Verifying Transactional Consistency of MongoDB

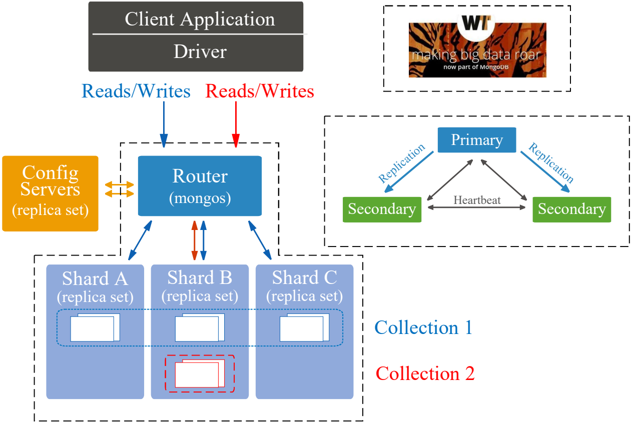
(submitted to VLDB'2022)

Hengfeng Wei

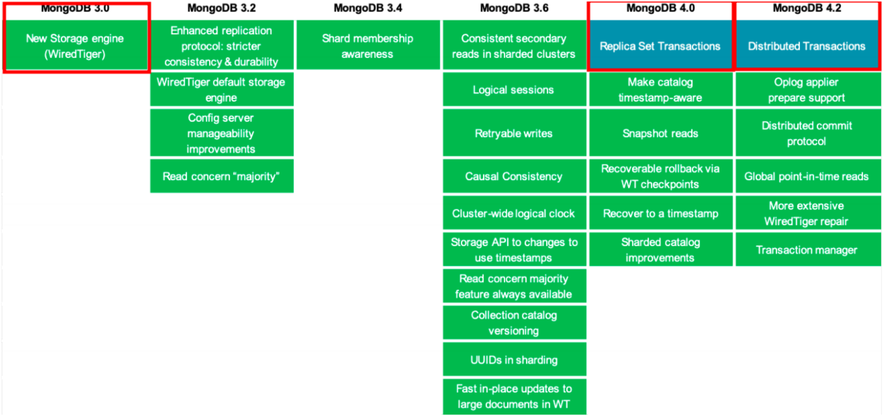
hfwei@nju.edu.cn

December 29, 2021

Transaction



MongoDB 的三种经典部署架构



MongoDB 事务的三阶段发展过程

一个基本问题:

每种部署下, MongoDB 事务协议提供怎样的事务一致性?

挑战一：MongoDB 官方规约不清楚, SI 有多种变体

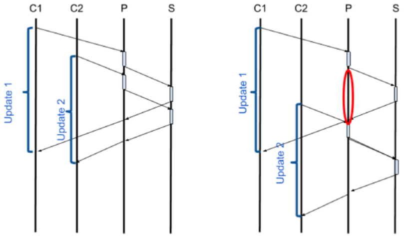
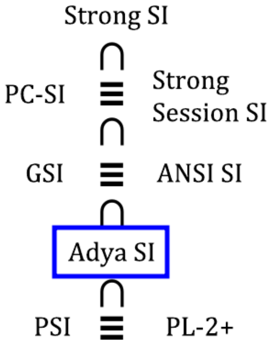
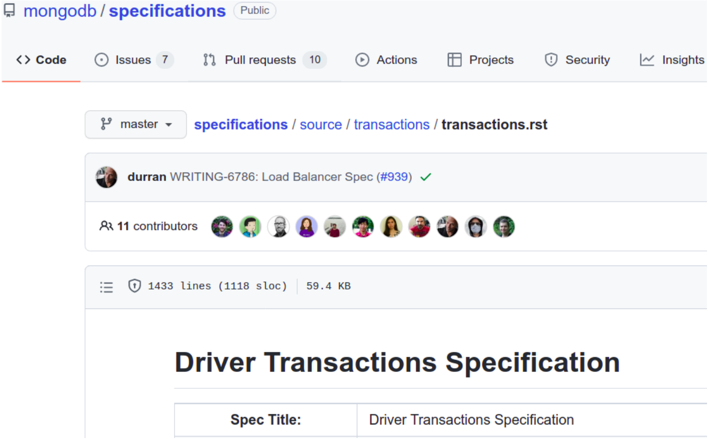


Figure 3: Back-to-Back Transactions with and without Speculative Snapshot Isolation



挑战二：MongoDB 缺少精简的事务协议描述, 更没有严格证明



挑战三: 在 Version Order 未知的情况下,

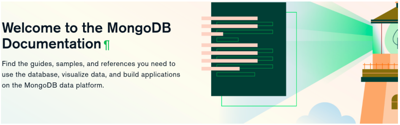
SI 检测问题是 NP-complete 的

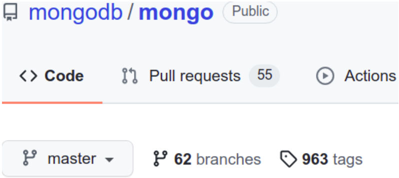
THEOREM 3.2. *For any criterion* $C \in \{$PREFIX CONSISTENCY, $\boxed{\text{SNAPSHOT ISOLATION}}$, SERIALIZABILITY$\}$ *the problem of checking whether a given history satisfies C is NP-complete.*

