Enabling Almost Strong Consistency for Quorum-replicated Datastores

Lingzhi Ouyang, Yu Huang, Hengfeng Wei, and Jian Lu

Abstract—Although it has been commercially successful to deploy weakly consistent replicated datastores, the tension between developing complex applications and obtaining only eventual consistency guarantees becomes more and more severe. The Almost Strong Consistency (ASC) tradeoff aims at helping the application developers out of the dilemma of choosing either low latency or strong consistency, and achieving both strong consistency and low latency in the common case. The ASC tradeoff has been explored in the case where only one single client can write the data. However, the more general case where multiple clients can read and write the data has not been studied. In this work, we study the ASC tradeoff in the multi-writer and multi-reader case. We first study the design options in implementing low-latency data access algorithms and propose a quorum-based algorithm schema. Then we analyze the consistency guarantees provided by the low-latency data access algorithms. We implement the data access algorithms in the ASC tradeoff over Cassandra, and evaluate the consistency-latency tradeoffs based on YCSB. The theoretical analysis and the experimental evaluations show that the ASC tradeoff can be achieved, even when faced with dynamic changes in the cloud computing environment.

Index Terms—Almost strong consistency, consistency-latency tradeoff, quorum-replicated datastore.

1 Introduction

Nowadays cloud-based distributed datastores are expected to provide always-available and highly responsive services for millions of user requests across the world [1], [2], [3]. To this end, data replication is typically employed. Through replicating data into multiple replicas across different machines or even across data centers, distributed datastores can not only reduce response time of user requests, but also tolerate certain degree of software/hardware failures and network partitions [4], [5]. Since cloud-based datastores must tolerate network partitions, according to the CAP theorem, once the datastore replicates data, the tradeoff between data consistency and latency comes up [6], [7]. Many real-world web services, such as Google, Amazon, EBay, etc., aim to provide an "always-on" experience and overwhelmingly favor availability and low latency over strong consistency [8]. It is claimed that a slight increase in user-perceived latency translates into concrete revenue loss [9].

Although it is widely used and commercially successful to deploy weakly consistent but highly responsive replicated datastores, the tension between developing complex upper-layer user applications and obtaining only eventual consistency guarantees becomes more and more severe [9], [10]. The application developer expects the illusion of programming on one single copy of local data, which can be emulated by atomicity (a.k.a. linearizability) [11], [12], [13]. Atomicity is an ideally strong consistency condition, requiring the execution to be equivalent to a legal sequential execution [11], [14]. While strong consistency shields the application developers from the underlying reality of large-scale distributed systems, this illusion comes at the cost of increased latency and reduced service availability [15].

The Almost Strong Consistency (ASC) tradeoff aims at helping the application developers out of the dilemma of choosing either low latency or strong consistency [16]. The ASC tradeoff achieves the best of both worlds: strong consistency and low latency in the common case. As for latency, the ASC tradeoff adopts "fast" data access algorithms, i.e., algorithms requiring one single round-trip of communication between the clients and the server replicas [15]. This is obviously optimal in terms of data access latency. As for data consistency, fast data access protocols cannot strictly guarantee strong consistency (atomicity) [15]. However, almost strong consistency can be achieved. Here, the abstract term "almost" needs to be interpreted in two orthogonal dimensions. On the one hand, "almost strong" means that the data accessed can be stale, but the staleness should be bounded. On the other hand, the probability of accessing stale data should be quite small. Combining both dimensions, clients are guaranteed to access up-to-date data most of the time.

The ASC tradeoff has been explored in the case where only one single client can write the data while multiple clients can read the data in our previous work [16]. In this SWMR (Single Writer Multiple Reader) case, it is obvious that write operations can finish in one single round-trip. It is shown that, when the clients read data using only one round-trip of communication with the server replicas, the clients can miss at most one data update and the probability of reading stale data is quite small. However, the more general and complex case where multiple clients can write and read the data (denoted MWMR) has not been studied. Moreover, the ASC tradeoff in the SWMR case was only

- *Corresponding author: Yu Huang, State Key Laboratory for Novel Software Technology, and Department of Computer Science and Technology, Nanjing University, China, 210023.*
  E-mail: yuhuang@nju.edu.cn.
- *Lingzhi Ouyang, Hengfeng Wei and Jian Lu are with the State Key Laboratory for Novel Software Technology, and Department of Computer Science and Technology, Nanjing University, China, 210023.*
  E-mail: lingzhi.ouyang@outlook.com, hfwei@nju.edu.cn, lj@nju.edu.cn.

evaluated in a mobile data sharing scenario. The more important scenario of cloud-based replicated datastores has not yet been studied experimentally.

To address the problems above, we study in this work the ASC tradeoff in the MWMR case. We first study the possible design options in implementations of fast data access algorithms, which potentially can be adopted in the ASC tradeoff. We model the client-server interaction as a quorum system [17], [18]. A quorum-based algorithm schema is proposed, which instructs us to tune the round-trips of writes and reads and obtain fast write and/or read algorithms.

Then we analyze the consistency guarantees in terms of the data staleness and the probability of atomicity violation. The key in our analysis is to interpret all possible cases of inconsistent data reads into two patterns, namely the Write Inversion (WI) and the Read Inversion (RI). We show that the inconsistent reads can always be captured by an RI or WI. Based on the construction of RI and WI, we can further construct possible executions with maximum data staleness using the adversary argument. Thus we obtain the bound of data staleness. We can also calculate the probability of atomicity violation by analyzing the probabilities of RI and WI.

We implement the fast data access algorithms in the quorum-replicated datastore Cassandra [3], [19]. We evaluate the consistency-latency tradeoffs based on the YCSB benchmark framework [20], [21]. The experiments not only show the accuracy of the theoretical analysis, but also show the effects of practical optimizations which are hard to analyze theoretically. The theoretical analysis and the experimental evaluations show the ASC tradeoff can be achieved by the one round-trip read and two round-trip write algorithm, even when faced with dynamic changes in the cloud computing environment.

The rest of this paper is organized as follows. Section 2 presents the algorithm schema for fast implementations. Section 3 and Section 4 present the theoretical analysis and the experimental evaluations respectively. Section 5 reviews the existing work. In Section 6, we conclude this work and discuss the future work.

2 Quorum-based Algorithm Schema

In this section we first present the system model and the consistency model. Then we propose the algorithm schema and discuss possible design options in implementing fast algorithms for the ASC tradeoff.

2.1 System Model

The replicated datastore consists of N *servers* that communicate with each other through point-to-point message communication. Each server stores one replica of the data item, and all the replicas collectively emulate the *shared register* abstraction for the *clients*, as shown in Fig. 1. The shared register is identified by its *key* and can be accessed through the *read* ($value \leftarrow read(key)$) and *write* ($write(key, value)$) operations. Since the data is replicated and can be updated concurrently, multiple versions of logically the same data may co-exist. We assume the asynchronous non-Byzantine

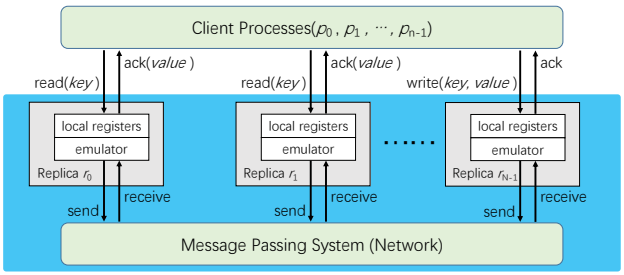

Fig. 1. System model of read/write register emulation.

model, where messages can be delayed, lost or delivered out of order due to process or link failures, but they will not be corrupted or duplicated. Besides, any number of clients and any minority of servers (less than $\frac{N}{2}$) may crash at any moment.

2.2 Consistency Model

We define an *execution history* (or *history* for short) of the clients accessing the shared register as a sequence of events where each event is either the invocation or the response of a read or write operation. As for shared registers that appear in the history, we assume that all operations in the history are applied to the same register. Note that this assumption is not restrictive. The consistency models considered in this work (atomicity and k-atomicity defined below) are local, i.e., for a history with multiple shared registers, it is atomic/k-atomic if and only if for each register accessed, the sub-history is atomic/k-atomic [22]. Given the locality of the consistency models, we can manage each data item independently in the replicated datastore.

Each event in the history is tagged with a unique time, and events appear in the history in increasing order of their timestamps. For a history σ, we can define the partial order between operations. Let $o.s$ and $o.f$ denote the timestamps of the invocation and the response events of operation o respectively. We define $o_1 \prec_\sigma o_2$ if $o_1.f < o_2.s$. We define $o_1 || o_2$ if neither $o_1 \prec_\sigma o_2$ nor $o_2 \prec_\sigma o_1$ holds. A history σ is *sequential* if σ begins with an invocation, and each invocation is immediately followed by its matching response. A history σ is *well-formed* if for each client p_i, $\sigma|p_i$ (the subsequence of σ restricted on p_i) is sequential. Given the notations above, we can define atomicity:

Definition 2.1. *A replicated datastore satisfies* ***atomicity*** *if, for each of its well-formed histories σ, there exists a permutation π of all operations in σ such that π is sequential and satisfying the following two requirements:*

- [***Real-time requirement***] *If $o_1 \prec_\sigma o_2$, then o_1 appears before o_2 in π.*
- [***Read-from requirement***] *Each read returns the value written by the latest preceding write in π.*

For the sake of defining almost strong consistency in the data staleness dimension, we generalize the definition of atomicity to k-atomicity [22] by generalizing the *read-from* requirement:

Definition 2.2. *A replicated datastore satisfies* ***k-atomicity*** *($k \in \mathbb{Z}^+$) if, for each of its well-formed histories σ, there exists a*

permutation π of all the operations in σ such that π is sequential and satisfying the following two requirements:

- [***Real-time requirement***] *If $o_1 \prec_\sigma o_2$, then o_1 appears before o_2 in π.*
- [***Parameterized read-from requirement***] *Each read returns the value written by one of the latest k preceding writes in π.*

Given the definitions above, it is obvious to see that atomicity is equivalent to 1-atomicity. Besides, if a history σ satisfies k-atomicity, then $\forall k' > k$, σ satisfies k'-atomicity $(k, k' \in \mathbb{Z}^+)$. In the following sections, when we say a history σ is k-atomic, we refer to the minimum k for which σ is k-atomic.

2.3 The Algorithm Schema of Read/Write Register Emulation

We first present the basic primitives for client-server interaction. Then we propose the algorithm schema by following the round-trips of client-server interaction we can tune and get 4 concrete algorithms. We also analyze the semantic guarantees provided by these 4 algorithms, which are essential to our analysis in the following Section 3.

2.3.1 The "Diamond" Schema and 4 Concrete Algorithms

The clients can obtain data updates from the replicas via the *query* operation, and can modify the replicas via the *update* operation. When a writer client updates the data, a sequence number paired with the id of the writer client, forming the version as $ver = (seq, id)$, would be attached to the data. Note that the version values are unique and totally ordered according to the lexicographical order, as in [13]. Upon receiving a query request from a client, the server replies to the client with the data. Upon receiving an update request, the server updates its replica if the data from the client is attached with a larger version value and replies to the client with an ACK. The implementation is presented in Algorithm 1. Note that the client-server interaction described here is abstract. Often it is implemented in such a way that the client contacts one replica, and this replica contacts all other replicas and replies to the client on behalf of all replicas, as in Cassandra [3], [19].

The interaction between clients and servers can be captured by a quorum system [17], [18]. Viewing a quorum system from the space dimension, clients need to contact multiple replicas to perform a query or an update, and the set of replicas contacted each time is called a query quorum or an update quorum respectively. All the query and update quorums form a quorum system if any two quorums have non-empty intersection. The intersection between quorums enables the data updates to be propagated among the clients and the server replicas. In this work we adopt the simple but efficient majority quorum system, where each query or update quorum contains a majority (more than $\frac{N}{2}$) of replicas. Viewing the quorum system from the time dimension, clients may communicate with the replicas via one or more round-trips of communication. The number of round-trips is the most important factor deciding data access latency.

Given the primitives for client-server interaction, the replicas can collectively emulate the *write* and the *read* operations for clients, as shown in Algorithm 2 and 3. To emulate an MWMR atomic register, both the write and the read operations require 2 round-trips of client-server interaction [13]. As for the write operation, the client first collects versions from a majority of replicas in the first round-trip. Then, it constructs a new version by increasing the sequence number of the largest version obtained by 1, and replacing the id with its own one. In the second round-trip, the client updates the data together with the new version to a majority of replicas. As for the read operation, the client first collects data from a majority of replicas and select the data with the largest version. Then, the client employs an additional round-trip to write-back the data into a majority of replicas.

The two round-trips of both write and read operations are able to strictly guarantee atomicity. Bearing ASC in mind, we can tune the write and/or read operations to one single round-trip in order to reduce latency. Specifically, the one round-trip write algorithm may omit the first round-trip of querying versions from replicas and directly update the replicas with a version constructed locally. The one round-trip read algorithm may omit the second round-trip of writing-back data and directly return the queried data that has the largest version value.

By tuning the number of round-trips of the write and the read operations, we can obtain 4 variants - namely W2R2, W2R1, W1R2 and W1R1 - of read/write register emulation algorithms. The W2R2 algorithm employs two round-trips for both read and write operations, and other 3 algorithms are named in a similar way. The 4 algorithms form a lattice as shown in Fig. 2. The lattice in the figure can be viewed

Algorithm 1: Client-server interaction

```
1  ▷ Code for client process p_i(0 ≤ i ≤ n − 1):
2  function query(key)
3      vals ← ∅
4      pfor each server s_j                    ▷ pfor: parallel for
5          send ['query', key] to s_j
6          v ←[key, val, ver] from s_j
7          vals ← vals ∪ v
8      until a majority of them respond
9      return vals
10 function update(key, value, version)
11     vals ← ∅
12     pfor each server s_j
13         send ['update', key, value, version] to s_j
14     wait for ['ACK']s from a majority of them
15 ▷ Code for server process s_i(0 ≤ i ≤ N − 1):
16 upon receive ['query', key] from p_j
17     send ['query − back', key, val, ver] to p_j
18 upon receive ['update', key, value, version] from
     p_j
19     pick [k, val, ver] with k == key
20     if ver < version then
21         val ← value
22         ver ← version
23     send ['ACK'] to p_j
```

Algorithm 2: Write algorithms for client p_i

```
1  procedure TwoRoundWRITE(key, value)
2      replicas ← query(key)
3      version ←(maxSeq(replicas)+1, i)
4      update(key, value, version)
5  procedure OneRoundWRITE(key, value)
6      localSeq[key] ← localSeq[key]+1
7      version ←(localSeq[key], i)
8      update(key, value, version)
```

Algorithm 3: Read algorithms for client p_i

```
1  procedure TwoRoundREAD(key)
2      replicas ← query(key)
3      version ←maxVer(replicas)
4      value ←valWithMaxVer(replicas, version)
5      update(key, value, version)
6      return value
7  procedure OneRoundREAD(key)
8      replicas ← query(key)
9      version ←maxVer(replicas)
10     value ←valWithMaxVer(replicas, version)
11     return value
```

as the Hasse diagram of the 4 algorithms. The algorithms in the upper layer provides stronger consistency guarantees than those in the lower layer, which is discussed in detail in the following Section 2.3.2.

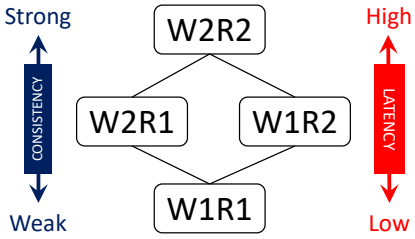

Fig. 2. Diamond schema of shared register emulation algorithms.

2.3.2 Consistency Guarantees Provided by the 4 Algorithms

In general, the more round-trips employed in the write or read algorithms, the more consistency guarantees provided. To define a fine-grained metric for measuring different levels of consistency guarantees, we explore the *monotonicity* properties between read and write operations. Basically, monotonicity means that when the operations have a certain temporal order in the history, they will have a certain semantic order concerning the version values of replicas. For any two operations, each could be either a write or a read. Thus we have four combinations, and define four types of monotonicity. We study which types of monotonicity that the 4 algorithms under the diamond schema can guarantee, as shown in Table 1.

Write-read monotonicity. All 4 algorithms have the write-read monotonicity that $w \prec_\sigma r \Rightarrow ver(w) \leq ver(r)$. This is guaranteed by the intersection requirement of quorum systems. When a read starts after a write has finished, the read will contact at least one replica which has been modified by the write due to the quorum intersection property. Thus, the version value obtained by the read is at least as large as that of the preceding write.

TABLE 1
4 types of monotonicity of the 4 algorithms

Properties	W2R2	W2R1	W1R2	W1R1
$w \prec_\sigma r \Rightarrow ver(w) \leq ver(r)$	√	√	√	√
$w \prec_\sigma w' \Rightarrow ver(w) < ver(w')$	√	√	×	×
$r \prec_\sigma r' \Rightarrow ver(r) \leq ver(r')$	√	×	√	×
$r \prec_\sigma w \Rightarrow ver(r) < ver(w)$	√	×	×	×

Write-write monotonicity. As for the monotonicity between writes for multi-writer registers, $w \prec_\sigma w' \Rightarrow ver(w) < ver(w')$ can be guaranteed only by the two round-trip write algorithms (i.e., W2R2 and W2R1). First employing one round-trip to obtain the largest version value is key to avoiding assigning a version value smaller than those of the previous writes.

Read-read monotonicity. As for the monotonicity between reads for multi-reader registers, $r \prec_\sigma r' \Rightarrow ver(r) \leq ver(r')$ can be guaranteed only with the two round-trip read algorithms (i.e., W2R2 and W1R2). The write-back process in the second round-trip is key to preventing later reads from returning more stale values.

Read-write monotonicity. As for the property $r \prec_\sigma w \Rightarrow ver(r) < ver(w)$, it can be guaranteed by the W2R2 algorithm only. As for the preceding read, the second round-trip writes back the newly acquired data. For the write, the first round-trip will first queries the replicas. The quorum intersection property guarantees that the version value of the read is smaller than that of the write. If either the write or the read employs only one round-trip, this property cannot be guaranteed.

The detailed proof of all the properties above can be found in Section 1 in the appendix. In the following Section 3, we study the consistency guarantees provided by algorithms following the diamond schema. We mainly discuss the W2R1 algorithm, which provides almost strong consistency. The W1R2 and W1R1 algorithms are briefly discussed since they cannot provide sufficient consistency guarantees.

3 Consistency Guarantees of W2R1: Data Staleness and Violation Probability

In this section, we mainly analyze the consistency guarantees provided by the W2R1 algorithm. The keys to our analysis are the patterns named *read inversion* and *write inversion*. We first transform all inconsistent reads into these two patterns. Then we leverage these two patterns to analyze the bound of data staleness and the probability of atomicity violation. After the analysis of the W2R1 algorithm, we also briefly discuss the consistency guarantees provided by other algorithms in the diamond schema.

