants of causal consistency, namely CC (Causal Consistency), CCv (Causal Consistency Convergence), and CM (Causal Memory). A history is CC if there exists a causal order that explains the return value of each operation.

Definition 2 (Causal Consistency). *A history* $h = (O, \mathtt{PO}, \ell)$ *is* CC *with respect to specification* $S_{\mathtt{RW}}$ *if there exists a strict partial order* $co \subseteq O \times O$ *called the causal order such that for each operation* $o \in O$*, there exists a sequence* $\rho_o \in S_{\mathtt{RW}}$ *satisfying*

$$\mathsf{AxCausal} \triangleq \mathtt{PO} \subseteq co,$$
$$\mathsf{AxCausalValue} \triangleq (co^{-1}(o), co, \ell)\{o\} \preceq \rho_o.$$

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

CCv ensures eventual convergence via a total arbitration order.

Definition 3 (Causal Convergence). *A history* $h = (O, \mathtt{PO}, \ell)$ *is* CCv *with respect to specification* $S_{\mathtt{RW}}$ *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 sequence* $\rho_o \in S_{\mathtt{RW}}$ *satisfying*

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

CM requires each client to be consistent with respect to the returned values it has observed before.

Definition 4 (Causal Memory). *A history* $h = (O, \mathtt{PO}, \ell)$ *is* CM *with respect to specification* $S_{\mathtt{RW}}$ *if there exists a strict partial order* $co \subseteq O \times O$ *called the causal order such that for each operation* $o \in O$*, there exists a sequence* $\rho_o \in S_{\mathtt{RW}}$ *satisfying*

$$\mathsf{AxCausal} \triangleq \mathtt{PO} \subseteq co,$$
$$\mathsf{AxCausalSeq} \triangleq (co^{-1}(o), co, \ell)\{\mathtt{PO}^{-1}(o)\} \preceq \rho_o.$$

Here $\mathtt{PO}^{-1}(o) \triangleq \{o' \mid o' \leqslant_{\mathtt{PO}} o\}$.

2.4 Causal Consistency Checking

The general decision problem of checking whether a history over read-write registers is causally consistent is NP-complete [11]. However, for differentiated histories in which the values written to the same register are distinct, it is polynomial time [11]. 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" [11]. Each causal consistency variant can be precisely characterized by lacking 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 on operations.

Definition 5 (Read-From Relation). *The read-from relation* $\mathtt{RF} \subseteq W \times R$ *associates a read with the write from which it obtains the value. Formally,*

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

Definition 6 (Causal Order). *The casual order* $\mathtt{CO} \subseteq O \times O$ *is defined as the transitive closure of program order and read-from relation. Formally,*

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

Definition 7 (Conflict Relation). *The conflict relation* $\mathtt{CF} \subseteq W \times W$ *orders two writes on the same register according to a third read operation. Formally,*

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

Example 1. Consider the history h *of Fig.3. Since* $\mathsf{wr}(x,2) <_{\mathtt{RF}} \mathsf{rd}(x) \triangleright 2$ *and* $\mathsf{rd}(x) \triangleright 2 <_{\mathtt{PO}} \mathsf{rd}(x) \triangleright 1$*, we have* $\mathsf{wr}(x,2) <_{\mathtt{CO}} \mathsf{rd}(x) \triangleright 1$*. In addition,* $\mathsf{wr}(x,1) <_{\mathtt{RF}} \mathsf{rd}(x) \triangleright 1$*. Thus,* $\mathsf{wr}(x,2) <_{\mathtt{CF}} \mathsf{wr}(x,1)$.

Definition 8 (Happened-Before Relation). *For each operation* $o \in O$*, the happened-before relation* $\mathtt{HB}_o \subseteq O \times O$ *of* o *is the smallest transitive relation satisfying that* $\mathtt{CO}_{|\mathtt{CO}^{-1}(o)} \subseteq \mathtt{HB}_o$ *and*

$$\forall w, w' \in W.\ (w, w') \in \mathtt{HB}_o \iff \exists r' \in R.\ \big(r' \leqslant_{\mathtt{PO}} o \wedge w' <_{\mathtt{RF}} r' \wedge \mathtt{var}(w) = \mathtt{var}(r') \wedge (w, r') \in \mathtt{HB}_o\big).$$

Here $\mathtt{CO}_{|\mathtt{CO}^{-1}(o)}$ *is the relation* CO *restricted on the set* $co^{-1}(o)$.

Example 2. Consider the history h *of Fig.4. Since* $\mathsf{wr}(x,1) <_{\mathtt{PO}} \mathsf{wr}(y,1) <_{\mathtt{RF}} \mathsf{rd}(y) \triangleright 1$ *and* $\mathsf{rd}(y) \triangleright 1 <_{\mathtt{PO}} \mathsf{rd}(x) \triangleright 2$*, we have* $\mathsf{wr}(x,1) <_{\mathtt{CO}} \mathsf{rd}(x) \triangleright 2$*. In addition, for operation* $\mathsf{rd}(x) \triangleright 2$*,* $\mathtt{CO}^{-1}(\mathsf{rd}(x) \triangleright 2) = \mathtt{CO}$*, therefore we have* $\mathtt{CO} \subseteq \mathtt{HB}_{\mathsf{rd}(x) \triangleright 2}$*. And since* $\mathsf{wr}(x,2) <_{\mathtt{RF}} \mathsf{rd}(x) \triangleright 2$ *and* $\mathsf{wr}(x,1) <_{\mathtt{HB}_{\mathsf{rd}(x) \triangleright 2}} \mathsf{rd}(x) \triangleright 2$*, we have* $\mathsf{wr}(x,1) <_{\mathtt{HB}_{\mathsf{rd}(x) \triangleright 2}} \mathsf{wr}(x,2)$*. For the transitivity, we can also get* $\mathsf{wr}(z,1) <_{\mathtt{HB}_{\mathsf{rd}(x) \triangleright 2}} \mathsf{rd}(z) \triangleright 0$.

