Received: 4 December 2016 | Revised: 13 February 2017 | Accepted: 18 February 2017

DOI: 10.1002/cpe.4129

WILEY

RESEARCH ARTICLE

# An experimental study on tuning the consistency of NoSQL systems

Xiangdong Huang[1] | Jianmin Wang[1] | Philip S. Yu[2,3] | Jian Bai[1] | Jinrui Zhang[1]

[1]School of Software, Tsinghua University, Beijing, China
[2]Department of Computer Science, University of Illinois at Chicago, IL, USA
[3]Institute for Data Science, Tsinghua University, Beijing, China

**Correspondence**
Xiangdong Huang, School of Software, Tsinghua University, Beijing 100084, China.
Email: huangxd12@mails.tsinghua.edu.cn

**Funding information**
NSFC, Grant/Award Number: 61325008; National Key R&D Program of China, Grant/Award Number: 2015BAF32B01; Tsinghua National Laboratory (TNList) Key Project

## Summary

The eventual consistency model has been widely adopted in NoSQL systems. By tolerating weak consistency, these systems attain high throughput and availability while sustaining side effects on user experience and developer friendliness. Trading off consistency from latency has been a common consensus. An important but widely ignored problem is how to control the consistency of an existing system without the necessity of modifying the system implementation. In this paper, we present a systematic study on the client-centric consistency of a NoSQL system, Cassandra, and disclose how the consistency can be substantially enhanced by tuning the system configurations when users use partial quorum settings. We use session guarantee as the consistency model and analyze the root cause of consistency violation, testifying that the length of the write queue is a reasonable indicator for consistency quantification. For inconsistency mitigation, we show through extensive experiments how the consistency is affected by the read and write processes of the system, and how the consistency can be improved by tuning system configurations. In particular, we provide developers with recommended configurations by changing the write thread number and the fine-grained quorum setting for enhanced consistency control. Because consistency anomalies do not occur uniformly, we discuss how to stabilize the consistency by analyzing system logs.

**KEYWORDS**

Cassandra, NoSQL, optimization, queue, replica consistency

## 1 | INTRODUCTION

A storage system providing weak replica consistency model is easier to achieve high availability, high throughput, and low latency. Therefore, many NoSQL systems, especially quorum systems such as DynamoDB, Voldemort, Riak, and Cassandra opt for eventual consistency, a typical weak replica consistency model.[1] These systems become popular choices by users with the awareness that they may read stale data with certain probability. Since eventually consistent systems make no rigorous guarantees on the staleness of data items returned, it is very important for users and developers to quantify how eventual the consistency is, how to program under the eventually consistent systems and how to provide stronger consistency while keeping its benefits.[2]

Ongoing research efforts are made to quantify the consistency in NoSQL systems.[3,4] Most works focus on the consistency–latency trade-off according to the PACELC criteria.[1] For example, Wada et al claimed that with eventual consistency, the probability of reading the latest data within 0 and 450 ms is 33% in a low workload using Amazon SimpleDB.[5] Bails et al showed that the average latency is 25 ms in LinkedIn's single-node Voldemort system and the maximal latency is 5s in Yammer's Riak cluster.[6] Some works consider how to program in these NoSQL systems. For example, the Consistency As Logical Monotonicity (CALM) theorem was proposed for guiding developers to design programs under the eventually consistent systems.[7] Other works focus on designing new read and write protocols to support stronger consistency.[8–11] For example, Bails et al proposed a "bolt-on" layer to support causal consistency.[11] Zhu and Wang proposed replica consistency-on-demand store to facilitate on-demand consistency by controlling the steps of read/write process.[12]

However, an important but widely ignored problem, especially for developers, is that given a workload for a storage system, whether there is a way to control the consistency of an existing system without the necessity of modifying the system implementation. That is, how developers can achieve a desired consistency degree by merely tuning the configuration parameters according to the hardware specification and how to make consistency anomalies uniformly distributed and avoid

 Copyright © 2017 John Wiley & Sons, Ltd. 

consistency anomalies outbreak, which is caused by system-level incidental events[13] and will severely hurt user experience since the bursty of consistency anomalies in which users read many stale data within a very short time, need to be considered.

In this paper, we address the above problems by presenting a systematic study on the client-centric consistency of a popular quorum NoSQL system, Cassandra. We analyze the root cause of consistency anomalies using the write queue and propose solutions to control the replica consistency under the session guarantee consistency model.[14] First, we analyze the read/write process and summarize why the consistency anomalies occur. We testify through extensive experiments that the consistency is closely related to the length of the write queue and can be substantially enhanced by tweaking the system configurations. Second, we empirically evaluate 2 potential enhancement methods for the replica consistency by modifying 2 kinds of system configurations. We devise the way to stabilize the distribution of consistency by mitigating the influence from system-level incidental events. While this study is mainly conducted on Cassandra, our conclusions are generalizable to other quorum systems such as Riak and Voldemort, given that they have similar architecture as Cassandra.

In summary, we make the following contributions in this paper:

- We theoretically reveal the root cause of consistency anomalies and analyze how the write queue impacts the replica consistency. We show that the length of write queue can be used as a reasonable indicator for quantifying consistency in real time because a longer queue leads to worse consistency, which is verified in our experiments under different workloads. We also summarize 3 bottlenecks to the write queue length that lead to the weak and unstable consistency.
- Two methods by tuning the write thread number according to the hardware specifications and by adjusting the quorum parameters are proposed to improve the replica consistency in Cassandra. In the experiments with different cluster environments, these methods can improve the replica consistency while somewhat sacrificing the read latency.
- Internal events in Cassandra that generate the consistency jitter are observed by analyzing the Cassandra logs in experiments. We find the event of flushing memtable may interrupt the write queue while a small memtable size can help make the distribution of consistency anomalies nearly uniform.
- Quantitative methods that optimize system configurations for controlling the consistency are summarized for practitioners:

    1. An appropriate number of the concurrent write threads can improve the replica consistency. The number of threads is determined by the CPU cores in that the best empirical value is twice of the CPU cores in different hardware environments.
    2. When the sum of read consistency level ($r$) and write consistency level ($w$), ie, $r + w$ keeps constant, changing $r$ and $w$ leads to different consistency and throughput trade-offs. The best configuration for maximized consistency is $w - r \leqslant 1$.
    3. A small memtable size is useful to alleviate the jitter of the write-queue length and hence mitigate the consistency anomalies outbreak in a short time for improved user experience. In our experiments, 512MB is suitable for the servers whose memory size is less than 32GB.

The remainder of the paper is organized as follows: In Section 2, we introduce the 4 types of session guarantee consistencies and some basic concepts in Cassandra. In Section 3, we sketch the read and write process in Cassandra and analyze the causes of inconsistency, based on which we propose the write-queue length as a reasonable indicator for consistency quantification. Then, we propose 3 methods to control the queue length and make the queue more stable. In Section 4, we describe the experiment environment and workloads setup. In Section 5, we experimentally study the relationship between the write-queue length and the consistency degree under different workloads (throughput, number of concurrent clients, number of replicas, read/write ratio, and read/write consistency level). In Section 6, we validate the effectiveness of the 3 consistency-control methods both independently and collectively and discuss their adverse effects. In Section 7, we compare the related works with ours. In Section 8, we conclude this study.

## 2 | BACKGROUND

### 2.1 | Session guarantee consistency models

As one of the well–known client-centric replica consistency models, session guarantee was first introduced in Terry et al[14] and further discussed and formalized in recent years.[5,15–17] Session guarantee contains 4 types of consistency models: read your writes, monotonic reads, writes follow reads, and monotonic writes.

Read your writes consistency: In 1 session, if a read $R$ follows a write $W$, the returned version of $R$ must be fresher than or equal to the version that $W$ writes. Then the system satisfies read your writes consistency (denoted by RYWC[16]).

Monotonic reads consistency: In 1 session, if a read gets a data version $v$, then the versions that the subsequent reads get must be fresher than or equal to the version $v$. Then, the system satisfies monotonic reads consistency (denoted by MRC[16]) .

Monotonic writes consistency: In 1 session, if a write updates a data item as version $v$, then the subsequent writes must be applied on replicas whose versions of this data item are fresher than or equal to $v$. Then the system satisfies monotonic writes consistency (denoted by MWC[16]).

Writes follow reads consistency: In 1 session, assuming that a write $W$ follows a read $R$ and the version which $R$ gets is $v$, $w$ can only be applied on the replicas whose data item version is fresher than or equal to $v$. Then the system satisfies writes follow reads consistency (denoted by WFRC[16]).

#### 2.1.1 | The scope of session guarantee consistency

To analyze the internal cause, we should clarify the meaning of "in a session". A session represents that 1 and only 1 connection is established between the client and the cluster. The most important constraint is that the client can send a new request only after it has received a response of the previous request. Therefore, there is no chance that the cluster receives the latter request earlier than the previous one.

However, the constraint does not restrict the concurrency of the 2 requests. For example, the cluster can respond immediately after it receives a request and then executes the request asynchronously. In this way, though the client has to send requests one by one, the cluster may execute them in parallel.

### 2.2 | Concepts in Cassandra

Coordinator. When a client connects to the Cassandra cluster and sends a request, the node that the client connects to is called the coordinator. Other nodes that participate in handling the request are called noncoordinators (abbr. non-coor). The coordinator may have a replica of the required data item or not. Each noncoordinator has 1 replica of the data item.

Quorum Size. Cassandra uses quorum consensus[18] to tuning the consistency. A quorum consists of the write consistency level $w$ and read consistency level $r$. Write consistency level $w$ means that the write operation must be finished for at least $w$ replicas. Read consistency level $r$ means the coordinator reads data from $r$ replicas and gets the latest version from them. Suppose the number of replicas for each data item is $n$. There is a common insight: If $w + r > n$, the system provides strong consistency. Otherwise, the system acts as a weakly consistent system and the consistency becomes stronger along with the increment of $w + r$. We call $w + r <= n$ as a partial quorum,[6] and the paper mainly focuses on this scenario.

Stage Event-Driven Architecture (SEDA). Cassandra is designed as a SEDA (SEDA[19]) to support high concurrency. Under this architecture, the computing resources are divided into several pieces and nonoverlapping thread resources are allocated to the read and write processes.

## 3 | THE WRITE QUEUE AND CONSISTENCY

### 3.1 | Read and write process in Cassandra

First, we define some terminologies and symbols. In this discussion, we only concern on the participated nodes (coordinator and noncoordinator nodes) for a request. The value of data item $a$ on the $i$th node (the $i$th replica, denoted by $N_i$) is $a[i]$. $Read(a)$ represents a read request for the data item $a$, $Read(a, v)$ represents the returned value of the read request for $a$ is $v$. $Write(a, v)$ represents modifying the value of $a$ as $v$.