3.1 Atomicity Violation Patterns

In the one round-trip read procedure of W2R1, the absence of the *write-back* phase before returning the data to the client

may violate the read-read monotonicity and the read-write monotonicity, as shown in Table 1. According to the violation of these two monotonicity properties, we can define two essential patterns of inversions accordingly:

Definition 3.1 (Read Inversion)**.** *The Read Inversion after a read (RI) involves two reads r, r' satisfying $(r \prec_\sigma r') \wedge (ver(r) > ver(r'))$.*

Definition 3.2 (Write Inversion)**.** *The Write Inversion after a read (WI) involves a read r and a write w satisfying $(r \prec_\sigma w) \wedge (ver(r) > ver(w))$.*

The RI and WI patterns are essential to further analysis of both the bound of data staleness and the probability of atomicity violation, which is depicted by the following Theorem 3.1. When the clients read and write a MWMR register using the W2R1 algorithm, and obtain the history σ, we have that:

Theorem 3.1. *If σ violates atomicity, then there exist some operations in σ that form either RI or WI.*

If σ violates atomicity, then for any permutation π of σ, we have a stale read r, the dictating write w of r and the interfering write w' satisfying: $w \prec_\pi w' \prec_\pi r$. To see why we inevitably have RI or WI, we exhaustively check all cases.

According to the definition of atomicity, two types of relations between operations are of our concern: the temporal real-time relation and the semantic read-from relation. We first enumerate all possible cases according to the semantic relation. Then for each case, we further enumerate all possible sub-cases according to the temporal relation.

Since w is the dictating write of r, we have that $ver(r) = ver(w)$. According to the semantic relation between versions of w and w', we have two complementing cases:

- Case 1: $ver(r) = ver(w) < ver(w')$.
- Case 2: $ver(r) = ver(w) > ver(w')$.

In each case, we then consider the temporal relation between operations, as shown in Fig. 3.

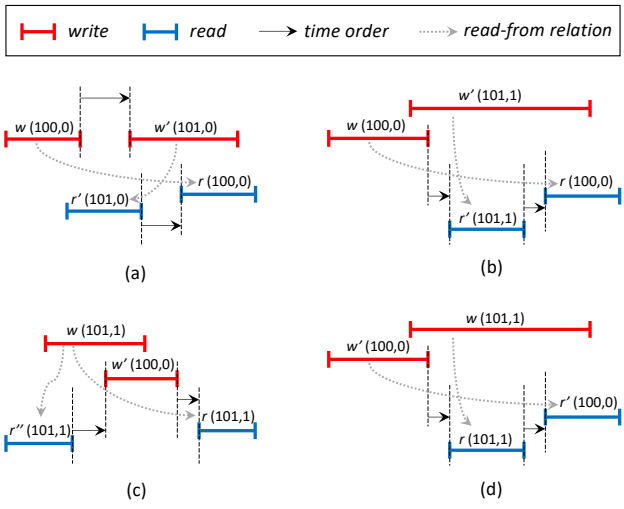

Fig. 3. Typical patterns of RI in Case 1 (Sub-fig (a) and (b)), and of WI (Sub-fig (c))and RI (Sub-fig (d)) in Case 2.

We discuss the basic rationale of the proofs of Case 1 and Case 2 in the following Section 3.2.1 and 3.2.2 respectively. Detailed proof of the theorem, which exhaustively checks all possible cases and constructs the RI or WI, is provided in Section 2.1 in the appendix.

3.1.1 *Case 1:* $ver(r) = ver(w) < ver(w')$

In Case 1, we further consider the temporal relation between w and w'. Since in the permutation π, $w \prec_\pi w'$, we have that in σ, $w \prec_\sigma w'$ or $w||_\sigma w'$, as shown in Fig. 3 (a) and (b) respectively.

Note that r must be concurrent with w'. This can be proved by contradiction. If $r \prec_\sigma w'$, w' can never be the interfering write of r. If $w' \prec_\sigma r$, according to the write-read monotonicity (see Table 1), r must return the version of w' (or return an even larger version), contradicting the fact that $ver(r) = ver(w) < ver(w')$.

When linearly extend σ into π, we inevitably have $w \prec_\pi w' \prec_\pi r$. Thus for concurrent operations w' and r, in order to "force" w' to appear before r in π, we must have a read operation r' satisfying that w' dictates r' and $r' \prec_\sigma r$. Thus we construct a certificate of RI, i.e., $r' \prec_\sigma r$, but $ver(r') > ver(r)$.

3.1.2 *Case 2:* $ver(r) = ver(w) > ver(w')$

Since $ver(r) = ver(w) > ver(w')$, w must be concurrent with w'. This can also be proved by contradiction. If $w \prec_\sigma w'$, according to the write-write monotonicity (see Table 1), the version of w' must be larger. If $w' \prec_\sigma w$, we can never get $w \prec_\pi w' \prec_\pi r$ when linearly extending σ to π.

Given that w and w' are concurrent, we must "force" w' to appear after w when linearly extending σ into π. This can be achieved in several different ways.

In one subcase, we have r'', satisfying that r'' is dictated by w and $r'' \prec_\sigma w'$, as shown in Fig. 3 (c). In this subcase, we got a certificate of WI, i.e., $r'' \prec_\sigma w'$, but $ver(r'') > ver(w')$.

In another typical subcase, we have a read r' dictated by w', and a read dictated by w, e.g. r, satisfying $w' \prec_\sigma r \prec_\sigma r'$, as shown in Fig. 3 (d). Besides, there exists no dictated read of w' that precedes w, so w' may appear after w in π and be the interfering write of r. In this subcase, we have a certificate for RI, i.e., $r \prec_\sigma r'$, but $ver(r) > ver(r')$.

3.2 Bound of Data Staleness

In this section, we calculate the tight bound of data staleness when accessing a MWMR register using the W2R1 algorithm. Denote the number of writer clients as n_w. Then we have that:

Theorem 3.2. *For any history σ, there exists a linear extension π of σ such that any read in π returns the value written by one of the latest B preceding writes. Here, $B = n_w + \frac{1}{2}n_w(n_w - 1) + 1$. Moreover, the bound B is tight, i.e., there exists a history σ and a linear extension π of σ in which some read returns the value written by the oldest write in the latest B preceding writes.*

The proof of Theorem 3.2 is based on an adversary argument: to insert as many inevitable interfering writes as possible for any inconsistent read. The proof also needs to consider the same two cases as defined in Section 3.1. Specifically, suppose r is an inconsistent read, and the dictating write of r is w. There must exist another interfering write w' such that $w \prec_\pi w' \prec_\pi r$.

According to the versions $ver(r)$ and $ver(w')$, we consider two cases. In Case 1 where $ver(r) = ver(w) < ver(w')$, we prove that there are at most $B_1 = n_w$ interfering writes which can inevitably be inserted between r and w in π. In Case 2 where $ver(r) = ver(w) > ver(w')$, we prove that there are at most $B_2 = \frac{1}{2}n_w(n_w - 1)$ interfering writes.

Since these two cases can appear at the same time in the history, we can construct a trace with $B_1 + B_2$ interfering writes. Counting the dictating write itself, we have that the read always returns the value written by one of the latest $B = B_1 + B_2 + 1$ preceding writes in π, i.e., the history σ always satisfies B-atomicity. During the proof, we explicitly construct the worst-case trace, which contains the read having $B_1 + B_2$ interfering writes. This proves that the bound is tight.

In the following Section 3.2.1 and 3.2.2, we derive the bound B_1 and B_2 respectively. Detailed proof of the bound B can be found in Section 2.2 in the appendix.

3.2.1 Bound of Staleness in Case 1 ($ver(r) < ver(w')$)

In this case, the pattern of operations which enables our construction of the trace with the most stale read is the pattern shown in Fig. 3 (a). The construction is illustrated in Fig. 4.

Since we have n_w writers in total and w and w' may not overlap, after the write operation w we force all n_w writers (including the writer of w itself) to issue the interfering writes w'. Thus we have the bound $B_1 = n_w$ in this case, and the bound B_1 is tight.

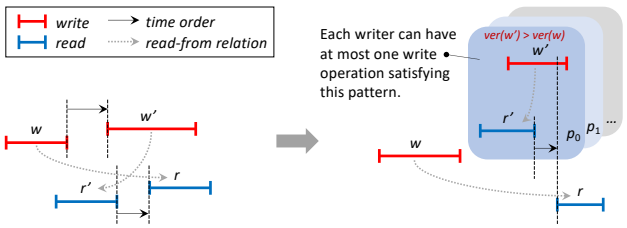

Fig. 4. Maximum number of interfering writes in Case 1.

3.2.2 Bound of Staleness in Case 2 ($ver(r) > ver(w')$)

In this case, the pattern of operations which enables our construction of the worst-case trace is the WI pattern, as shown in Fig. 3 (c).

In WI, since all the interfering writes w' must be concurrent with w, other than the writer of operation w, we can have at most $n_w - 1$ writers to generate interfering writes. The challenge in the construction is that, for each interfering writer client, we cannot insert as many interfering write operations as we want, due to the constraint of $ver(w') < ver(w)$ and $r'' \prec_\sigma w'$.

The further reasons can be described as follows. On the one hand, the interfering write operation w' will have the version smaller than $ver(w)$ only when w' queries a majority of replicas whose maximum version is smaller than $ver(w)$ in the first round-trip. Therefore, for adversary argument we should keep the maximum version of some replica majority as small as possible before w is finished. On the other hand, w will not have $ver(w) = (seq, id)$ unless it queries a majority of replicas whose maximum version is $(seq - 1, id')$ in the first round-trip. Therefore, there must exist some writer holding the sequence number not small than $seq - 1$, and this writer is not able to issue interfering write operations versioned smaller than $(seq - 1, id')$.

For the ease of illustration, we use a concrete example as shown in Fig. 5. Assume $n_w = 4$, and the writer clients are p_0, p_1, p_2 and p_3. Let the operation w issued by p_3 have the version value $(100, 3)$. Client p_3 can write with this version value only when it queries a majority of replicas whose maximum version is $(99, id)$, where id can be any writer's identifier. Let p_2 have the write operation with the version value $(99, 2)$. Similarly, let p_1 have the write operation versioned $(98, 1)$, so p_2 can write with the version value $(99, 2)$; let p_0 have the write operation versioned $(97, 0)$, so p_1 can write with the version value $(98, 1)$.

Assume r'' reads from w versioned $(100, 3)$. Since r'' won't write back the acquired data into replicas, after r'' finishes, there may still exist a majority of replicas whose maximum version is smaller than $(100, 3)$. Now we focus on what could happen after r'' finishes. Assume that when p_0 finishes the write operation versioned $(97, 0)$, none of p_1, p_2 or p_3 has finished the write operation versioned $(98, 1)$, $(99, 2)$ or $(100, 3)$ respectively. Thus, p_0 may query a majority of replicas whose maximum version value is $(97, 0)$, then issue an interfering write operation versioned $(98, 0)$ and later issue interfering write operations versioned $(99, 0)$ and $(100, 0)$. Similarly, when p_1 finishes the write operation versioned $(98, 1)$ while none of the write operations versioned larger than $(98, 1)$ has been finished, a majority of replicas may still hold the maximum version $(98, 1)$. Then, p_1 can issue interfering write operations versioned $(99, 1)$ and later $(100, 1)$. As for p_2, after it finishes the write versioned $(99, 2)$ while all the write operations versioned larger than $(99, 2)$ stay unfinished, p_2 can still make an interfering write operation versioned $(100, 2)$.

Generalize the typical example above, we have that for all the $n_w - 1$ interfering writer clients, we can insert:

$$B_2 = 1 + 2 + \cdots + (n_w - 2) + (n_w - 1) = \frac{1}{2}n_w(n_w - 1)$$

interfering writes.

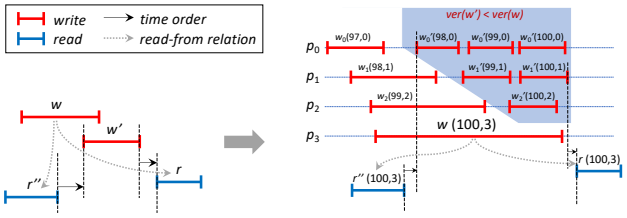

Fig. 5. Maximum number of interfering writes in Case 2.

3.3 Probability of Atomicity Violation

The probability calculation of atomicity violation is also based on destructing the violation into the RI and WI patterns. To further calculate the probabilities of RI and WI, we reduce the calculation in the multi-writer case to that in the single-writer case.

First note that the occurrence of either RI or WI is the necessary but not sufficient condition for atomicity violation, as shown in Fig. 6, but this will not prevent us from obtaining the upper bound of the violation probability:

$$\begin{aligned}\mathbb{P}\{\text{Violation}\} &\leq \mathbb{P}\{\text{RI} \vee \text{WI}\} \\ &= \mathbb{P}\{\text{RI}\} + \mathbb{P}\{\text{WI}\} - \mathbb{P}\{\text{RI} \wedge \text{WI}\} \\ &\leq \mathbb{P}\{\text{RI}\} + \mathbb{P}\{\text{WI}\} \end{aligned} \tag{3.1}$$

Fig. 6. The relation between atomicity violation and the occurrences of RI and WI.

To calculate the probabilities of RI and WI, we view RI and WI as the concurrent occurrences of multiple atomicity violation patterns in the single-writer case. According to our previous work [16], atomicity violation in the single-writer case is equivalent to the pattern named Old New Inversion (ONI). We further destruct the ONI into the temporal Concurrency Pattern (CP) and the semantic Read Write Pattern (RWP). Then we obtain the probability of ONI:

$$\begin{aligned}\mathbb{P}\{\text{ONI}\} &= \sum_{m\geq 1} \mathbb{P}\{\text{CP} \mid R' = m\} \cdot \mathbb{P}\{\text{RWP} \mid R' = m\} \\ &\approx \sum_{m=1}^{n_r} \left(\left(\sum_{k=0}^{n_r-1} \binom{n_r}{k} \binom{m-1}{n_r-k-1} p_0^k r^{n_r-k} s^m \right) \right. \\ &\quad \cdot e^{-q\lambda_w t} \frac{\alpha^q B(q, \alpha(N-q)+1)}{B(q, N-q+1)} \\ &\quad \left. \cdot \left(1 - \left(\frac{J_1}{B(q, N-q+1)} \right)^m \right) \right) \end{aligned} \tag{3.2}$$

where R' is a random variable denoting the number of reads r' in Definition 3.1. Please refer to our previous work [16] for detailed explanations of Equation 3.2.

Given the analysis of the ONI in the single-writer case, we can see that ONI is a special case of RI when there exists only one writer client. As for WI, it can be modeled by ONI in principle with some extra mappings of operations (see Section 2.3 in the appendix). By modeling the occurrences of multiple writers, we can calculate the probabilities of RI and WI based on that of ONI. We first model the occurrences of multiple writers with the following queuing model. The workload of each (reader or writer) client is modeled as an independent queue characterized by the rate of operations and the service time of each operation. We assume a Poisson process with parameter λ for the scenario of each client issuing a sequence of read/write operations, and assume an exponential distribution with parameter μ for the service time of each operation. We then have n independent, parallel $M/M/1/1/\infty/FCFS$ queues Q_i $(1 \leq i \leq n)$ (i.e., a single-server exponential queuing system, whose capacity is 1 with the "first come first served" discipline) [23]. All queues have arrival rate λ and service rate μ. For each queue, if there is any operation in service, no more operations can enter it. Let $X^i(t)$ be the number of operations in queue i at time t. Then $X^i(t)$ is a continuous-time Markov chain with two states: 0 when the queue is empty and 1 when some operation is being served. Its stationary distribution $P_s \triangleq P(X^i(\infty) = s), s \in \{0, 1\}$ is: $P_0 = \frac{\mu}{\mu+\lambda}$ and $P_1 = \frac{\lambda}{\mu+\lambda}$.

As for RI, given a read operation r in Q_i, let W'_{cr} be a random variable denoting the number of writes w' satisfying $r_{st} \in [w'_{st}, w'_{ft}]$ in RI. The probability that r starts during the service period of some write w' in Q_j equals the probability that when r arrives Q_i, it finds Q_i empty, as well as finds Q_j full as a bystander (with the constraint that Q_j is a writer queue). Since the events in different queues are independent, by the PASTA property [23], we have:

$$\mathbb{P}\{W'_{cr} = x\} = \binom{n_w}{x} \left(\frac{\lambda}{\mu+\lambda} \right)^x \left(\frac{\mu}{\mu+\lambda} \right)^{n_w-x+1} \tag{3.3}$$

Conditional on $W'_{cr} = x$, the probability of RI is no more than the sum of each w' forming RI with r separately. By considering all possible choices of w', we have:

$$\begin{aligned}\mathbb{P}\{\text{RI}\} &= \sum_{x=1}^{n_w} \mathbb{P}\{W'_{cr} = x\} \cdot \mathbb{P}\{\text{RI} \mid W'_{cr} = x\} \\ &\leq \sum_{x=1}^{n_w} \mathbb{P}\{W'_{cr} = x\} \cdot x\mathbb{P}\{\text{ONI}\} \end{aligned} \tag{3.4}$$

As for WI, given a specific write w' in Definition 3.2, let $W_{cw'}$ be a random variable denoting the number of writes w satisfying $w'_{st} \in [w_{st}, w_{ft}]$ in WI. Similarly, we have:

$$\mathbb{P}\{W_{cw'} = x\} = \binom{n_w-1}{x} \left(\frac{\lambda}{\mu+\lambda} \right)^x \left(\frac{\mu}{\mu+\lambda} \right)^{n_w-x} \tag{3.5}$$

Then we can bound the probability of WI by:

$$\begin{aligned}\mathbb{P}\{\text{WI}\} &= \sum_{x=1}^{n_w-1} \mathbb{P}\{W_{cw'} = x\} \cdot \mathbb{P}\{\text{WI} \mid W_{cw'} = x\} \\ &\leq \sum_{x=1}^{n_w-1} \mathbb{P}\{W_{cw'} = x\} \cdot x\mathbb{P}\{\text{ONI}\} \end{aligned} \tag{3.6}$$

Substituting Formula (3.2)-(3.6) into (3.1), we obtain a bound of the probability of atomicity violation:

$$\begin{aligned}&\mathbb{P}\{\text{Violation}\} \\ &\leq \sum_{x=1}^{n_w} \mathbb{P}\{W'_{cr} = x\} \cdot x\mathbb{P}\{\text{ONI}\} + \sum_{x=1}^{n_w-1} \mathbb{P}\{W_{cw'} = x\} \cdot x\mathbb{P}\{\text{ONI}\} \\ &\approx \frac{(2n_w-1)\lambda\mu}{(\lambda+\mu)^2} \cdot \sum_{m=1}^{n_r} \left(\left(\sum_{k=0}^{n_r-1} \binom{n_r}{k} \binom{m-1}{n_r-k-1} p_0^k r^{n_r-k} s^m \right) \right. \\ &\quad \left. \cdot e^{-q\lambda_w t} \frac{\alpha^q B(q, \alpha(N-q)+1)}{B(q, N-q+1)} \cdot \left(1 - \left(\frac{J_1}{B(q, N-q+1)} \right)^m \right) \right) \end{aligned}$$

To better illustrate the theoretical analysis results, we present the numerical results in Fig. 7. The results in Fig. 7 are obtained when setting $\lambda = \mu = 10s^{-1}$ and $\lambda_r = \lambda_w = 20s^{-1}$. The source code for calculation of the numerical results can be found in our open source project on line [1]. More detailed numerical results can be found in Section 2.3 in the appendix. The numerical results show that

1. https://github.com/Lingzhi-Ouyang/Almost-Strong-Consistency-Cassandra/tree/master/numerical_results

the probability of atomicity violation is quite low (mostly below 0.1% and can be below 10^{-9}) and will decrease when the number of replicas increases. Moreover, the probability of the concurrency pattern is close to 1. The probability of the read-write pattern is significantly less and decreases as the number of replicas increases. The probability of the read-write pattern ensures that the probability of the atomicity violation is almost zero. We will conduct more comprehensive experimental evaluations in Section 4, which further confirm our theoretical analysis results.

