Checking Causal Consistency of MongoDB

Hong-Rong Ouyang, Heng-Feng Wei, Member, CCF, Hai-Xiang Li, Member, CCF, An-Qun Pan, Member, CCF, and Yu Huang, 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 (TEC) of Tencent, Tencent Inc., Shenzhen 518054, China

E-mail: m203300566@mail.nju.edu.cn; hwsei@nju.edu.cn; blueseal@tencent.com; aaronpan@tencent.com; yuhuang@nju.edu.cn

Received July 13, 2021; accepted October 13, 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 Bonnici et al. and test MongoDB against three very 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, causal consistency, Jepsen, consistency checking, TLA⁺

1 Introduction

MongoDB is a general-purpose, document-oriented distributed NoSQL database. 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.

MongoDB achieves scalability by partitioning the data into shards and fault-tolerance by replicating each shard across a set of nodes. 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 PACECL theorem, an extension to the CAP theorem, 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\(^\text{\textcircled{2}}\). It provides clients with session guarantees including read-your-writes, monotonic-reads, monotonic-writes, and writes-follow-reads\(^\text{\textcircled{6}}\). As the Jepsen team\(^\text{\textcircled{7}}\) denoted, MongoDB is one of the first commercial databases that implement causal consistency\(^\text{\textcircled{9}}\).

Being a production database, MongoDB’s implementation of causal consistency requires multi-dimensional evaluation criteria on performance, scalability, and security\(^\text{\textcircled{2}}\). It combines several research ideas, including hybrid logical clocks\(^\text{\textcircled{7}}\), explicit dependency tracking\(^\text{\textcircled{8}}\), Raft-based replication consensus protocol\(^\text{\textcircled{10}}\), 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.\(^\text{\textcircled{11}}\) 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)\(^\text{\textcircled{12,13}}\), causal memory (CM)\(^\text{\textcircled{14}}\), and causal convergence (CCv)\(^\text{\textcircled{15}}\). Not all of them are comparable\(^\text{\textcircled{18}}\). 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\(^\text{\textcircled{15}}\), i.e., the composition of a set of keys satisfying causal consistency may not be causally consistent. Thus, the test cases are too restrictive for causal consistency checking.
- Given the specific test cases above, the official

Jepsen testing present 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\(^\circ\). 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, namely 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.
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-off between consistency and latency, at a per operation level\cite{DBLP:conf/sigmod/KorthH96}. writeConcern specifies the number of replica set members that must acknowledge the write before returning to a client. In particular, \textbf{v : majority} requires a write operation to be acknowledged by a majority of the replica set members before being returned to the client. \textbf{readConcern} determines what consistency guarantees data returned to a client must satisfy. The default value of \textbf{readConcern} is \textbf{level : local}, which allows to return the local data in a single replica set member. In contrast, \textbf{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 \textbf{majority readConcern} and writes with \textbf{majority writeConcern}. In a good condition, however, write operations with \textbf{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\cite{DBLP:journals/tods/AbadiKV92}.

2.3.1 Replicated Objects

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

2.3.2 Histories

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

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

- \(O\) is a set of operation identifiers, or simply operations; we use \(R\) and \(W\) to denote the set of read and write operations, respectively.
- \(\mathcal{PO}\) is a partial-ordered set of operations called program order; for \(o_1, o_2 \in O\), \(o_1 \leq_{\mathcal{PO}} o_2\) 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 \(\ell(o) = (m, \text{arg}, rv) \in M \times V \times V\) indicates that \(o\) is an invocation of method \(m\) with input argument \(\text{arg}\), returning value \(rv\). We sometimes denote \(\ell(o)\) by \(m(\text{arg}) > rv\).
- We use \(\text{wr}(x,v) > \perp\) (or simply \(\text{wr}(x,v)\)) to denote a write of value \(v \in V\) to register \(x \in X\) returning \(\perp \in V\), and \(\text{rd}(x) > v\) a read of \(x\) returning \(v\). In addition, for an operation \(o\) with \(\ell(o) = \text{wr}(x,v)\) or \(\ell(o) = \text{rd}(x) > v\), we define \(\text{var}(o) = x\) and \(\text{val}(o) = v\).

