# Research on A Tunable Consistency Strategy of Distributed Database

Chaozhi Yang
School of Information Science and Technology
Dalian Maritime University
Dalian, Liaoning, China 0411-84729544
Email: 18342208289@163.com

Tingting Cai
School of Information Science and Technology
Dalian Maritime University

Zhihuai Li
School of Information Science andTechnology
Dalian Maritime University
Dalian, Liaoning, China 0411-84727856
Email: qhlee@dlmu.edu.cn

***Abstract*—This paper designs a strategy of distributed databases with tunable consistency. This strategy can divide the consistency dynamically for each request. We add a request scheduling layer to schedule requests according to user's demand. The tunable consistency strategy can coordinate the consistency level dynamically and ensure the security of its data.**

## I. Introduction

In the context of massive data, the use of distributed database system has become an inevitable choice.The distributed database must ensure that its partition is fault tolerant, so the designer needs a trade-off between the usability and consistency of the entire system, so that the application on this platform achieves the desired level [1]. The requirements of the traditional strong consistency brought a problem of delayed increasing. Database services developed towards the weak consistency gradually. The demand of consistency for each request is different. So it is ideal to differentiate the consistency dynamically for every request. Such a distributed database system can meet the needs of consistency and reduce the delay. Cassandra database has achieved a division of consistency at the operational level [2]. Cassandra allows users to use different conformance.

This paper designs a strategy of distributed databases with tunable consistency [3]. This strategy can divide the consistency dynamically for each request. We add a request scheduling layer between interface layer and business logic layer to schedule requests according to user's demand. We introduced the replica factor number adaptive algorithm and stale read rate to calculate the consistency of every request.

The experimental results show that the tunable consistency strategy proposed in this paper can coordinate the consistency level dynamically and ensure the security of its data.

The paper is organized as follows. Section I introduces the tunable consistency strategy. Section II presents the experiment. Section III gives the experimental results. Section IV summarizes the conclusion.

## II. Tunable Consistency Strategy

### A. Overall design framework of the strategy

Commonly used systems divide business applications into three layers: User Interface Layer, Business Logic Layer, Data access Layer. In our strategy, we adds a request scheduling layer between the interface layer and the business logic layer. The request schedules layer added in the strategy are the

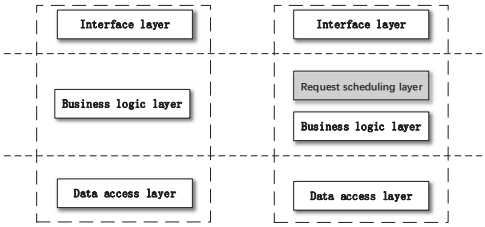



Fig. 1: The comparison of the structure before and after adding the tunable consistency strategy.

core architectures that achieve tunable consistency. The request scheduling layer is a consistent control layer and an implementation of an adaptive consistency strategy. This layer receives the data from the client and performs the consistency level processing. This layer processes the data from the business logic layer and outputs it according to the consistency level.

As shown in Figure 2, the request scheduling layer in this

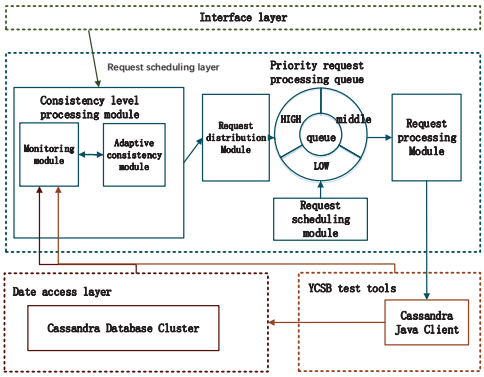



Fig. 2: System Module diagram

paper designed a consistency level processing module, request distribution module, priority request processing queue, request scheduling module and request processing module.

The consistency level processing module is divided into a monitoring module and an adaptive consistency module. The

monitoring module is responsible for collecting the relevant information from the Cassandra storage cluster through the Cassandra Node tool and passing the information to the adaptive consistency module. The adaptive consistency module uses the replica factor number adaptive algorithm. The adaptive consistency module calculates how many replications of the request are processed to achieve the requirement for consistency of the application, resulting in a system consistency level.

The request distribution module calculates the threshold for real-time high, medium, and low queues based on the consistency level of each request obtained by concurrency. And the received requests are placed into the three queues according to the threshold.

The priority request processing queue consists of three queues. These queues determine which requests are stored in the queue based on the working mechanism of the requesting distribution module. The requests in the queue are processed in the order of the high queue, the middle queue, and the low queue.

The request processing module reads the request from the three queues according to the priority and passes the request's corresponding consistency level to the Cassandra Java Client in the YCSB [4] module. While the module is responsible for passing the read data to the interface layer.

The Cassandra Java Client [5] is responsible for reading the number of replicationtions that are consistent with the request consistency level from the data access layer.