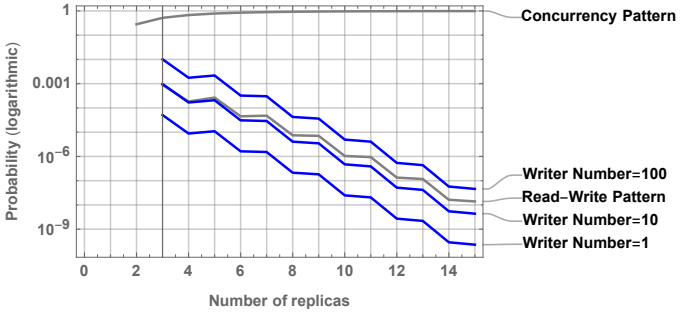

Fig. 7. The probabilities of atomicity violation.

3.4 Discussions

We also analyze the bound of data staleness and the probability of atomicity violation for the W1R2 and W1R1 algorithms. Although for SWMR registers, the W1R2 algorithm (also recognized as the ABD algorithm in the literature [12]) guarantees atomicity and the W1R1 algorithm (also called the PA2AM algorithm in [16]) achieves the ASC tradeoff, for MWMR registers, we observe that neither the W1R2 algorithm nor the W1R1 algorithm can provide sufficient consistency guarantees in terms of the data staleness and the probability of atomicity violation.

The reason mainly lies in the one round-trip write algorithm. Since the writers try to directly update the replicas without querying the existing versions of them first, the updates may often take no effect on the replicas. Specifically, it occurs frequently that, after a majority of replicas have already been updated with a larger version, some writer still tries to update the data with a smaller version. Moreover, the data written by the no-effect writes will be "invisible" to any following read. Clients may miss an arbitrary number of updates, so the data returned by a read operation can be arbitrarily stale.

The theoretical analysis and numerical results for the W1R2 and W1R1 algorithms are provided in Section 3.1 and 3.2 in the appendix. These theoretical analysis results are also confirmed by further experimental evaluations, as shown in Section 3.3 in the appendix.

4 Experiments and Evaluations

In this section, we conduct experiments to study whether the W2R1 algorithm can achieve the ASC tradeoff in quorum-replicated datastores. We first explain important implementation details and present the experiment design. Then we discuss the evaluation results.

4.1 Implementation

Our implementation is based on the open-source distributed datastore Cassandra [3], [19], which enables high availability and low data access latency based on quorum replication. Our implementation basically reuses the quorum mechanism in Cassandra except that we implement the versioning of replicas. Cassandra relies on synchronized clocks among replica servers and uses timestamps following the UTC standard, while following the diamond schema, we use the discrete timestamp $ver = (seq, id)$. We transfer the 64-bit timestamp of Cassandra into a pair of integers, letting the higher 32-bits store the sequence number and the lower 32-bits store the process id. In this way, we can reuse the timestamp maintenance and comparison schemes of Cassandra.

Besides, Cassandra implements a variety of optimizations, which are beyond accurate modeling in the theoretical analysis. Thus, we implement knobs which enable users to turn on/off the optimizations. Tuning off all the optimizations enables us to validate our theoretical analysis with the experimental evaluations. Turning on one optimization (and turning off other optimizations) each time enables us to study in depth the effect of this optimization. The optimizations we consider in the experiments include:

- *Snitch*. Cassandra uses the snitch mechanism to determine relative host proximity for each node and select a preferred group of replicas for data access [24].
- *Digest*. During a read, Cassandra makes a number of replicas (determined by the consistency level) send digest messages except the nearest one (decided by the snitch mechanism). This is speculative and can often save bandwidth. When the messages hold different versions of data, the replicas are forced to retransmit the concrete data [24].
- *Read repair*. During a read, besides returning the latest replica to the client, Cassandra may find some servers holding stale replicas. The read repair mechanism makes the servers that hold stale replicas synchronize their data in the background [24].

4.2 Experiment Design

The experiments are conducted on a ThinkStation P710 server with the Intel Xeon(R) E5-2620 8-core 16-thread CPU(2.10GHz), equipped with 64 GB DDR4 memory and 7200 RPM SATA disks. The operation system is Ubuntu Linux 18.04. We run up to 9 instances of Cassandra to simulate a cloud storage system with up to 3 data centers, each consisting of no more than 3 instances, as shown in Fig. 8. Each Cassandra instance, also denoted a node, acts as a server replica in our system model.

We implement the client reading and writing algorithms using the database interface provided by the YCSB framework [20], [21]. We also use YCSB to generate a variety of workloads for our experiments. Here we only present the evaluation results concerning the W2R1 algorithm. We evaluate the W2R1 algorithm with no optimizations (denoted W2R1), as well as the W2R1 algorithm with only one optimization turned on (denoted Snitch, Digest and Repair respectively). Moreover, we implement the W2R2 algorithm, mainly for the sake of performance comparison. Evaluation

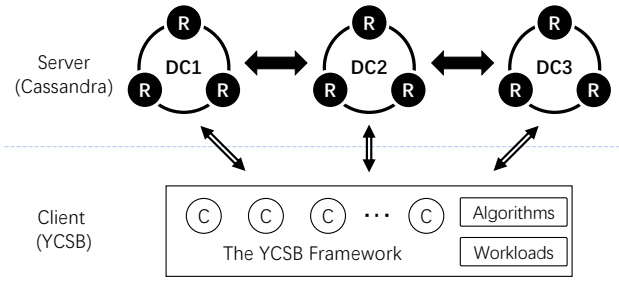

Fig. 8. Architecture of the experiment system.

results concerning the W1R2 and W1R1 algorithms are presented in Section 3.3 in the appendix.

The performance metrics are the read latency for the clients and the data consistency provided. As for data consistency, we consider both the staleness of accessed data (measured by the k-atomicity model) and the probability of atomicity violation.

In the first experiment, we study whether the W2R1 algorithm can achieve the ASC tradeoff. The evaluation is conducted under the default environment setting. The default workload pattern is: 30 clients issuing 3,000 operations each. Among all the operations, 90% are read operations and others are write operations. Each client issues operations at the speed of 5 operations per second. The default replica configuration is one replica in each of the three data centers. The injected one-way inter-data center delays of network communication are normally distributed with the average of 50ms and the standard deviation of 25ms (denoted $\mathbb{N}(50, 25^2)$), while the intra-data center delays follow $\mathbb{N}(5, 1^2)$. The delays between clients and servers also follow $\mathbb{N}(5, 1^2)$. Each experiment results reported are the average over 10 runs.

In the second experiment, we study whether the ASC tradeoff can be achieved in different environments. We mainly tune three types of environment factors. We first tune the workload parameters, including the client number and the read ratio. Then we tune the number of replicas in each data center. Finally, we tune the communication delay. In each experiment, only one parameter would be tuned while others remain the default values. The experiment configurations are listed in Table 2.

4.3 Evaluation Results

4.3.1 The ASC Tradeoff

From the evaluation results under the default setting, as shown in Table 3, we can see that W2R1 can achieve the ASC tradeoff. Specifically, as for data consistency, W2R1 produces up to 3-atomic traces in the worst case, but the probabilities $\mathbb{P}(k = 2)$ and $\mathbb{P}(k = 3)$ are quite small [2]. The overall probability of violation is less than 0.02%, and the optimizations can further reduce the probability down to approximately 0.002%.

As for read latency, we measure the ratio of the latency of W2R1 to that of W2R2. The read latency of W2R1 (with and without optimizations) is about 60%. This is mainly due to the one round-trip saved. The ratio is not 50% because besides the round-trip communication, the read latency is also affected by other factors, e.g., the processing latency of the replica servers and the waiting time in the buffer due to the concurrency control. With optimizations, the latency can be further influenced in different ways. In particular, the snitch optimization may further reduce read latency by efficiently routing requests, while the repair optimization takes a little more time for replica synchronization. For the digest optimization, the speculative nature and the possible retransmissions may increase the read latency, but the gain is mainly the saving of transmission bandwidth.

2. $\mathbb{P}(k = 1)$ means the proportion of reads that reads from the latest preceding write, $\mathbb{P}(k = 2)$ means the proportion of reads that reads from the second latest preceding write, and so on.

4.3.2 ASC Tradeoffs in Different Environments

We also change the environment settings to explore whether the W2R1 algorithm can achieve the ASC tradeoffs in different environments. As for data consistency, we find that the probabilities of consistent reads in different environments are all over 99.7% and in most cases over 99.97%. The data staleness in the worst-case is $k = 4$. This is much less than the theoretical upper bound. The tight upper bound with 30 clients (at most 29 writer clients and 1 reader client) is $B = 29 + \frac{1}{2} \times 29 \times (29 - 1) + 1 = 436$. Different optimizations have certain impact on the consistency guarantees. However, since the consistency guarantees are quite close to 100% atomicity, the impacts of different optimizations are limited.

As for read latency, the W2R1 algorithm (including W2R1 with different optimizations) can reduce about 40% of time compared to the W2R2 algorithm. Different optimizations do not have significant impacts on the read latency except for the digest optimization. Detailed discussions on the impact of the environment factors are provided in Section 4 in the appendix.

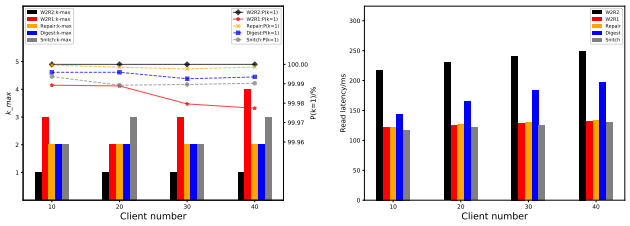

Fig. 9. Consistency and latency results by tuning the number of clients.

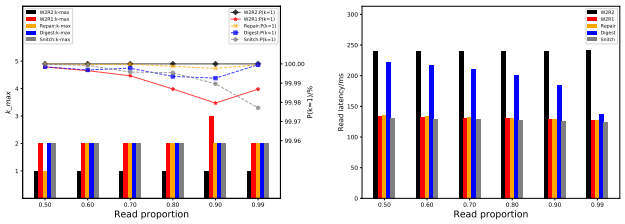

Fig. 10. Consistency and latency results by tuning read ratio.

TABLE 2
Experiment configurations

	Parameter	Tuning values	Default value
Workload	Client number	10 / 20 / 30 / 40	30
	Read ratio	0.5 / 0.6 / 0.7 / 0.8 / 0.9 /0.99	0.9
Replica	Replica configuration*	3 / 1_1_1 / 3_1_1 / 3_3_3	1_1_1
Network	Inter-DC delay/ms	$\mathbb{N}(\mu, \sigma^2)$, μ = 10 / 20 / 30 / 40 / 50, $\sigma = \frac{\mu}{2}$	$\mathbb{N}(50, 25^2)$

* Replica configuration specifies the number of nodes that stores replicas in the Cassandra cluster. For example, *3* means using 3 replicas and storing all of them in one single DC; *3_1_1* means using 5 replicas but storing 3, 1, 1 replicas in DC1, DC2, DC3 respectively.

TABLE 3
ASC tradeoff under the default setting

	k_{max}	$\mathbb{P}(k=1)$	$\mathbb{P}(k=2)$	$\mathbb{P}(k=3)$	$\mathbb{P}(staleness)$	Read latency
W2R2	1	100%	0	0	0	100%
W2R1	3	99.9796%	0.0203%	0.0001%	0.0204%	53%
Snitch	2	99.9896%	0.0104%	0	0.0104%	52.3%
Digest	2	99.9926%	0.0074%	0	0.0074%	76.5%
Repair	2	99.9977%	0.0023%	0	0.0023%	53.9%

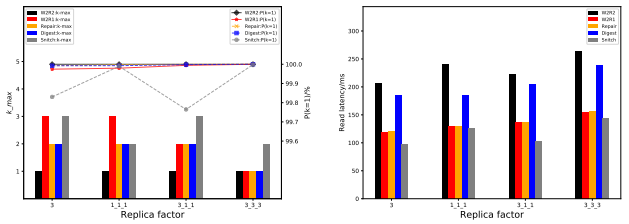

Fig. 11. Consistency and latency results by tuning replica configuration.

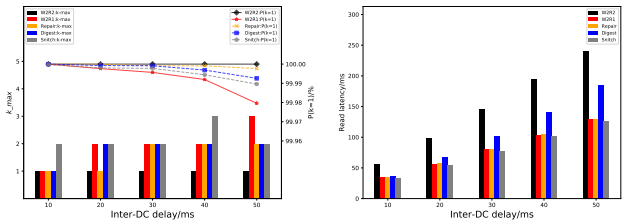

Fig. 12. Consistency and latency results by tuning inter-DC communication delay.

4.3.3 *Discussions*

From the evaluation results we can see that the W2R1 algorithm (including its optimizations) achieves the best of both worlds. Specifically, it achieves both (almost) strong consistency and low latency in the common case. Although the details vary in different workload patterns, replica configurations or network conditions, the ASC tradeoff is achieved in different environments.

Note that the W2R1 algorithm is only a fast read algorithm, and the write operations still require 2 round-trips of communication. However, both the theoretical analysis and the experimental evaluations show that the 2 round-trips of writes are necessary in the multi-writer case when using the version $ver = (seq, id)$. Fast (1 round-trip) write algorithms have poor consistency guarantees when multiple clients can write, no matter the read is fast or not.

5 Related Work

Consistency-latency tradeoff is the essential issue in distributed storage system design. Eventually consistent datastores which ensures low latency have been widely used and commercially successful [1], [2], [3]. The ever growing demand of new application development also calls for datastores providing stronger consistency guarantees. Strong consistency models, e.g., atomicity or linearizability [11] and sequential consistency [25], may greatly ease application development. Spanner [26] supports linearizable distributed transactions, partly motivated by the complaints received from users that Bigtable [27] can be difficult to use for applications that need strong consistency in the presence of wide-area replication. Megastore [28] provides strong consistency guarantees by using semi-relational data model and synchronous replication.

Even when using weak consistency models, stronger (though not strong) consistency semantics may also greatly simplify the resolution of conflicts among replicas and thus ease the development of upper-layer applications. The Eiger system provides causal consistency, which is the strongest consistency guarantee possible when the system can be partitioned [9]. It also supports read-only and write-only transactions. The read-only transactions in Eiger normally complete in one round of local reads, and two rounds in the worst case. The concept of hybrid consistency is presented in [10], [29], with which strict strong consistency can be provided, but only for a selected part of important operations. For not-important operations, eventual consistency is provided. All existing works need to find certain way to circumvent the impossibility results [6], [30], in order to achieve both strong consistency and low latency in certain sense. The ASC tradeoff mainly tackles the challenge by relaxing strictly strong consistency to almost strong consistency, as long as the data inconsistency perceived by the user is close to zero in the average case. In scenarios where low latency is important and the data reads are frequent, the statistically strong consistency guarantees may be well accepted, as long as the read latency can be significantly

reduced.

The emulation of atomic registers in distributed storage systems is the theoretical foundation of the ASC tradeoff. The ABD algorithm [12] uses quorum replication to emulate the atomic SWMR registers in unreliable, asynchronous networks where only a minority of replicas can fail. The ABD algorithm completes each read in two round-trips. For MWMR registers, atomicity can be guaranteed by 2 round-trips of both read and write. It is proved in [15] that when requiring both read and write to be fast (using only one round-trip of communication), it is not possible to guarantee atomicity. This impossibility result motivates us to propose the notion of "almost strong" consistency.

The consistency-latency tradeoff needs to and has been studied via comprehensive experiments. Many storage systems are equipped with tunable quorum mechanisms to meet a variety of consistency requirements in different scenarios [1], [3], [31], [32]. Many practical techniques for tradeoff tuning were developed over the above-mentioned tunable systems [33], [34], [35]. An adaptable SLA-aware consistency tuning framework for quorum-based stores [36] was also implemented and tested on Cassandra. In our previous work [16], the ASC tradeoff is only studied via experiments in a mobile file sharing scenario. In this work, we implement the diamond schema of 4 shared register emulation algorithms over Cassandra, mainly for the sake of exploration of the ASC tradeoff in the MWMR register emulation scenarios.

6 Conclusion and Future Work

In this work we study the ASC tradeoff in quorum-replicated datastores, where all clients can both read and write data replicas. We propose the quorum-based algorithm schema to study the possible design options in low latency read/write implementations. Then we analyze the consistency guarantees in the staleness and the probability dimensions. We implement the low latency data access algorithms in the quorum-replicated data store Cassandra, and evaluate the consistency-latency tradeoffs based on the YCSB benchmark framework. The experiments not only show the accuracy of the theoretical analysis, but also show the effects of practical optimizations which is hard to analyze theoretically. The theoretical analysis and the experimental evaluations show the ASC tradeoff can be achieved by the W2R1 algorithm.

In our future work, we need to prove the lower bound that we inevitably need 2 round-trips for both reads and writes to strictly guarantee atomicity. We also plan to study the ASC tradeoff in large scale geo-replicated datastores, where software/hardware failures and network partitions are common. More consistency metrics for data inconsistency measurement are also needed, in order to interpret the abstract notion of "almost strong" consistency in various realistic application scenarios.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 61690204, 61772258). Yu Huang is the corresponding author.