Figure 1 shows the read and write processes in Cassandra. The figure illustrates that a read request $Read(a)$ follows a write request $Write(a, 2)$ and then a write request $Write(a, 3)$ comes in 1 session. The coordinator $N_0$ is responsible for 1 replica of item $a$. Before the client sends $Write(a, 2)$ request, the values of $a$ in all the replicas are 1.

Bails et al named the read and write processes of Cassandra as the Write, Acknowledge, Read, and reSponse model (WARS)[6] and showed how message reordering gives rise to the staleness. Steps 1, 4, 5, and 10 in Figure 1 constitute the WARS model. We extend the WARS model to explain why consistency anomalies occur in 1 session by splitting the write/read process into fine-grained steps.

In our extended WARS model, a write operation can be divided into 4 steps: write, propagate, response, and acknowledge (Step 1 ∼ Step 4 in Figure 1). Note that Cassandra will send response to the client once the update is finished for $w$ replicas instead of all replicas. The updates of the rest replicas are executed asynchronously.

A read operation can be divided into 6 steps: read, digest propagate, digest response, propagate, response, and return result (Step 5 ∼ Step 10 in Figure 1). Cassandra only requests data from $r$ replicas that are considered as the fastest nodes by the coordinator.

### 3.2 | Scenarios of consistency anomalies

Read your writes consistency. In Figure 1, $Write(a, 2)$ and $Read(a, 1)$ trigger a read your writes anomaly. From the figure, we can find that the direct reason of $Read(a)$ getting a stale value is that noncoordinators $N_{w+2} \sim N_{w+r}$ reorder the read and write operations so that the read operation is finished before the write operation.

Considering the constraint of "in 1 session" in Section 2.1.1, the only reason of the reordering is that the write operation waits too long time for being executed in at least $r$ nodes.

Monotonic writes consistency. $Write(a, 2)$ and $Write(a, 3)$ trigger a monotonic writes consistency anomaly. The reason of the anomaly is the first write operation waits too long time for being executed and Cassandra executes the latter operations in parallel.

However, we cannot observe this anomaly in client-centric view. After $Write(a, 3)$ is finished, the old write operation $Write(a, 2)$ will be aborted due to Cassandra's conflict reconciliation mechanism. Cassandra uses the operation's time stamp to determine which operation is the latest. Then Cassandra aborts the older write operation when it finds a more recent version having been written into memory. To observe the anomaly, we need add logs in Cassandra.

Monotonic reads consistency and writes follow reads consistency. These 2 consistency anomalies are subclasses of read your writes and monotonic writes[15] anomalies. Therefore, the reasons for these 2 anomalies are similar with the above discussion.

### 3.3 | Consistency metrics

In this paper, we do not sentence that whether a system satisfies or violates a kind of consistency model. Instead, we count how many operations violate a given consistency model and calculate the rate of consistency to indicate the consistency intensity of the system. It is similar with Wada et al.[5] In Wada et al,[5] they counted the rate of inconsistency phenomenon in a time window.

Consistency Intensity. In an experiment for session guarantee consistency model $C$, suppose the observer client sends $m$ request pairs. In these $m$ request pairs, there are $k$ pairs that violate the consistency model $C$. Then the consistency intensity is

$$I_C = 1 - \frac{k}{m} \tag{1}$$

Request pair is defined as the following: If the consistency model $C$ is the read your writes consistency model, a write operation and a read operation form a request pair. If $C$ is the monotonic reads consistency model, a write operation for updating the item value and

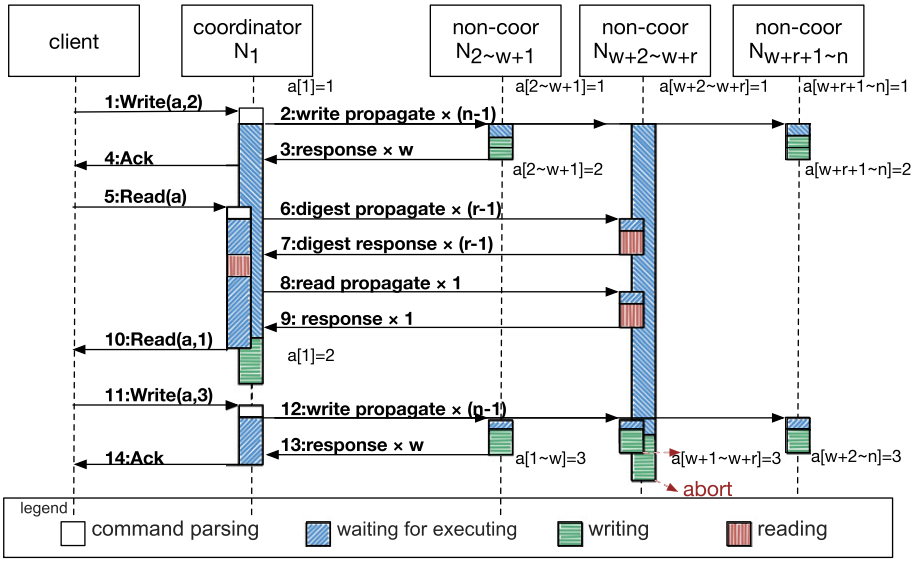


**FIGURE 1** Sequence diagram of read/write processes in Cassandra

2 read operations form a request pair. If *C* is the monotonic writes consistency model, 2 write operations form a request pair. If *C* is the write follows reads consistency model, a write operation for updating the item value, a read operation and another write operation form a request pair.

## 3.4 | The root cause of consistency anomalies

According to the discussions about scenarios of consistency anomalies, we summarize a major reason, which causes the inconsistency: Some write requests wait too long to be finished so that the global execution ordering is not guaranteed (reordering happens). If the write requests can be finished in time before the read request comes, the inconsistency is avoided.

A corresponding structure for writing data in Cassandra is the write queue, which obeys the First In, First Out (FIFO) strategy. All the write requests a node receives enter the write queue (ie, enqueue) and wait to be executed (ie, dequeue). The longer the write queue is, the higher probability that the condition of read your writes anomaly could be met. Therefore, we conjecture that the length of the write queue can be a reasonable indicator for quantifying the degree of consistency.

Figure 2 is a sketch of how to use the write-queue length to quantify the consistency in real time. We acquire the queue length by Java Management Extensions (JMX) in real time for each node. If the queue length of 1 node is exceptionally long, we can conjecture that the consistency anomalies outbreak may happen. If the queue length becomes shorter, we can make a decision that the consistency becomes stronger.

Cassandra also has a read queue to cache the read requests. We do not consider the read queue for 2 reasons: (1) Read operation cannot change the value of the replica, while write queue is closely related to the essence of inconsistency between replicas. (2) In our experiments (Section 5), we find the length of the read queue is always less than 200 even when we use a read-intensive workload. Hence, the read queue is not suitable for observing the replica consistency.

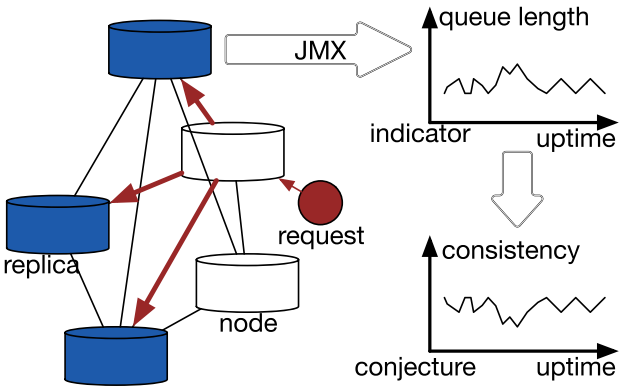


**FIGURE 2** The write queue as a real-time indicator of consistency

## 3.5 | System configurations and events that impact the consistency

Because a long write queue implies high inconsistency, in this subsection, we discuss what factors give rise to a long write queue. Tackling the bottlenecks of long write queues can help improve the replica consistency. Figure 3 shows the write process in Cassandra and its 3 potential bottlenecks (the green dashed boxes).

### 3.5.1 | Insufficient service capacity

Considering the write queue and the write threads as a queueing system, the queueing system arrives at stable state if and only if

$$\rho = \lambda/\varphi \leqslant 1, \tag{2}$$

where $\lambda$ is the enqueue speed of the write requests and $\varphi$ is the dequeue speed. Therefore, one of the immediate causes of a long write queue is that $\varphi$ is smaller than $\lambda$, which means that the node executes write operations slower than requested. We denote the bottleneck as the insufficient service capacity of the node. Solving the bottleneck can improve the degree of the consistency.

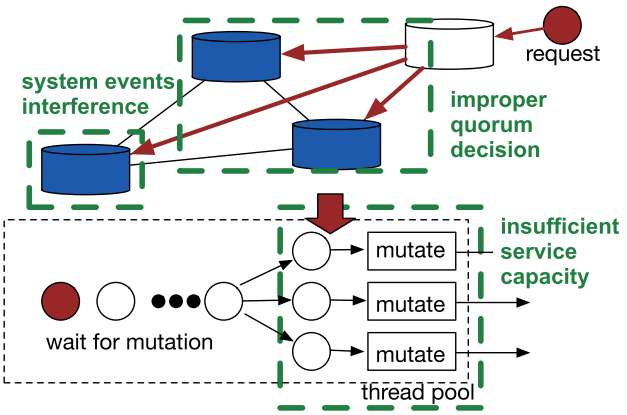


**FIGURE 3** The write process in Cassandra and potential bottlenecks

### 3.5.2 | Improper quorum decision

According to the discussion of Section 3.2, when the read your writes consistency anomaly occurs, at least $r$ nodes have not finished the write operation and only these nodes receive the propagated read operation. The phenomenon is triggered when users choose an improper quorum decision (read consistency level and write consistency level in Cassandra). We denote the bottleneck as the improper quorum decision. Solving the bottleneck can also improve the degree of the consistency.

### 3.5.3 | System events interference

We find that the queue length is not absolutely stable in our experiments. When the peaks of the queue length appear, the consistency anomalies will outbreak in a short time, which is undesirable for user experiences. There are 2 reasons that may lead to the queue length jitter. One is the query distribution changes, which depends on the application requirements and the other is that some internal events in Cassandra impact the departure speed of the write queue. We denote the latter as system events interference. Finding out what events interrupt the write process and alleviate the adverse effects can make the consistency more stable.