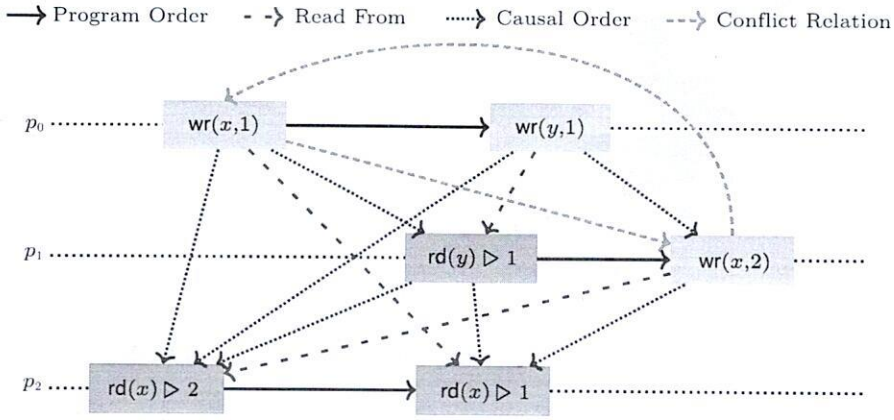


Fig.3. A history h that is not CCv. (The arrows for CO that are implied by transitivity are not shown.)

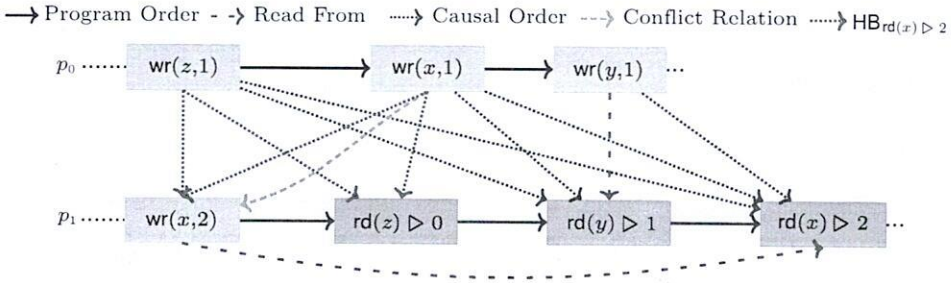


Fig.4. A history h that is not CM. (The arrows for CO that are implied by transitivity are not shown.)

The following theorem characterizes CC , CCv, and CM in terms of bad patterns defined in Table 1.

Theorem 1 [11]. *A history h is* CC *if and only if h does not exhibit any bad patterns of CyclicCO, WriteCOInitRead, ThinAirRead or WriteCORead.*

A history h is CCv *if and only if it is* CC *and does not exhibit any bad patterns of CyclicHF.*

A history h is CM *if and only if it is* CC *and does not exhibit any bad patterns of WriteHBInitRead or CyclicHB.*

Example 3. Consider the history h of Fig.3. 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*.

Example 4. Consider the history h of Fig.4. It is not CM. Since we have $\mathsf{wr}(z,1) <_{\mathsf{HB}_{\mathsf{rd}(x)\triangleright 2}} \mathsf{rd}(z) \triangleright 0$ and $\mathsf{rd}(z) \triangleright 0 <_{\mathsf{PO}} \mathsf{rd}(x) \triangleright 2$, it exhibits the bad pattern WriteHBInitRead.

2.5 Jepsen

Jepsen[7] is a library for black-box testing of distributed systems. A typical Jepsen testing of a dis-

Table 1. Definitions of Bad Patterns [11]

Bad Pattern	Description
CyclicCO	$\mathtt{PO} \cup \mathtt{RF}$ is cyclic
ThinAirRead	$\exists r \in R.\ \mathtt{val}(r) \neq 0 \wedge (\nexists w \in W.\ w <_{\mathtt{RF}} r)$
WriteCOInitRead	$\exists r \in R, w \in W.\ w <_{\mathtt{CO}} r \wedge \mathtt{var}(w) = \mathtt{var}(r) \wedge \mathtt{val}(r) = 0$
WriteCORead	$\exists w_1, w_2 \in W, r_1 \in R.\ \mathtt{var}(w_1) = \mathtt{var}(w_2) \wedge w_1 <_{\mathtt{CO}} w_2 <_{\mathtt{CO}} r_1 \wedge w_1 <_{\mathtt{RF}} r_1$
CyclicCF	$\mathtt{CF} \cup \mathtt{CO}$ is cyclic
WriteHBInitRead	$\exists o \in O, r \in R, w \in W.\ r \leqslant_{\mathtt{PO}} o \wedge w <_{\mathtt{HB}_o} r \wedge \mathtt{var}(w) = \mathtt{var}(r) \wedge \mathtt{val}(r) = 0$
CyclicHB	$\exists o \in O.\ \mathtt{HB}_o$ is cyclic

[7] Jepsen Library. https://github.com/jepsen-io/jepsen, Oct. 2021.

tributed 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 nemeses continuously inject faults or rare events (such as data movement among shards) into the database deployment.

2.6 TLA$^+$

TLA$^+$ is a high-level formal specification language developed by Lamport [19]. It was designed for modeling and reasoning about programs and systems, especially concurrent and distributed ones.

TLA$^+$ is based on TLA, the Temporal Logic of Actions [20]. With TLA, a system can be modeled as a state machine which is described by its initial states and actions. Since we focus on the specification of causal consistency, we omit the temporal operators in TLA$^+$ here.

TLA$^+$ combines TLA with first-order logic and Zermelo-Fraenkel set theory. Table 2 summarizes the (non-temporal) operators that we use in [21]. Interested readers are referred to the complete version of Summary of TLA$^+$⑧.

A specification in TLA$^+$ consists of modules. In a module, we can declare constants (CONSTANTS) and variables (VARIABLES), and define operators like $Op(p_1, \cdots, p_n) \triangleq exp$. We can also import the declarations, definitions, and operators from other modules $M_1, \ldots, M_n$, by writing EXTENDS $M_1, \cdots, M_n$ in M.

TLC is an explicit-state model checker for TLA$^+$ [22]. It verifies the TLA$^+$ specifications by exploring the whole state space of finite-state instances of them. In this paper, we use TLC only to evaluate constant expressions.

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, 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 Official Jepsen Testing

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

3.1.1 Specification

The Jepsen team claimed that they have tested MongoDB against causal consistency⑨. However, they

Table 2. Summary of TLA$^+$ Operators Used in This Paper

Category	Operator	Meaning
Set	SUBSET S	Powerset of S
	UNION S	Union of all elements of S
	$\{e : x \in S\}$	Set of elements e such that x is in S
	$\{x \in S : p\}$	Set of elements x in S satisfying p
Function	DOMAIN f	Domain of function f
	$f[e]$	Function application
	$[x \in S \mapsto e]$	Function f such that $f[x] = e$ for $x \in S$
Record	$e.h$	h-field of record e
	$[h_1 \mapsto e_1, \cdots, h_n \mapsto e_n]$	record whose h_i field is e_i
	$[h_1 : S_1, \cdots, h_n : S_n]$	Set of all records with h_i field in S_i
Tuple	$e[i]$	The i-th component of tuple e
	$\langle e_1, \cdots, e_n \rangle$	The n-tuple whose i-th component is e_i
Sequence	$SubSeq(s, m, n)$	Sequence $\langle s[m], s[m+1], \ldots, s[n] \rangle$
	$Range(s)$	Set of elements of sequence s

⑧Leslie Lamport. Summary of TLA$^+$. http://lamport.azurewebsites.net/tla/summary-standalone.pdf, May 2021.