References

[1] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. ACM, 2007, pp. 205–220. [Online]. Available: http://doi.acm.org/10.1145/1294261.1294281

[2] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008. [Online]. Available: http://dx.doi.org/10.14778/1454159.1454167

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[4] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, "Replicated data types: Specification, verification, optimality," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14. New York, NY, USA: ACM, 2014, pp. 271–284. [Online]. Available: http://doi.acm.org/10.1145/2535838.2535848

[5] S. J. Park and J. K. Ousterhout, "Exploiting commutativity for practical fast replication," in *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019.*, 2019, pp. 47–64.

[6] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC'00. New York, NY, USA: ACM, 2000, p. 7.

[7] S. Gilbert and N. A. Lynch, "Perspectives on the cap theorem," *Computer*, vol. 45, no. 2, pp. 30–36, 2012.

[8] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan, "Declarative programming over eventually consistent data stores," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 413–424. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737981

[9] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 313–328. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482657

[10] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 265–278. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387906

[11] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 463–492, July 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[12] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, Jan. 1995. [Online]. Available: http://doi.acm.org/10.1145/200836.200869

[13] N. A. Lynch and A. A. Shvartsman, "Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts," in *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*, June 1997, pp. 272–281.

[14] L. Lamport, "On interprocess communication. part i: Basic formalism," *Distributed Computing*, vol. 1, no. 2, pp. 77–85, 1986. [Online]. Available: http://dx.doi.org/10.1007/BF01786227

[15] P. Dutta, R. Guerraoui, R. R. Levy, and M. Vukolić, "Fast access to distributed atomic memory," *SIAM J. Comput.*, vol. 39, no. 8, pp. 3752–3783, Dec. 2010. [Online]. Available: http://dx.doi.org/10.1137/090757010

[16] H. Wei, Y. Huang, and J. Lu, "Probabilistically-atomic 2-atomicity: Enabling almost strong consistency in distributed storage systems," *IEEE Trans. Comput.*, vol. 66, no. 3, pp. 502–514, Mar. 2017. [Online]. Available: https://doi.org/10.1109/TC.2016.2601322

[17] M. Naor and A. Wool, "The load, capacity, and availability of quorum systems," *SIAM J. Comput.*, vol. 27,

no. 2, pp. 423–447, Apr. 1998. [Online]. Available: http://dx.doi.org/10.1137/S0097539795281232
[18] M. Vukolic, *Quorum Systems With Applications to Storage and Consensus*. Morgan & Claypool Publishers, 2012.
[19] [Online]. Available: http://cassandra.apache.org
[20] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1807128.1807152
[21] [Online]. Available: https://github.com/brianfrankcooper/YCSB
[22] W. Golab, X. SteveLi, A. López-Ortiz, and N. Nishimura, "Computing k-atomicity in polynomial time," *SIAM Journal on Computing*, vol. 47, no. 2, pp. 420–455, 2018. [Online]. Available: https://doi.org/10.1137/16M1056389
[23] S. M. Ross, *Introduction to probability models*. Academic press, 2014.
[24] E. Hewitt, *Cassandra: The Definitive Guide*, 2010.
[25] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979. [Online]. Available: https://doi.org/10.1109/TC.1979.1675439
[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proc. OSDI'12, USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2012, pp. 251–264. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387905
[27] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816
[28] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: providing scalable, highly available storage for interactive services," in *Proc. CIDR'11, Conference on Innovative Data System Research*, 2011, pp. 223–234.
[29] C. Li, J. a. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis, "Automating the choice of consistency levels in replicated systems," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 281–292. [Online]. Available: http://dl.acm.org/citation.cfm?id=2643634.2643664
[30] D. J. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, Feb 2012.
[31] C. Meiklejohn, "Riak pg:distributed process groups on dynamo-style distributed storage," in *Twelfth Acm Sigplan Workshop on Erlang*, 2013.
[32] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*. USENIX Association, 2012, pp. 18–18.
[33] D. Bermbach, M. Klems, S. Tai, and M. Menzel, "Metastorage: A federated cloud storage system to manage consistency-latency tradeoffs," in *2011 IEEE 4th International Conference on Cloud Computing*. IEEE, 2011, pp. 452–459.
[34] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 309–324.
[35] M. McKenzie, H. Fan, and W. Golab, "Fine-tuning the consistency-latency trade-off in quorum-replicated distributed storage systems," in *2015 IEEE International Conference on Big Data (Big Data)*. IEEE, 2015, pp. 1708–1717.
[36] S. Sidhanta, W. Golab, S. Mukhopadhyay, and S. Basu, "Optcon: An adaptable sla-aware consistency tuning framework for quorum-based stores," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 388–397.

Appendix to "Enabling Almost Strong Consistency for Quorum-replicated Datastores"

Lingzhi Ouyang, Yu Huang, Hengfeng Wei, and Jian Lu

1 Monotonicity Properties of Shared Register Emulation Algorithms

In this section, we study the monotonicity properties of quorum-based algorithms theoretically. For simplicity, we assume that all operations in the history are applied to the same register. Some conventions in this appendix are specified as follows:

- σ: the execution history of the clients accessing the shared register.
- π: the linear extension / permutation of operations in σ.
- $\mathbb{W}$: the set of write operations.
- $\mathbb{R}$: the set of read operations.
- $\mathbb{O} = \mathbb{W} \cup \mathbb{R}$: the set of operations including writes and reads.
- $[o_{st}, o_{ft}]$: the time interval of the operation o. The time of invocation event and response event of o are denoted o_{st} and o_{ft} separately.
- $ver(o)$: the version attached to the returned value if operation $o \in \mathbb{R}$, or the version attached to the written value if $o \in \mathbb{W}$.
- $ver(s, t)$: the version stored in the server s at time t.
- $maxVer(A, t)/minVer(A, t)$: the maximum / minimum version stored in the server set A at a specific time t.
- $maxVer(A)/minVer(A)$: the maximum / minimum version collected in a query, where A is the responding server set of the query. Note that each server in A may respond at different time.

The pseudo-code for client-server interaction appears in Algorithm 1. It can be easily proved that:

Lemma 1.1 (Query-after-Update Property). *For any update u and any query q, if $u \prec_\sigma q$, and q receives responses from the server set Q, then $maxVer(Q) \geq ver(u)$.*

Proof. According to the server procedures in Algorithm 1, the version stored in servers keep non-decreasing as time goes by. Therefore, for any update u, when it is finished at time t_1, there exist a set of servers, denoted U, storing replicas with the version no lower than $ver(u)$. Namely, $minVer(U, t_1) \geq ver(u)$.

Now we consider any query q that starts after u. Namely, $u \prec_\sigma q$. Let Q denote the server set that q receives responses from.

According to the client procedures in Algorithm 1, any query or update will only be finished until a *majority* of servers reply, so $U \cap Q \neq \emptyset$. Assume the server $s \in U \cap Q$, and s returns the version at t_2 in the query q. Since q starts after u, then $t_2 > t_1$. Therefore, $maxVer(Q) \geq ver(s, t_2) \geq ver(s, t_1) \geq minVer(U, t_1) \geq ver(u)$. □

The pseudo-codes for read/write operations appear in Algorithm 2 & 3. Here, we mainly focus on the properties in terms of monotonicity of quorum-based algorithms, as shown in Table 1. These properties depict the relations between the temporal real-time order and the semantic read-from order of operations. The detailed proofs are mainly based on the *Query-after-Update Property* and displayed in Section 1.1 - 1.4 respectively.

TABLE 1
Algorithm properties in terms of monotonicity

Properties	W2R2	W2R1	W1R2	W1R1
$w \prec_\sigma r \Rightarrow ver(w) \leq ver(r)$	√	√	√	√
$w \prec_\sigma w' \Rightarrow ver(w) < ver(w')$	√	√	×	×
$r \prec_\sigma r' \Rightarrow ver(r) \leq ver(r')$	√	×	√	×
$r \prec_\sigma w \Rightarrow ver(r) < ver(w)$	√	×	×	×

1.1 Common Property: Write-Read Monotonicity

The quorum-based write / read algorithms with one or two communication round-trips guarantee the *write-read monotonicity* for multi-writer registers. Specifically,

Theorem 1.1 (Write-Read Monotonicity). *In the execution history σ, for any operations w, r using the quorum-based algorithms(see Algorithm 1, 2, 3), where $w \in \mathbb{W}, r \in \mathbb{R}$, if $w \prec_\sigma r$, then $ver(w) \leq ver(r)$.*

Proof. $\forall w \in \mathbb{W}, r \in \mathbb{R}$, let u denote the update period of w, q denote the query period of r, and Q denote the server set that q receives responses from. Then $ver(w) = ver(u), ver(r) = maxVer(Q)$.

- *Corresponding author: Yu Huang, State Key Laboratory for Novel Software Technology, and Department of Computer Science and Technology, Nanjing University, China, 210023.*
  E-mail: yuhuang@nju.edu.cn.
- *Lingzhi Ouyang, Hengfeng Wei and Jian Lu are with the State Key Laboratory for Novel Software Technology, and Department of Computer Science and Technology, Nanjing University, China, 210023.*
  E-mail: lingzhi.ouyang@outlook.com, hfwei@nju.edu.cn, lj@nju.edu.cn.

Algorithm 1: Client-server interaction

```
▷ Code for client process p_i(0 ≤ i ≤ n − 1):
function query(key)
    vals ← ∅
    pfor each server s_j                    ▷ pfor: parallel for
        send ['query', key] to s_j
        v ←[key, val, ver] from s_j
        vals ← vals ∪ v
    until a majority of them respond
    return vals
function update(key, value, version)
    vals ← ∅
    pfor each server s_j
        send ['update', key, value, version] to s_j
    wait for ['ACK']s from a majority of them
▷ Code for server process s_i(0 ≤ i ≤ N − 1):
upon receive ['query', key] from p_j
    send ['query − back', key, val, ver] to p_j
upon receive ['update', key, value, version] from p_j
    pick [k, val, ver] with k == key
    if ver < version then
        val ← value
        ver ← version
    send ['ACK'] to p_j
```

Algorithm 2: Write algorithms for client p_i

```
procedure TwoRoundWRITE(key, value)
    replicas ← query(key)
    version ←(maxSeq(replicas) +1, i)
    update(key, value, version)
procedure OneRoundWRITE(key, value)
    localSeq[key] ← localSeq[key]+1
    version ←(localSeq[key], i)
    update(key, value, version)
```

Algorithm 3: Read algorithms for client p_i

```
procedure TwoRoundREAD(key)
    replicas ← query(key)
    version ←maxVer(replicas)
    value ←valWithMaxVer(replicas, version)
    localSeq[key] ← version.seq          ▷ Optional.
    update(key, value, version)
    return value
procedure OneRoundREAD(key)
    replicas ← query(key)
    version ←maxVer(replicas)
    value ←valWithMaxVer(replicas, version)
    localSeq[key] ← version.seq          ▷ Optional.
    return value
```

If $w \prec_\sigma r$, then, it's trivial to have $u \prec_\sigma q$. By Lemma 1.1, $maxVer(Q) \geq ver(u)$. Therefore, $ver(w) = ver(u) \leq maxVer(Q) = ver(r)$. □

1.2 Two-Round-Trip Write Property: Write-Write Monotonicity

The two-round-trip write algorithm guarantees the *write-write monotonicity* for multi-writer registers. Specifically,

Theorem 1.2 (Write-Write Monotonicity). *In the execution history σ, for any write operations w_1, w_2 using the quorum-based algorithm that completes each write operation in 2 communication round-trips (see TwoRoundWRITE in Algorithm 2), if $w_1 \prec_\sigma w_2$, then $ver(w_1) < ver(w_2)$.*

Proof. For two-round-trip write operations w_1, w_2, let u_1 denote the update period of w_1, q_2 denote the query period of w_2, and Q_2 denote the server set that q_2 receives responses from. Then $ver(w_1) = ver(u_1), ver(w_2) > maxVer(Q_2)$ (Note: w_2 will construct a larger version according to the largest version in Q_2).

If $w_1 \prec_\sigma w_2$, then, it's trivial to have $u_1 \prec_\sigma q_2$. By Lemma 1.1, $maxVer(Q_2) \geq ver(u_1)$. Therefore, $ver(w_1) = ver(u_1) \leq maxVer(Q) < ver(w_2)$. □

In comparison, the one-round-trip write algorithm (see *OneRoundWRITE* in Algorithm 2) can only guarantee the *write-write monotonicity* for *single-writer* registers, where the only writer can execute updates with *monotonically increasing local versions*. However, for *multi-writer* registers, one-round-trip write may result in *write inversion anomalies after a write*, which means that a *previous* write assigns a *larger* version for its written value than a *later* write (Namely, $(w_1 \prec_\sigma w_2) \cap (ver(w_1) > ver(w_2))$). What's worse, the procedure without the promise of assigning a version larger than previous writes may lead to unsuccessful updates on servers. More details will be discussed in Section 3 in this appendix.

1.3 Two-Round-Trip Read Property: Read-Read Monotonicity

The two-round-trip read algorithm guarantees the *read-read monotonicity* for multi-writer registers. Specifically,

Theorem 1.3 (Read-Read Monotonicity). *In the execution history σ, for any read operations r_1, r_2 using the quorum-based algorithm that completes each read operation in 2 communication round-trips (see TwoRoundREAD in Algorithm 3), if $r_1 \prec_\sigma r_2$, then $ver(r_1) \leq ver(r_2)$.*

Proof. For two-round-trip read operations r_1, r_2, let u_1 denote the update period of r_1, q_2 denote the query period of r_2, and Q_2 denote the server set that q_2 receives responses from. Then $ver(r_1) = ver(u_1), ver(r_2) = maxVer(Q_2)$.

If $r_1 \prec_\sigma r_2$, then, it's trivial to have $u_1 \prec_\sigma q_2$. By Lemma 1.1, $maxVer(Q_2) \geq ver(u_1)$. Therefore, $ver(r_1) = ver(u_1) \leq maxVer(Q) = ver(r_2)$. □

In comparison, the one-round-trip read algorithm (see *OneRoundREAD* in Algorithm 3) can guarantee the *read-read monotonicity* for *single-reader* registers only if clients store previous read records locally. However, for *multi-reader*

registers, the one-round-trip read algorithm may lead to *read / write inversion anomalies after a read*, which means that the version of a later operation (a read or a write) is smaller than the version of a *previous* read. The anomalies involve several patterns and more details will be discussed in Section 2 in this appendix.

1.4 Exclusive Property of W2R2: Read-Write Monotonicity

Besides all above properties, the W2R2 algorithm also guarantees the *read-write monotonicity* for MWMR registers.

Theorem 1.4 (Read-Write Monotonicity). *In the execution history σ, for any operations w, r using the W2R2 algorithm, where $w \in \mathbb{W}, r \in \mathbb{R}$, if $r \prec_\sigma w$, then $ver(r) < ver(w)$.*

Proof. $\forall w \in \mathbb{W}, r \in \mathbb{R}$, let u_r denote the update period of r, q_w denote the query period of w, and Q_w denote the server set that q receives responses from. Then $ver(r) = ver(u_r), ver(w) > maxVer(Q_w)$ (Note: w will construct a larger version according to the largest version in Q_w).

If $w \prec_\sigma r$, then, it's trivial to have $u_w \prec_\sigma q_w$. By Lemma 1.1, $maxVer(Q_w) \geq ver(u_r)$. Therefore, $ver(r) = ver(u_r) \leq maxVer(Q_w) < ver(w)$. □

2 Atomicity Violation of W2R1: Staleness and Probability

In this section, we first prove the possible atomicity violation patterns in W2R1 (see Algorithm 4), then we prove the bound of data staleness and calculate the probability of atomicity violation in W2R1 based on the violation patterns.

Algorithm 4: W2R1

```
1  Code for client process p_i(0 ≤ i ≤ n − 1):
2  procedure TwoRoundWRITE(key, value)
3      replicas ← query(key)
4      version ←(maxSeq(replicas)+1,i)
5      update(key, value, version)
6  procedure OneRoundREAD(key)
7      replicas ← query(key)
8      version ←maxVer(replicas)
9      value ←valWithMaxVer(replicas, version)
10     return value
```

2.1 Proof of Atomicity Violation Patterns

We aim to prove that atomicity violation incurred in W2R1 are composed of RI or WI. When clients read and write the shared register using the W2R1 algorithm and obtain the history σ, we have that:

Theorem 2.1. *If σ violates atomicity, then there exist some operations in σ that form either RI or WI.*

Proof. If σ violates atomicity, then for any permutation π of σ, we have a stale read r, the dictating write w of r and the interfering write w' satisfying: $w \prec_\pi w' \prec_\pi r$. Then, we exhaustively check all cases that make r a stale read.

According to the definition of atomicity, two types of relations between operations are of our concern: the temporal real-time relation and the semantic read-from relation. We first enumerate all possible cases according to the semantic relation. Then for each case, we further enumerate all possible sub-cases according to the temporal relation.

Since w is the dictating write of r, we have that $ver(r) = ver(w)$. According to the semantic relation between versions of w and w', we have two complementing cases: $ver(r) = ver(w) < ver(w')$ and $ver(r) = ver(w) > ver(w')$. In each case, we then consider the temporal relation between operations.

Case 1: $ver(r) = ver(w) < ver(w')$.

Note that r *must be concurrent with* w'. Namely, $r \parallel_\sigma w'$. This can be proved by contradiction. If $r \prec_\sigma w'$, w' can never be the interfering write of r. If $w' \prec_\sigma r$, according to the *write-read monotonicity* (Theorem 1.1), r must return the version of w' (or return an even larger version), contradicting the fact that $ver(r) = ver(w) < ver(w')$.

Next we exhaustively check all possible sub-cases according to the temporal relation to prove that: *there must exist a read operation r' that reads from w', and r' precedes r.* Namely, $\exists r' = R(w') : r' \prec_\sigma r$.

We first enumerate all possible cases according to the temporal relation between w and w'. Since in the permutation π, $w \prec_\pi w'$, we have that in σ, $w \prec_\sigma w'$ or $w || _\sigma w'$.

Case 1.1: $w \prec_\sigma w'$. The permutation between w and w' must be *determined* as $w \prec_\pi w'$. Then, we enumerate all cases as follows:

- × $\nexists r' = R(w')$. Note that $w' \parallel_\sigma r$, so r can be ordered before w'. Thus, the permutation among w, w' and r must be $w \prec_\pi r \prec_\pi w'$, which contradicts the permutation $w \prec_\pi w' \prec_\pi r$.
- × $\forall r' = R(w') : r' \not\prec_\sigma r$. Similarly, the permutation among w, w' and r must be $w \prec_\pi r \prec_\pi w'$, contradicting the permutation $w \prec_\pi w' \prec_\pi r$.
- ✓ $\exists r' = R(w') : r' \prec_\sigma r$. Then the permutation among w, w' and r must be $w \prec_\pi w' (\prec_\pi r') \prec_\pi r$ (see Fig. 1 (a)).

Case 1.2: $w \parallel_\sigma w'$. In this case, the permutation between w and w' can not be *determined* only by the temporal relation of these two write operations.