## 3.6 | Methods for improving consistency

The first 2 bottlenecks lead to a long queue length and the third one incurs the jitter of the queue length. In this section, we firstly propose 2 methods for controlling the queue length. When we consider the write queue in 1 node, we adjust its service capacity to shorten the queue length. When we consider the whole cluster, we can control the request propagation among nodes to decrease the queue length. Then, we investigate what internal events of Cassandra affect the stabilization of the queue and reduce that impact.

### 3.6.1 | Changing the write thread number

The first method is to control the write queue of each node. If we enhance the service capacity of the node (increase dequeue speed $\varphi$), the sojourn time ($W$) of write operations will be reduced. According to Little's law, queue length $L = \lambda W$, hence $L$ will decrease accordingly. For a queueing system, we can either improve the capacity of each counter or add more counters to increase $\varphi$.

Because the counter is the write thread in Cassandra, improving the capacity of the counters means improving the capacity of the write threads. To achieve that, we have to scale up the cluster by upgrading the hardware especially CPU specification, because the CPU performance decides the concurrency capacity and the speed of the threads. Upgrading hardware is a trivial solution that increases the budget. Hence, adding more counters is more sensible.

Adding counters means that increasing the write threads in the thread pool. Fortunately, Cassandra use the SEDA architecture so that modifying the thread number is easy to achieve. Cassandra has a configuration parameter `concurrent_writes`. It represents the number of write threads in 1 node. We can tune this parameter to control the queue length henceforth tuning the consistency.

However, as a limited resource, write threads cannot be increased unlimitedly. Give a hardware specification, how many threads should we assign is empirically discussed in the experimental study.

There is another way to reduce $\varphi$: speeding up each counter by improving the write implementation locally. In Cassandra, when a write thread begins to process a write operation, it firstly writes data into the commitlog, and then writes data into the memtable. Memtable will be flushed on disk in background. To speedup each counter, we have to modify the above process. For example, sending acknowledgment immediately once the server writes data into the commitlog. After that, the server writes data into the memtable in background. However, the above improvement violates our original purpose: improving the consistency without the necessity of modifying the system implementation.

### 3.6.2 | Changing the quorum in fine-grained

Currently, Cassandra uses the sum of read and write consistency level, ie, $w + r$ to tune the quorum size. The common insight is that increasing $w + r$ can improve the consistency. Some works focus on discussing the relation between $w + r$ and the consistency. For example, Probabilistically Bounded Staleness (PBS)[6] predicts the consistency according to $w$, $r$, and operation time cost.

However, increasing $w + r$ brings high read and write latencies. What we consider is that if we keep $w + r$ constant, whether changing $w$ and $r$ impacts the consistency. To date, no previous works have discussed this problem theoretically.

Considering an operation sequence: *Write(a)* and *Read(a)*. To simplify the running example, we suppose only $w$ of $n$ nodes finish the write operation. According to the discussion in Section 3.2, if the operation sequence satisfies read your writes consistency, the read operation must be executed on at least one of the $w$ nodes. If $n = 4$ and $(r, w) = (1, 3)$, the probability is $\frac{C_3^1}{C_4^1} = \frac{3}{4}$. If $(r, w) = (3, 1)$, the probability is $\frac{C_3^2}{C_4^3} = \frac{3}{4}$. If $(r, w) = (2, 2)$, there are 2 cases: the first is reading 1 replica from the $w$ nodes and reading another from the $n - w$ nodes. The second is reading 2 replicas from the $w$ nodes. So the probability is $\frac{(C_2^1}{C_2^1+C_2^2)}C_4^2 = \frac{5}{6}$. In all, the read your writes consistency intensity when $(r, w) = (2, 2)$ is better than $(1, 3)$ or $(3, 1)$. Therefore, we conjecture that the better policy is that let $w - r \leqslant 1$ in this case.

**Theorem 1.** In a quorum-based system, if $r + w < n$ and $r + w$ is fixed constant, then the optimal choice is to let $w - r \leqslant 1$.

*Proof.* Suppose $r + w = k$, $k$ is constant and $r, w \in [1, k)$. To simplify the problem, we suppose only $w$ of $n$ nodes finish the write operation. As discussed above, if the operation sequence satisfies read your writes consistency, the read operation must be executed on at least one of the $w$ nodes. Mark the probability of this case as $f(r)$. If $w + r > n$, then $f(r) = 1$. Otherwise

$$f(r) = 1 - \frac{C_{n-w}^{r}}{C_{n}^{r}} = 1 - \frac{(n-w)!(n-r)!}{n!(n-w-r)!} \triangleq 1 - g(r).$$

If we increase $r$ and want to increase the probability $f(r)$, then

$$\begin{aligned} f(r+1) > f(r) &\Rightarrow g(r+1) < g(r) \\ &\Rightarrow g(r+1)/g(r) < 1 \\ &\Rightarrow \frac{n-w+1}{n-r} < 1 \\ &\Rightarrow w - r > 1. \end{aligned}$$

So $w - r = 1$ is the inflexion point of $f(r)$. we can increase $r$ until $w - r \leqslant 1$ to increase the probability. Besides, both of $w$ and $r$ are integers. Therefore, if $k \bmod 2 = 0$, $r = w = k/2$ is the best choice. Otherwise $r = \lfloor k/2 \rfloor \pm 1$ is the best choice. □

#### 3.6.3 | Stabilizing the consistency

We investigate when the query distribution keeps constant, what internal events incur the jitter of the queue length by analyzing the Cassandra logs. Based on this analysis, we can tweak the relevant parameters to mitigate the adverse effects.

The memtable flushing event. One of the most frequent events in the Cassandra log is flushing a memtable into the disk. Memtable is a memory data structure in Cassandra for storing data, which will be flushed into the disk if it spends too much memory. At this time, Cassandra will generate a new memtable to store data and then flush the old one asynchronously. Switching the memtables may lead to redo of some write requests because Cassandra insures row-level atomicity for write operations, which means either all the columns in a row are written into the memtable or none of them is written successfully. Besides, flushing the memtable into the disk has I/O cost. According to our experiments in Section 6.3, if we decrease the duration for a memtable flushing event, the peak of the queue can be eliminated to some extent.

Other events. Another frequent event in Cassandra log is JVM GC (garbage collection) event. The impact of GC is discussed in Fan et al.[20] In our experiment in Section 6.3, Cassandra executes GC periodically, and the impact of the normal GC event is negligible for the session guarantee consistency.

However, the phenomenon of "pause"[20] in Cassandra exists. The phenomenon is caused by a self-inspection function, which is called as StatusLogger in Cassandra. Many scenarios can trigger the function and one of them is a long configurable time GC, while we conjecture these GC events are the things discussed in Fan et al.[20] Therefore, GC is not a special event for us. Besides, StatusLogger just counts the statistical information of the system, prints them into the console or log files and we can disable it without any adverse effect. Therefore, we do not consider the event of GC in this paper.

## 4 | EXPERIMENT SETUP

First, we design experiments to verify whether the queue length can be a good indicator of the consistency. In this section, we observe the relation between them in different workloads, such as different throughput, concurrent client number, and query distribution. Read-intensive, write-intensive, and mix workload are covered in the experiments. Second, we verify whether the first 2 methods are effective for improving the consistency. The adverse effects (read and write latency) are also considered in these experiments. Third, we verify the third method by analyzing why the queue jitters and discuss how to configure Cassandra to alleviate the wave. Then, we give an example to combine all the 3 methods to tune the consistency in Cassandra. In the paper, we design experiments to compare our conclusion with 2 latest related works, which include consistency prediction model PBS[6] and an explanation for atomicity consistency model in Cassandra.[20]

### 4.1 | Heavy workload setup

When we reproduce the experiment in Wada et al,[5] we find session guarantee anomalies exist in Cassandra, but the number of anomalies is few (0.03%~1%). We conjecture the workload in Wada et al[5] is so light that all the replicas are consistent before the read operation comes.

To observe the consistency anomaly more clearly, we design a heavy workload for Cassandra. We classify the clients as 2 classes: the first class is called as pressure client, which is a normal Yahoo! Cloud Serving Benchmarking (YCSB)[21] client and sends read/write request quickly. The second class is called as observer client, which sends particular requests to the cluster and records all the results to judge whether the read or write operation violates the consistency models. We design the workload with 3 reasons:

a. (1) This workload simulates the scenario of session guarantee well. Session guarantee focuses on the feeling of 1 session. We use the observer client to observe the feeling of 1 session. Besides, it is impossible that only 1 user use the whole cluster. Therefore, we run many pressure clients to provide background workloads.
b. (2) This workload is suitable for analysis. If all the clients are observer clients, the whole workload is out of control because controlling the write speed of the observer client is more complex than normal YCSB clients. Besides, too many observation behaviors may impact the results like the observer effect of Schrodinger's Cat.
c. (3) Heavy workload is more suitable for benchmarking NoSQL systems because many applications have heavily write workloads. For example, the peak write speed may be 100 000 records/s in China Sany Group's time serial application in our practice.

To cover all the types of workload (read intensive, write intensive, and mix workload), we change the read and write ratios of the pressure clients: 0% write and 100% read, 25% write and 75% read, 50% write and 50% read, 75% read and 25% write, and 100% write and 0% read.

The observer client has special behaviors. After making a decision about whether the operation violates some consistency models, the client triggers a strong consistent write operation to make sure all the replicas have been updated to the latest status. For example, if the observer client observes a read your writes consistency anomaly, it sends a write operation with Consistency ALL level declaration in Cassandra. If the observer client does not observe any anomaly, it also sends a strong consistency write operation because observing none anomaly does not mean that all the replicas are consistent. A strong consistency write can eliminate the cumulative anomalies. To our knowledge, the behavior of the observer client is similar with the method in Bermbach and Tai.[22]

When we experiment with the heavy workload, we make some other settings. First, we disable the read repair mechanism in Cassandra like what Bailis et al[6] does in their experiments. Read repair, which acts like an additional write for each read who reads old data, is an additional remedial mechanism to improve the consistency level. Second, the write latency may be very high with the heavy workload. Therefore, we set the operation time-out time as 100 seconds to avoid the time-outs.

Under this workload, we can find about 1%~ 50% operations violate the session guarantee consistency.

## 4.2 | Experiment environment

To make the experiments more reliable, we run our experiments on different clusters: local cluster and cluster in the cloud.