Let \(\rho = (O, <, \ell)\) be an \(M \times V \times V\) 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 \((M \times V) \cup (M \times V \times V)\) labeled poset \((O, <, \ell')\) where for all \(o \in O'\), \(\ell'(o) = \ell(o)\), and for all \(o' \in O \setminus O'\), \(\ell'(o') = (m, \text{arg})\) if \(\ell(o) = (m, \text{arg}, rv)\). We denote \(\rho(O')\) by \(\rho(o)\) if \(O' = \{o\}\).

Let \(\rho = (O, <, \ell)\) and \(\rho' = (O, <', \ell')\) be two \((M \times V) \cup (M \times V \times V)\) labeled posets. \(\rho' \preceq \rho\) means that \(\rho'\) 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 \(\ell'(o) = (m, \text{arg})\) if \(\ell(o) = (m, \text{arg}, rv)\).

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 \text{rd} operation returns the value of the latest preceding \text{wr} on the same register, or the initial value \(0\) if there are no such prior writes. Formally, the sequential semantics \(S_{\text{seq}}\) of read-write registers is the smallest set of sequences labeled by \(M \times V \times V\) satisfying:

- \(\epsilon \in S_{\text{seq}}\), where \(\epsilon\) is the empty sequence;
- if \(\rho \in S_{\text{seq}}\), then \(\rho \cdot \text{wr}(x,v) \in S_{\text{seq}}\);
- if \(\rho \in S_{\text{seq}}\) contains no writes on \(x\), then \(\rho \cdot \text{rd}(x) = 0 \in S_{\text{seq}}\);
- if \(\rho \in S_{\text{seq}}\) and the last write in \(\rho\) on register \(x\) is \(\text{wr}(x,v)\), then \(\rho \cdot \text{rd}(x) > v \in S_{\text{seq}}\).

\(\odot\) 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 \( h = (O, P_0, \ell) \) is CC with respect to specification \( S_{\text{CC}} \) 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 \( \rho_o \in S_{\text{CC}} \) satisfying

\[
\begin{align*}
\text{AxCausal} & \triangleq P_0 \subseteq co, \\
\text{AxCausalValue} & \triangleq (co^{-1}(o), co, \ell) \{ o \} \preceq \rho_o.
\end{align*}
\]

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

CCv ensures eventual convergence via a total arbitration order.

**Definition 3** (Causal Convergence). A history \( h = (O, P_0, \ell) \) is CCv with respect to specification \( S_{\text{CCv}} \) 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 \( \rho_o \in S_{\text{CCv}} \) satisfying

\[
\begin{align*}
\text{AxCausal} & \triangleq P_0 \subseteq co, \\
\text{AxArb} & \triangleq co \subseteq arb, \\
\text{AxCausalArb} & \triangleq (co^{-1}(o), arb, \ell) \{ o \} \preceq \rho_o.
\end{align*}
\]

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

**Definition 4** (Causal Memory). A history \( h = (O, P_0, \ell) \) is CM with respect to specification \( S_{\text{CM}} \) 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 \( \rho_o \in S_{\text{CM}} \) satisfying

\[
\begin{align*}
\text{AxCausal} & \triangleq P_0 \subseteq co, \\
\text{AxCausalSeq} & \triangleq (co^{-1}(o), co, \ell) \{ P_0^{-1}(o) \} \preceq \rho_o.
\end{align*}
\]

Here \( P_0^{-1}(o) \) is \( \{ o' \mid o' \preceq P_0, o \} \).