贡献一：使用 (VIS, AR) 框架，为多种 SI 变体提供形式化规约

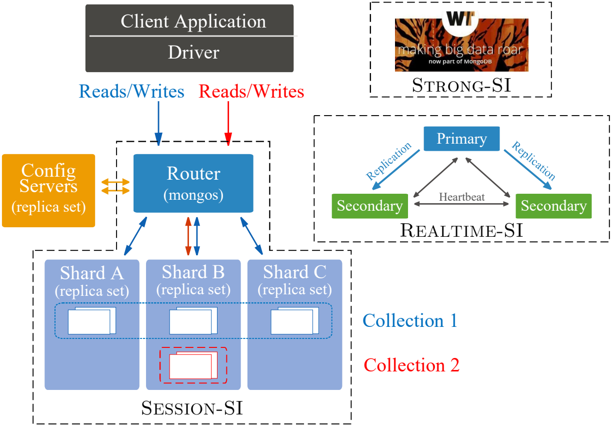
SessionSI RealtimeSI GSI StrongSI . . .

贡献二：为 MongoDB 事务一致性协议提供精简的伪代码描述

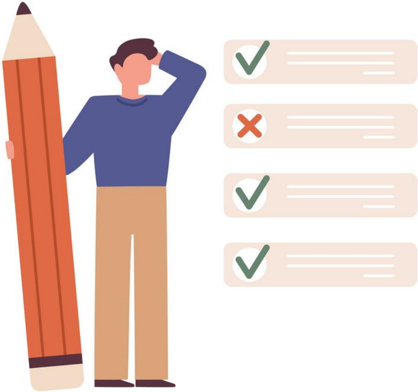




贡献三：证明 WiredTiger、ReplicaSet、ShardedCluster 事务协议分别满足 StrongSI、RealtimeSI、SessionSI 变体



贡献四：设计并评估了多项式时间 SI 变体白盒检测算法




SPECIFICATION

1. 事务 $T : (E, \mathsf{po})$
   - po : 程序序 (Program Order)
   - $start(T)$: 事务开始时间
   - $commit(T)$: 事务提交时间

2. 历史 $\mathcal{H} : (\mathbb{T}, \textsc{so})$
   - $\mathbb{T}$: 已提交事务集合
   - $\textsc{so}$: 会话序 (Session Order)

3. 执行 $\mathcal{A} : (\mathcal{H}, \textsc{vis}, \textsc{ar})$
   - $\textsc{vis}$: 可见 (Visibility) 偏序关系
   - $\textsc{ar}$: 仲裁 (Arbitration) 全序关系
   - $\textsc{vis} \subseteq \textsc{ar}$

一个事务一致性模型可定义为一组一致性公理的集合 Φ

历史 $\mathcal{H}$ 满足事务一致性模型 Φ, 如果存在 $\textsc{vis}$ 与 $\textsc{ar}$ 使得

$$\exists \textsc{vis}, \textsc{ar}.\ (\mathcal{H}, \textsc{vis}, \textsc{ar}) \models \Phi$$

$\textsc{ReadAtomic} = \textsc{Int} \wedge \textsc{Ext}$

$\forall (E, \mathsf{po}) \in \mathcal{H}.\ \forall e \in \mathsf{Event}.\ \forall key, val.\ \big(\mathsf{op}(e) = \mathsf{read}(key, val) \wedge \{f \mid (\mathsf{op}(f) = \_(key, \_) \wedge f \xrightarrow{\mathsf{po}} e\} \neq \emptyset\big) \implies \mathsf{op}(\max_{\mathsf{po}}\{f \mid \mathsf{op}(f) = \_(key, \_) \wedge f \xrightarrow{\mathsf{po}} e\}) = \_(key, val)$ (Int)	
$\forall T \in \mathcal{H}.\ \forall key, val.\ T \vdash \mathsf{read}(key, val) \implies \max_{\mathrm{AR}}(\mathrm{VIS}^{-1}(T) \cap \mathsf{WriteTx}_{key}) \vdash \mathsf{write}(key, val)$ (Ext)	
$\mathrm{SO} \subseteq \mathrm{VIS}$ (Session)	$\mathrm{AR} ; \mathrm{VIS} \subseteq \mathrm{VIS}$ (Prefix)
$\mathrm{RB} \subseteq \mathrm{VIS}$ (ReturnBefore)	$\mathrm{CB} \subseteq \mathrm{AR}$ (CommitBefore)
$\mathrm{VIS} \subseteq \mathrm{RB}$ (RealtimeSnapshot)	$\forall S, T \in \mathcal{H}.\ S \bowtie T \implies (S \xrightarrow{\mathrm{VIS}} T \vee T \xrightarrow{\mathrm{VIS}} S)$ (NoConflict)