⑨Jepsen Testing of MongoDB 3.6.4. https://jepsen.io/analyses/mongodb-3-6-4, Oct. 2021.

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 five operations as follows:

$$(r, w1, r, w2, r),$$

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 is sufficient 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 five-node deployment of MongoDB, the network will be split into two disconnected parts: one (denoted P_1) consists of two nodes, one of which is the original primary node, and the other (denoted P_2) consists of three nodes. Since three nodes in P_2 constitute a majority (of five nodes), one of them will be 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 3, we improve 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 [11].

3.2.2 Test Case Generation

In our design, the generator generates an arbitrary differentiated operation sequence for each client using YCSB [23] [10]. 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 [11].

3.2.4 Testing Scenarios

Besides partition-random-halves in the official Jepsen testing, we introduce two additional nemeses called node-failure and data-mover. The node-failure nemesis randomly selects a database node, suspends it for a while, and then recovers it. This may trigger leader election. The data-mover nemesis periodically

Table 3. Comparison between the Official Jepsen Testing and Our Design

	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 [11]
Testing Scenarios	Network partition	Network partition, data movement, node failure

[10] YCSB. https://github.com/brianfrankcooper/YCSB, Oct. 2021.

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 implement the checking algorithms of [11] and check histories produced by MongoDB 4.2.3 against CC, CM, and CCv. We use the Jepsen testing framework of version 0.1.17[11]. Table 4 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 five 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 5 shows the experimental results of checking causal consistency of MongoDB.

4.2.1 *Causal Consistency Checking*

The preliminary experimental results confirm the claim in MongoDB's documentation 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`.

4.2.2 *Performance*

Fig.5 demonstrates the performance of checking whether histories satisfy causal consistency. According to [11], 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 Fig.5 exhibit such a substantial performance gap.

4.3 Unexpected ThinAirRead Bad Patterns

We observe some unexpected ThinAirRead bad patterns in our preliminary evaluations, marked $\otimes$ in Table 5. They appear in some histories that are produced without nemesis and consist of reads with `majority readConcern` and writes with `majority writeConcern`. Table 6 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 [11] 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.

Table 4. Hardware Configurations

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

[11] Jepsen Library 0.1.17. https://github.com/jepsen-io/jepsen/tree/0.1.17, Oct. 2021.

Table 5. Experimental Results of Causal Consistency Checking of MongoDB

Number 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	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
1 000	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1 100	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
1 200	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
1 300	⊗	⊗	⊗	✓	✓	✓	✓	✓	✓	✓	✓	✓
1 400	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1 500	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1 600	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
1 700	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
1 800	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
1 900	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓
2 000	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
2 500	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
3 000	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
3 500	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
4 000	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
4 500	⊗	⊗	⊗	✗	✗	✗	✓	✓	✓	✓	✓	✓
5 000	✓	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	✓

Note: ✓: satisfaction; ✗: violation; ⊗: unexpected ThinAirRead bad patterns discussed in Subsection 4.3.

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 TLA$^+$ Specification of Causal Consistency and Checking Algorithms

In this section, we formally specify both the specification of causal consistency and the "bad patterns" based checking algorithms in [11] in TLA$^+$, and verify them using TLC model checker. We explain how TLA$^+$ specification can be further related to Jepsen testing in Subsection 5.5. Table 7 summarizes the auxiliary operators we defined in this paper.

5.1 TLA$^+$ Specification of Causal Consistency

We follow the way how the specification of causal consistency is developed in Subsection 2.3.

5.1.1 *Replicated Objects*

In module *ReplicatedObjects* (Fig.6(a)), we assume single-character keys and take values from natural numbers for read/write registers. Following [11], we set the initial value of each key to 0. We assume that each operation is associated with a unique identifier.

5.1.2 *History*

We define *Session* and *History* in module *History* (Fig.6(b)). A session $s \in Session$ is a sequence of operations issued by the same client, and a history $h \in History$ consists of a set of sessions. The program order $PO(h)$ of a history h is a union of strict total