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 \( P_0 \), 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 \( RF \subseteq W \times R \) associates a read with the value from which it obtained the value. Formally,

\[
\forall w \in W, r \in R, (w, r) \in RF \iff \text{var}(w) = \text{var}(r) \land \text{val}(w) = \text{val}(r).
\]

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

\[
CO = (P_0 \cup RF)^+.
\]

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

\[
\forall \omega, w' \in W, (\omega, w') \in CF \iff \exists r \in R, (\omega, r) \in RF \land (w', r) \in RF \land \text{var}(\omega) = \text{var}(w') \land (\text{val}(\omega) = \text{val}(w') \lor \text{val}(\omega) \neq \text{val}(w')).
\]

**Example 1.** Consider the history \( h \) of Fig. 3. Since \( w(x, 2) <_{RF} r(x) > 2 \) and \( r(x) < 1 <_{RF} r(x) > 1 \), we have \( w(x, 2) <_{CO} r(x) > 1 \). In addition, \( w(x, 1) <_{RF} r(x) > 1 \). Therefore, \( w(x, 2) <_{RF} w(x, 1) \) is caused by \( w(x, 2) \).

**Definition 8** (Happened-Before Relation). For each operation \( o \in O \), the happened-before relation \( HB_o \subseteq O \times O \) of \( o \) is the smallest transitive relation satisfying that \( CO_{o-1}(o) \subseteq HB_o \) and

\[
\forall w, w' \in W, (w, w') \in HB_o \iff \exists r \in R, (r) \preceq r' \preceq (w, w') \text{ and } \forall r \in R, \{ r \leq (w, w') \} \text{ and } \forall r \in R, \{ r \leq (w, w') \}.
\]

Here \( CO_{o-1}(o) \) is the relation \( CO \) restricted on the set \( o^{-1}(o) \). (Let us recall the example above.)

**Example 2.** Consider the history \( h \) of Fig. 4. Since \( w(x, 1) <_{RF} r(y, 1) <_{RF} r(y, 1) > 1 \) and \( r(y, 1) > 1 <_{RF} r(x) > 2 \), we have \( w(x, 1) <_{CO} r(x) > 2 \). In addition, for operation \( r(x) > 2 \), \( CO^{-1}(r(x) > 2) = CO \) therefore we have \( CO \subseteq HB_{r(x) > 2} \). And since \( w(x, 2) <_{RF} r(x) > 2 \) and \( w(x, 1) <_{RF} r(x) > 2 \), we have \( w(x, 2) <_{HB_{r(x) > 2}} w(x, 2) \). For the transitivity, we can also get \( w(x, 1) <_{HB_{r(x) > 2}} r(x) > 2 \).
The following theorem characterizes CC, CCv, and CM in terms of bad patterns defined in Table 1.

**Theorem 1.** 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 $\text{wr}(x,1) <_{\text{CO}} \text{wr}(x,2) <_{\text{CO}} \text{rd}(x) > 1$, and $\text{wr}(x,1) <_{\text{CF}} \text{rd}(x) > 1$, it exhibits the bad pattern WriteCORead. In addition, there is a cycle in $\text{CF}$: $\text{wr}(x,1) <_{\text{CF}} \text{wr}(x,2) <_{\text{CF}} \text{wr}(x,1)$. Thus, it also exhibits the bad pattern CyclicCF.

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

### 2.5 Jepsen

Jepsen is a library for black-box testing of distributed systems. A typical Jepsen testing of a dis-
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 nemesis 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. 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. 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+

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

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

TLC is an explicit-state model checker for TLA+\(^\circ\). 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 consistency\(^\ominus\). However, they