Local cluster: We use Fujitsu Primergy RX600 S6, which has 4 × 10 cores processors, 512GB memory and 8TB disk spaces, as our physical server. The physical server provides tens of virtual machines by VMware vSphere software. In this environment, we can change the specification of virtual machines easily. For example, in our experiments, we modify the CPU cores of these virtual machines from 1 core to 4 cores to observe the impact of different hardware performances. Both of our Cassandra cluster and workload clients are running on the physical server. To screen unknown factors, there is no other virtual machines on the physical server.

Cluster in the cloud: We deploy the Cassandra cluster and workload clients on Amazon EC2. Amazon EC2 is another way to provide a pay-as-you-go virtualization solution. We use 2 types of specifications virtual machines in our experiments. The first is *t2.small*, which has 1 virtual CPU and 2GB memory. The second is *t2.medium*, which has 2 virtual CPUs and 4GB memory. The price of the latter is twice of the former.

Table 1 summarizes the experiment environments. Most of the experiments run on Amazon EC2 cluster $C_5$. But when we consider the different network environment and CPU capacity, we use all of the 5 clusters.

It is no doubt that YCSB[21] has been a standard benchmark tool for NoSQLs. We use YCSB as the workload generator and performance reporter of the pressure client. In our experiments, the row key of data is a long integer and the value size of a column is 10 bytes. We set the YCSB maximal executing time as 500 seconds. For each experiment, we run 5 rounds and discard the result of the first round to avoid the problem of cold start. Then, we calculate the average throughput as the final results. Because official YCSB does not support Cassandra 2.0, we use Cassandra 1.2.11 in experiments.

**TABLE 1** Environment setup with different cluster specifications

| Cluster | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ |
|---|---|---|---|---|---|
| Virtualization | VMware sPhere | | | Amazon EC2(us-west-2) | |
| Physical server | Fujitsu Primergy RX600 S6 | | | - | |
| | 40 cores, 512GB memory | | | | |
| Network | Gigabit ethernet | | | Low to moderate | |
| Instance | 2GHz (CPU) | | | t2.small | t2.medium |
| CPU Cors | 1 | 2 | 4 | 1 | 2 |
| Memory | 8GB | 8GB | 8GB | 2GB | 4GB |
| Disk | 200GB SATA with RAID0 | | | 40GB SSD | |
| Cluster size | 5 nodes with Cassandra 1.2.11 | | | | |
| YCSB clients | Up to 6 nodes with YCSB 0.1.4 | | | | |

Because monotonic writes consistency and write follows reads consistency are invisible in Cassandra, additional information should be collected for observing the 2 anomalies. Therefore, we add additional logs when a write operation dequeues and is finished.

# 5 | QUEUE LENGTH AS AN INDICATOR OF CONSISTENCY

## 5.1 | The relation between the queue and the consistency

To show the effectiveness of using the queue length as an indicator in real time, we collect the queue length and occurrence time of the consistency anomalies in real time.

Figure 4 describes an instance of Figure 2. It shows the length of the write queue on 1 node per 1 ms and the occurrence time of the consistency anomalies when the query distribution is uniform. The red line shows the changes of the queue of 1 node in the cluster. The length of the queue fluctuates sometimes in the figure. After collecting the CPU utilization of the Cassandra process and plotting the result, we can see when the CPU utilization changes, the length of the queue waves.

The bars show how many consistency anomalies occur in 10 seconds. We can find that the consistency anomalies assemble when the instantaneous length of the queue is long. The result obeys our conjecture about the relation of the length of the queue and the consistency.

Therefore, we can use the queue length to observe the consistency in real time. When the query distribution is uniform, observe any 1 node is adequate. But if the query distribution is skewed, we should monitor

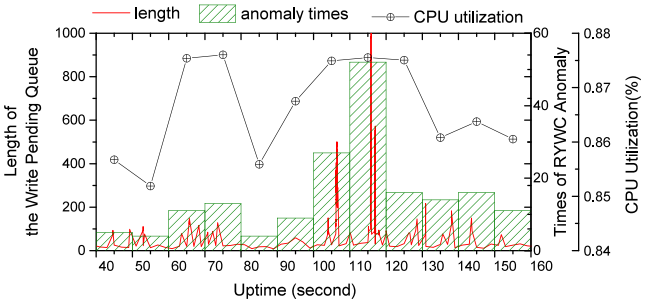


**FIGURE 4** The relationship between the change of the queue length and the number of read your writes consistency anomalies

all the nodes because any node has a long queue may leads to serious inconsistency. We will show the result in Section 5.2.

In our experiments, read your writes consistency anomalies are frequent while other consistency anomalies are rare (1%~ 5%). Therefore, we mainly discuss the relation between the queue and read your writes consistency model in the paper.

## 5.2 | Generalizability of the relation between the queue and the consistency

To prove the generalizability of the conjecture about the relation between the queue and the consistency, we reproduce the above experiment with different workloads, such as different throughput, different read/write ratio and query distribution skewness scenario, in different clusters.

In this section, we draw the average length of the queue of all the nodes and the consistency intensity for each experiment when the data distribution is uniform. Otherwise, we draw the average lengths of the queues in different nodes.

### 5.2.1 | Concurrent client number and throughput

In this experiment, we change the throughput from 47 500 to 60 000 per second and the concurrent client number from 1500 to 2500. The replica number is 3 and read/write consistency levels are 1. The data distribution and client requests are uniform. To eliminate the network's effect and show the relation between the queue length and the consistency intensity on different hardwares, we run the experiment on both vMware vSphere (Cluster $C_2$) and Amazon EC2 environment (Cluster $C_5$).

Figure 5 shows the results. The intensity of read your writes consistency decreases along with the increment of the throughput and the number of the concurrent client. At the same time, the length of the queue increases. The results prove that the intensity of the consistency is negatively correlated to the length of the queue.

### 5.2.2 | Different read and write ratios

In the experiments above, we use write-intensive workload, which the write/read ratio is 1:0. To prove the generalizability of the conjecture, we change the write/read ratio in the following experiments.

Figure 6 shows the relation between the intensity of the consistency and the queue length with a mix-intensive workload, which the read/write ratio is 1:1. The experiment also proves that when the write queue length increases, the consistency intensity decreases. Besides, in this experiment, the lengths of the read queue are always less than 200. Comparing with the length of the write queue, the change of the read queue is smaller so that it is inappropriate as an indicator of the consistency intensity. The result matches the discussion in Section 3.4.

However, the results above may mislead us that the throughput can also indicate the consistency. If so, using the length of queue as an

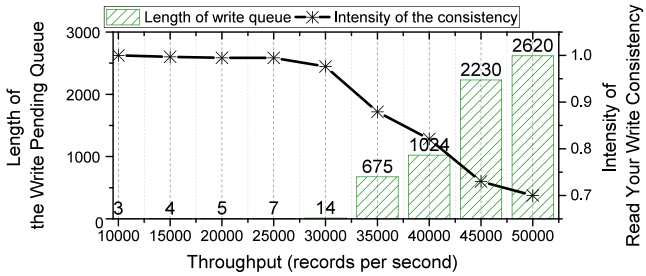


**FIGURE 6** The relation between the queue length and consistency in mix-intensive workload (50% read and 50% write). The consistency is also negatively correlated to the queue length in this case

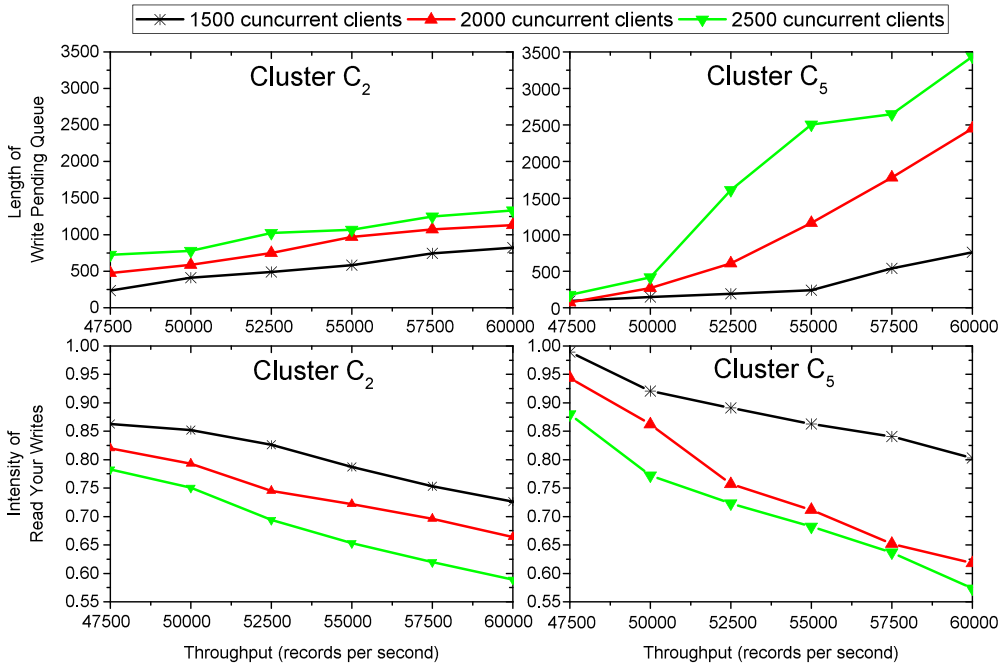


**FIGURE 5** The relation between queue length and consistency intensity with different throughputs and concurrent client numbers in different environment. When the length of write queue increases, the intensity of the consistency decreases

indicator of the consistency is needless. In fact, they cannot indicate the consistency correctly. To prove that, we design another experiment on Amazon EC2 $C_5$. In this experiment, we change the read/write ratio and observe the queue length and throughput. Figure 7 shows the results. In this experiment, when the write ratio increases, the intensity of the consistency decreases and the length of the write pending queue becomes longer. However, the throughput decreases at the same time. The phenomenon to conflict the conjecture that increasing the throughput will decrease the consistency intensity. Therefore, we cannot judge the consistency just according to the throughput. Second, we explain why the throughput decreases when the write ratio increases. In a quorum data storage system with $n$ replicas, the write operations need to be propagated $n$ times and the read operations need only $r$ times ($r \leqslant n$). Suppose a client sends $k$ requests, which includes $a$ writes and $k - a$ reads. The cluster needs to process $n \times a + r \times (k - a)$ tasks. If $a$ increases, the cluster will need to do more tasks. That is why the throughput decreases.