$$S \xrightarrow{\mathrm{RB}} T \iff commit(S) < start(T)$$

$$S \xrightarrow{\mathrm{CB}} T \iff commit(S) < commit(T)$$

$$\mathrm{SI} = \textsc{Int} \land \textsc{Ext} \land \textsc{Prefix} \land \textsc{NoConflict}$$

$$\textsc{SessionSI} = \text{SI} \wedge \textsc{Session}$$

$$\textsc{SessionSI} = \text{SI} \wedge \textsc{Session}$$

$$\textsc{RealtimeSI} = \text{SI} \wedge \textsc{ReturnBefore} \wedge \textsc{CommitBefore}$$

$$\textsc{SessionSI} = \text{SI} \land \textsc{Session}$$

$$\textsc{RealtimeSI} = \text{SI} \land \textsc{ReturnBefore} \land \textsc{CommitBefore}$$

$$\text{GSI} = \text{SI} \land \textsc{RealtimeSnapshot} \land \textsc{CommitBefore}$$

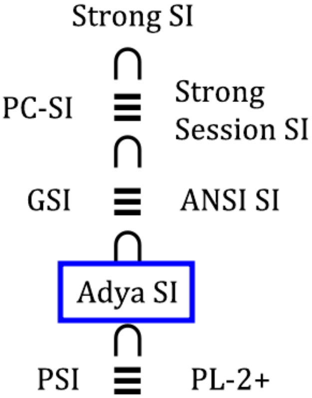
$$\textsc{SessionSI} = \text{SI} \land \textsc{Session}$$

$$\textsc{RealtimeSI} = \text{SI} \land \textsc{ReturnBefore} \land \textsc{CommitBefore}$$

$$\text{GSI} = \text{SI} \land \textsc{RealtimeSnapshot} \land \textsc{CommitBefore}$$

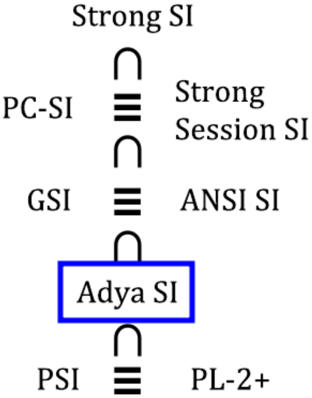
$$\textsc{StrongSI} = \text{GSI} \land \textsc{ReturnBefore}$$

- ANSI-SI
- SI
- GSI
- StrongSI
- StrongSessionSI
- PSI
- WriteSI
- NMSI
- PC-SI



$$\textsc{TransVis} \triangleq (\textsc{vis}^{+} = \textsc{vis})$$

- ANSI-SI
- SI
- GSI
- StrongSI
- StrongSessionSI
- PSI
- WriteSI
- NMSI
- PC-SI



$$\text{PSI} = \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{TransVis} \wedge \textsc{NoConflict}$$

2013 IEEE 32nd International Symposium on Reliable Distributed Systems

Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems

Masoud Saeida Ardekani
Université Pierre-et-Marie Curie (UPMC-LIP6)
Paris, France
masoud.saeida-ardekani@lip6.fr

Pierre Sutra
University of Neuchâtel
Neuchâtel, Switzerland
pierre.sutra@unine.ch

Marc Shapiro
INRIA & UPMC-LIP6
Paris, France
http://lip6.fr/Marc.Shapiro/

SI/PSI 扩展性差, 作者提出四种可扩展性性质:

Wait-Free Queries: 只读事务不被阻塞且总是被成功提交

Genuine Partial Replication: 仅被读写的节点参与分布式事务协议

Minimal Commitment Synchronization: 仅在发生写冲突时等待

Forward Freshness: 允许读在当前事务开始后提交的数据

2013 IEEE 32nd International Symposium on Reliable Distributed Systems

Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems

Masoud Saeida Ardekani
Université Pierre-et-Marie Curie (UPMC-LIP6)
Paris, France
masoud.saeida-ardekani@lip6.fr

Pierre Sutra
University of Neuchâtel
Neuchâtel, Switzerland
pierre.sutra@unine.ch

Marc Shapiro
INRIA & UPMC-LIP6
Paris, France
http://lip6.fr/Marc.Shapiro/

Definition (NMSI)