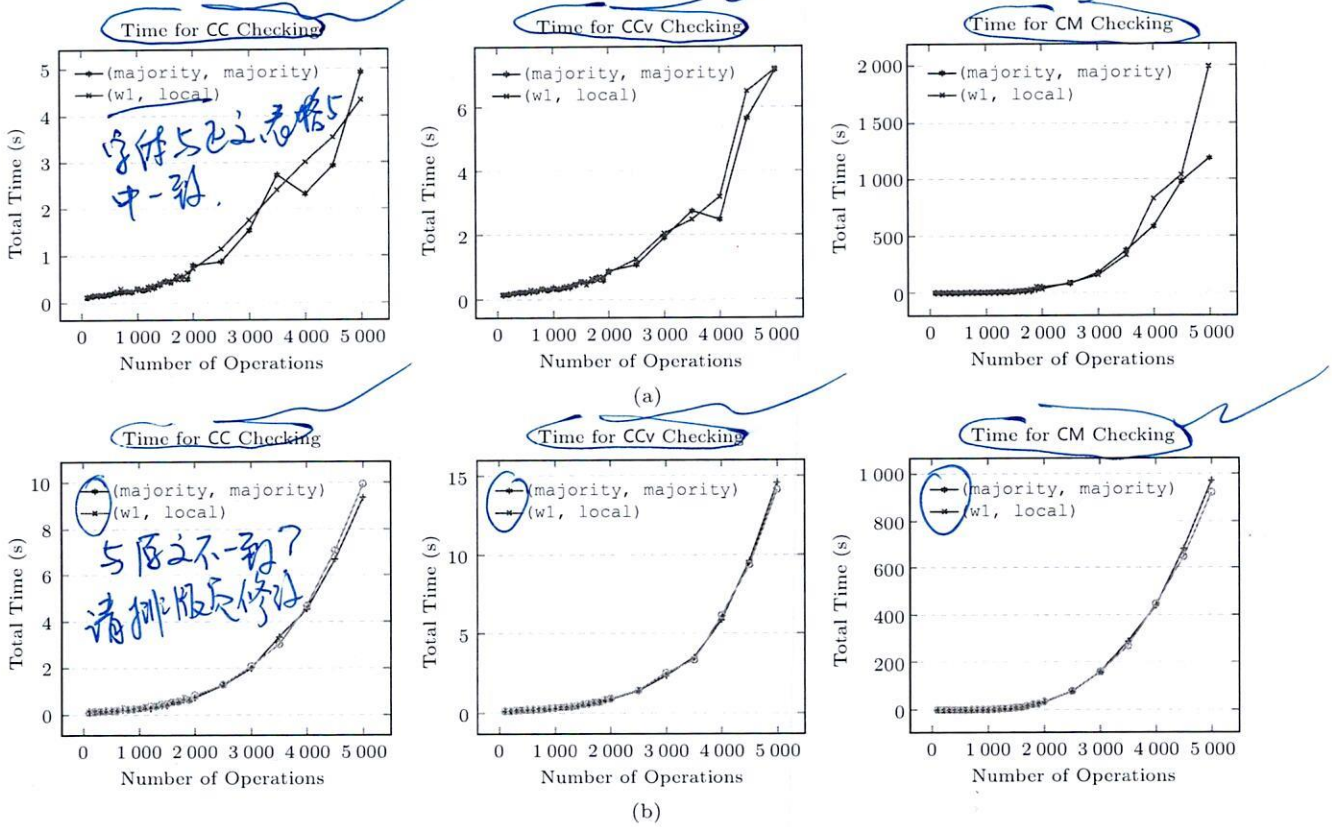


Fig.5. Time of checking whether histories satisfy causal consistency. (a) Time for causal consistency checking with nemesis. (b) Time for causal consistency checking without nemesis.

Table 6. 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	No Exception	majority
1266	rd(85) ▷ 5	No Exception	majority
1336	rd(20) ▷ 5	No Exception	majority
3756	rd(20) ▷ 5	No Exception	majority

orders among operations in the same session.

5.1.3 Sequential Semantics

The operator $RWRegSemantics(seq, o)$ in module *RWRegSemantics* (Fig.6(c)) checks whether the operation o is legal with respect to the sequential semantics when it is appended to the operation sequence seq.

5.1.4 Causal Consistency

The module *Axioms* (Fig.7(a)) defines the axioms used in the specification of variants of causal consistency, which are shown in the module *CausalDefinition* (Fig.7(b)).

The axiom *AxCausalValue* requires that for an operation o, there exists a linear extension seq of the causal order co when restricted on the set of operations preceding o such that $RWRegSemantics(seq, o)$ is satisfied.

The axiom *AxCausalArb* requires that for an operation o, the arbitration order arb when restricted on the set of operations preceding o in causal order co is legal with respect to the sequential semantics.

The axiom *AxCausalSeq* requires that for an operation o, there exists a linear extension seq of the causal order co when restricted on the set of operations preced-

Table 7. Summary of Auxiliary Operators Defined in this Paper

Operator	Meaning
$PreSeq(s, e)$	The prefix of the sequence s ending with element e (which is unique in s)
$Seq2Rel(s)$	Convert a sequence s into a strict total order relation
$SelectSeq(s, Test(\_))$	The subsequence of s consisting of all elements $s[i]$ such that $Test(s[i])$ is true
$R\|S$	Restriction of relation R on set S
$AllLinearExtensions(R, S)$	All possible linear extensions of the partial order R defined on the set S
$AnyLinearExtension(R, S)$	An arbitrary linear extension of the partial order R defined on the set S
$Respect(R, T)$	Does the relation R respect relation T?
$TC(R)$	Transitive closure of the relation R
$POPast(h, o)$	The set of operations that precede $o \in Operation$ in program order in history $h \in History$ (including o)
$StrictCausalPast(co, o)$	The set of operations that precede $o \in Operation$ in causal order co
$CausalPast(co, o)$	The set of operations that precede $o \in Operation$ in causal order co (including o)
$StrictCausalHist(co, o)$	The restriction of causal order co to the operations in $StrictCausalPast(co, o)$
$CausalHist(co, o)$	The restriction of causal order co to the operations in $CausalPast(co, o)$
$StrictCausalArb(co, arb, o)$	The restriction of arbitration arb to the operations in $StrictCausalPast(co, o)$
$CausalArb(co, arb, o)$	The restriction of arbitration arb to the operations in $CausalPast(co, o)$
$Ops(h)$	The set of all operations in history $h \in History$
$ReadOps(h)$	The set of all read operations in history $h \in History$
$ReadOpsOnKey(h)$	The set of all read operations on key $k \in Key$ in history $h \in History$
$WriteOps(h)$	The set of all write operations in history $h \in History$
$WriteOpsOnKey(h, k)$	The set of all write operations on key $k \in Key$ in history $h \in History$
$KeyOf(h)$	The set of keys read or written in $h \in History$

```
MODULE ReplicatedObjects
Key ≜ Range("abcdefghijklmnopqrstuvwxyz")
Val ≜ Nat
InitVal ≜ 0
Oid ≜ Nat

Operation ≜ [type : {"read", "write"}, key : Key, val : Val, oid : Oid]
R(k, v, oid) ≜ [type ↦ "read", key ↦ k, val ↦ v, oid ↦ oid]
W(k, v, oid) ≜ [type ↦ "write", key ↦ k, val ↦ v, oid ↦ oid]
```

(a)

```
MODULE History
Session ≜ Seq(Operation)
History ≜ SUBSET Session
PO(h) ≜ UNION {Seq2Rel(s) : s ∈ h}
```

(b)

```
MODULE RWRegSemantics
RWRegSemantics(seq, o) ≜
  IF o.type = "write" THEN TRUE ELSE
  LET wseq ≜ SelectSeq(seq, LAMBDA op : op.type = "write" ∧ op.key = o.key)
      IN  IF wseq = ⟨⟩ THEN o.val = InitVal
          ELSE o.val = wseq[Len(wseq)].val
```

(c)

Fig.6. TLA$^+$ modules for replicated read-write registers. (a) TLA$^+$ module *ReplicatedObjects*. (b) TLA$^+$ module *History*. (c) TLA$^+$ module *RWRegSemantics*.

```
MODULE Axioms
AxCausalValue(co, o) ≜
  LET seqs ≜ AllLinearExtensions(StrictCausalHist(co, o), StrictCausalPast(co, o))
  IN  ∃ seq ∈ seqs : RWRegSemantics(seq, o)

AxCausalArb(co, arb, o) ≜
  LET seq ≜ AnyLinearExtension(StrictCausalArb(co, arb, o), StrictCausalPast(co, o))
  IN  RWRegSemantics(seq, o)

AxCausalSeq(h, co, o) ≜
  LET seqs ≜ AllLinearExtensions(CausalHist(co, o), CausalPast(co, o))
  IN  ∃ seq ∈ seqs : ∀ o2 ∈ POPast(h, o) :
        RWRegSemantics(PreSeq(seq, o2), o2)
```