### 5.2.3 | Query distribution skewness

All the above show the relation between the write pending queue and the consistency when the data distribution and client requests are uniform. We design new experiments to investigate the length of queue and the consistency when the data distribution and requests are not uniform. In this experiment, we generate data and operations by power law distribution (20% data items are involved in 80% operations). Then, we calculate the queue length of the coordinator and the average length of the noncoordinators. Figure 8 shows the lengths of the queues for imbalanced data distribution and the consistency intensity. Comparing with the green and the carmine bars, we can find that when the data

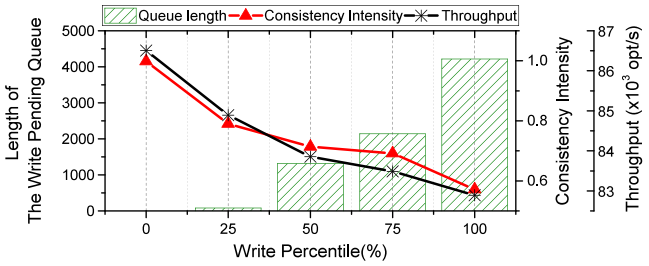


**FIGURE 7** The counterexample of the relation that the consistency is negatively correlated to the throughput. Therefore, throughput cannot reflect the consistency intensity

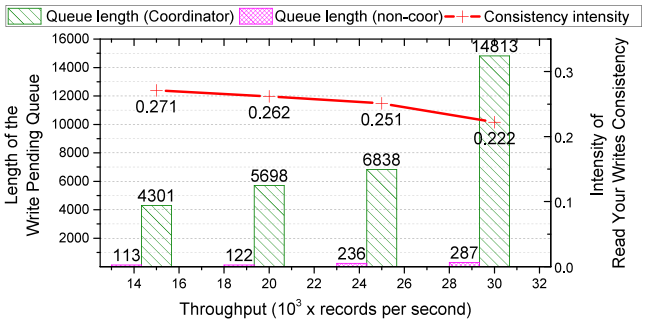


**FIGURE 8** The consistency intensity when the query distribution is skew (power law distribution). The consistency is negatively correlated to the queue length in this case and the system is heavily inconsistent

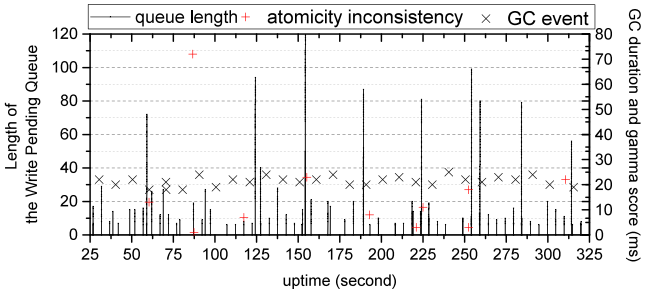


**FIGURE 9** Queue length and the atomicity consistency model with $\Gamma$ metric. When the peak of the queue length appears, there is high probability to find out the atomicity inconsistency

distribution is imbalanced, the queue length difference between the coordinator and noncoordinator is large. At this time, the intensity of the consistency is low to less than 0.3 (the line with +). The experiment shows that when the query distribution is skew, the conjecture of that the consistency intensity is negatively correlated with the queue length is still true. Besides, when the query distribution is skew, the system is heavily inconsistent.

### 5.2.4 | Relation with other consistency models

All the experiments above show the queue length can indicate the session guarantee consistency model well. Actually, because the write queue reveals the essence of the inconsistency—a long queue length prevents the global ordering of the requests in the system; we conjecture that the write queue can also explain other consistency models. For example, Hua Fan et al measure the atomicity consistency[13] with $\Gamma$ metric and explain the phenomenon with JVM GC pause.[20] We make a comparison with the work to investigate that whether the queue can explain the atomicity consistency.

Figure 9 shows the reproduction of Fan et al.[20] The read cross "+" represents the atomicity consistency anomaly and the black "x" refers to the JVM GC. The figure shows that when the atomicity consistency anomaly occurs, the length of the queue is long. Therefore, the queue can explain the atomicity consistency with $\Gamma$ metric well. However, many GC events do not lead to atomicity inconsistency. Therefore, we think the write queue is more suitable than GC events for explaining the consistency.

All the experiments in this section show that queue length is a good indicator of the consistency intensity. Therefore, if the queue lengths of some nodes in the cluster are high, there will be many consistency anomalies.

## 6 | IMPROVEMENT AND STABILIZATION OF THE CONSISTENCY

In Section 3.6, we propose 2 methods to improve the consistency. One is enhancing the service capacity of nodes in the cluster by adding the write thread number, the other is change the quorum in fine grained. In Section 3.6.3, we make a conjecture that reduce the memtable size to stabilize the write queue. In this section, we verify the effectiveness of these methods.

## 6.1 | Changing the write thread number

We conjecture adding the write thread number can improve the consistency. However, a quantitative conclusion is not proposed. We will make an empirically quantitative conclusion in this section.

### 6.1.1 | Effectiveness verification

First, we show how the thread number changes impacts the queue length. We monitor the queue length in real time on cluster $C_2$. In this experiment, we increase the enqueue speed $\lambda$ by adding concurrent clients from 0 to 2500 until the queue arrives to the steady state. Then, we change the thread number to modify the dequeue speed $\varphi$. As discussed in Section 3.6.1, the modification of the $\varphi$ is implemented by changing the parameter of concurrent writes in Cassandra. The modification of concurrent writes is dynamic by JMX technology according to modifying the core_thread_number of the write thread pool.

Figure 10 shows the result. At first, the queue length is short because $\lambda$ is small. When the number of the concurrent clients reaches to 2500, the queue length is out of the steady state and the queue length increases slowly. Then, we add the write thread number from 2 to 4, the queue arrives to the steady state again and the length keeps as about 500. Then, we continue to increase the thread number from 4 to 8 and 16, the queue length stops the increment but begins to wave. After we set the thread number to 4 again, the queue length keeps stable again. The experiment proves that thread number can control the queue length well and there is a best value for the thread number. The number of the CPU cores of the cluster $C_2$ is 2, so that the best thread number is the twice of the CPU cores in this scenario.

Besides, we can find that when we increase the thread number, the queue increases quickly (250 s, 350 s, and 450 s in the figure). It is caused by the dynamic modification by JMX. Dynamic increasing the thread pool size leads to some exception in the thread pool. However, the thread pool can recover autonomously.

### 6.1.2 | Generalizability of the quantitative conclusion of the thread number

The experiment above shows that the best thread number is the twice of the CPU cores. In this section, we design experiments to modify the write thread number in different CPU specifications to investigate that whether the quantitative conclusion is suitable for other hardwares. In these experiments, we use cluster $C_1 \sim C_5$ to observe the behavior of different thread number and CPU specifications. In the local environment, we change the cluster instances from 1 CPU core to 4 cores (Cluster $C_1$ to $C_3$). In Amazon EC2 environment, we change the cluster instances from *t2.small* to *t2.medium* (Cluster $C_4$ and $C_5$). We set the replica number as 3 and the read/write consistency level as 1. The concurrent client is 2500.

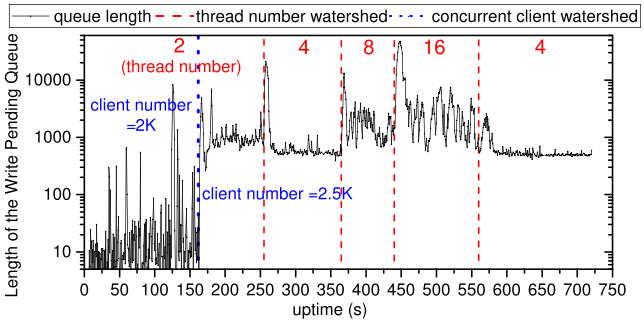


**FIGURE 10** The changes of the queue length in real time on Cluster $C_2$. When the concurrent client number is less than 2500, the service capacity of the node is enough. The node cannot handle so many requests when the client number is equal with 2500. When the thread number is equal with 2 × the number of CPU cores ($4 = 2 \times 2$), the node can provide the maximal service capacity

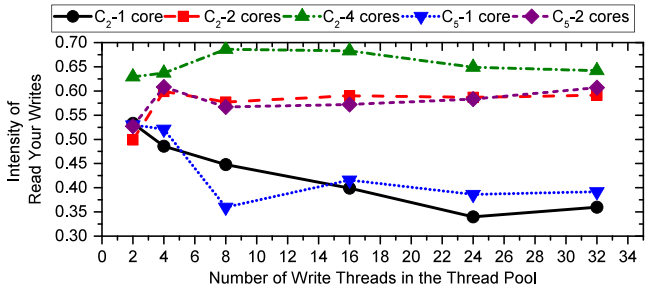


**FIGURE 11** The consistency intensity and length of queue with different thread numbers and CPU specifications. In different hardwares, the best thread numbers are twice of the CPU cores

Read your writes consistency. Figure 11 shows the results of the read your writes consistency intensity and the length of the write pending queue. When the write thread number keeps constant, a better CPU specification generally brings better read your writes consistency intensity. But if the write thread number is not appropriate, we can get some opposite results. For example, when the number of the write thread is 2, the read your writes consistency intensity of the cluster $C_2$ whose node has 2 cores is worse than $C_1$ and $C_4$ whose node has 1 core.

For a particular CPU specification (except 1 core), the read your writes consistency intensity increases first and then decreases when the concurrent write thread increases. We notice that the peak of the intensity always occurs when the number of the write threads is 2 × *cores*. For example, when the CPU has 2 cores, the system gets the highest intensity when the number is 4 and when the CPU has 4 cores, the system gets the highest intensity when the number is 8. This principle stands in all the 5 clusters. We believe the cluster with 1 core instances also obey the principle because at this time, the number of the thread as 2 is the best. Unfortunately, we cannot verify the result when the number of the thread is 1 because Cassandra limits the number greater than 1.

Other consistency models. According to the cause of the monotonic writes and the write follows reads anomaly, the write thread number is the most important factors. If there is only 1 write thread, because Cassandra uses FIFO policy for the write pending queue, the latter write operation never dequeues earlier than the previous operations. But if there are more than 1 write thread, the latter write operation has opportunities to be executed in parallel with the previous operations and then be finished earlier than the previous operations.