$$\text{NMSI} = \text{ACA} \wedge \text{CONS} \wedge \text{WCF}$$

Avoiding Cascading Aborts (ACA): 避免读未提交数据

Consistent Snapshot (CONS): 不遗漏所依赖事务的写操作

Write-Conflict Freedom (WCF): 没有 (并发) 写冲突

A Critique of Snapshot Isolation

Daniel Gómez Ferro   Maysam Yabandeh *

Yahoo! Research

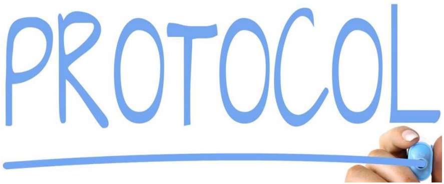
Barcelona, Spain

{danielgf,maysam}@yahoo-inc.com

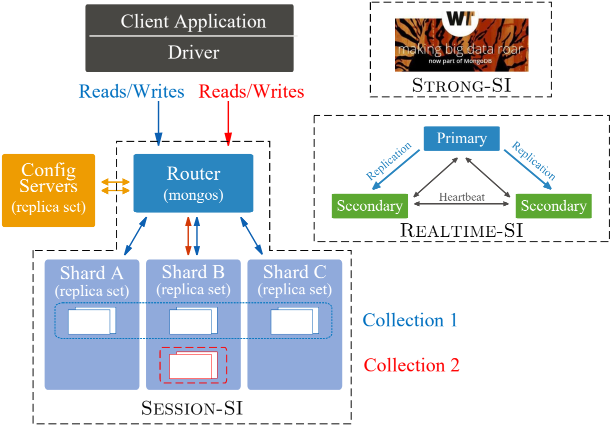
WRITESI 避免读-写冲突, 而不是写-写冲突 (SI: READSI)

$$\text{WRITESI} \subseteq \text{SER} \subset \text{SI}$$

(暂时存疑)


PROTOCOL


pRoof



重点在于如何确定每个事务的 “读快照” (Read Snapshot),

也就是对该事务可见的所有事务构成的集合

$$\textsc{WiredTiger} \models \textsc{StrongSI}$$

每个 WiredTiger 事务 $txn \in \mathsf{WT\_TXN}$ 有一个唯一标识号

$$txn.\mathsf{tid} \in \mathsf{TID} = \mathbb{N} \cup \{-1, \bot_{\mathsf{tid}}\}$$

- 事务开始时 (WT_START), $txn.\mathsf{tid} = 0$
- 事务第一个写操作成功执行后 (WT_UPDATE), $txn.\mathsf{tid} > 0$
- 事务因写冲突回滚时 (WT_ROLLBACK), $txn.\mathsf{tid} = -1$
- $\bot_{\mathsf{tid}}$ 表示不存在这个事务

每个 WiredTiger 事务 $txn \in \mathsf{WT\_TXN}$ 有一个唯一标识号

$$txn.\mathsf{tid} \in \mathsf{TID} = \mathbb{N} \cup \{-1, \bot_{\mathsf{tid}}\}$$

- 事务开始时 (WT_START), $txn.\mathsf{tid} = 0$
- 事务第一个写操作成功执行后 (WT_UPDATE), $txn.\mathsf{tid} > 0$
- 事务因写冲突回滚时 (WT_ROLLBACK), $txn.\mathsf{tid} = -1$
- $\bot_{\mathsf{tid}}$ 表示不存在这个事务

对于只读事务 txn, 始终有 $txn.\mathsf{tid} = 0$

- 客户端通过会话 (Session) 与 WiredTiger 进行交互
- 每个会话有一个唯一会话标识号 $wt\_sid \in \mathsf{WT\_SID} = \mathbb{N}$

- 客户端通过会话 (Session) 与 WiredTiger 进行交互
- 每个会话有一个唯一会话标识号 $wt\_sid \in \mathsf{WT\_SID} = \mathbb{N}$
- WiredTiger 维护数据结构

  $\mathsf{wt\_txn\_global} \in [\mathsf{current\_tid} : \mathsf{TID}, \mathsf{states} : \mathsf{WT\_SID} \to \mathsf{TID}]$

  current_tid : 当前分配的最大事务标识号

  states : 会话与会话之上当前事务之间的映射关系

- 客户端通过会话 (Session) 与 WiredTiger 进行交互
- 每个会话有一个唯一会话标识号 $wt\_sid \in \mathsf{WT\_SID} = \mathbb{N}$
- WiredTiger 维护数据结构
  $\mathsf{wt\_txn\_global} \in [\mathsf{current\_tid} : \mathsf{TID}, \mathsf{states} : \mathsf{WT\_SID} \to \mathsf{TID}]$

  current_tid : 当前分配的最大事务标识号

  states : 会话与会话之上当前事务之间的映射关系

$$\text{事务 } txn \text{ 提交或回滚时, } \mathsf{wt\_txn\_global.states}[wt\_sid] \leftarrow \perp_{\mathsf{txn}}$$

每个事务只能观察到在它开始之前提交的事务

$\textsc{WiredTiger} \models \textsc{RealtimeSI}$

每个事务只能观察到在它开始之前提交的事务

$$\textsc{WiredTiger} \models \textsc{RealtimeSI}$$

每个事务在开始时 ($\textsc{wt\_start}$) 根据

wt_txn_global 维护的信息确定它的 “读快照”

$\textsc{WiredTiger}$ 事务协议从反面入手计算, 排除不可见事务集合

每个事务 txn 维护以下信息:

txn.snapshot： 正在进行的、已获取事务标识号的事务集合

txn.snap_max： txn 开始时, 当前最大的事务标识号

$$\text{wt\_txn\_global} \in [\text{current\_tid} : \text{TID}, \text{states} : \text{WT\_SID} \to \text{TID}]$$

procedure $\text{TXN\_VISIBLE}(txn, tid)$

  return $\neg(tid = -1 \lor tid \in txn.\mathsf{snapshot} \lor (tid \geq txn.\mathsf{snap\_max} \land tid \neq txn.\mathsf{tid}))$