(a)

```
MODULE CausalDefinition
CC(h) ≜  ∃ co ∈ SUBSET (Ops(h) × Ops(h)) :
           ∧ Respect(co, PO(h))                          AxCausal
           ∧ IsStrictPartialOrder(co, Ops(h))
           ∧ ∀ o ∈ Ops(h) : AxCausalValue(co, o)         AxCausalValue

CCv(h) ≜  ∃ co ∈ SUBSET (Ops(h) × Ops(h)) :
           ∧ ∃ arb ∈ SUBSET (Ops(h) × Ops(h)) :
               ∧ IsStrictPartialOrder(co, Ops(h))
               ∧ IsStrictTotalOrder(arb, Ops(h))
               ∧ Respect(co, PO(h))                      AxCausal
               ∧ Respect(arb, co)                        AxArb
               ∧ ∀ o ∈ Ops(h) : AxCausalArb(co, arb, o)  AxCausalArb

CM(h) ≜  ∃ co ∈ SUBSET (Ops(h) × Ops(h)) :
           ∧ IsStrictPartialOrder(co, Ops(h))
           ∧ Respect(co, PO(h))                          AxCausal
           ∧ ∀ o ∈ Ops(h) : AxCausalSeq(h, co, o)        AxCausalSeq
```

(b)

Fig. 7. TLA$^+$ modules for the definition of variants of causal consistenry. (a) TLA$^+$ module *Axioms*. (b) TLA$^+$ module *CausalDefinition*.

ing o in co such that for each operation $o2$ preceding o in program order, $RWRegSemantics(PreSeq(seq, o2), o2)$ is satisfied.

5.2 TLA$^+$ Specification of Causal Consistency Checking Algorithms

The module *Relations* (Fig.8) defines the relations including RF, CO, CF, and HB on the set of operations in histories. The module *BadPatterns* (Fig.9(a)) then defines all the bad patterns mentioned in Subsection 2.4. Finally, the module *Algorithm* (Fig.9(b)) specifies the "bad patterns" based checking algorithms for CC, CCv, and CM.

5.3 Optimizations

We observe that model checking histories against *CC*, *CCv*, or *CM* as defined in Fig.7(b) is prohibitively inefficient. In this subsection, we propose several optimizations, taking *CCv* as an example (see Fig.10).

5.3.1 *CCv*1: Rearranging Clauses

In *CCv*, we first enumerate all possible relations on *ops* as candidates for *co* and *arb*. In this way, for a history with n operations, the number of all possible combinations of *co* and *arb* is 2^{2n^2}. To eliminate undesired *co* candidates as early as possible, we move the two constraints $IsStrictPartialOrder(co, ops)$ and $Respect(co, PO(h))$ on *co* to the front, before enumerating *arb* (see *CCv*1 in Fig.10).

5.3.2 *CCv*2: Computing Linear Extensions of *co* As Candidates for *arb*

The axiom AxArb requires $co \subseteq arb$. Therefore, we can directly compute the linear extensions of *co* as candidates for *arb*, instead of enumerating all possible relations on *ops* (see *CCv*2 in Fig.10).

```
MODULE Relations
RF(h) ≜ {⟨w, r⟩ ∈ WriteOps(h) × ReadOps(h) : w.key = r.key ∧ w.val = r.val}
CO(h) ≜ TC(PO(h) ∪ RF(h))
CF(h) ≜ LET co ≜ CO(h)rf ≜ RF(h)
        IN {⟨w1, w2⟩ ∈ WriteOps(h) × WriteOps(h) :
            ≜ w1.key = w2.key
            ∧ w1.val ≠ w2.val
            ∧ ∃ r ∈ ReadOps(h) : ⟨w1, r⟩ ∈ co ∧ ⟨w2, r⟩ ∈ rf}
HBo(h, o) ≜ LET base ≜ CO(h) | CausalPast(co, o)
              RECURSIVE HBoRE(_)
              HBoRE(hbo) ≜
                LET update ≜ {
                      ⟨w1, w2⟩ ∈ WriteOps(h) × WriteOps(h) :
                        ≜ w1.key = w2.key
                        ∧ w1.val ≠ w2.val
                        ∧ ∃ r2 ∈ ReadOps OnKey(h, w2.key) :
                            ≜ r2.val = w2.val
                            ∧ ⟨w1, r2⟩ ∈ hbo
                            ∧ ∨ r2 = o ∨ ⟨r2, o⟩ ∈ PO(h)}
                    hbo2 ≜ update ∪ hbo
                IN IF hbo2 = hbo THEN hbo ELSE HBoRE(TC(hbo2))
            IN TC(HBoRE(base))
```

Fig.8. TLA+ module *Relations*.

```
MODULE BadPatterns
CyclicCO(h) ≜ Cyclic(PO(h) ∪ RF(h))
WriteCOInitRead(h) ≜ ∃ k ∈ KeyOf(h) : ∃ r ∈ ReadOpsOnKey(h, k),
                                       w ∈ WriteOpsOnKey(h, k) :
                       ≜ ⟨w, r⟩ ∈ CO(h) ∧ r.val = InitVal
ThinAirRead(h) ≜ ∃ k ∈ KeyOf(h) :
                   ∃ r ∈ ReadOpsOnKey(h, k) :
                     ≜ r.val ≠ InitVal
                     ∧ ∀ w ∈ WriteOpsOnKey(h, k) : ⟨w, r⟩ ∉ RF(h)
WriteCORead(h) ≜ ∃ k ∈ KeyOf(h) :
                   ∃ w1, w2 ∈ WriteOpsOnKey(h, k),
                     r1 ∈ ReadOpsOnKey(h, k) :
                     ≜ ⟨w1, w2⟩ ∈ CO(h) ∧ ⟨w2, r1⟩ ∈ CO(h)
                     ∧ ⟨w1, r1⟩ ∈ RF(h)
CyclicCF(h) ≜ Cyclic(CF(h) ∪ CO(h))
WriteHBInitRead(h) ≜ ∃ o ∈ Ops(h) :
                       ∃ r ∈ POPast(h, o) :
                         ≜ r.val = InitVal
                         ∧ LET writes ≜ WriteOpsOnKey(h, r.key)
                           IN ∃ w ∈ writes : ⟨w, r⟩ ∈ HBo(h, o)
CyclicHB(h) ≜ ∃ o ∈ Ops(h) : Cyclic(HBo(h, o))
```