Figure 12 shows the results and verifies our conjecture. The intensity of monotonic writes and write follows reads decrease along

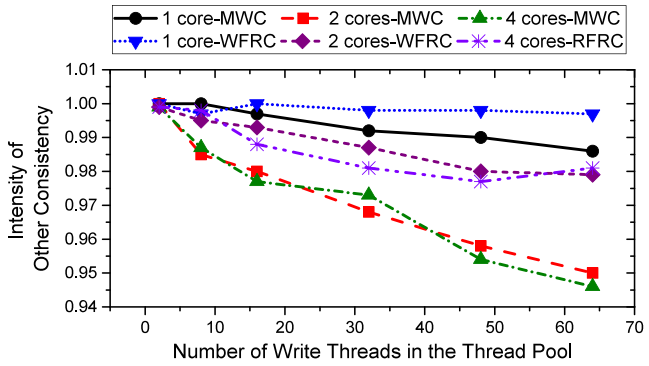


**FIGURE 12** The other consistency intensities and thread number. The consistency intensity decreases along with the increment of the thread number

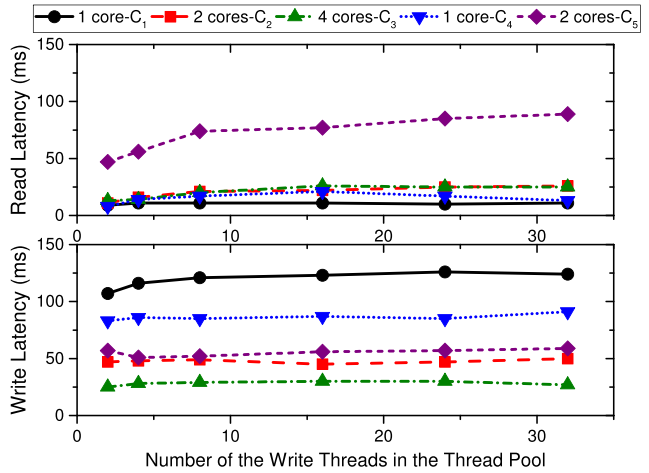


**FIGURE 13** Read and write latency with different thread numbers and CPU specifications. Increasing the write thread number brings many read latencies while adding the write latency little

with the increment of write thread number. Besides, we notice that when the nodes in the cluster have only 1 core per node, the intensity of monotonic writes decreases more slowly than the nodes with 2 or 4 cores. We conjecture the reason is that the concurrent capacity for a single core is weaker than multicores. However, we should point out that though the downtrend is obvious, the difference is little. For example, the maximum difference comes from the *4 cores-MWC* scenario and the difference is less than 6%.

### 6.1.3 | Adverse effect of changing the thread number

As one of the limited resources, the more of threads does not mean the better. Imbalanced resource allocation leads to other modules lack of the computing resources. That is why increasing the write thread number will bring some adverse effects.

Figure 13 shows the write latency and read latency of the experiment in Section 6.1.2. The write latency increases little along with the increment of the write thread number. However, the read latency increases substantially. To analyze why the read latency increases, we collect the detailed information of CPU under each number of write threads by using "dstat" command on Ubuntu. First, in all the rounds of the experiment, the utilization rates of CPU are all about 90%. Second, when the number of write threads are 2, 4, 8, 16, 24, and 32, there are 21.5K, 22.9K, 23.3K, 24.4K, 25K, and 25.4 K system interruption events in 1 second. Therefore, we conjecture that it is because the write module of Cassandra takes so many threads that the read module does not have enough threads. As a result, many thread switch events occur and thereby they increase the latency.

## 6.2 | Optimizing the quorum in fine grained

We have proved that if we keep the sum of $r + w$ constant and $r + w \leqslant n$, a better policy is that let $|r - w| < 1$ in Section 3.6.2. In this section, we investigate how different write and read consistency level impact the consistency by experiments first. Second. we analyze the adverse effects of read/write consistency level. The experiment runs on Amazon EC2 cluster $C_5$. 5 YCSB clients act as pressure clients, and they establish 2500 concurrent connections with the cluster. We change the replica number from 2 to 4. We also compare our results with PBS[6] to show the conclusion is suitable for more scenarios.

### 6.2.1 | Effectiveness verification

The Figure 14 shows the result of read your writes consistency. When $r + w < n$, e.g., $n = 2$ and $(r, w)$ is $(1, 2)$, $(2, 1)$ or $(2, 2)$ or $n = 3$ and $(r, w) = (2, 2)$, the intensity of read your writes consistency is 100% all the time. We omit these result in the figure for the conciseness. But when $w+r < n$, the intensity of read your writes consistency is less than 100% (see $n = 4$ and $(r, w) = (1, 1), (1, 2), (2, 1), (3, 1)$ or $(1, 3)$). Besides, the intensity of read your writes consistency increases along with the sum of $r + w$. All of these observations obey common insights and are reasonable. When user uses strong consistency level ($w + r > n$), the scenario does not satisfy the condition we discuss in Section 3.2 and we cannot find read your writes anomaly. When $w + r < n$, the smaller the $w+r$ is, the easier the condition of read your writes consistency anomaly is satisfied.

Now, we analyze the consistency difference when we keep $w+r$ constant. In Figure 14, the pink bars show the read your writes consistency intensity when the replica number is 4. The read your writes consistency intensity when $(r, w) = (2, 2)$ is higher than the intensity when $(r, w) = (3, 1)$ or $(1, 3)$. The result is coincident with the conclusion in Section 3.6.2.

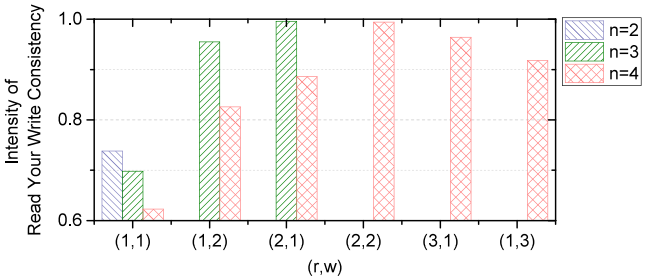


**FIGURE 14** Intensity of read your writes consistency with different quorum size and replica number (the cases of $r + w > n$ are omitted). If $r + w \leqslant n$ and $r + w$ keeps constant, the consistency are the best when $w - r \leqslant 1$

### 6.2.2 | Generalizability of the quorum decision

The experiment above only shows that when $n = 4$, the conclusion that $(r, w) = 2$ is the best. In this section, we will use PBS to simulate more cases of replica number $n$, read consistency level $r$, and write consistency level $w$ to prove the conclusion.

Bailis et al proposed PBS to predict the consistency.[6] They believed the quorum size impacts the consistency significantly and given a formula to calculate the probability of k-staleness.[6] But they did not discuss the relation between $r$ and $w$ when $r + w$ keeps constant. We will use the formula to simulate more scenarios to show whether our conclusion is right.

Because 1-staleness in PBS is similar with read your writes consistency, we use PBS to simulate the probabilities of 1-staleness with different quorum ($r$, $w$, and $n$). Figure 15 shows the simulation. We can find that if $r+w$ keeps constant, the system can get the best consistency when $w - r \leqslant 1$. For example, when $n = 6$ and $r + w = 4$, $(r, w) = (2, 2)$ is the best. When $n = 6$ and $r+w = 6$, $(r, w) = (3, 3)$ is the best. The simulations show our conclusion is suitable for more scenarios and coincident with PBS.

### 6.2.3 | The adverse effect of changing the quorum

The common insight is that increasing $r + w$ will decrease the throughput. However, Figure 16 shows another interesting phenomenons about quorum and throughput: When we keep the quorum size $r + w$ constant, the throughput differs. For example, the throughput with $(r, w) = (1, 3)$ is higher than the throughput with $(r, w) = (2, 2)$ and more than twice of that with $(r, w) = (3, 1)$. Besides, when the replica number $n$ and read consistency level $r$ keep constant, the throughput changes little along with the change of write consistency level $w$. For example, the throughputs when $(r, w) = (1, 1), (1, 2)$ are approximate. $(r, w) = (2, 1), (2, 2)$ show the same result.

According to the experiments, we make a conclusion that increasing the read consistency level will decrease the throughput sharply while increasing the write consistency level impacting the throughput little. We give an explanation: Write operation is propagated $n$ times no matter what $w$ is. $w$ only impacts the number of *ack* messages the coordinator receives. However, $r$ decides how many times the read operation will be propagated. Therefore, the cluster has to do more operations when $r$ increases. That is why in Figure 16, the throughput changes little when $n$ and $r$ are constant and the throughput decreases sharply when $r$ increases.

We omit the results of read and write latencies because they obey common insights: read (write) latency increases along with the increment of $r$ ($w$).

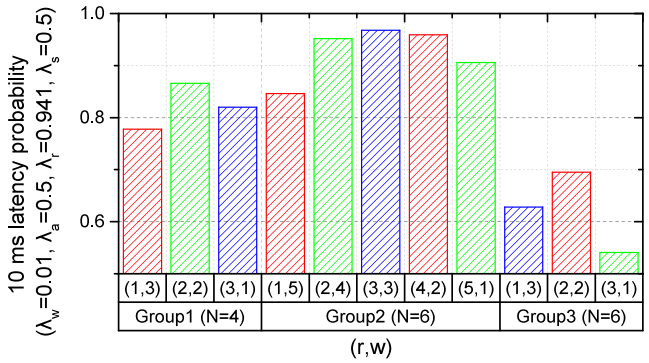


**FIGURE 15** One-staleness simulation of Probabilistically Bounded Staleness. All the 3 groups of the simulations match our conclusion that when $r + w$ keeps constant, $w - r \leqslant 1$ can get the best consistency intensity

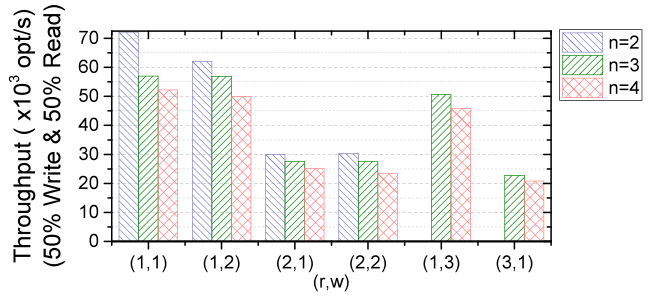


**FIGURE 16** Throughput with different quorum. Increasing the read consistency level will decrease the throughput sharply while increasing the write consistency level impacting the throughput little

## 6.3 | Stabilizing the write queue

Figure 4 has showed that the consistency anomalies assemble when the instantaneous length of the queue is long. For example, about 20% consistency anomalies occurs in (110 s and 120 s) in the figure. However, the skew distribution of the consistency is what we do not want. In this section, we collect the logs of Cassandra to analyze that what events disturb the stable of consistency and queue length. Then, we discuss how to make the queue length more stable.