Therefore, we then focus on other operations related to these two clusters [1]. We first consider the temporal relation between w' and any read operation r'' that reads from w. Notice that r is a special instance of r'', thus r'' must exist.

Similarly, by contradiction we have $w' \not\prec_\sigma r''$; otherwise r'' must return the version of w' (or return an even larger version) according to the *write-read monotonicity* (Theorem 1.1). Thus, the temporal relation between w' and r'' can only be $r'' \prec_\sigma w'$ or $r'' \parallel_\sigma w'$.

1. Here, we use the concept *cluster*, a terminology introduced by Gibbons and Korach [1], for our analysis. A *cluster* is a subset of operations in an execution history that consists of a write and all of its dictated reads. In [1] , Gibbons and Korach proved that atomicity violation indicates certain patterns related to different zones of clusters, where all the dictated reads of the related writes may impact the permutation. Therefore, for the soundness of proof, atomicity violation and the permutation concerned with w, w', r should be analyzed without missing other operations in the related clusters, i.e. all dictated reads of w besides r, as well as all dictated reads of w'.

Case 1.2.1: $\exists r'' = R(w) : r'' \prec_\sigma w'$. By the temporal relation between r'' and w' , as well as read-from order between r'' and w, the permutation between w and w' must be *determined* as $w(\prec_\pi r'') \prec_\pi w'$.

- × $\nexists r' = R(w')$. Note that $w' \parallel_\sigma r$, so r can be ordered before w'. Thus, the permutation among w, w' and r must be $w \prec_\pi r \prec_\pi w'$, contradicting the permutation $w \prec_\pi w' \prec_\pi r$.
- × $\forall r' = R(w') : r' \not\prec_\sigma r$. Similarly, the permutation among w, w' and r must be $w \prec_\pi r \prec_\pi w'$, contradicting the permutation $w \prec_\pi w' \prec_\pi r$.
- ✓ $\exists r' = R(w') : r' \prec_\sigma r$. Then the permutation must be $w \prec_\pi w'(\prec_\pi r') \prec_\pi r$ (see Fig. 1 (b)).

Case 1.2.2: $\forall r'' = R(w) : r'' \parallel_\sigma w'$. Then, We consider the temporal relation between w and any read operation r' that reads from w'. Note that r' may not exist.

Case 1.2.2.1: $\nexists r' = R(w')$. Then, there exists no determining factor to deciding the permutation between w and w'. Thus, the permutation among w, w', r is $w' \prec_\pi w \prec_\pi r$ or $w \prec_\pi r \prec_\pi w'$, both contradict the assumption that w, w' and r form an inconsistent read. [×]

Case 1.2.2.2: $\exists r' = R(w') : r' \prec_\sigma w$. By the temporal relation between r' and w , as well as read-from order between r' and w', the permutation between w and w' is *determined* as $w'(\prec_\pi r') \prec_\pi w$, which contradicts $w \prec_\pi w'$. [×]

Case 1.2.2.3: $\forall r' = R(w') : r' \not\prec_\sigma w$. Then, there exists no determining factor to deciding the permutation between w and w'.

- × $\forall r' = R(w') : r' \not\prec_\sigma r$. Thus, the permutation among w, w' and r is $w' \prec_\pi w \prec_\pi r$ or $w \prec_\pi r \prec_\pi w'$, both violating the given permutation $w \prec_\pi w' \prec_\pi r$.
- ✓ $\exists r' = R(w') : r' \prec_\sigma r$. Then the permutation among w, w', r and r' is either $w \prec_\pi w' \prec_\pi r' \prec_\pi r$ or $w' \prec_\pi w \prec_\pi r' \prec_\pi r$ (see Fig. 1 (c)), which involves atomicity violation inevitably. If the former permutation is selected, then r is a stale read [2].

Above all, for $ver(w) < ver(w')$, then w, w' and r can be linearly extended to $w \prec_\pi w' \prec_\pi r$ only when $\exists r' = R(w') : r' \prec_\sigma r$. Obviously, r and r' form RI.

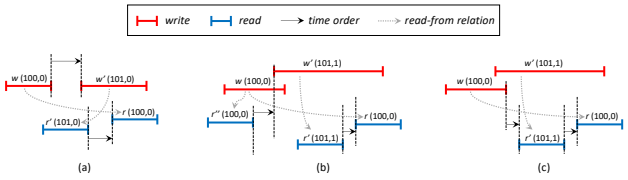

Fig. 1. Typical patterns of RI in Case 1.

Case 2: $ver(r) = ver(w) > ver(w')$.

Note that w must be concurrent with w'. This can be proved by contradiction. If $w \prec_\sigma w'$, according to *write-write monotonicity* that the two-round-trip write algorithm holds (Theorem 1.2), the version of w' must be larger. If $w' \prec_\sigma w$, we can never get $w \prec_\pi w' \prec_\pi r$ when linearly extending σ to π. Besides, according to the permutation $w' \prec_\pi r$, we have $w' \prec_\sigma r$ or $w' \parallel_\sigma r$.

Next we enumerate all possible sub-cases according to the temporal relation to prove that: there exists a read operation r'' (can be r) that reads from w, satisfying that r'' precedes either w' or another read operation r' that reads from w'. Namely, $\exists r'' = R(w) : (r'' \prec_\sigma w') \vee (\exists r' = R(w') : r'' \prec_\sigma r')$.

We first consider the temporal relation between w' and any read operation r'' that reads from w. Notice that r is a special instance of r'', thus r'' must exist.

Case 2.1: $\exists r'' = R(w) : r'' \prec_\sigma w'$. Then, the permutation between w and w' must be *determined* as $w(\prec_\pi r'') \prec_\pi w'$. Note that r cannot take the role of r'' here, because $r \not\prec_\sigma w'$.

Case 2.1.1: $w' \prec_\sigma r$. Then the permutation among w, w' and r must be $w \prec_\pi w' \prec_\pi r$ (see Fig. 2 (a)). [✓]

Case 2.1.2: $w' \parallel_\sigma r$.

- × $\nexists r' = R(w')$. Note that $w' \parallel_\sigma r$, so r can be ordered before w'. Thus, the permutation among w, w' and r is $w \prec_\pi r \prec_\pi w'$, contradicting the assumption.
- × $\forall r' = R(w') : r' \not\prec_\sigma r$. Similarly, the permutation among w, w' and r is $w \prec_\pi r \prec_\pi w'$, contradicting the assumption.
- ✓ $\exists r' = R(w') : r' \prec_\sigma r$. Then the permutation among w, w' and r must be $w(\prec_\pi r'') \prec_\pi w'(\prec_\pi r') \prec_\pi r$ (see Fig. 2 (b)).

Case 2.2: $\forall r'' = R(w) : r'' \not\prec_\sigma w'$. Then, We consider the temporal relation between w and any read operation r' that reads from w'. Note that r' may not exist.

Case 2.2.1: $\nexists r' = R(w')$. Then, there exists no determining factor to deciding the permutation between w and w'.

- × $w' \prec_\sigma r$. Then the permutation among w, w' and r must be $w' \prec_\pi w \prec_\pi r$, contradicting the assumption.
- × $w' \parallel_\sigma r$. Thus, the permutation among w, w', r is $w' \prec_\pi w \prec_\pi r$ or $w \prec_\pi r \prec_\pi w'$, both contradicting the assumption.

Case 2.2.2: $\exists r' = R(w') : r' \prec_\sigma w$. Then the permutation between w and w' is *determined* as $w'(\prec_\pi r') \prec_\pi w$,which contradicts $w \prec_\pi w'$. [×]

Case 2.2.3: $\forall r' = R(w') : r' \not\prec_\sigma w$. Then, We consider the temporal relation between any read operation r'' that reads from w and any read operation r' that reads from w'.

Case 2.2.3.1: $\forall r' = R(w'), r'' = R(w) : r'' \not\prec_\sigma r'$.

- × $w' \prec_\sigma r$. Then the permutation among w, w' and r must be $w' \prec_\pi w \prec_\pi r$, contradicting the assumption.
- × $w' \parallel_\sigma r$. Thus, the permutation among w, w', r is either $w' \prec_\pi w \prec_\pi r$ or $w \prec_\pi r \prec_\pi w'$, both contradicting the assumption.

Case 2.2.3.2: $\exists r' = R(w'), r'' = R(w) : r'' \prec_\sigma r'$.

By contradiction we have $w \not\prec_\sigma r'$; otherwise r' must return the version of w (or return an even larger version) according to the *write-read monotonicity* (Theorem 1.1). Besides, r'' can't precede its dictating write w due to the regularity property. Thus, the temporal relation can only be $w \parallel_\sigma r''$ as well as $w \parallel_\sigma r'$. The permutation among w, r'', r' must be $w \prec_\pi r'' \prec_\pi r'$. Then the permutation can be $w \prec_\pi w' \prec_\pi r'' \prec_\pi r'$ or $w' \prec_\pi w \prec_\pi r'' \prec_\pi r'$, which involves atomicity violation inevitably. If $w' \prec_\sigma r$ (or r just

2. In this subcase (as well as **Case 2.2.3.2**), only one of the possible permutations will make r a stale read. However, we aim to prove the necessary conditions of atomicity violation in the proof, thus this possibility needs to be counted.

takes the role of r''), then r may become a stale read [3] (see Fig. 2 (c)). [✓]

Above all, for $ver(w) > ver(w')$, then w, w' and r can be linearly extended to $w \prec_\pi w' \prec_\pi r$ only when there exists a read operation r'' (can be r) that reads from w, satisfying that r'' precedes either w' or another read operation r' that reads from w'. Namely, $\exists r'' = R(w) : (r'' \prec_\sigma w') \vee (\exists r' = R(w') : r'' \prec_\sigma r')$, where r'' and w' form WI, or r'' and r' form RI.

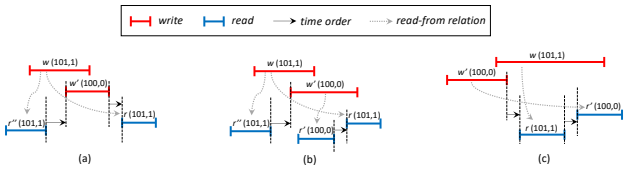

Fig. 2. Typical patterns of WI (Sub-fig (a) and (b))and RI (Sub-fig (c)) in Case 2.

In conclusion, by assuming r is a stale read, we exhaustively check all cases according to the temporal real-time relations and the semantic read-from relations of relative operations, At last, we have proved that the necessary conditions to make r a stale read involve RI or WI. Therefore, Theorem 2.1 is true. □

From the proof of Theorem 2.1 we can obtain some more detailed information, which is useful for our further analysis of staleness bound.

Proposition 2.1. *Suppose σ is the history obtained by clients using the W2R1 algorithm. If σ violates atomicity, then for any permutation π of σ, there exists a stale read r, the dictating write w of r and the interfering write w' that form $w \prec_\pi w' \prec_\pi r$, satisfying **Case 1** or **Case 2**, where*

- ***Case 1:*** *$ver(w) < ver(w')$. w' and r are concurrent, and there exists a read r' that reads from w', as well as precedes r. Namely, $(w' \parallel_\sigma r) \wedge (\exists r' = R(w') : r' \prec_\sigma r)$;*
- ***Case 2:*** *$ver(w) > ver(w')$. w and w' are concurrent, and there exists a read r'' (can be r) that reads from w, satisfying that r'' precedes either w' or another read r' that reads from w'. Namely, $(w \parallel_\sigma w') \wedge (\exists r'' = R(w) : (r'' \prec_\sigma w') \vee (\exists r' = R(w') : r'' \prec_\sigma r'))$.*

2.2 Proof of Bound of Data Staleness

In this section, we calculate the tight bound of data staleness when accessing the shared register using the W2R1 algorithm. Suppose the distributed storage system consists of N replicas($N \geq 3$). Denote the number of writer clients as n_w. We have that:

Theorem 2.2. *For any history σ, there exists a linear extension π of σ such that any read in π returns the value written by one of the latest B preceding writes. Here, $B = n_w + \frac{1}{2}n_w(n_w - 1) + 1$. Moreover, the bound B is tight, i.e., there exists a history σ and a linear extension π of σ in which some read returns the value written by the oldest write in the latest B preceding writes.*

3. **Case 2.2.3.2** and **Case 1.2.2.3** are nearly the same except the version relation between the clusters of w and w' is exchanged. However, in the proof we are not supposed to miss any possible subcase.

Proof. The proof of Theorem 2.2 is based on an adversary argument, to insert as many interfering writes as possible before the inconsistent read. The proof also needs to consider the same two cases as defined in Section 2.1. Specifically, suppose r is an inconsistent read, and the dictating write of r is w. There must exist another interfering write w' such that $w \prec_\pi w' \prec_\pi r$.

According to the versions $ver(r)$ and $ver(w')$, we consider two cases. In Case 1 where $ver(r) = ver(w) < ver(w')$, we prove that there are at most $B_1 = n_w$ interfering writes which can be inserted between r and w. In Case 2 where $ver(r) = ver(w) > ver(w')$, we prove that there are at most $B_2 = \frac{1}{2}n_w(n_w - 1)$ interfering writes.

Case 1: $ver(w) = ver(r) < ver(w')$.

From Case 1 of Proposition 2.1 we know that the interfering write w' must be concurrent with r, and there exists a read operation r' preceding r and dictated by w'. Note that the invocation of w' must be *earlier* than that of r, or r' is not able to read from w' when preceding r . Since each writer client can have at most one write operation that starts before r and is concurrent with r, the number of interfering writes in this case is no more than n_w. Thus we have the bound $B_1 = n_w$ in this case, and the bound B_1 is tight. One construction of the most stale read in Case 1 is displayed in Fig. 3, where the pattern of operations is shown in Fig. 1 (a).

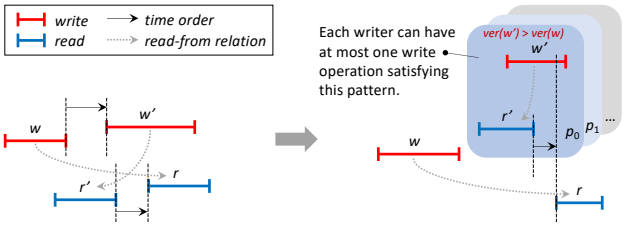

Fig. 3. Maximum number of interfering writes in Case 1.

Case 2: $ver(w) = ver(r) > ver(w')$.

From Case 2 of Proposition 2.1 we know that the interfering write w' must be concurrent with w. Other than the writer client of operation w, we can have at most $n_w - 1$ writer clients to generate interfering writes. The challenge in the construction is that, for each interfering writer client, we cannot insert as many interfering write operations w' as we want, due to the constraint of $ver(w') < ver(w)$ and $w' \parallel_\sigma w$.

Denote the write operation versioned (seq, i) as $w(seq, i)$. In this case, both WI and RI are able to coexist to result in stale reads, we need to consider both patterns in our adversary argument. We first construct a simplified worst-case trace using WI alone, then we take both patterns into consideration.

The construction of WI without RI

In WI, there exists a read r'' that reads from $w(seq, i)$ and precedes w'. We focus on the maximum number of interfering write operations w' that can be inserted after r'' while still concurrent with $w(seq, i)$. Let w have the version (seq, i). The key point of our adversary argument is to keep the *maximum* version of some replica majority as *small* as possible before $w(seq, i)$ is done, so the interfering writes can have the chance to query that replica majority and write with the version value smaller than (seq, i).

Now let's describe the construction with adversary argument in detail. First of all, at the finish time of r'', if the maximum version of any replica majority become not smaller than (seq, i), then no interfering two-round-trip write operation after r'' can have the version value smaller than (seq, i). Therefore, there still exists some replica majority whose maximum version value is smaller than (seq, i) after the finish time of r'', and the interfering two-round-trip write operations may still have the chance to have the version value smaller than (seq, i).

Be aware that w won't have $ver(w) = (seq, i)$ unless it queries a majority of replicas whose maximum version is $(seq-1, i_1)$ in its first round-trip. After the finish time of r'', there exist two possibilities. If the maximum version of *any* replica majority becomes not smaller than $(seq-1, i_1)$, then any interfering two-round-trip write operation can only have the version not smaller than (seq, i'), where $i' < i$. If there still exists some replica majority whose maximum version is smaller than $(seq-1, i_1)$ (the write operation $w(seq-1, i_1)$ is not yet finished at this time for sure), take $(seq-2, i_2)$ for instance, then the interfering two-round-trip write operations may still have the chance to have the version smaller than $(seq-1, i_1')$. Since $(seq-1, i_1') < (seq, i')$, for adversary argument, we choose the latter possibility, i.e., let $w(seq-1, i_1)$ be unfinished at the finish time of r'', so more interfering write operations can be inserted after r''.

Recursively, be aware that a write operation won't have the version value $(seq-1, i_1)$ unless it queries a majority of replicas whose maximum version is $(seq-2, i_2)$ in its first round-trip. For adversary argument, let $w(seq-2, i_2)$ be unfinished at the finish time of r'', so more interfering write operations can be inserted after r''. Similarly, let $w(seq-3, i_3)$, $w(seq-4, i_4)$, ..., $w(seq-n+1, i_{n-1})$ be unfinished at the finish time of r'' until the operations of all writer clients (except the writer client of the operation $w(seq, i)$) are in.

Above all, we adversely make a construction that allows the most interfering write operations in the pattern of WI. Note that all of $w(seq-1, i_1)$, $w(seq-2, i_2)$, ..., $w(seq-n+1, i_{n-1})$ can be ordered before r'' in π. Then, after $w(seq-n+1, i_{n-1})$ is finished while $w(seq-1, i_1)$, $w(seq-2, i_2)$, ..., $w(seq-n+2, i_{n-2})$ are processing, the writer client i_{n-1} may query a majority of replicas whose maximum version value is $w(seq-n+1, i_{n-1})$ and can issue an interfering write operation $w(seq-n+2, i_{n-1})$. Similarly, after $w(seq-n+1, i_{n-1})$ and $w(seq-n+2, i_{n-2})$ are finished while $w(seq-1, i_1)$, $w(seq-2, i_2)$, ..., $w(seq-n+3, i_{n-3})$ are processing, the writer clients i_{n-1} and i_{n-2} may query a majority of replicas whose maximum version value is $w(seq-n+2, i_{n-2})$, and then issue interfering write operations $w(seq-n+3, i_{n-1})$ and $w(seq-n+3, i_{n-2})$ respectively. In this way, after all of $w(seq-1, i_1)$, $w(seq-2, i_2)$, ..., $w(seq-n+1, i_{n-1})$ are done, all writer clients except i may query a majority of replicas whose maximum version value is $w(seq-1, i_{n-1})$, and then issue interfering write operations $w(seq, i'), i' < i$. Let i be the maximum client identifier. Then, the maximum number of interfering write operations using WI is:

$$B_2^{WI} = 1 + 2 + \cdots + (n_w - 2) + (n_w - 1) = \frac{1}{2} n_w (n_w - 1)$$

Fig. 4 shows a concrete example of the most stale read with WI when $n_w = 4$, where the pattern of operations is shown in Fig. 2 (a). The read operation r returns the value written by $w(100, 3)$ issued by the writer client with the maximum identifier.