(a)

```
MODULE Algorithm
CCAlg(h) ≜ ∧ ¬CyclicCO(h) ∧ ¬WriteCOInitRead(h)
           ∧ ¬ThinAirRead(h) ∧ ¬WriteCORead(h)
CCvAlg(h) ≜ ∧ CCAlg(h) ∧ ¬CyclicCF(h)
CMAlg(h) ≜ ∧ CCAlg(h) ∧ ¬WriteHBInitRead(h) ∧ ¬CyclicHB(h)
```

(b)

Fig.9. TLA+ modules for the "bad patterns" based checking algorithm of variants of causal consistenry. (a) TLA+ module *BadPatterns*. (b) TLA+ module *Algorithm*.

```
MODULE Optimization
CCv1(h) ≜ LET ops ≜ Ops(h)
          IN ∃ co ∈ SUBSET (ops × ops) :
               ∧ Respect(co, PO(h))
               ∧ IsStrictPartialOrder(co, ops)
               ∧ ∃ arb ∈ SUBSET (ops × ops) :
                    ∧ Respect(arb, co)
                    ∧ IsStrictTotalOrder(arb, ops)
                    ∧ ∀ o ∈ ops : AxCausalArb(co, arb, o)

CCv2(h) ≜
    LET ops ≜ Ops(h)
    IN ∃ co ∈ SUBSET (ops × ops) :
         ∧ Respect(co, PO(h))
         ∧ IsStrictPartialOrder(co, ops)
         ∧ ∃ arb ∈ {Seq2Rel(le) : le ∈ AllLinearExtensions(co, ops)} :
              ∧ ∀ o ∈ ops : AxCausalArb(co, arb, o)

CCv3(h) ≜
    LET ops ≜ Ops(h)
    IN ∃ co ∈ StrictPartialOrderSubset(ops) :
         ∧ Respect(co, PO(h))
         ∧ ∃ arb ∈ {Seq2Rel(le) : le ∈ AllLinearExtensions(co, ops)} :
              ∧ ∀ o ∈ ops : AxCausalArb(co, arb, o)
```

Fig.10. TLA$^+$ module *Optimization*.

5.3.3 *CCv3: Enumerating Strict Partial Order As Candidates for co*

In *CCv2*, we still need to enumerate all possible relations on *ops* as candidates for *co*, and then eliminate the ones that are not strict partial orders. In *CCv3* (Fig.10), we directly compute all possible strict partial orders on *ops*. To this end, we implement the efficient partial order enumeration algorithm of [24] in Python, and let TLC call it when necessary[12].

5.4 Model Checking Results

We verify the TLA$^+$ specification of causal consistency and their "bad patterns" based checking algorithms against five sample histories from [11] using the TLC model checker. The sample histories are described in TLA$^+$ in module *Samples* (Fig.11). It is quite easy

```
MODULE Samples
hasa ≜ ⟨W("x", 1, 1), R("x", 2, 2)⟩
hasb ≜ ⟨W("x", 2, 3), R("x", 1, 4)⟩
ha ≜ {hasa, hasb}  CM but not CCv

hbsa ≜ ⟨W("z", 1, 1), W("x", 1, 2), W("y", 1, 3)⟩
hbsb ≜ ⟨W("x", 2, 4), R("z", 0, 5), R("y", 1, 6), R("x", 2, 7)⟩
hb ≜ {hbsa, hbsb}  CCv but not CM

hcsa ≜ ⟨W("x", 1, 1)⟩
hcsb ≜ ⟨W("x", 2, 2), R("x", 1, 3), R("x", 2, 4)⟩
hc ≜ {hcsa, hcsb}  CC but not CM nor CCv

hdsa ≜ ⟨W("x", 1, 1), W("y", 2, 2), R("y", 2, 3)⟩
hdsb ≜ ⟨W("y", 1, 4), R("x", 1, 5), R("y", 1, 6)⟩
hd ≜ {hdsa, hdsb}  CC, CM, and CCv

hesa ≜ ⟨W("x", 1, 1), W("y", 1, 2)⟩
hesb ≜ ⟨R("y", 1, 3), W("x", 2, 4)⟩
hesc ≜ ⟨R("x", 2, 5), R("x", 1, 6)⟩
he ≜ {hesa, hesb, hesc}  not CC (nor CM, nor CCv)
```

Fig.11. TLA$^+$ module *Samples*.

[12] Technically, we need to wrap it in Java first.

to manually check them against each causal consistency variant.

As shown in Table 8, the "bad patterns" based checking algorithms meet their corresponding specifications as expected. It also confirms the satisfaction or violation of the sample histories. This demonstrates, though on test cases of relatively small scales, the correctness of the checking algorithms. Note that it takes much longer to check the history *hb* which consists of two sessions and seven operations directly against the specifications than to use the polynomial "bad patterns" based checking algorithms.

Table 9 shows the time for checking histories *ha*, *hb*, and *hd* against different versions of *CCv* proposed in Subsection 5.3. It demonstrates that each optimization is quite effective in reducing the checking time. Note that *CCv*3 is the only version that is feasible for history *hb* with seven operations.

5.5 Relating TLA$^+$ Specification to Jepsen Testing

As summarized in Fig.12, we have two TLA$^+$ specifications, one for causal consistency variants and the other for "bad patterns" based checking algorithms. We have also a Java implementation of these checking algorithms used in Jepsen testing of MongoDB. Now we explain how they can interact with each other.

On the one hand, utilizing TLC we are able to automatically generate as many histories as possible of various kinds from the TLA$^+$ specification of causal consistency variants. One of the most interesting kinds of histories are those satisfy or violate some or all causal consistency variants. They can be used as test oracles for both the specification and our Java implementation of the checking algorithms. On the other hand, it is convenient for MongoDB to generate arbitrarily long histories in real deployment. By checking them against both the TLA$^+$ specification and our Java implementation of the checking algorithms, we can gain more confidence in our implementation.

6 Related Work

6.1 Jepsen Testing of MongoDB

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

- In 2013, the team tested the election and data replication protocol of MongoDB 2.4.3[13]. It showed that acknowledged writes may be lost under network partitions at all consistency levels.
- In 2015, the team tested the single-document consistency of MongoDB 2.6.7[14]. 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.