### 6.3.1 | The JVM GC event

Figure 17 shows the length of the write queue, JVM GC and memtable flushing events in real time when we write data uniformly. We find that JVM GC is triggered periodically and does not impact the queue length distinctly. But when the memtable flushing event occurs, the queue length gets a peak. Therefore, in our experiments, we make a conclusion that JVM GC has no impact for the queue length while the memtable flushing event impacts the queue length obviously.

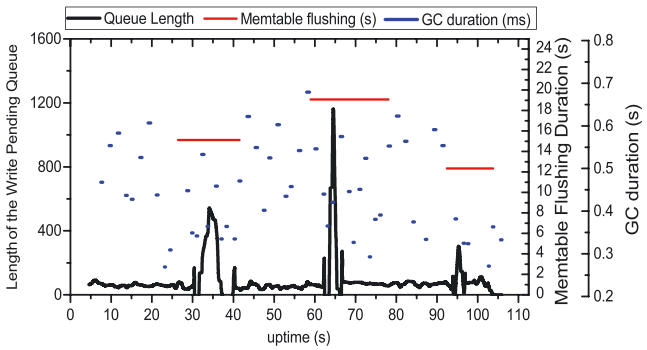


**FIGURE 17** Queue length and other events in realtime. The garbage collection event occurs periodically and impacts the queue length little. But the memtable flushing event impacts the queue obviously

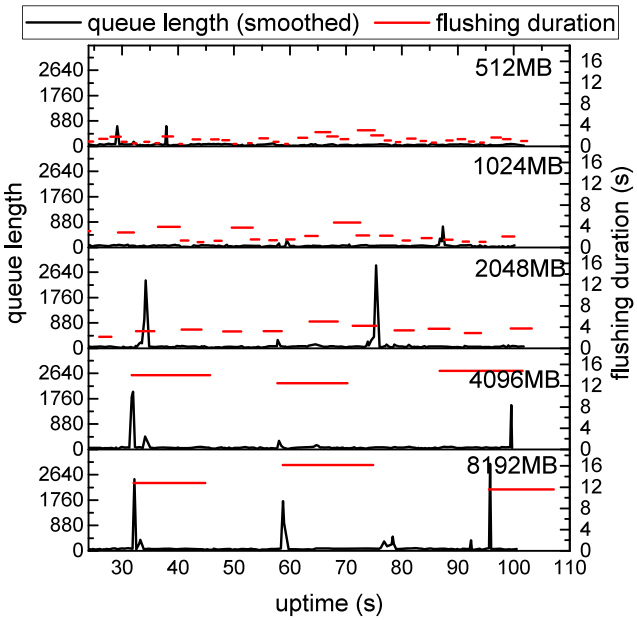


**FIGURE 18** Queue and memtable flushing with different memtable size parameter. Smaller memtable size makes the queue length more stable

### 6.3.2 | The memtable flushing event

The reason why the memtable flushing event impacts the queue length has been discussed in Section 3.6.3. To alleviate the jitter of the queue, we change the memtable_total_size_in_mb parameter of Cassandra, which means the total memory size the memtables can use. To get more experiments results and screen other unknown reasons, we use 5 physical servers, which have 32GB RAM instead of $C_1$ to $C_5$. In this experiment, we set the parameter memtable_total_size_in_mb as 512MB ~ 8GB.

Figure 18 shows the results. When the memtable size is 512MB, the duration of flushing a memtable is less than 1 second and only a part of the flushing events leads to the queue slightly waves. The duration of flushing a memtable increases along with the increment of the memtable size. At the same time, the queue has higher peaks than the case with 512MB.

Considering the results in Figure 4 and Figure 18 together, we can make a conclusion that the memtable_total_size_in_mb is not "the larger, the better," because the consistency anomalies will assemble nearby the peak of the queue. Therefore, an appropriate memtable_total_size_in_mb configuration can bring users a more uniform consistency experience. In our experiments in the clusters $C_1$ to $C_3$, which each node has 8GB memory, setting the memtable size as 512MB is a good choice.

Changing the value of memtable_total_size_in_mb not only impacts the consistency but also has some adverse effects. Figure 19 shows the results. According to the experiment, we find that a small memtable can reduce the average write latency. However, the impaction is little: the system just enhances about 1 ms when we double the memtable size.

Meanwhile, the size of each sstable is positively correlated with the parameter. Besides, small sstable size will lead to amount of sstable files. If there is no compaction mechanism, the large amount of sstable files will reduce the read speed. It is because (1) Data in 1 sstable is ordered and indexed, so that Cassandra reads data from 1 sstable is fast even though the size of the sstable is large; (2) For each read operation, Cassandra has to scan all the sstable files. The above analysis is corresponding to our experiment results: if there is only 2 sstable files, the read latency is about 28 ms. If there are about 100 sstables, the read latency is about 684 ms in our experiment. If the compaction mechanism is enabled, small sstable files may increase the frequency of compaction, which is a little harmful for hard disk. Fortunately, the compaction impacts the write throughput little because Cassandra limits the computation resources of the compaction.

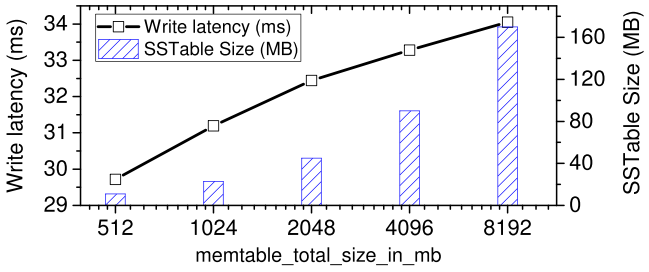


**FIGURE 19** Adverse effects of increasing the value of memtable_total_size_in_mb

## 6.4 | Methods summarization

In this section, we verify the 3 methods to tune and stabilize the consistency and investigate their adverse effects.

For the first method, we make the conclusions: (1) Setting the concurrent write thread number as twice of the CPU cores is an appropriate choice. (2) Increasing the write thread number is not free; the read latency increases while the write latency has little increment. Therefore, for a read latency intensive application, the method is not suitable.

According to the second method, we recommend some policies to determine how to set the value of read and write consistency level ($r$ and $w$). (1) Increasing $r+w$ improves the consistency of Cassandra while decreasing the throughput. (2) When we keep the sum of $r + w$ constant, we can get the best consistency if $w - r \leqslant 1$. (3) When we keep the sum of $r+w$ constant, a larger $r$ leads to a lower throughput. Therefore, for applications with high-throughput workload, increase the read consistency level $r$.

Conclusively, we summarize the first 2 methods in Table 2.

For the third method, we propose that a smaller memtable size is helpful to stabilize the write queue. In our experiments, 512MB is a better size if the memory size is less than 32GB.

We combine all the 3 methods together to show how to improve the consistency in Cassandra. We run the experiment on Cluster $C_3$. First, we set the total_memtable_size as 512MB. After measuring the read your writes consistency intensity, we set the concurrent_thread_number from 2 to 8. Finally, we set the read consistency level and write consistency level from $(r, w) = (1, 3)$ to $(2, 2)$. We run 5 rounds of the experiments and collect the consistency intensity, write latency, and read latency. Figure 20 shows the average values. After the 3 methods are used totally, the consistency intensity is the largest. The figure also shows that the adverse effect of changing the write threads

**TABLE 2** Summary of first 2 methods for tuning consistency

| Tuning Method | Best Choice for Consistency | Adverse Effect (trends of latency or throughput) |
|---|---|---|
| Increasing the write threads | number of write threads =2× CPU cores | read latency ↗ write latency ↦ |
| Increasing read consistency level (keeping $r + w$ constant) | $w - r \leqslant 1$ | throughput ↘ |

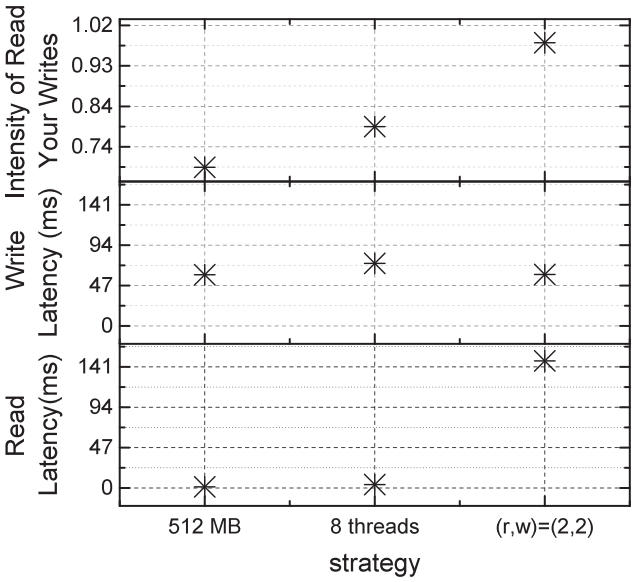


**FIGURE 20** Combining the 3 tuning methods in 1 cluster

can be eliminated by decreasing the write consistency level $w$. Besides, increasing the read consistency level $r$ takes many read latencies and it cannot be eliminated by other methods. The experiment proves that our improvements still obey PACELC criterion.[1]

## 7 | RELATED WORKS

Many theorems, such as Consistence, Availability and network Partitioning (CAP)[23] and PACELC[1] indicated that developers have to make many trade-offs when they design distributed systems. For example, the CAP theorem pointed out that a system has to give up one of the 3 features: consistency, availability, and partitioning. Cassandra allows users choosing the quorum size to tune the consistency. In this paper, we choose partial quorum, which is $w + r \leqslant n$, to observe the weak consistency. For another example, PACELC pointed out another scenario: if there is no network partitioning, users need to make a trade-off between consistency and latency. In this paper, we also discuss the changes of latencies when we tune the consistency.

When we say the terminology of consistency, it means the replica consistency model. Consistency model is defined as a contract between processes and the data store.[24] There are many kinds of consistency models, and they can be classified as data centric and client centric.[24] Data-centric consistency model defines that which ordering of read and write operations on shared, replicated data is right.[25] Lineariability,[26] sequential consistency,[27] causal consistency,[28] and FIFO consistency[29] (also called as PRAM consistency) belong to data-centric consistency model. Client-centric consistency model defines that what the data should be from a client view.[24] Therefore, client-centric consistency model is more intuitive for terminal users. In this paper, we focus on a kind of client-centric consistency model, which is called as session guarantees. Alvaro et al[7] also focused on the same consistency model. However, it just considered the measurement of the consistency, while our purpose is tuning and enhancing the consistency.