<table>
<thead>
<tr>
<th>Category</th>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set</td>
<td>Subset $S$</td>
<td>Powerset of $S$</td>
</tr>
<tr>
<td></td>
<td>UNION $S$</td>
<td>Union of all elements of $S$</td>
</tr>
<tr>
<td></td>
<td>${x : x \in S}$</td>
<td>Set of elements $x$ such that $x$ is in $S$</td>
</tr>
<tr>
<td></td>
<td>$(x \in S : p)$</td>
<td>Set of elements $x$ in $S$ satisfying $p$</td>
</tr>
<tr>
<td>Function</td>
<td>DOM $f$</td>
<td>Domain of function $f$</td>
</tr>
<tr>
<td></td>
<td>$f[e]$</td>
<td>Function application</td>
</tr>
<tr>
<td>Record</td>
<td>$e.h$</td>
<td>$h$-field of record $e$</td>
</tr>
<tr>
<td></td>
<td>$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$</td>
<td>Set of all records with $h_i$ field in $e_i$</td>
</tr>
<tr>
<td>Tuple</td>
<td>$e[i]$</td>
<td>The $i$-th component of tuple $e$</td>
</tr>
<tr>
<td></td>
<td>$(e_1, \ldots, e_n)$</td>
<td>The $n$-tuple whose $i$-th component is $e_i$</td>
</tr>
<tr>
<td>Sequence</td>
<td>$\text{SubSeq}(s, m, n)$</td>
<td>Sequence $[s[m], \ldots, s[n]]$</td>
</tr>
<tr>
<td></td>
<td>Range(s)</td>
<td>Set of elements of sequence $s$</td>
</tr>
</tbody>
</table>


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.

<table>
<thead>
<tr>
<th>Official Jepsen Testing</th>
<th>Our Design of Jepsen Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td>Unspecified</td>
</tr>
<tr>
<td>Test Case Generation</td>
<td>Restricted on keys and operation sequences</td>
</tr>
<tr>
<td>Checking Algorithms</td>
<td>Ad hoc for restricted test cases</td>
</tr>
<tr>
<td>Testing Scenarios</td>
<td>Network partition</td>
</tr>
<tr>
<td></td>
<td>Three well-known variants: CC, CM, and CCv</td>
</tr>
<tr>
<td></td>
<td>General for differentiated histories</td>
</tr>
<tr>
<td></td>
<td>Full implementation of [11]</td>
</tr>
<tr>
<td></td>
<td>Network partition, data movement, node failure</td>
</tr>
</tbody>
</table>

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 causal 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]. 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, ... 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 nemesis 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

moves data among shards. In an execution, partition-
random-halves, node-failure, and data-mover are
encoded 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\(^{(2)}\). Table 4 shows the hardware config-
urations 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 op-
erations and appends them into a queue. For each reg-
ister, the ratio between the number of read operations
and that of write operations is 3 : 1. Each client cre-
ares 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 corre-
sponding client. Moreover, we cover both the scenarios
with and without nemesia. 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 presen-
ce of nemesia (such as partition-random-halves,
node-failure, and data-mover), causally consistent ses-
sions guarantee causal consistency only for reads with
majority readConcern and writes with majority
writeConcern. In contrast, in the presence of nemes-
ia, the histories with local readConcern and w
writeConcern may violate any of three causal con-
sistency variants. On the other hand, without nemes-
ia, MongoDB can provide all three variants of causal
consistency even with local readConcern and w
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 \(O(n^2)\) 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 pat-
terns in our preliminary evaluations, marked \(\odot\) in Ta-
ble 5. They appear in some histories that are produced
without nemesia 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 success-
ful, this write operation is considered failed and dis-
carded from the history. However, a later read op-
eration rd(85) of No. 1266 obtains the value 5 from key
85, indicating that the write operation wr(85, 5) has ac-
tually written its value to the database. This gives rise
to a ThinAirRead bad pattern during checking.