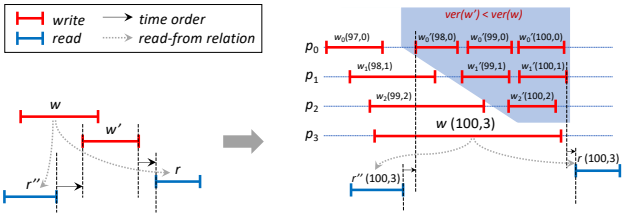

Fig. 4. Maximum number of interfering writes in the pattern of WI in Case 2.

The construction of WI and RI

Now we discuss the situation with the additional effects of RI. For RI, according to Case 2 of Proposition 2.1, there exists a read r'' (can be r) that's dictated by w preceding another read r' dictated by w'. For adversary argument, let r be the read operation with the latest starting time dictated by w, so it must be the most stale read among those dictated by w.

Consider any write operation w' that's concurrent with w and versioned smaller than $ver(w)$. As for the dictated read r'' of w, we mainly focus on the one that *ends earliest*, because the position of w in π is largely decided by it compared to other dictated reads of w. Note that if w' is an interfering write of r, then either w' forms WI with r'', or a dictated read of w' forms RI with r''. Without losing generality, w' can be preceding, concurrent with or starting after r''.

Now we describe an approach to linearly extending σ to a permutation π where the number of interfering writes in Case 2 is no more than B_2^{WI}. For any w' that's preceding or concurrent with r'', let w' precede w and r'' in π, preventing it from interfering the dictated reads of w. For any w' that starts after r'', w' must be ordered after w and r''. Thus, each of these w' will interfere the dictated read(s) of w that start after it. Repeat above procedures recursively for all write operations. In this way, all w' that start not after r'' won't interfere the dictated reads of w, and only those w' that starts after r'' can have the chance to interfere the dictated reads of w.

We now analyze the maximum staleness of each read dictated by w' or w in above permutation. For any read r' dictated by w', if it starts after some read operation r''' dictated by w , then r' and r''' form RI. Since $ver(r') = ver(w') < ver(w)$, according to the worst-case construction using RI in Case 1, the interfering write operations of r' is no more than $B_1 = n_w$. For any read r''' dictated by w, if it starts after some write w', then w' and r''' form the pattern of WI. According to the worst-case construction using WI in Case 2, the number of interfering write operations versioned smaller than $ver(w)$ and starting after r'' is no more than $B_2^{WI} = \frac{1}{2} n_w (n_w - 1)$.

Note that this approach only promises to obtain a legal permutation π of any given σ. Whether or not there exists some other legal permutations with smaller maximum staleness of all reads, the interfering writes of all dictated reads

of w in Case 2 (versioned smaller than $ver(w)$) are definitely no more than B_2^{WI}.

Since we have constructed the worst-case trace with B_2^{WI} interfering writes using WI before, we have the bound $B_2 = B_2^{WI} = \frac{1}{2}n_w(n_w - 1)$, and the bound B_2 is tight.

Given a specific read operation r, the interfering write operations in Case 1 and Case 2 can appear at the same time in the history σ, so we can construct a trace with $B_1 + B_2$ interfering writes. Counting the dictating write itself, we have that the read always returns the value written by one of the latest

$$B = B_1 + B_2 + 1 = n_w + \frac{1}{2}n_w(n_w - 1) + 1$$

preceding writes, i.e. the history σ always satisfies B-atomicity. During the proof, we explicitly construct the worst-case trace, which contains the read having $B_1 + B_2$ interfering writes. This proves that the bound is tight [4]. □

An example of the worst-case construction is illustrated in Fig. 5. The process of version updates on replicas is shown in Table 2.

Till now, we prove the correctness of Theorem 2.2. The bound of data staleness using the W2R1 algorithm is related to the number of writer clients. Specially, for *single-writer* multiple-reader registers, W2R1 satisfies 2-atomicity, which conforms to the conclusion in [2].

2.3 Probability of Atomicity Violation

In this subsection we quantify the atomicity violation incurred in W2R1. The quantification follows from Theorem 2.1 that, atomicity violation incurred in W2R1 are composed of either RI or WI. We simplify the quantification problem into the following one: what's the probability that RI or WI incurred in W2R1? The simplification helps us to quantify atomicity violation by ignoring unnecessary details. Note that the occurrence of either RI or WI is merely the condition *necessary* but *not sufficient* for atomicity violation in W2R1, but this will not prevent us from obtaining the upper bound of the violation probability:

$$\begin{aligned} & \mathbb{P}\{\text{Violation}\} \leq \mathbb{P}\{\text{RI} \vee \text{WI}\} \\ = \ & \mathbb{P}\{\text{RI}\} + \mathbb{P}\{\text{WI}\} - \mathbb{P}\{\text{RI} \wedge \text{WI}\} \\ \leq \ & \mathbb{P}\{\text{RI}\} + \mathbb{P}\{\text{WI}\} \end{aligned} \tag{2.1}$$

According to the case-by-case proof of Theorem 2.1, RI and WI can only occur within certain conditions in W2R1. Intuitively, an execution with higher degree of concurrency would produce more atomicity violation, but it's still not sufficient without certain read-write patterns. Thus, we decompose RI and WI incurred in W2R1 into a *concurrency pattern* and a *read-write pattern* separately. The *concurrency pattern(CP)* describes the constraints of real-time orders among events of operations, and the *read-write pattern(RWP)* limits read/write semantics among operations. From the pattern analysis in Theorem 2.1, we have

4. Actually, the proof of some simple corner cases (e.g. $n_w < 3$) is a little different in details and not elaborated here, but the bound is not affected.

Definition 2.1 (The RI Pattern in W2R1). *The RI in W2R1 involves one write w' and two reads r, r', satisfying the requirements of*

- *the long-lived-write concurrency pattern(CP):*
  1) $r_{st} \in [w'_{st}, w'_{ft}]$,
  2) $r'_{ft} \in [w'_{st}, r_{st}]$;
- *the non-monotonic read-write pattern(RWP):*
  3) $r' = R(w')$;
  4) $r \neq R(w')$.

Definition 2.2 (The WI Pattern in W2R1). *The WI in W2R1 involves two writes w, w' and one read r'', satisfying the requirements of*

- *the long-lived-write concurrency pattern(CP):*
  1) $w'_{st} \in [w_{st}, w_{ft}]$,
  2) $r''_{ft} \in [w_{st}, w'_{st}]$;
- *the non-monotonic read-write pattern(RWP):*
  3) $r'' = R(w)$;
  4) $ver(w') \neq ver(w)$.

To calculate the probabilities of RI and WI, we view RI and WI as the concurrent occurrences of multiple atomicity violation patterns in the single-writer case. According to our previous work [2], atomicity violation in the single-writer case is equivalent to the pattern named Old New Inversion (ONI). We further destruct the ONI into the temporal Concurrency Pattern (CP) and the semantic Read Write Pattern (RWP). Then we obtain the probability of ONI:

$$\begin{aligned} \mathbb{P}\{\text{ONI}\} &= \sum_{m\geq 1} \mathbb{P}\{\text{CP} \mid R' = m\} \cdot \mathbb{P}\{\text{RWP} \mid R' = m\} \\ &\approx \sum_{m=1}^{n_r} \left(\left(\sum_{k=0}^{n_r-1} \binom{n_r}{k} \binom{m-1}{n_r-k-1} p_0^k r^{n_r-k} s^m \right) \right. \\ &\quad \cdot e^{-q\lambda_w t} \frac{\alpha^q B(q, \alpha(N-q)+1)}{B(q, N-q+1)} \\ &\quad \left. \cdot \left(1 - \left(\frac{J_1}{B(q, N-q+1)} \right)^m \right) \right). \end{aligned} \tag{2.2}$$

where R' is a random variable denoting the number of r's in Definition 2.1. It is an approximation since the timed balls-into-bins model used for calculating the probability of read-write patterns assumes that there is at most one such r' in each reader queue, which can be justified by numerical results. For further details, please refer to our previous work [2] for detailed explanations of Equation 2.2.

Through observation we can see that ONI is a special case of RI when there exists one single writer. Based on the quantification of ONI, we can quantify RI for MWMR registers with extra consideration of the concurrency effects of multiple writers. As for WI, by replacing r, w', r' in Definition 2.1 with w', w, r'' accordingly, it can be analyzed with similar process. Above all, we simplify MWMR quantification by reusing results of ONI and taking the extra effects of multiple writers into consideration together.

We first model the occurrences of multiple writers with the following queuing model. Each client's workload is recognized as an independent queue characterized by the rate of operations and the service time of each operation (i.e.

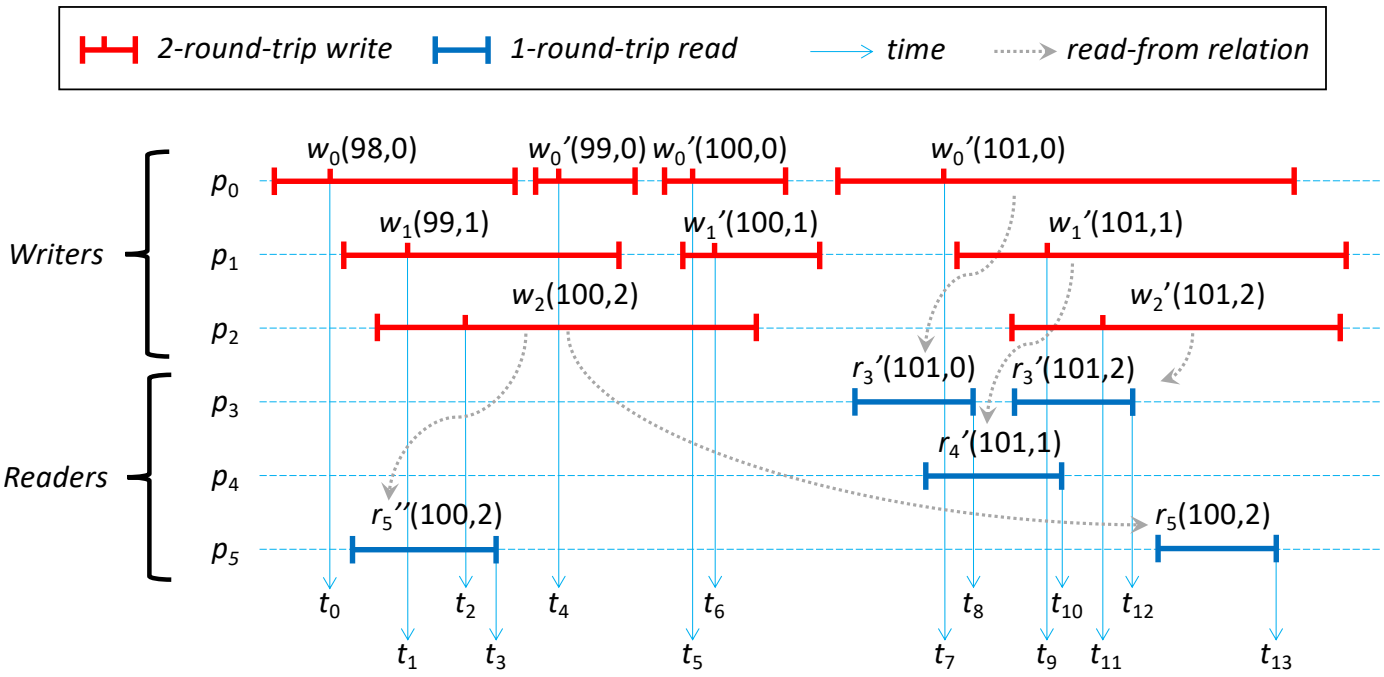

Fig. 5. An example of worst case with 3 writer clients and 3 reader clients using the W2R1 algorithm. The *version* that a *write* assigns or a *read* returns are displayed in brackets. One permutation of these operations can be: $w_0(98,0) \prec_\pi w_1(99,1) \prec_\pi \mathbf{w_2(100,2)} \prec_\pi r_5''(100,2) \prec_\pi w_0'(99,0) \prec_\pi w_0'(100,0) \prec_\pi w_1'(100,1) \prec_\pi w_0'(101,0) \prec_\pi r_3'(101,0) \prec_\pi w_1'(101,1) \prec_\pi r_4'(101,1) \prec_\pi w_1'(101,2) \prec_\pi r_3'(101,2) \prec_\pi \mathbf{r_5(100,2)}$. For $r_5(100,2)$, it returns the value of the latest 7 preceding write in π. Interfering write operations w' of r include $w_0'(99,0), w_0'(100,0), w_1'(100,1), w_0'(101,0), w_1'(101,1), w_2'(101,2)$. Among these operations, $r_5''(100,2)$ and each of $w_0'(99,0), w_0'(100,0), w_1'(100,1)$ form the pattern of WI respectively; $r_5(100,2)$ and each of $r_3'(101,0), r_4'(101,1), r_3'(101,2)$ form the pattern of RI respectively.

TABLE 2
The process of version updates on replicas in Fig. 5. Suppose the distributed storage system contains 3 replicas.

Time	Replica 1	Replica 2	Replica 3	Description
t_0	**(97,0)**	**(97,0)**	(97,0)	p_0 then assigns $(98,0)$ after the query returns maximum version $(97,0)$ in 1^{st} round-trip.
t_1	**(98,0)**	**(97,0)**	(97,0)	p_1 then assigns $(99,1)$ after the query returns maximum version $(98,0)$ in 1^{st} round-trip.
t_2	**(99,1)**	**(97,0)**	(97,0)	p_2 then assigns $(100,2)$ after the query returns maximum version $(99,1)$ in 1^{st} round-trip.
t_3	**(100,2)**	**(97,0)**	(97,0)	p_5 returns the value versioned $(100,2)$ in the read.
t_4	(100,2)	**(98,0)**	**(97,0)**	p_0 then assigns $(99,0)$ after the query returns maximum version $(98,0)$ in 1^{st} round-trip.
t_5	(100,2)	**(99,0)**	**(97,0)**	p_0 then assigns $(100,0)$ after the query returns maximum version $(99,0)$ in 1^{st} round-trip.
t_6	(100,2)	**(99,0)**	**(97,0)**	p_1 then assigns $(100,1)$ after the query returns maximum version $(99,0)$ in 1^{st} round-trip.
t_7	(100,2)	**(100,2)**	**(97,0)**	p_0 assigns $(101,0)$ after the query returns maximum version $(100,2)$ in 1^{st} round-trip.
t_8	(100,2)	**(101,0)**	**(97,0)**	p_3 returns the value versioned $(101,0)$ in the read.
t_9	**(100,2)**	(101,0)	**(97,0)**	p_1 assigns $(101,1)$ after the query returns maximum version $(100,2)$ in 1^{st} round-trip.
t_{10}	(100,2)	**(101,1)**	**(97,0)**	p_4 returns the value versioned $(101,1)$ in the read.
t_{11}	**(100,2)**	(101,1)	**(97,0)**	p_2 assigns $(101,2)$ after the query returns maximum version $(100,2)$ in 1^{st} round-trip.
t_{12}	(100,2)	**(101,2)**	**(97,0)**	p_3 returns the value versioned $(101,2)$ in the read.
t_{13}	**(100,2)**	(101,2)	**(97,0)**	p_5 returns the value versioned $(100,2)$ in the read.

$[o_{st}, o_{ft}]$). Assume a Poisson process with parameter λ for the scenario of each client issuing a sequence of read/write operations, and an exponential distribution with parameter μ for the service time of each operation. We then have n independent, parallel $M/M/1/1/\infty/FCFS$ queues (i.e., a single-server exponential queuing system, whose capacity is 1 with the "first come first served" discipline) [3]. All the queues have arrival rate λ and service rate μ. For simplicity, queues of writes and reads are separate; and if there is any operation in service, no more operations can enter it in that queue.

Let $X^i(t)$ be the number of operations in queue i at time t. Then $X^i(t)$ is a continuous-time Markov chain with two states: 0 when the queue is empty and 1 when some operation is being served. Its stationary distribution $P_s \triangleq P(X^i(\infty) = s), s \in \{0,1\}$ is:

$$P_0 = \frac{\mu}{\mu+\lambda}, P_1 = \frac{\lambda}{\mu+\lambda}.$$

Now we consider the concurrency conditions of RI and WI respectively. For RI, given a read r in Q_i, let W'_{cr} be a random variable denoting the number of w's satisfying $r_{st} \in [w'_{st}, w'_{ft}]$ in RI. The probability of the event that r starts during the service period of some write w' in Q_j equals the probability that when r arrives Q_i, it finds Q_i empty, as well as finds Q_j full as a bystander (with the constraint that Q_j is a writer queue). Since the events in different queues are independent, by the PASTA property

and through combinatorial analysis, we have

$$\mathbb{P}\{W'_{cr} = x\} = \binom{n_w}{x}\left(\frac{\lambda}{\mu+\lambda}\right)^x\left(\frac{\mu}{\mu+\lambda}\right)^{n_w-x+1} \quad (2.3)$$

Conditioning on $W'_{cr} = w$, the probability of RI is no more than the sum of each w' forming RI with r seperately. Therefore, a probabilistic and combinatorial analysis shows that

$$\begin{aligned}\mathbb{P}\{\text{RI}\} &= \sum_{x=1}^{n_w}\mathbb{P}\{W'_{cr}=x\}\cdot\mathbb{P}\{\text{RI}\mid W'_{cr}=x\}\\ &\le \sum_{x=1}^{n_w}\mathbb{P}\{W'_{cr}=x\}\cdot x\mathbb{P}\{\text{ONI}\} \quad (2.4)\end{aligned}$$

As for WI, given a specific write w' in Definition 2.2, let $W_{cw'}$ be a random variable denoting the number of writes w satisfying $w'_{st} \in [w_{st}, w_{ft}]$ in RI. Similarly, through probabilistic and combinatorial analysis we have

$$\mathbb{P}\{W_{cw'}=x\} = \binom{n_w-1}{x}\left(\frac{\lambda}{\mu+\lambda}\right)^x\left(\frac{\mu}{\mu+\lambda}\right)^{n_w-x} \quad (2.5)$$

$$\begin{aligned}\mathbb{P}\{\text{WI}\} &= \sum_{x=1}^{n_w-1}\mathbb{P}\{W_{cw'}=x\}\cdot\mathbb{P}\{\text{WI}\mid W_{cw'}=x\}\\ &\le \sum_{x=1}^{n_w-1}\mathbb{P}\{W_{cw'}=x\}\cdot x\mathbb{P}\{\text{ONI}\} \quad (2.6)\end{aligned}$$

Substituting Formula (2.2)-(2.6) into (2.1), we obtain an upper bound of the rate of violating atomicity incurred in W2R1.

$$\begin{aligned}&\mathbb{P}\{\text{Violation}\}\\ \le&\sum_{x=1}^{n_w}\mathbb{P}\{W'_{cr}=x\}\cdot x\mathbb{P}\{\text{ONI}\}+\sum_{x=1}^{n_w-1}\mathbb{P}\{W_{cw'}=x\}\cdot x\mathbb{P}\{\text{ONI}\}\\ \approx&\frac{(2n_w-1)\lambda\mu}{(\lambda+\mu)^2}\sum_{m=1}^{n_r}\left(\left(\sum_{k=0}^{n_r-1}\binom{n_r}{k}\binom{m-1}{n_r-k-1}p_0^k r^{n_r-k}s^m\right)\right.\\ &\left.e^{-q\lambda_w t}\frac{\alpha^q B(q,\alpha(N-q)+1)}{B(q,N-q+1)}\left(1-\left(\frac{J_1}{B(q,N-q+1)}\right)^m\right)\right)\end{aligned}$$

The results in Table 3 and Fig. 6 are obtained when setting $\lambda = \mu = 10s^{-1}$ and $\lambda_r = \lambda_w = 20s^{-1}$. The source code for calculation of the numerical results can be found in our open source project on line [5]. The numerical results show that the probability of atomicity violation is quite low (mostly below 0.1% and can be below 10^{-9}) and will decrease when the number of replicas increases. Moreover, the probability of the concurrency pattern is close to 1. The probability of the read-write pattern is significantly less and decreases as the number of replicas increases. The probability of the read-write pattern ensure that the probability of the atomicity violation is almost zero. Besides, the probability of atomicity violation using the W2R1 algorithm has positive correlation with the number of concurrent writers. The comprehensive experimental results in Section 4 in the appendix further confirm our theoretical analysis.