对于事务 txn, 满足以下条件的、标识号等于 tid 的事务对 txn 不可见:

- $tid = -1$: 该事务已回滚
- $tid \in txn.snapshot$: 该事务与 txn 并发
- $\mathsf{tid} \geq txn.\mathsf{snapshot} \land \mathsf{tid} \neq txn.\mathsf{tid}$:
  - $\mathsf{tid} \geq txn.\mathsf{snapshot}$: 该事务开始得比 txn 晚
  - $\mathsf{tid} \neq txn.\mathsf{tid}$: 允许 txn 观察到自身

每个事务观察到在它开始之前提交的事务

Definition (可见关系 $\textsc{vis}_{\text{WT}}$)

$$\textsc{vis}_{\text{WT}} \triangleq \textsc{ReturnBefore}$$

每个事务观察到在它开始之前提交的事务

Definition (可见关系 $\textsc{vis}_{\textsc{wt}}$)

$$\textsc{vis}_{\textsc{wt}} \triangleq \textsc{ReturnBefore}$$

在逻辑上, 所有事务按实时序依次提交

Definition (仲裁关系 $\textsc{ar}_{\textsc{wt}}$)

$$\textsc{ar}_{\textsc{wt}} \triangleq \textsc{CommitBefore}$$

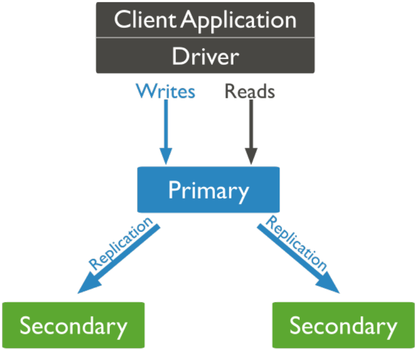
Lemma (冲突事务的提交顺序)

$$\forall txn, txn' \in \mathsf{WT\_TXN}.$$

$$txn \bowtie txn' \implies \left(txn \xrightarrow{AR_{WT}} txn' \iff txn.\mathsf{tid} < txn'.\mathsf{tid}\right).$$

该引理对 StrongSI 检测算法有辅助作用

$$\textsc{StrongSI} = \text{SI} \land \textsc{RealtimeSnapshot} \land \textsc{ReturnBefore} \land \textsc{CommitBefore}$$



$\textsc{ReplicaSet} \models \textsc{RealtimeSI}$

$$readConcern = \text{“}snapshot\text{”} \text{ and } writeConcern = \text{“}majority\text{”}$$

$readConcern = \text{“}snapshot\text{”}$: 保证事务读取到一致性的、被多数节点提交的快照

$writeConcern = \text{“}majority\text{”}$: 保证事务中的写操作以及读到的数据被多数节点提交

主节点维护 oplog, 并负责决定事务的提交顺序

主节点维护 oplog, 并负责决定事务的提交顺序

每个事务 txn 被赋予一个唯一的逻辑提交时间戳 txn.commit_ts

这些逻辑时间戳确定了事务在 ReplicaSet 层的 (逻辑) 提交顺序

当事务 txn 开始时, 计算它的 “读时间戳” (txn.read_ts)

条件: oplog 中所有提交时间戳小于 txn.read_ts 的事务均已提交

(txn.read_ts 保证可见的事务在 oplog 中不造成空洞)

procedure TXN_VISIBLE(txn, tid, ts)

return $\neg\big(tid = -1 \lor tid \in txn.\mathsf{snapshot} \lor (tid \geq txn.\mathsf{snap\_max} \land tid \neq txn.\mathsf{tid})\big) \land \big(ts \neq \bot_{\mathsf{ts}} \land ts \leq txn.\mathsf{read\_ts}\big)$

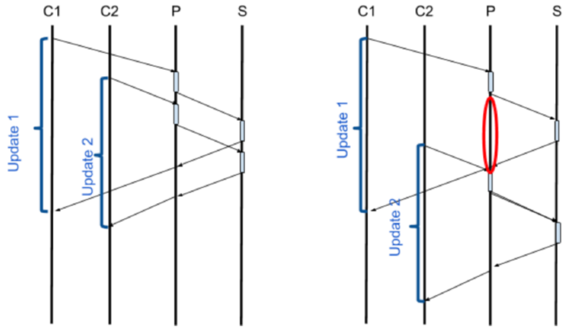


Figure 3: Back-to-Back Transactions with and without Speculative Snapshot Isolation