There are many ways to measure consistency. The first way is assertion. That is, if a system never violates a kind of consistency model, we say that the system supports that consistency. For example, Cassandra supports eventual consistency[30] and HBase supports timeline-consistency*. This metric is coarse because some applications just require systems guarantee a given consistency model in a high probability. Some works focus on analyzing the source code for inferring the consistency. For example, Jiang et al proposed code verification for synchronization,[31,32] which can be used for replica consistency analytics. The other way is using the probability to describe a system's consistency. For example, Facebook claimed that there are 0.0004% read operations returning different results in its linearizable system.[33] Bailis et al[6,34] proposed a probability model to measure and predict the consistency of quorum systems. The consistency intensity in this paper also belongs to this kind of measurement.

Another view to describe the consistency is using latency or data item version. For example, t-staleness and $\Delta$-consistency are proposed and predicted.[6] $\Gamma$ metric[13] is proposed to describe the atomicity consistency more accurately. Some explanation about $\Gamma$ is also proposed.[20] In this paper, we show that the write queue length can indicate the $\Gamma$ metric with atomicity consistency better.

There are many benchmarks for consistency measurement. For example, Bailis et al[35] presented a client-centric benchmark and Bermbach, Zhao, and Sakr[36] proposed a more comprehensive benchmark. Different from just measuring the consistency as above, our another purpose is to reveal the cause of inconsistency and help users tuning the consistency.

Many new protocols and methods are proposed for enhancing the consistency. For example, Google used atomic clock to implement strong consistency in globally data centers.[37] Bails et al proposed using a middle layer to support the causal consistency.[11] They implemented the consistency by storing all data's causal relationship in the middle layers' memory. Zhu et al proposed replica consistency-on-demand store[10] to provide on-demand consistency. The main idea is if the operation execution time has exceeded user's expectation, the system returns results (eg, null, stale version, or latest data) to the user immediately.

*https://www.cloudera.com/documentation/enterprise/5-4-x/topics/admin&uscore;hbase&uscore;read&uscore;replicas.html

Different from the above works, our purpose is for enhancing the consistency without the necessity of modifying the existing system implementation.

## 8 | SUMMARY AND DISCUSSION

In this section, we summarize the key insights from the results presented in this paper and discuss how the conclusions help developers and researchers to improve their works.

In this paper, we investigate the read and write process in Cassandra to analyze whether Cassandra satisfies four session guarantee consistencies: read your writes, monotonic reads, monotonic writes, and write follow reads consistency. By analyzing why the inconsistency occurs, we propose using the length of the write pending queue as an indicator of the consistency. Then, we propose how to improve the consistency: changing the thread number, configuring the quorum in fine grained, and set a small memtable size according to the memory size. We investigate the distribution of the consistency anomalies by analyzing the events in Cassandra logs, verify the effectiveness of these methods, and discuss the adverse effects about the methods. In the experiments, we compare our work with 2 latest related works, atomicity consistency model with $\Gamma$ metric and PBS prediction in the experiments. The comparisons show that the queue length is also suitable for atomicity consistency and our conclusion is coincident with PBS.

It is believable that our work is helpful for both of developers and researchers.

Developer. The observation of queue length is easy to be implemented by JMX technology. Therefore, developers can observe the current length of the queue to monitor the consistency in real time. When the workload of the application changes, developers can tune the consistency intensity to guarantee the user experiences by using proposed 3 methods. When using the methods, they can also make a trade-off between the consistency and latency or throughput. In a word, developers can configure the system better according to our conclusions in the paper.

Researcher. The paper proposes a new view to control the consistency. Researchers can analyze the system with queueing theory and design new consistency control methods according to the information of the queue length. For 1 example, we can limit the maximal length of the queue by simply blocking the following requests. For another example, we can use simple consistency control methods when the queue length is short. But when the queue length is long, we can use complicated control methods to guarantee the consistency.

### ACKNOWLEDGMENT

The paper is supported by NSFC (61325008), National Key R&D Program of China (2015BAF32B01) and Tsinghua National Laboratory (TNList) Key Project.

### REFERENCES

1. Abadi DJ. Consistency tradeoffs in modern distributed database system design. *IEEE Comput Mag*. 2012;45(2):37–42.
2. Bailis P, Ghodsi A. Eventual consistency today: limitations, extensions, and beyond. *Commun ACM*. 2013;56(5):55–63.
3. Bermbach D, Zhao L, Sakr S. Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services. *Performance Characterization and Benchmarking*: Springer International Publishing: Trento; 2014:32–47.
4. Bailis P, Venkataraman S, Franklin MJ, Hellerstein JM, Stoica I. Quantifying eventual consistency with PBS. *Commun ACM*. 2014;57(8):93–102.
5. Wada H, Fekete A, Zhao L, Lee K, Liu A. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. *Fifth Biennial CIDR*, Asilomar; 2011:134–143.
6. Bailis P, Venkataraman S, Franklin MJ, Hellerstein JM, Stoica I. Probabilistically bounded staleness for practical partial quorums. *Proc VLDB Endowment*. 2012;5(8):776–787.
7. Alvaro P, Conway N, Hellerstein J, Marczak WR. Consistency analysis in bloom: a CALM and collected approach. *CIDR 2011*, Asilomar; 2011:249–260.
8. Liu R, Aboulnaga A, Salem K. DAX: a widely distributed multitenant storage service for DBMS hosting. *Proc VLDB Endowment*. 2013;6(4):253–264.
9. Terry DB, Prabhakaran V, Kotla R, Balakrishnan M, Aguilera MK, Abu-Libdeh H. Consistency-based service level agreements for cloud storage. *Proceedings of the Twenty-Fourth ACM SOSP*, SOSP '13. ACM, New York, NY, USA; 2013:309–324.
10. Zhu Y, Yu PS, Wang J. RECODS: replica consistency-on-demand store. *29th IEEE ICDE, April 8-12, 2013*, Brisbane, Australia; 2013:1360–1363.
11. Bailis P, Ghodsi A, Hellerstein JM, Stoica I. Bolt-on causal consistency. *Proceedings of the ACM SIGMOD*, New York; 2013:761–772.
12. Wang X, Sun H, Deng T, Huai J. Consistency or latency? A quantitative analysis of replication systems based on replicated state machines. *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Budapest; 2013:1–12.
13. Golab WM, Rahman MR, AuYoung A, Keeton K, Gupta I. Client-centric benchmarking of eventual consistency for cloud storage systems. *IEEE 34th ICDCS*, Madrid; 2014:493–502.
14. Terry DB, Demers AJ, Petersen K, Spreitzer MJ, Theimer MM, Welch BB. Session guarantees for weakly consistent replicated data. *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, Austin; September 1994:140–149.
15. Zhu Y, Wang J. Client-centric consistency formalization and verification for system with large-scale distributed data storage. *Future Gener Comput Syst*. 2010;26(8):1180–1188.
16. Bermbach D, Kuhlenkamp J. Consistency in distributed storage systems. *Networked Systems*. Springer, Berlin Heidelberg; 2013:175–189.
17. Bernstein PA, Das S. Rethinking eventual consistency. *Proceedings of the ACM SIGMOD, June 22-27, 2013*, New York, NY, USA; 2013:923–928.
18. Gifford DK. Weighted voting for replicated data. *Proceedings of the Seventh ACM SOSP*, SOSP '79. ACM, New York, NY, USA; 1979:150–162.
19. Welsh M, Culler D, Brewer E. SEDA: an architecture for well-conditioned, scalable internet services. *Proceedings of the 8th ACM SOSP*, Banff; 2001:230–243.
20. Fan H, Ramaraju A, McKenzie M, Golab W, Wong B. Understanding the causes of consistency anomalies in Apache Cassandra. *Proc VLDB Endowment*. 2015;8(7):810–813.
21. Cooper BF, Silberstein A, Tam E, Ramakrishnan R, Sears R. Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM Symposium on Cloud Computing*, Indianapolis; 2010:143–154.
22. Bermbach D, Tai S. Eventual consistency: how soon is eventual? An evaluation of Amazon S3's consistency behavior. *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. ACM, Lisbon; 2011:1:1–1:6.
23. Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM Sigact News*. 2002;33(2):51–59.

24. Tanenbaum AS, Van Steen M. *Distributed Systems*. Upper Saddle River: Prentice-Hall; 2007.
25. Bermbach D, Kuhlenkamp J. Consistency in distributed storage systems. *Networked Systems*. Springer, Berlin; 2013:175–189.
26. Herlihy MP, Wing JM. Linearizability: a correctness condition for concurrent objects. *ACM Trans Program Languages Syst (TOPLAS)*. 1990;12(3):463–492.
27. Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans Comput*. 1979;100(9):690–691.
28. Ahamad M, Neiger G, Burns JE, Kohli P, Hutto PW. Causal memory: definitions, implementation, and programming. *Distrib Comput*. 1995;9(1):37–49.
29. Lipton RJ, Sandberg JS. *Pram: A Scalable Shared Memory*: Princeton University, Department of Computer Science, Princeton; 1988.
30. Lakshman A, Malik P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Syst Rev*. 2010;44(2):35–40.
31. Jiang Y, Zhang H, Li Z, et al. Design and optimization of multiclocked embedded systems using formal techniques. *IEEE Trans Ind Electron*. 2015;62(2):1270–1278.
32. Jiang Y, Zhang H, Zhang H, et al. Design of mixed synchronous /asynchronous systems with multiple clocks. *IEEE Trans Parallel Distrib Syst*. 2015;26(8):2220–2232.
33. Lu H, Veeraraghavan K, Ajoux P, et al. Existential consistency: measuring and understanding consistency at facebook. *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, Monterey; 2015:295–310.
34. Bailis P, Venkataraman S, Franklin MJ, Hellerstein JM, Stoica I. Quantifying eventual consistency with pbs. *The VLDB J*. 2014;23(2):279–302.
35. Rahman M, Golab W, AuYoung A, Keeton K, Wylie J. Toward a principled framework for benchmarking consistency. *HotDep (Workshop on Hot Topics in System Dependability)*, Hollywood; 2012:8–14.
36. Bermbach D, Zhao L, Sakr S. Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services. *Technology Conference on Performance Evaluation and Benchmarking*. Springer, Cham; 2013:32–47.
37. Corbett JC, Dean J, Epstein M, et al. Spanner: Googles globally distributed database. *ACM Trans Comput Syst (TOCS)*. 2013;31(3):8:1–8:22.


**How to cite this article:** Huang X, Wang J, Yu PS, Bai J, Zhang J. An experimental study on tuning the consistency of NoSQL systems. *Concurrency Computat: Pract Exper*. 2017;29:e4129. https://doi.org/10.1002/cpe.4129