Table 4. Hardware Configurations

<table>
<thead>
<tr>
<th>Component</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Node</td>
<td>Intel® Core™ i5-9500 CPU @ 3.00 GHz; 16 GB; Ubuntu 20.04</td>
</tr>
<tr>
<td>Database Node</td>
<td>Intel® Xeon® Platinum 8269CY CPU @ 2.50 GHz; 4 GB; Ubuntu 16.04</td>
</tr>
<tr>
<td>Checker Server</td>
<td>Intel® Core™ i9-9900X CPU @ 3.50 GHz; 32 GB; Ubuntu 16.04</td>
</tr>
</tbody>
</table>

Table 5. Experimental Results of Causal Consistency Checking of MongoDB

<table>
<thead>
<tr>
<th>Number of Operations</th>
<th>With Nemesis (majority, majority)</th>
<th>Without Nemesis (majority, majority)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CC</td>
<td>CM</td>
</tr>
<tr>
<td>100</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>200</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>300</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>400</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>500</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>600</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>700</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>800</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>900</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1000</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1100</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1200</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1300</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1400</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1500</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1600</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1700</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1800</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>1900</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2000</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2500</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3000</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3500</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4000</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4500</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5000</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

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 define 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 \( s \in \text{Session} \) is a sequence of operations issued by the same client, and a history \( h \in \text{History} \) consists of a set of sessions. The program order \( PO(h) \) of a history \( h \) is a union of strict total
orders among operations in the same session.

5.1.3 Sequential Semantics

The operator RWRegSemantics(seq,o) in module RWRegSemantics (Fig.6(a)) 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 AzCausalValue 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 AzCausalArb 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 AzCausalSeq 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

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

(a)

**MODULE ReplicatedObjects**

\[
\begin{align*}
\text{Key} & \triangleq \text{Range}(\text{"abcdefghijklmnopqrstuvwxyz"}) \\
\text{Val} & \triangleq \text{Nat} \\
\text{InitVal} & \triangleq 0 \\
\text{Oid} & \triangleq \text{Nat} \\
\text{Operation} & \triangleq \text{type} : \{\text{"read", "write"}, \text{key} : \text{Key}, \text{val} : \text{Val}, \text{oid} : \text{Oid}\} \\
R(k, v, oid) & \triangleq \{\text{type} \to \text{"read"}, \text{key} \to k, \text{val} \to v, \text{oid} \to \text{oid}\} \\
W(k, v, oid) & \triangleq \{\text{type} \to \text{"write"}, \text{key} \to k, \text{val} \to v, \text{oid} \to \text{oid}\}
\end{align*}
\]

(b)

**MODULE History**

\[
\begin{align*}
\text{Session} & \triangleq \text{Seq(Operation)} \\
\text{History} & \triangleq \text{SUBSET Session} \\
\text{PO}(h) & \triangleq \text{UNION \{Seq2Rel(s) : s ∈ h\}}
\end{align*}
\]

(c)

**MODULE RWRegSemantics**

\[
\begin{align*}
\text{RWRegSemantics(seq, o)} & \triangleq \\
& \text{IF o.type = "write" THEN TRUE ELSE} \\
& \text{LET useq = SelectSeq(seq, Lambda op : op.type = "write" ∧ op.key = o.key) IN} \\
& \text{IF useq = () THEN a.val = InitVal ELSE a.val = useq[Len(useq)].val}
\end{align*}
\]

Fig. 6. TLA\textsuperscript{+} modules for replicated read-write registers. (a) TLA\textsuperscript{+} module ReplicatedObjects. (b) TLA\textsuperscript{+} module History. (c) TLA\textsuperscript{+} module RWRegSemantics.
Fig. 7. TLA+ modules for the definition of variants of causal consistency. (a) TLA+ module Axioms. (b) TLA+ module CausalDefinition.

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: 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 CCv1 in Fig. 10).