- 事务读取 WiredTiger 层当前最新的数据, 而不限于已被多数节点提交的数据
- 当事务提交时, 等待读取到的数据被多数节点提交

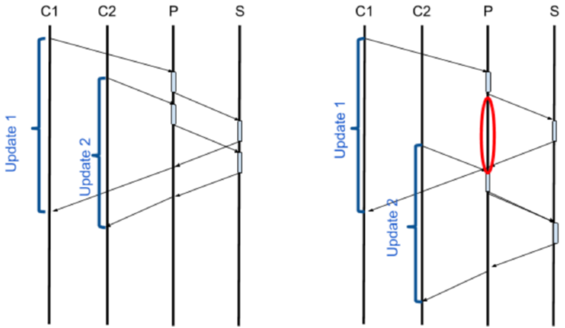


Figure 3: Back-to-Back Transactions with and without Speculative Snapshot Isolation

- 事务读取 WiredTiger 层当前最新的数据, 而不限于已被多数节点提交的数据
- 当事务提交时, 等待读取到的数据被多数节点提交

只读事务在提交时发起 “noop” 操作, 然后等待该操作被多数节点提交

(physical component, logical component)

Coarse synchronization of time intervals using NTP

Order within coarse intervals with a monotonic counter

---

procedure TICK()
    if $\mathsf{ct}.sec \geq \mathsf{clock}$ **then**
        return $\langle \mathsf{ct}.sec, \mathsf{ct}.counter + 1 \rangle$
    else
        return $\langle \mathsf{clock}, 0 \rangle$

---

procedure TICK()
  if ct.$sec \geq$ clock **then**
    return $\langle$ct.sec, ct.$counter + 1\rangle$
  else
    return $\langle$clock, 0$\rangle$

- 主节点的 cluster time (ct) 仅在产生新的 oplog 项时增加
- 每个消息都携带发送方的 ct
- 每个节点 (包括客户端) 维护它已知的最大 ct

Definition (可见关系 $\textsc{vis}_{\text{RS}}$)

$$\forall txn_1, txn_2 \in \mathsf{RS\_TXN}.\ txn_1 \xrightarrow{\text{VIS}_{\text{RS}}} txn_2 \iff txn_1.\mathsf{commit\_ts} \le txn_2.\mathsf{read\_ts}.$$

Lemma (WiredTiger 与 ReplicaSet 层可见关系)

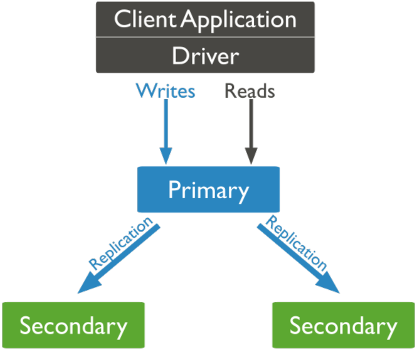
$$VIS_{RS} \subseteq VIS_{WT}$$

ReplicaSet 使用逻辑时间戳 “覆写” (override) 了 WiredTiger 中的事务标识号

Definition (仲裁关系 AR_{RS})

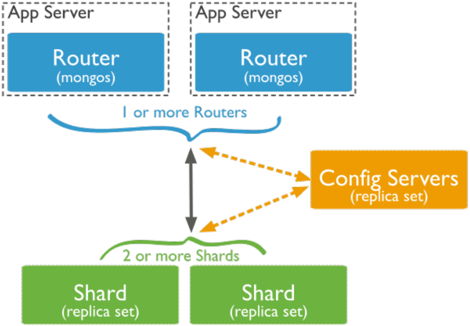
- 非只读事务按照它们的 commit_ts 在 AR_{RS} 中排序
- 对于每个客户端, 将只读事务按照会话序依次插入 AR_{RS}。

  具体而言, 会话 $rs\_session$ 上的只读事务 txn 紧随以下事务之后:
  - txn 在 SO_{RS} 序下的前一个事务 (如果有),
  - 非只读事务 txn' 满足 $txn' \xrightarrow{\text{VIS}_{\text{RS}}} txn$。



$$\textsc{RealtimeSI} = \text{SI} \land \textcolor{red}{\textsc{ReturnBefore}} \land \textcolor{red}{\textsc{CommitBefore}}$$