Table 8. Model Checking Results on Sample Histories Defined in Fig.11

History	Number of Sessions	Number of Operations	Specifications						Checking Algorithms					
			CC		*CCv*		*CM*		*CCAlg*		*CCvAlg*		*CMAlg*	
			Result	Time (ms)	Result	Time (ms)	Result	Time (ms)	Result	Time (ms)	Result	Time (ms)	Result	Time (ms)
ha	2	4	✓	1 161	✗	1 155	✓	938	✓	898	✗	802	✓	1 140
hb	2	7	✓	83 089	✓	79 089	✗	82 930	✓	867	✓	990	✗	1 886
hc	2	4	✓	1 073	✗	836	✗	940	✓	950	✗	885	✗	1 321
hd	2	6	✓	2 326	✓	2 318	✓	2 296	✓	945	✓	951	✓	1 166
he	3	6	✗	2 620	✗	3 237	✗	2 673	✗	921	✗	769	✗	868

Note: ✓: satisfaction; ✗: violation.

Table 9. Time of Checking Histories Against Different Versions of *CCv*

	ha (4 Operations)	*hd* (6 Operations)	*hb* (7 Operations)
CCv	48 min 47 sec	> 24 hour	> 24 hour
*CCv*1	2 051 ms	20 hour 17 min	> 24 hour
*CCv*2	1 469 ms	1 min 25 sec	> 24 hour
*CCv*3	1 161 ms	2 326 ms	1 min 23 sec

[13] Jepsen: MongoDB. https://aphyr.com/posts/284-call-me-maybe-mongodb, Oct. 2021.
[14] Jepsen: MongoDB stale reads. https://aphyr.com/posts/322-jepsen-mongodb-stale-reads, Oct. 2021.

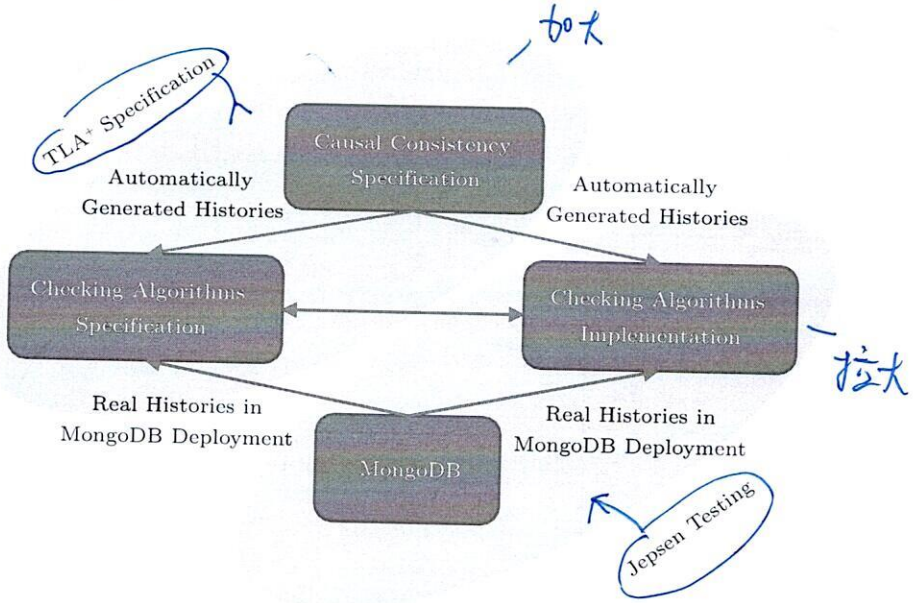


Fig.12. Relating TLA+ specification to Jepsen testing.

• In 2017, the team tested the v0 and v1 replication protocols of MongoDB 3.4.0-rc3[15]. 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, the team tested the causal consistency protocol of MongoDB 3.6.4. 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 identify several drawbacks of this testing in terms of specification, test case generation, implementation of causal consistency checking algorithms, and testing scenarios. We also propose a more thorough design of Jepsen testing of the causal consistency protocol of MongoDB.

• In 2020, the team tested the transactional consistency models of MongoDB 4.2.6[16]. It showed that MongoDB failed to preserve snapshot isolation, even for reads with `majority readConcern` and writes with `majority writeConcern`.

6.2 Consistency Checking Problem

Much work has been devoted to the problem of checking whether a given history satisfies a desirable consistency model. Gibbons and Korach [25] systematically studied the complexity of the checking problem against strong consistency models, including linearizability [26] and sequential consistency [27]. Regarding weak consistency models, Wei *et al.* [28] addressed the problem of checking PRAM consistency [29] 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.* addressed the problem of checking causal consistency [11]. 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 fully implement these efficient checking algorithms and utilize them to test the causal consistency protocol of MongoDB.

7 Conclusions

We proposed a thorough design of Jepsen testing of the causal consistency protocol of MongoDB. It strengthened the official Jepsen testing in 2018 in terms of specification, test case generation, implementation of

[15] Jepsen Testing of MongoDB 3.4.0-rc3. https://jepsen.io/analyses/mongodb-3-4-0-rc3, Oct. 2021.
[16] Jepsen Testing of MongoDB 4.2.6. https://jepsen.io/analyses/mongodb-4.2.6, Oct. 2021.

causal consistency checking algorithms, and testing scenarios. We conducted a preliminary evaluation of our design and more intensive experiments are needed. We also developed formal specifications of causal consistency and their checking algorithms in TLA$^+$. We will explore the issues discussed in Subsection 5.5 in future work.

We plan to improve the official Jepsen testing of the transaction protocols of MongoDB 4.2.6. 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$^+$, verify them using the TLC model checker, and develop mechanical correctness proofs for them using TLAPS[17].

References

[1] Schultz W, Avitabile T, Cabral A. Tunable consistency in MongoDB. *Proc. VLDB Endow.*, 2019, 12(12): 2071-2081. DOI: 10.14778/3352063.3352125.

[2] Tyulenev M, Schwerin A, Kamsky A, Tan R, Cabral A, Mulrow J. Implementation of cluster-wide logical clock and causal consistency in MongoDB. In *Proc. the 2019 International Conference on Management of Data*, June 30-July 5, 2019, pp.636-650. DOI: 10.1145/3299869.3314049.

[3] Abadi D. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 2012, 45(2): 37-42. DOI: 10.1109/MC.2012.33.

[4] Brewer E A. Towards robust distributed systems (abstract). In *Proc. the 19th Annual ACM Symposium on Principles of Distributed Computing*, July 2000, Article No. 7. DOI: 10.1145/343477.343502.