5.3.2 CCv2: Computing Linear Extensions of co As Candidates for arb

The axiom AxAarb 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 CCv2 in Fig. 10).
\textbf{MODULE Relations}

\[ RF(h) \triangleq \{(w, r) \in \text{WriteOps}(h) \times \text{ReadOps}(h) : w.\text{key} = r.\text{key} \land w.\text{val} = r.\text{val}\} \]

\[ CO(h) \triangleq \text{TC}(\text{PO}(h) \cup RF(h)) \]

\[ CF(h) \triangleq \text{LET} \ co \triangleq \text{CO}(h) \land rf \triangleq \text{RF}(h) \]

\[ \text{LET} \ \{(w_1, w_2) \in \text{WriteOps}(h) \times \text{WriteOps}(h) : \]
\[ w_1.\text{key} = w_2.\text{key} \land w_1.\text{val} \neq w_2.\text{val} \land \exists r \in \text{ReadOps}(h) : (w_1, r) \in co \land (w_2, r) \in rf\} \]

\[ \text{LET} \ base \triangleq \text{CO}(h) \land \text{CausalPost}(co, o) \]

\[ \text{RECURSIVE} \ HBoRE(-) \]

\[ HBoRE(hbo) \triangleq \]

\[ \text{LET} \ \text{update} \triangleq \{ \]
\[ (w_1, w_2) \in \text{WriteOps}(h) \times \text{WriteOps}(h) : \]
\[ w_1.\text{key} = w_2.\text{key} \land w_1.\text{val} \neq w_2.\text{val} \land \exists r_2 \in \text{ReadOpsOnKey}(w_2, 2.\text{key}) : \]
\[ r_2.\text{val} = w_2.\text{val} \land (w_1, r_2) \in \text{hbo} \land (r_2, o) \in \text{PO}(h)\}\]

\[ \text{hbo2} \triangleq \text{update} \cup \text{hbo} \]

\[ \text{IF} \ \text{hbo2} = \text{hbo} \ \text{THEN} \ \text{hbo} \ \text{ELSE} \ HBoRE(\text{TC}(hbo2)) \]

\[ \text{IN} \ \text{TC}(\text{HBoRE}(\text{base})) \]

\textbf{Fig. 8. TLA\textsuperscript{+} module Relations.}

\textbf{MODULE BadPatterns}

\[ \text{CyclicCO}(h) \triangleq \text{Cyclic}(\text{PO}(h) \cup RF(h)) \]

\[ \text{WriteCOInitRead}(h) \triangleq \exists k \in \text{KeyOf}(h) : \exists r \in \text{ReadOpsOnKey}(h, k), \]
\[ w \in \text{WriteOpsOnKey}(h, k) : \]
\[ w.\text{key} = r.\text{key} \land w.\text{val} = \text{InitVal} \]

\[ \text{ThinAirRead}(h) \triangleq \exists k \in \text{KeyOf}(h) : \]
\[ \exists r \in \text{ReadOpsOnKey}(h, k) : \]
\[ r.\text{val} \neq \text{InitVal} \land \forall w \in \text{WriteOpsOnKey}(h, k) : (w, r) \notin RF(h) \]

\[ \text{WriteCORead}(h) \triangleq \exists k \in \text{KeyOf}(h) : \]
\[ \exists w_1, w_2 \in \text{WriteOpsOnKey}(h, k), \]
\[ \exists r_1 \in \text{ReadOpsOnKey}(h, k) : \]
\[ w_1.\text{key} = w_2.\text{key} \land (w_1, r_1) \in \text{CO}(h) \land (w_2, r_1) \in \text{CO}(h) \]

\[ \text{CyclicCF}(h) \triangleq \text{Cyclic}(CF(h) \cup CO(h)) \]

\[ \text{WriteHBBInitRead}(h) \triangleq \exists o \in \text{Ops}(h) : \]
\[ \exists r \in \text{PO}(h, o) : \]
\[ r.\text{val} = \text{InitVal} \land \exists \text{LET} \ \text{writes} \triangleq \text{WriteOpsOnKey}(h, r.\text{key}) \]
\[ \text{IN} \ \exists w \in \text{writes} : (w, r) \in HBo(h, o) \]

\[ \text{CyclicHB}(h) \triangleq \exists o \in \text{Ops}(h) : \text{Cyclic}(HBo(h, o)) \]