由主从架构、复制与多数节点提交机制、读时间戳设置策略共同保障

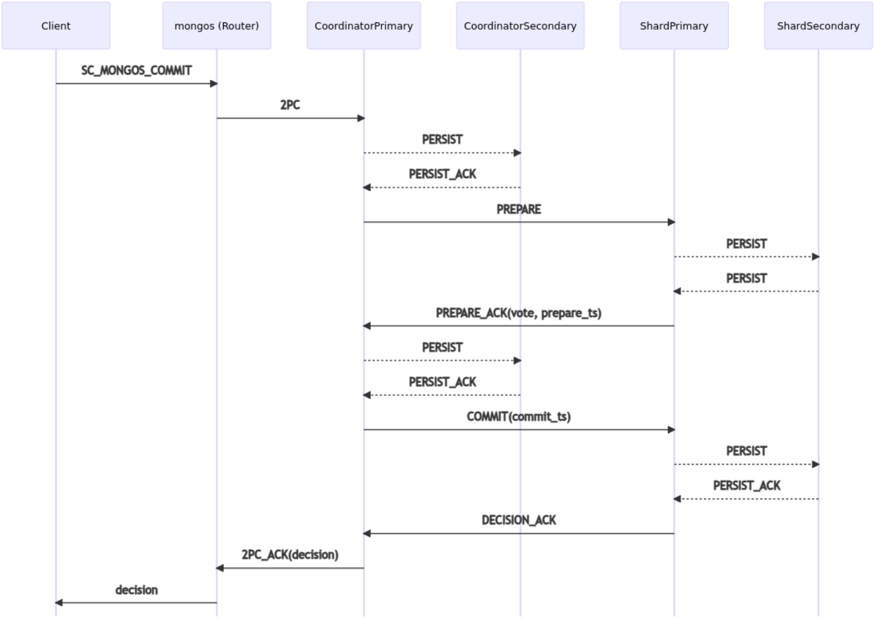


$$\textsc{ShardedCluster} \models \textsc{SessionSI}$$

分布式事务

不同分区间使用 2PC 协议, 保证原子性

同一分区间使用共识复制协议, 提高容错性


Client
mongos (Router)
CoordinatorPrimary
CoordinatorSecondary
ShardPrimary
ShardSecondary
SC_MONGOS_COMMIT
2PC
PERSIST
PERSIST_ACK
PREPARE
PERSIST
PERSIST
PREPARE_ACK(vote, prepare_ts)
PERSIST
PERSIST_ACK
COMMIT(commit_ts)
PERSIST
PERSIST_ACK
DECISION_ACK
2PC_ACK(decision)
decision

monogs 为事务指定快照时间戳 (read_ts)

逻辑时间戳是松散同步的, 在某些分区上该快照尚不可得

monogs 为事务指定快照时间戳 (read_ts)

逻辑时间戳是松散同步的, 在某些分区上该快照尚不可得

CASE-CLOCK-SKEW: 如果 ct < read_ts, 则先推进 ct

CASE-PENDING-COMMIT-READ: 如果可见的写操作是 WT_PREPARED 状态, 则重试读

CASE-PENDING-COMMIT-UPDATE 如果可见的写操作是 WT_PREPARED 状态, 则先重试读

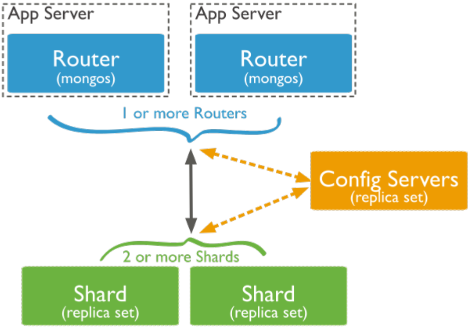
CASE-HOLES: 等待直到 oplog 中 commit_ts $\le$ read_ts 的事务均提交

Definition (可见关系 $\textsc{vis}_{\textsc{sc}}$)

$$\forall txn_1, txn_2 \in \mathsf{SC\_TXN}.\ txn_1 \xrightarrow{\textsc{vis}_{\textsc{sc}}} txn_2 \iff txn_1.\mathsf{commit\_ts} \le txn_2.\mathsf{read\_ts}.$$

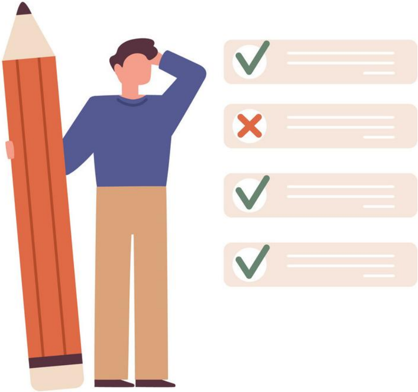
Definition (仲裁关系 $\mathrm{AR}_{\mathrm{SC}}$)

- 非只读事务按照它们的 commit_ts 在 $\mathrm{AR}_{\mathrm{SC}}$ 中排序
  - 相同的 commit_ts 根据事务协调者所在的分区 shard_id 定序
- 对于每个客户端，将只读事务按照会话序依次插入 $\mathrm{AR}_{\mathrm{SC}}$。
  具体而言，会话 $sc\_session$ 上的只读事务 txn 紧随以下事务之后:
  - txn 在 $\mathrm{SO}_{\mathrm{SC}}$ 序下的前一个事务 (如果有),
  - 非只读事务 txn' 满足 $txn' \xrightarrow{\mathrm{VIS}_{\mathrm{SC}}} txn$。



$$\textsc{SessionSI} = \text{SI} \wedge \textsc{Session}$$

由逻辑时间戳的维护与分发机制保障



多项式时间 SI 变体白盒检测算法

(1) 收集事务协议产生的历史执行

(2) 根据证明中的定义构造 VIS 与 AR 关系

(3) 检查 VIS 与 AR 关系是否满足相应的 SI 变体所需的公理

(1) 收集事务协议产生的历史执行

(2) 根据证明中的定义构造 VIS 与 AR 关系

(3) 检查 VIS 与 AR 关系是否满足相应的 SI 变体所需的公理

第 (2) 中构造的 AR 是全序关系，提供了 Version Order，

第 (3) 步可以在多项式时间内完成

如何从历史执行中获取必要的信息, 以构造 VIS 与 AR 关系:

WIREDTIGER : 记录事务的开始与提交时间, 利用前述引理校正

REPLICASET : 分别从 `mongod.log` 与 `oplog.rs` 中获取事务的读时间戳与提交时间戳

SHARDEDCLUSTER : 分别从 `mongod.log` 与分区主节点 oplog 中获取事务的读时间戳与提交时间戳

Lemma (冲突事务的提交顺序)

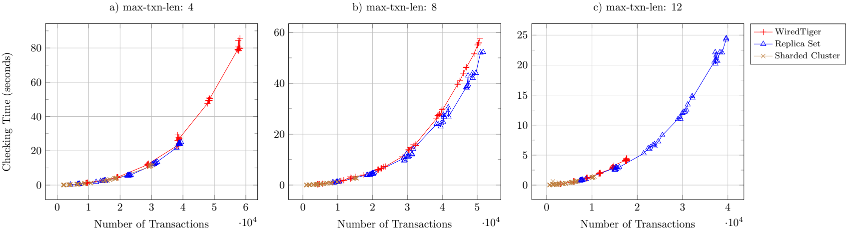
$$\forall txn, txn' \in \mathsf{WT\_TXN}.$$
$$txn \bowtie txn' \implies (txn \xrightarrow{AR_{WT}} txn' \iff txn.\mathsf{tid} < txn'.\mathsf{tid}).$$

<table>
<tr><th>Deployment</th><th>Version</th><th>Configuration</th><th>OS</th></tr>
<tr><td>Standalone</td><td>WiredTiger 3.3.0</td><td>A WiredTiger storage engine</td><td>Ubuntu 20.04</td></tr>
<tr><td>Replica Set</td><td rowspan="2">MongoDB 4.2.8</td><td>A replica set of 5 nodes</td><td rowspan="2">Each database node runs in a Docker
container built with the Debian 10 image.</td></tr>
<tr><td>Sharded Cluster</td><td>A cluster consists of 1 config server and 2 shards.
Each shard is a replica set of 3 nodes.
Each node has a mongos instance.</td></tr>
</table>

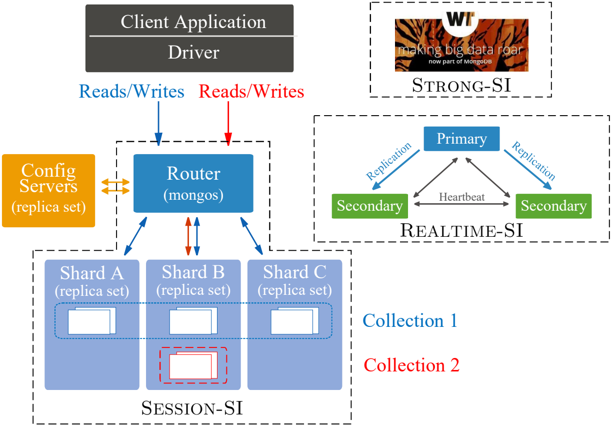
MongoDB 部署配置

Parameter	Value(s)	Description
key-count	10	There are 10 distinct keys at any point for generation.
key-dist	exponential	Probability distribution on keys.
max-txn-length	{4, 8, 12}	Each transaction contains at most 4/8/12 operations.
max-updates-per-key	128	There are at most 128 updates on each key.
read:update ratio	1 : 1	The default (and fixed) read:update ration in Jepsen.
wt-duration	{10s, 20s, 30s, 40s, 50s, 60s}	The duration of an experiment on WiredTiger.
rs-duration	{1min, 2min, 3min, 4min, 5min}	The duration of an experiment on replica set .
sc-duration	{1min, 2min, 3min, 4min, 5min, 8min}	The duration of an experiment on sharded cluster.
wt/rs/sc-timeout	5s/10s/30s	Transaction timeout for experiments on WiredTiger, replica set, and sharded cluster, respectively.

事务生成器相关参数 (Jepsen 支持)



- 13ms 的时钟误差内, 可认为 WiredTiger $\models$ StrongSI
- 10s 检测包含 30, 000 个事务的历史执行
  - Intel Core i5-9500 CPU @ 3.00GHZ and 16GB RAM


Client Application
Driver
Reads/Writes
Reads/Writes
Config Servers (replica set)
Router (mongos)
Shard A (replica set)
Shard B (replica set)
Shard C (replica set)
SESSION-SI
making big data roar
now part of MongoDB
STRONG-SI
Primary
Replication
Replication
Heartbeat
Secondary
Secondary
REALTIME-SI
Collection 1
Collection 2

规约、协议、证明、检测算法

本工作是迈向 MongoDB 形式化验证的一小步

(1) 建模事务协议中的并发机制
(2) MongoDB 支持事务操作与非事务操作混合执行
(3) MongoDB Raft 等相关容错与恢复机制
(4) 利用 SMT 求解器设计高效的事务一致性检测算法

Hengfeng Wei (hfwei@nju.edu.cn)