5. https://github.com/Lingzhi-Ouyang/Almost-Strong-Consistency-Cassandra/tree/master/numerical_results

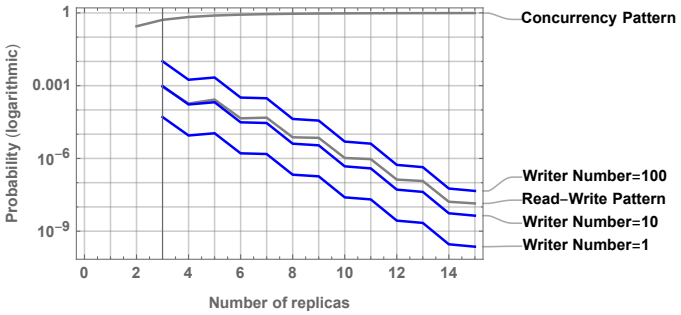

Fig. 6. The probabilities of atomicity violation.

3 W1R2 and W1R1: Theoretical Analysis and Experimental Evaluation

From the diamond schema we have one-round-trip write algorithms W1R2 & W1R1, as shown in Algorithm 5 & 6, which complete each write in only one communication round-trip. When a writer client initiates a write request, it would constructs a version for the written key-value pair based on its local sequence number record. The version assigned for its written value is only guaranteed to be larger than any version returned by its previous reads or assigned by its previous writes, but not others'. Therefore, one-round-trip write works well only for *single-writer* registers since the only writer client can promise to preserve the version for a register monotonically increasing without conflicting with other writer clients. For *multiple-writer* registers, one-round-trip write may result in *write inversion anomalies after a write*: a later write updates a value with a smaller version than a previous write from a different writer client. Here, we define a write is *invisible* if before the start time of the write, the maximum version of any server replica majority is larger than its assigned version for the update. Obviously, the data from an invisible write cannot be returned by any read.

In this section, we tend to demonstrate that one-round-trip write algorithms are useless from two aspects. First we prove that neither W1R2 nor W1R1 can guarantee bounded staleness of data for reads in Section 3.1. Then we demonstrate that both algorithms lead to frequent unsuccessful updates by quantifying the rate of invisible writes in Section 3.2.

3.1 Proof of Atomicity Violation in One-Round-Trip Write

We prove that, in the W1R2 or W1R1 algorithm, the value returned by each read can be any stale. That is,

Theorem 3.1. *For MWMR registers, given $k \in \mathbb{N}^*$, the W1R2 or W1R1 algorithm doesn't satisfy k-atomicity.*

Proof. Assume W1R2 or W1R1 satisfies k-atomicity, where $k \in \mathbb{N}^*$. Then, we can raise a counter-example to show the absurdity of above assumption. The counter-example

TABLE 3
Numerical results on the probabilities of concurrency patterns, read-write patterns, and old-new inversions.

# Replicas	$\mathbb{P}\{\text{CP}\}$	$\mathbb{P}\{\text{RWP}\}$	$\mathbb{P}\{n_w = 1\}$	$\mathbb{P}\{n_w = 10\}$	$\mathbb{P}\{n_w = 100\}$
2	0.28125	0.	0.	0.	0.
3	0.518555	0.00088802	0.0000509207	0.000967493	0.0101332
4	0.677307	0.000183791	8.82396×10^{-6}	0.000167655	0.00175597
5	0.781222	0.000266569	1.09295×10^{-5}	0.000207661	0.00217498
6	0.849318	0.0000450835	1.62306×10^{-6}	0.0000308382	0.00032299
7	0.89429	0.0000478926	1.5218×10^{-6}	0.0000289142	0.000302839
8	0.924335	0.43561×10^{-6}	2.13453×10^{-7}	4.0556×10^{-6}	0.0000424771
9	0.9447	7.06025×10^{-6}	1.82686×10^{-7}	3.47103×10^{-6}	0.0000363545
10	0.95874	1.04312×10^{-6}	2.48339×10^{-8}	4.71844×10^{-7}	4.94195×10^{-6}
11	0.968604	9.37995×10^{-7}	2.04234×10^{-8}	3.88044×10^{-7}	4.06425×10^{-6}
12	0.975675	1.34085×10^{-7}	2.72056×10^{-9}	5.16906×10^{-8}	5.41391×10^{-7}
13	0.98085	1.16911×10^{-7}	2.19289×10^{-9}	4.1665×10^{-8}	4.36386×10^{-7}
14	0.984717	1.63195×10^{-8}	2.87944×10^{-10}	5.47094×10^{-9}	5.73009×10^{-8}
15	0.987662	1.39573×10^{-8}	2.29571×10^{-10}	4.36184×10^{-9}	4.56846×10^{-8}

Algorithm 5: W1R2

```
1  Code for client process p_i(0 ≤ i ≤ n − 1):
2  procedure OneRoundWRITE(key, value)
3      localSeq[key] ← localSeq[key]+1
4      version ←(localSeq[key], i)
5      update(key, value, version)
6  procedure TwoRoundREAD(key)
7      replicas ← query(key)
8      version ←maxVer(replicas)
9      value ←valWithMaxVer(replicas, version)
10     localSeq[key] ← version.seq    ▷ Recommended.
11     update(key, value, version)
12     return value
```

Algorithm 6: W1R1

```
1  procedure OneRoundWRITE(key, value)
2      localSeq[key] ← localSeq[key]+1
3      version ←(localSeq[key],i)
4      update(key, value, version)
5  procedure OneRoundREAD(key)
6      replicas ← query(key)
7      version ←maxVer(replicas)
8      value ←valWithMaxVer(replicas, version)
9      localSeq[key] ← version.seq    ▷ Recommended.
10     return value
```

is constructed as follows. Suppose there exist two writer clients whose identifiers are p_0 and p_1, and we only focus on the same register. At time t_0, p_0 stores v_0 as its local sequence number for the register, and p_1 stores v_1, satisfying $v_1 > v_0$. During the period from t_0 to t_1, p_1 executes k writes successively while p_0 initiates no writes. Thus, at time t_1, p_1's local version is increased to be $(v_1 + k, 1)$ while p_0 still keeps v_0. After that, during the period from t_1 to t_2, only p_0 executes k writes successively and its local version is increased to be $v_0 + k$ at t_2. Since $(v_0 + k, p_0) < (v_1 + k, p_1)$, the written value versioned $(v_0 + k, p_0)$ is invisible. Then at t_3, a read occurs but still reads the value versioned $(v_1 + k, p_1)$. In above case, the permutation of operations on p_0 and p_1 goes like this: (The *version* that a *write* assigns or a *read* returns are shown in brackets.) $w(v_1 + 1, p_1) \prec_\pi w(v_1 + 2, p_1) \prec_\pi \ldots \prec_\pi \underline{w(v_1 + k, p_1)} \prec_\pi w(v_0 + 1, p_0) \prec_\pi w(v_0 + 2, p_0) \prec_\pi \ldots \prec_\pi \underline{w(v_0 + k, p_0)} \prec_\pi \underline{r(v_1 + k, p_1)}$. Since $r(v_1 + k, p_1)$ returns a value not written by one of the latest k preceding writes in π, so the history violates k-atomicity. By contradiction, we prove the correctness of Theorem 3.1. □

3.2 Quantification of Visible Writes in One-Round-Trip Write

The one-round-trip write algorithms including W1R2 and W1R1 not only can't provide bounded k-atomicity, but also lead to frequent invisible writes that cannot be read. Here, we quantify the rate of visible writes incurred in W1R2 & W1R1 using the following assumption. Assume a Poisson process with parameter λ for the scenario of each client issuing a sequence of write/read operations, but *ignore the duration of each operation*. That is, each write/read is regarded as an instant event taking effect on the register at its starting moment. Suppose there are n_w writer clients($n_w > 1$), identified 0, 1, ..., $n_w - 1$ separately. Then we have n_w independent, parallel write sequences, each satisfying a Poisson process with parameter λ. Assume that at the initial moment, all n_w writer clients are informed of the version (v_0, id) for some specific register that we focus on here. Let $N^i(t)$ be the number of write operations issued by writer client i from the initial moment till time t, where $0 \leq i \leq n_w - 1$. According to the formula of Poisson probability, we have

$$\mathbb{P}\{N^i(t) = k\} = \frac{(\lambda t)^k e^{-\lambda t}}{k!}$$

Assume that writer client i stores the local sequence number $k - 1$ at time t. Let $V^i(t, k)$ be a random variable denoting the visibility of the un-occurred write versioned $(v_0 + k, i)$ at time t. Then, $V^i(t, k)$ has two status: invisible(denoted 0) and visible(denoted 1). $V^i(t, k) = 1$ occurs if and only if all other writer clients' local sequence number stayed smaller than $(v_0 + k, i)$ at time t. Since write

sequences are independent, we have

$$\begin{aligned}&\mathbb{P}\{V^i(t,k)=1\}\\&=\prod_{j=0}^{i-1}\mathbb{P}\{N^j(t)<k+1\}\prod_{j=i+1}^{n_w-1}\mathbb{P}\{N^j(t)<k\}\\&=\left(\sum_{j=0}^{k}\frac{(\lambda t)^j e^{-\lambda t}}{j!}\right)^{i}\left(\sum_{j=0}^{k-1}\frac{(\lambda t)^j e^{-\lambda t}}{j!}\right)^{n_w-1-i}\end{aligned}$$

Here, we quantify the rate when $k=1$ at the expected arrival time $t=\frac{1}{\lambda}$, when the probability of $N^j(t)=1$ is the highest.

By setting $\lambda=10s^{-1}$, we present the numerical results in Fig. 7 and Table 4. The numerical results show that the probability of visible writes will fall as there exist multiple writer clients. When the number of writer clients is up to 10, the probability of visible writes for the writer client with the maximum id is about 0.063, while for the writer client with the minimum id, the probability is about 10^{-4}.

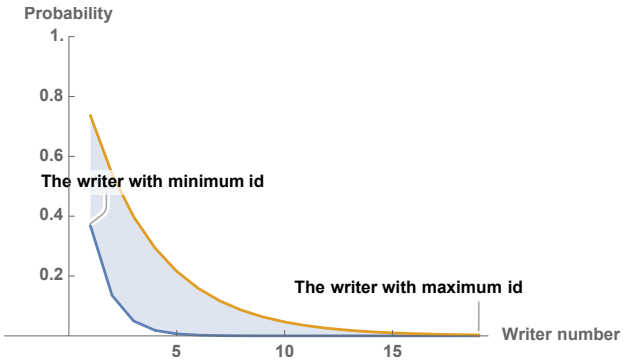

Fig. 7. The probabilities of visible writes from writer clients with different identifiers ($\lambda=10s^{-1}, t=100ms$).

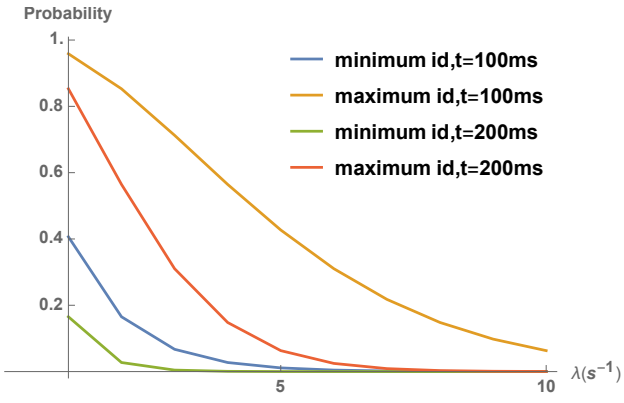

Fig. 8. The probabilities of visible writes from the writer client with maximum / minimum identifier by varying λ ($n_w=10$).

3.3 Experimental Results of One-Round-Trip Write Algorithms

The rough theoretical analysis in Section 3.2 doesn't take the duration of each operation into consideration, so we conduct more comprehensive experimental evaluations of W1R2 and W1R1 in quorum-replicated datastores. The experimental design and the environment settings are described in the experiments of W2R1, and the experimental results of W1R2 and W1R1 are presented in Table 5.

For read operations, the overall probability of obtaining an up-to-date value is varied from 46.6% to 91.1%. However, under most environment settings, the probability is around 50%. That is, nearly half of the read operations will return stale data. Moreover, the staleness of the returned data can be quite high. We only calculate the rate of k up to 10 in terms of k-atomicity. Specifically, the probability of returning an value with the staleness of 5 is varied from 0.7% to 8.1%. When the staleness is up to 10, the probability is around 1%. Note that if the returned data is with staleness of 5 or even more, it is nearly of no use for users. Thus, reader clients may frequently obtain stale or even invalid data. From the aspect of writes, the considerable possibility of issuing invisible updates is the main reason for reads to return stale data. Therefore, the algorithms of both W1R2 and W1R1 without any improvement are far away from use [6].

3.4 Improvement of One-Round-Trip Write Algorithms

Imagine slightly modifying the procedure of $update$ function. When a replica server receives an "update" message and finds the intended update version smaller than it stores, the server additionally fills the ACK with extra information (like the sequence number of its stored version) instead of nothing and reply to the client. A client would have to wait for ACKs from a majority of replicas and update its local sequence number according to the information in the ACKs. Then, an invisible one-round-trip write may work like the $query$ function and help update the local sequence number on the client side. In this way, W1R2 can be proved to satisfy n_w-atomicity given n_w writer clients.

Here we quantify the rate of visible writes in the improved W1R2 & W1R1 algorithms. Suppose the writer client i issues a write w after time t from its last write operation w'. Let $U^i(t)$ be a random variable denoting the visibility of the write w. Similarly, $U^i(t)$ has two status: invisible(denoted 0) and visible(denoted 1). Here, we only focus on a specific but frequent pattern that would inevitably make w invisible: during the period from the finish time of w' to the invocation of w, there exists a writer client identified smaller than i completing at least three writes, or a writer client identified larger than i completing at least two writes [7]. Since write sequences are independent, through probabilistic and

6. For MWMR registers, the W1R2 and W1R1 algorithms that use discrete version (seq, id) are far away from use. However, the one-round-trip write algorithms in Cassandra and some other datastores use timestamps following the UTC standard, which is different from our algorithms. Theoretically speaking, these datastores cannot always promise to produce unique timestamps for different data, and that's out of scope in this work.

7. This can be proved through a case-by-case analysis.

TABLE 4
Numerical results on the probabilities of **invisible** writes in one-round-trip write algorithms without extra information in ACK ($\lambda = 10s^{-1}, t = 100ms$).

Writer numbers	$id=0$	$id=1$	$id=2$	$id=3$	$id=4$	$id=5$	$id=6$	$id=7$	$id=8$	$id=9$
$n_w=2$	0.632121	0.264241								
$n_w=3$	0.864665	0.729329	0.458659							
$n_w=4$	0.950213	0.900426	0.800852	0.601703						
$n_w=5$	0.981684	0.963369	0.926737	0.853475	0.70695					
$n_w=6$	0.993262	0.986524	0.973048	0.946096	0.892193	0.784386				
$n_w=7$	0.997521	0.995042	0.990085	0.98017	0.96034	0.92068	0.84136			
$n_w=8$	0.999088	0.998176	0.996352	0.992705	0.98541	0.97082	0.94164	0.883279		
$n_w=9$	0.999665	0.999329	0.998658	0.997316	0.994633	0.989265	0.97853	0.957061	0.914122	
$n_w=10$	0.999877	0.999753	0.999506	0.999013	0.998025	0.996051	0.992102	0.984204	0.968407	0.936814

TABLE 5
Experimental results of one-round-trip write algorithms without extra information in ACK ($\lambda = 10s^{-1}, t = 100ms$).

Algorithm	Quorum level	Replica	Inter-DC delay/ms	Client num.	Read ratio	$\mathbb{P}(k=1)$	$\mathbb{P}(k=5)$	$\mathbb{P}(k=10)$
W1R2	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.50	0.911	0.007	0.010
W1R2	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.60	0.878	0.014	0.009
W1R2	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.541	0.058	0.011
W1R2	QUORUM	3	$\mathbb{N}(50, 25^2)$	30	0.90	0.494	0.057	0.006
W1R2	QUORUM	3_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.518	0.069	0.004
W1R2	QUORUM	3_3_3	$\mathbb{N}(50, 25^2)$	30	0.90	0.529	0.064	0.016
W1R2	EACH_QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.540	0.071	0.016
W1R1	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.50	0.864	0.017	0.012
W1R1	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.60	0.816	0.024	0.021
W1R1	QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.507	0.052	0.023
W1R1	QUORUM	3	$\mathbb{N}(50, 25^2)$	30	0.90	0.496	0.091	0.009
W1R1	QUORUM	3_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.466	0.061	0.011
W1R1	QUORUM	3_3_3	$\mathbb{N}(50, 25^2)$	30	0.90	0.563	0.059	0.014
W1R1	EACH_QUORUM	1_1_1	$\mathbb{N}(50, 25^2)$	30	0.90	0.496	0.053	0.019

TABLE 6
Numerical results on the probabilities(lower bound) of **invisible** writes in one-round-trip write algorithms if sequence number is included in ACK ($\lambda = 10s^{-1}, t = 100ms$).