[5] Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 2002, 33(2): 51-59. DOI: 10.1145/564585.564601.

[6] Brzezinski J, Sobaniec C, Wawrzyniak D. From session causality to causal consistency. In *Proc. the 12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Feb. 2004, pp.152-158. DOI: 10.1109/EMPDP.2004.1271440.

[7] Kulkarni S S, Demirbas M, Madappa D, Avva B, Leone M. Logical physical clocks. In *Proc. the 18th International Conference on Principles of Distributed Systems*, Dec. 2014, pp.17-32. DOI: 10.1007/978-3-319-14472-6_2.

[8] Du J, Iorgulescu C, Roy A, Zwaenepoel W. GentleRain: Cheap and scalable causal consistency with physical clocks. In *Proc. the ACM Symposium on Cloud Computing*, Nov. 2014, Article No. 4. DOI: 10.1145/2670979.2670983.

[9] Akkoorath D D, Tomsic A Z, Bravo M, Li Z, Crain T, Bieniusa A, Preguiça N, Shapiro M. Cure: Strong semantics meets high availability and low latency. In *Proc. the 36th International Conference on Distributed Computing Systems*, June 2016, pp.405-414. DOI: 10.1109/ICDCS.2016.98.

[10] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In *Proc. the 2014 USENIX Conference on USENIX Annual Technical Conference*, June 2014, pp.305-320.

[11] Bouajjani A, Enea C, Guerraoui R, Hamza J. On verifying causal consistency. In *Proc. the 44th ACM Symposium on Principles of Programming Languages*, Jan. 2017, pp.626-638. DOI: 10.1145/3009837.3009888.

[12] Burckhardt S. Principles of eventual consistency. *Found. Trends Program. Lang.*, 2014, 1(1-2): 1-150. DOI: 10.1561/2500000011.

[13] Perrin M, Mostéfaoui A, Jard C. Causal consistency: Beyond memory. In *Proc. the 21st ACM Symposium on Principles and Practice of Parallel Programming*, March 2016, Article No. 26. DOI: 10.1145/2851141.2851170.

[14] Ahamad M, Neiger G, Burns J E, Kohli P, Hutto P W. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 1995, 9(1): 37-49. DOI: 10.1007/BF01784241.

[15] Lynch N A. Distributed Algorithms. Morgan Kaufmann Publishers Inc., 1996.

[16] Ouyang H R, Wei H F, Huang Y. Checking causal consistency of MongoDB. In *Proc. the 12th Asia-Pacific Symposium on Internetware*, Nov. 2020, pp.209-216. DOI: 10.1145/3457913.3457928.

[17] Lamport L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978, 21(7): 558-565. DOI: 10.1145/359545.359563.

[18] Lesani M, Bell C J, Chlipala A. Chapar: Certified causally consistent distributed key-value stores. In *Proc. the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2016, pp.357-370. DOI: 10.1145/2837614.2837622.

[19] Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers (1st edition). Addison-Wesley Professional, 2002.

[20] Lamport L. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 1994, 16(3): 872-923. DOI: 10.1145/177492.177726.

[21] Wei H F, Tang R Z, Huang Y, Lv J. Jupiter made abstract, and then refined. *Journal of Computer Science and Technology*, 2020, 35(6): 1343-1364. DOI: 10.1007/s11390-020-0516-0.

[22] Yu Y, Manolios P, Lamport L. Model checking TLA$^+$ specifications. In *Proc. the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, Sept. 1999, pp.54-66. DOI: 10.1007/3-540-48153-2_6.

[23] Cooper B F, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. In *Proc. the 1st ACM Symposium on Cloud Computing*, June 2010, pp.143-154. DOI: 10.1145/1807128.1807152.

[24] Bowles J, Caminati M B. A verified algorithm enumerating event structures. In *Proc. the 10th International Conference on Intelligent Computer Mathematics*, July 2017, pp.239-254. DOI: 10.1007/978-3-319-62075-6_17.

[25] Gibbons P, Korach E. Testing shared memories. *SIAM Journal on Computing*, 1997, 26(4): 1208-1244. DOI: 10.1137/S0097539794279614.

[17] TLA$^+$ Proof System (TLAPS). https://tla.msr-inria.inria.fr/tlaps/content/Home.html

[26] Herlihy M P, Wing J M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 1990, 12(3): 463-492. DOI: 10.1145/78969.78972.

[27] Attiya H, Welch J L. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 1994, 12(2): 91-122. DOI: 10.1145/176575.176576.

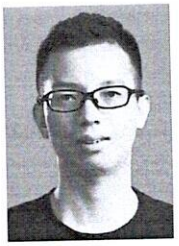
[28] Wei H, Huang Y, Cao J, Ma X, Lv J. Verifying Pipelined-RAM consistency over read/write traces of data replicas. *IEEE Transactions on Parallel and Distributed Systems*, 2013, 27(5): 1511-1523. DOI: 10.1109/TPDS.2015.2453985.

[29] Lipton R J, Sandberg J. PRAM: A scalable shared memory. Technical Report, Department of Computer Science, Princeton University, 1988. https://www.cs.princeton.edu/research/techreps/TR-180-88, Aug. 2021.

Hong-Rong Ouyang received his B.S. degree in computer science and technology from Nanjing University, Nanjing, in 2020. He is currently a Master student with the Department of Computer Science and Technology at Nanjing University, Nanjing. His research interests include distributed databases and formal methods.

Heng-Feng Wei received his B.S. and Ph.D. degrees in computer science and technology from Nanjing University, Nanjing, in 2009 and 2016, respectively. He is currently a research assistant with Software Institute at Nanjing University. His research interests include distributed computing and formal methods. He is a member of CCF.

Hai-Xiang Li is currently a chief researcher and chief architect of the distributed database system TDSQL at Tencent Inc.. His research interests include distributed computing, cloud database, transaction processing, and query optimization. He is a member of CCF.



An-Qun Pan is the technical director of Tencent Billing Platform Department. He has more than 15 years' experience in the research and development of distributed computing and storage systems. He is currently responsible for the research and development of the distributed database system TDSQL. He is a member of CCF.

Yu Huang received his B.S. and Ph.D. degrees in computer science from the University of Science and Technology of China, Hefei, in 2002 and 2007, respectively. He is currently a professor with the Department of Computer Science and Technology at Nanjing University, Nanjing. His research interests include distributed algorithms, distributed systems, formal methods, and system reliability. He is a member of CCF.