(a)

\textbf{MODULE Algorithm}

\[ \text{CCA}(h) \triangleq \neg \text{CyclicCO}(h) \land \neg \text{WriteCOInitRead}(h) \]
\[ \land \neg \text{ThinAirRead}(h) \land \neg \text{WriteCORead}(h) \]

\[ \text{CCe}(h) \triangleq \neg \text{CCA}(h) \land \neg \text{CyclicCF}(h) \]

\[ \text{CM}(h) \triangleq \neg \text{CCA}(h) \land \neg \text{WriteHBBInitRead}(h) \land \neg \text{CyclicHB}(h) \]

(b)

\textbf{Fig. 9. TLA\textsuperscript{+} modules for the “bad patterns” based checking algorithm of variants of causal consistency. (a) TLA\textsuperscript{+} module BadPatterns. (b) TLA\textsuperscript{+} module Algorithm.}
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\(^\S\).

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

\(^\S\) 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 of 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 \( CCv3 \) 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 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 one 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. 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. 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>
<thead>
<tr>
<th>History</th>
<th>Number of Sessions</th>
<th>Number of Operations</th>
<th>Specifications</th>
<th>Checking Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>cc</td>
<td></td>
<td></td>
<td>( CCv )</td>
<td>( CCv )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( CCv3 )</td>
</tr>
<tr>
<td>ha</td>
<td>2</td>
<td>4</td>
<td>( CCv )</td>
<td>( CCv3 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( CCv1 )</td>
<td>( CCv3 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( CCv2 )</td>
<td>( CCv3 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>( CCv3 )</td>
<td>( CCv3 )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>History</th>
<th>Specifying</th>
<th>Checking Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>ha (4 Operations)</td>
<td>( CCv )</td>
<td>( CCv3 )</td>
</tr>
<tr>
<td>hb (7 Operations)</td>
<td>( CCv3 )</td>
<td>( CCv3 )</td>
</tr>
<tr>
<td>hd (6 Operations)</td>
<td>( CCv3 )</td>
<td>( CCv3 )</td>
</tr>
</tbody>
</table>

Note: \( \checkmark \): satisfaction; \( \times \): violation.

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

Table 9. Time of Checking Histories Against Different Versions of \( CCv \)

\( CCv \) \( CCv1 \) \( CCv2 \) \( CCv3 \)

\( 48 \text{ min 37 sec} \) \( 2051 \text{ ms} \) \( 1469 \text{ ms} \) \( 1161 \text{ ms} \)

\( 20 \text{ hour 17 min} \) \( 1 \text{ min 25 sec} \) \( 2396 \text{ ms} \)

\( > 24 \text{ hour} \) \( > 24 \text{ hour} \) \( > 24 \text{ hour} \)

\( 48 \text{ min 37 sec} \) \( 2051 \text{ ms} \) \( 1469 \text{ ms} \) \( 1161 \text{ ms} \)

\( > 24 \text{ hour} \) \( > 24 \text{ hour} \) \( > 24 \text{ hour} \)

\( 48 \text{ min 37 sec} \) \( 2051 \text{ ms} \) \( 1469 \text{ ms} \) \( 1161 \text{ ms} \)

\( > 24 \text{ hour} \) \( > 24 \text{ hour} \) \( > 24 \text{ hour} \)


- In 2017, the team tested the v0 and v1 replication protocols of MongoDB 3.4.0-rc3®. 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®. 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® systematically studied the complexity of the checking problem against strong consistency models, including linearizability and sequential consistency. Regarding weak consistency models, Wei et al.® addressed the problem of checking PRAM consistency 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. They considered three well-known variants of causal consistency, namely CC, CM, and CHeC. They proved that checking whether a general history of arbitrary replicated objects satisfies CC, CM, or CHeC 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 strengthens the official Jepsen testing in 2018 in terms of specification, test case generation, implementation of

---

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®.

References


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.

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.

Hong-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.