Writer numbers	$id=0$	$id=1$	$id=2$	$id=3$	$id=4$	$id=5$	$id=6$	$id=7$	$id=8$	$id=9$
$n_w=2$	0.264241	0.0803014								
$n_w=3$	0.458659	0.323324	0.154154							
$n_w=4$	0.601703	0.502129	0.377662	0.222077						
$n_w=5$	0.70695	0.633687	0.542109,	0.427636	0.284545					
$n_w=6$	0.784386	0.730482	0.663103	0.578878	0.473598	0.341997				
$n_w=7$	0.84136	0.8017	0.752125	0.690156	0.612695	0.515869	0.394836			
$n_w=8$	0.883279	0.854099	0.817624	0.77203	0.715037	0.643796	0.554745	0.443431		
$n_w=9$	0.914122	0.892652	0.865815	0.832269	0.790336	0.73792	0.6724	0.5905	0.488125	
$n_w=10$	0.936814	0.921018	0.901272	0.87659	0.845738	0.807172	0.758965	0.698707	0.623383	0.529229

combinatorial analysis we have

$$\begin{aligned}&\mathbb{P}\{U^i(t)=0\}\\ \geq&1-\prod_{j=0}^{i-1}\mathbb{P}\{N(t)<3\}\prod_{j=i+1}^{n_w-1}\mathbb{P}\{N(t)<2\}\\ =&1-e^{-\lambda t(n-1)}\cdot(1+\lambda t+\frac{1}{2}(\lambda t)^2)^i\cdot(1+\lambda t)^{n_w-1-i}\end{aligned}$$

Then, we have

$$\begin{aligned}&\mathbb{P}\{U^i(t)=1\}=1-\mathbb{P}\{U^i(t)=0\}\\ \leq&e^{-\lambda t(n-1)}\cdot(1+\lambda t+\frac{1}{2}(\lambda t)^2)^i\cdot(1+\lambda t)^{n_w-1-i}\end{aligned}$$

Here, we quantify the rate at the expected time $t=\frac{1}{\lambda}$. By setting $\lambda = 10s^{-1}$, we present the numerical results in Fig. 9 and Table 6. The numerical results of the improved W1R2 and W1R1 are similar to those without improvement in Section 3.2, although the probability of visible writes raises a little. When the number of writer clients is up to 10, the probability of visible writes for the writer client with the maximum id is about 0.047, while for the writer client with the minimum id, the probability is about 0.063. However, the improvement is not significant. Actually, the modification in *update* doesn't prevent the occurrence of invisible writes, so the problem of the one-round-trip write algorithm still exists.

4 W2R1: Experimental Evaluations

In this section, we study the ASC tradeoff mainly with the W2R1 algorithm empirically. First we explore the impacts of different environment factors, then we conclude the influences of different ASC algorithms on data consistency and read latency.

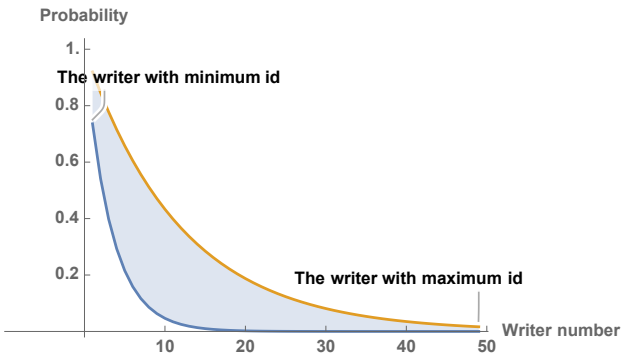

Fig. 9. The probabilities of visible writes from writer clients with different identifiers using the improved update ACK ($\lambda = 10s^{-1}, t = 100ms$).

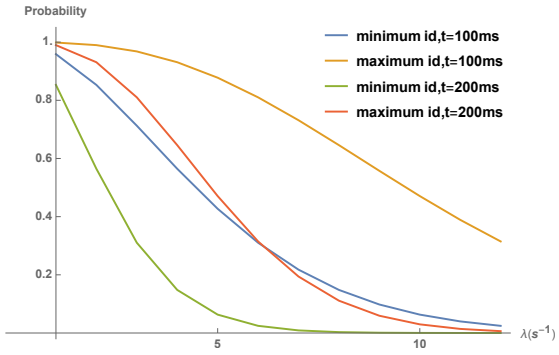

Fig. 10. The probabilities of visible writes from the writer client with maximum / minimum identifier by varying λ using the improved update ACK ($n_w = 10$).

4.1 ASC Tradeoff in Different Environments

We investigate the impacts of different environment factors on the consistency and latency performance of the W2R1 algorithm (with / without optimization) running on Cassandra. Besides, the W2R2 algorithm is explored for comparison. The environmental settings are varied from workload pattern (client number and read ratio), replica configuration, communication delay to read quorum level.

4.1.1 Impact of Client Number

The client number reflects the concurrency degree of operations to some extent. Intuitively, more clients are able to result in higher concurrency. In the experiments, each client is acted by one YCSB instance. All clients are allowed to initiate read/write requests concurrently, while each of them can only execute at most one request at any time. By varying the client number from 10 to 40, we derive the results shown in Fig. 11.

First we have a look at the data consistency performance of the algorithms. W2R2 guarantees atomicity all the time as is expected, while other algorithms can lead to atomicity violation more or less. From the perspective of worst case we obtained, W2R1 without any optimization produced some non-atomic traces with k up to 4 in terms of k-atomicity, while others resulted in better performance. From the perspective of average consistency, W2R1 also has the worst performance compared to those with one of the Cassandra's optimizations(snitch, digest or read repair). However, although all but W2R2 can't satisfy atomicity all the time, they guarantee atomicity most of the time, with a confidence that more than 99.97% read requests can obtain the most up-to-date data. In overall, the average consistency performance becomes weaker as the concurrent client number increases, and we produced the worst case of $k = 4$ when client number is up to 40. We speculate that the rate of returning fresh data for reads would decrease with more concurrent clients. However, most of the time these non-atomic algorithms can still promise atomicity. Therefore, we believe that W2R1 is applicable for many applications and scenarios, especially when concurrent clients operating on single shared register are no more than 40.

As for latency, we observe that the average read latency is positively relative to the client number in a subtle degree. The main reason is obvious: the processing capacity of each server is limited. With the concurrency control, the waiting time in the buffer will become longer as the user throughput increases. From another aspect, W2R2 costs higher *read* latency compared with other four algorithms, while it gains stronger consistency all the time. Different algorithms behaves in a slightly different way and we'll discuss it in detail in Section 4.2.

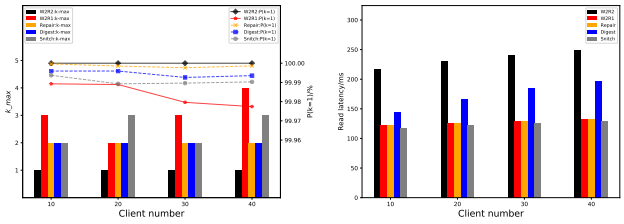

Fig. 11. Consistency and latency results by tuning the number of clients.

4.1.2 Impact of Read Ratio

In the experiments, each client can initiate read and write requests. The ratio between read and write can also be a critical factor to impact the chance to produce violation patterns. Here, we use *read ratio*(the ratio of read numbers to the total operation numbers) to describe the relations between read and write numbers. By varying the read ratio from 0.50 to 0.99, we derive the results shown in Fig. 12. As we can see, W2R2 still guarantees atomicity while others do not. W2R1 with snitch leads to a monotonically decreasing atomicity degree with higher read ratio. For the W2R1 algorithm without optimization or with digest / repair, higher read ratio within certain range (0.5 - 0.9) can result in weaker consistency guarantee in average. However, when the read ratio is up to 0.99, the consistency performance reversely becomes better. One potential reason is that, k-atomicity violation is described by the staleness of data returned by reads, so within certain range, more reads will have more chances to form more complicated read-write patterns that make reads stale. However, as the ratio of reads grows too high, writes will become too rare and sparsely distributed for reads to obtain stale written values.

As for latency, we observe that the average read latency is approximately irrelative to read ratio for all algorithms except W2R1 with digest. The reason why request latency

for W2R1 with digest grows higher when read ratio gets lower is mainly that, when write proportion becomes relatively higher, more frequent updates will make digest mismatch occur more frequently, triggering extra round-trip for collecting full data.

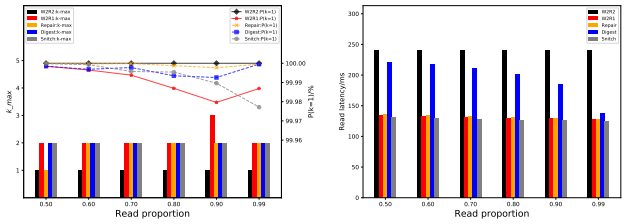

Fig. 12. Consistency and latency results by tuning read ratio.

4.1.3 Impact of Replica configuration

The replica factor in Cassandra specifies the number of replicas in each data center. The number of replicas impacts the consistency and latency performance in some degree, while the data center location of replicas impacts the request latency. Given a number of replicas, besides spreading them into different data centers, we can also place them all into a single data center for the convenience of local access. In the experiments, the replica factor is varied in 3(inside single data-center), 1_1_1, 3_1_1, and 3_3_3. As is shown in Fig. 13, more replicas would result in stronger consistency but higher latency in overall. The reason is obvious: the latency of an operation depends on the slowest responding replica which costs the longest time to communicate. In the network where delays are randomly distributed, it's more probable to have longer delays when communicating with more replicas. However, the replica configuration of 3_1_1 is an exception. 3_1_1 may sometimes select a quorum of replicas all in the local data center, which can reduce the overall read latency at the cost of consistency lost compared to 1_1_1. Besides, W2R1 with snitch behaves in a similar way. Applying smart routing through the snitch strategy makes the algorithm always process each read/write request by accessing local replicas in priority, which tends to lower the overall latency, but replica updates in remote data centers are usually not able to spread in time. The results under the setting of the 3_1_1 replica configuration with snitch is a powerful evidence for above explanations.

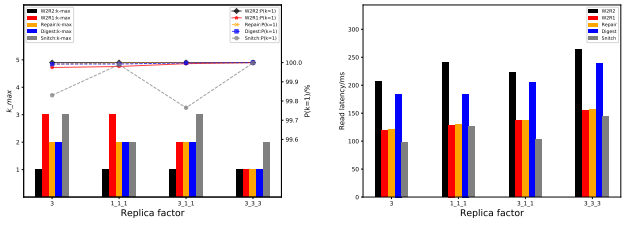

Fig. 13. Consistency and latency results by tuning replica configuration.

4.1.4 Impact of Network Delay

In our experiments, we mainly focus on inter-data center delay because it impacts the performance in a significant way compared to intra-data center delay. The default injected one-way inter-data center delays in network are normally distributed with the average of $50ms$ and the standard deviation of $25ms$ (denoted $\mathbb{N}(50, 25^2)$). By varying μ from $10ms$ to $50ms$, and σ from $5ms$ to $25ms$ respectively, we obtain the results shown in Fig. 14 and Fig. 15 . We observe that, the average latency of read operations is basically in proportion to the average network delays across data centers. Besides, the consistency performance become weaker when the average network delays or jitters grow higher and higher. This is due to, not only higher network delay would make each operation duration stay longer, but also larger jitters or variances of delays would aggravate the occurrence of out-of-sync replicas. Then, the joint effects of above both lead to frequent atomicity violation patterns.

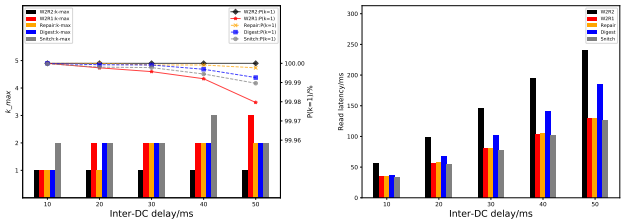

Fig. 14. Consistency and latency results by tuning the average value of inter-DC delay ($\mu = 10/20/30/40/50ms$, $\sigma = \frac{\mu}{2}$, accordingly).

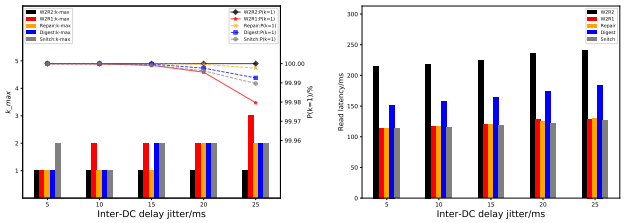

Fig. 15. Consistency and latency results by tuning inter-DC delay jitter ($\mu = 50ms$, $\sigma = 5/10/15/20/25ms$).

4.1.5 Impact of Read Quorum Level

We test two types of read quorum level in Cassandra: QUORUM and EACH_QUORUM [8]. The level of QUORUM requires to access a majority of replicas from all data-centers, which is actually the same as our theoretical model. The level of EACH_QUORUM requires to access a majority of replicas in each data-centers, which requires more replicas for responses compared to our theoretical model. As is shown in Fig. 16, EACH_QUORUM promises better consistency performance in average at the cost of extra latency; QUORUM lowers the average latency at the sacrifice of a little bit lower atomicity rate. The tradeoff between consistency and latency is well presented in this experiment.

TABLE 7
ASC tradeoff under the default setting

	k_{max}	$\mathbb{P}(k=1)$	$\mathbb{P}(k=2)$	$\mathbb{P}(k=3)$	$\mathbb{P}(staleness)$	Read latency
W2R2	1	100%	0	0	0	100%
W2R1	3	99.9796%	0.0203%	0.0001%	0.0204%	53%
Snitch	2	99.9896%	0.0104%	0	0.0104%	52.3%
Digest	2	99.9926%	0.0074%	0	0.0074%	76.5%
Repair	2	99.9977%	0.0023%	0	0.0023%	53.9%

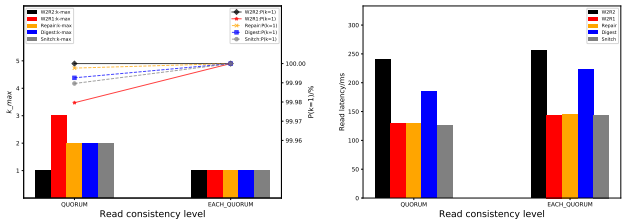

Fig. 16. Consistency and latency results by tuning read quorum level.

4.2 ASC Tradeoff with Different Algorithms

From all above experimental results, we observe that distinct algorithms vary in the ASC tradeoffs. Here, we summarize some insights from the following several perspectives.

4.2.1 Consistency

We first compare the data consistency provided by W2R1 and W2R2. No matter under what circumstance, W2R2 provides atomicity all the time, while W2R1 (with / without optimization) does not. The W2R1 algorithm produced non-atomic traces with k up to 4 in terms of k-atomicity, although the probability of atomicity violation is quite low. Specifically, the W2R1 algorithm produced stale reads with the probability lower than 0.3% in our experiments under all circumstances, and lower than 0.02% under most circumstances. For example, from the experimental results under the default setting in Table 7, the rate of 2-atomic reads is lower than 0.03%, and 3-atomic reads even lower, not to mention higher values of k in terms of k-atomic reads.

Next, we make a comparison among different W2R1 optimizations. W2R1 with repair performs best among all W2R1 algorithms, and its probability of consistent reads in average is merely a little inferior than W2R2. W2R1 with digest or snitch doesn't perform so well as W2R1 with repair, but better than W2R1 without any optimization in overall. Above all, we believe that W2R1 (with / without optimization) rarely violates atomicity, and still promises atomicity most of the time.

4.2.2 Read Latency

We mainly focus on read latency. Normally, W2R1 costs lower read latency than W2R2, which is expected. Among different W2R1 optimizations, W2R1 with digest has the largest read latency in average. We speculate in two reasons. Firstly, using digest requests means that in the first communication round-trip of read, only one replica would be requested with full data, and all others with digest. Once there exists some inconsistent replica responded, an extra round-trip for data retransmission is required. Secondly, compared to raw data collection, the digest request reduces data size in communication but requires extra work for data hashing in replica servers, which takes extra time. For W2R1 with repair, it leads to approximately the same or just slightly higher read latency compared to W2R1 without repair. This is due to that it applies asynchronous mechanism to repair stale replicas in the background after the response of the read, which has nearly no effect on the read response time. As for W2R1 with snitch, it gains the highest efficiency for reads mainly because it always selects the replicas with high proximity and route requests to them, which significantly decreases latency.

4.2.3 Consistency-Latency Tradeoff

First we conclude the two theoretical algorithms. W2R2 provides atomicity all the time at the sacrifice of extra communication round-trip for write-back in reads, which inevitably leads to high read latency. By omitting the second round-trip of read, W2R1 without any optimization gains lower latency but worse data consistency, although it still guarantees atomicity most of the time.

Based on the theoretical W2R1 algorithm, the optimization in Cassandra can help speedup request process by selecting replicas smartly(*snitch*), provide stronger consistency through replica synchronization(*repair*), or reduce communication data size(*digest*). W2R1 with snitch has the lowest read latency due to its smart routing, but provides the weakest consistency among three. W2R1 with repair performs better in providing atomicity without increasing much read latency since it puts repair work in the background. W2R1 with digest performs worse than W2R1 with repair but better than W2R1 with snitch in data consistency provided, while its read latency is a lot higher than the other two.

Above all, the W2R1 algorithm with or without optimization achieves the ASC tradeoff in the common case. Specially, W2R1 with *repair* performs significantly well. The major wisdom of this optimization lies in the *divergence-oriented communication*, i.e., only when inconsistency has been found will the algorithm take extra action for repair. In this way, one communication round-trip is enough for those consistent responding replicas. From this perspective, W2R1 with *repair* can be regarded as a combination of W2R2 and W2R1. Moreover, repairing replicas in the background asynchronously is an efficient design to provide both strong consistency and low latency.

8. Cassandra also provides a special quorum level: LOCAL_QUORUM. The level of LOCAL_QUORUM only requires to access a majority of replicas in the local data-center, which speeds up a lot while does actually not meet the requirement of majority-replication. Therefore, it's out of scope for this work.

REFERENCES

[1] P. B. Gibbons and E. Korach, "Testing shared memories," *SIAM Journal on Computing*, vol. 26, no. 4, pp. 1208–1244, 1997.
[2] H. Wei, Y. Huang, and J. Lu, "Probabilistically-atomic 2-atomicity: Enabling almost strong consistency in distributed storage systems," *IEEE Trans. Comput.*, vol. 66, no. 3, pp. 502–514, Mar. 2017. [Online]. Available: https://doi.org/10.1109/TC.2016.2601322
[3] S. M. Ross, *Introduction to probability models*. Academic press, 2014.