### *B. Realization of the strategy*

The general idea of the tunable consistency strategy can be shown as Alg.1

**Algorithm 1**: Tunable consistency strategy

```
1  QueHigh = newList();
2  QueMiddle = newList();
3  QueLow = newList();
4  X_n = Calconsistence(T_p, app_stale_rate, λ_r, λ_w);
5  // get consistency level based on Alg.2;
6  T_h = Calthreshold()// calculate threshold;
7  PushQue(T_h, X_n);
8  if (time > ΔT_1); then
9      Pop(QueMiddle, N_1)); // prevent starvation
10 else
11     if (time > ΔT_2); then
12         Pop(QueLow, N_2);// prevent starvation
13     else
14         DealQue(); // process queue normally
```

$X_n$ is the number of replication, $T_p$ is the time required to write or update the request to propagate to all copies, $app\_stale\_rate$ is need of the application for consistency, $\lambda_r$ and $\lambda_w$ are the parameters of the exponential distribution followed by the random variable read time and write time, $T_h$ is the threshold, $N_1$ and $N_2$ is the number of replications, $\Delta T_1$ and $\Delta T_2$ is the interval of the initialization time.

*1) Consistency level processing:* In the adaptive consistency module, we introduce the replica factor number adaptive algorithm [6] to get the consistency level of the request [7]. This algorithm uses stale read rate [8] to precisely define the consistency requirements of the application. $App\_stale\_rate$ describe the application's need for consistency. In order to meet the consistency requirements of the application, it should be $app\_stale\_rate \geq \theta_{stale}$ . $\theta_{stale}$ is the stale read rate.

The main algorithm of Consistency level processing is as follows:

**Algorithm 2**: Consistency level processing

```
1 if (app_stale_rate ≥ θ_stale); then
2     Choose eventuall consistency (Consistency Level =
      One);
3 else
4     Compute X_n the number of always consistent
      replicas necessary to have app_stale_rate ≥ θ_stale;
5 Choose consistency level based on X_n;
```

(1) Calculate stale read rate
The process of arriving is often seen as a poisson distrib ution, which is used in both literatures [9] and [10] to describe transactional access.
The formula of stale read rate is as follows:

$$P_{stale\_read} = \frac{(N-1)(1-e^{-\lambda_r T_p})(1+\lambda_r\lambda_w)}{N\lambda_r\lambda_w} \quad (1)$$

(2) Calculate the number of replications
In order to meet the consistency requirements of the application(ie $app\_stale\_rate >= \theta_{stale}$), We need to calculate the number of replications for the read requests(ie $X_n$).
$X_n$ satisfy the following inequality:

$$X_n \geq \frac{N((1-e^{-\lambda_r T_p})(1+\lambda_r\lambda_w) - ASR\lambda_r\lambda_w)}{(1-e^{-\lambda_r T_p})} \quad (2)$$

ASR is the $app\_stale\_rate$.

*2) Consistency level processing:* This strategy designs three priority task queues of HQ (High priority task queue), MQ (Middle priority task queue) and LQ (Low priority task queue). This strategy creates a counter CO in each queue to hold the total number of requests in the current queue and needs to create an accumulator AC (Accumulator) to record the sum of the consistency levels of all requests currently in the queue.

1. The module creates empty queues HQ, MQ, LQ, and initializes counter CO and accumulator AC to zero.
2. When the consistency level processing module receives the request, the monitoring module will collect information related to the data from Cassandra Node tool and pass the information to the adaptive consistency module.
3. After receiving the information, the adaptive consistency module will calculate the consistency of the request level according to the replica factor number adaptive algorithm.

4. After receiving the information, the adaptive consistency module will calculate the consistency of the request level according to the replica factor number adaptive algorithm.The request distribution module distribute requests to their corresponding queues and pass the consistency level to the corresponding HQ, MQ, LQ queues. Then the system update the counters and accumulators. So that the threshold between HQ, MQ, and LQ should be dynamically updated.
5. After receiving the information, the adaptive consistency module will calculate the consistency of the request level according to the replica factor number adaptive algorithm.The request processing module read $X_n$ replications from Cassandra database according to the order.
6. The system loop through steps 2-6 until the database exception or stop the service.

*3) Request distribution and request dispatch processing:* The formula for the threshold of the priority queue:

$$Th_{low} = \frac{\sum_{i=1}^{n} err\_val_i}{2n} \tag{3}$$

$$Th_{high} = \frac{2\sum_{i=1}^{n} err\_val_i}{n} \tag{4}$$

$Th_{low}$ is the threshold of the current low-priority queue, $Th_{high}$ is the threshold of the current high-priority queue, $err\_val_i$ is the consistency level of the i-th task in the priority request processing queue(ie the value in the current accumulator ). N is the total number of requests in current queue (ie the current counter value). The request distribution module obtains each request and its corresponding consistency level from the consistency level processing module. The module compares the consistency level with the current $Th_{low}$ and $Th_{high}$, and the request is delivered to the corresponding consistency according to the result of the comparison queue. The priority queue request processing flow is as follows:

1. If the new request's consistency level is less than $Th_{low}$, the request dispatch module will add the request to the end of the low priority queue.
2. If the consistency level of the new request is less than or equal to $Th_{high}$, and greater than or equal $Th_{low}$, the request distribution module will add the request to the end of the priority queue.
3. If the new request's consistency level is greater than $Th_{high}$, the request distribution module will add the request to the end of the high priority queue.
4. Counter CO is incremented by 1. The accumulator AC adds a new request for the consistency level, and the threshold is updated.
5. The system cycle 1-5 step, until the database exception or stop the service.

*4) Consistency level processing:* The flow of request processing is as follows:

1. Request processing module to see whether the high priority queue is empty, if not empty, then from the high priority queue header to take out a request to perform.
2. If the high priority queue is empty, then it determines if the priority queue is empty, and if it is not empty, the module takes a request from the priority queue header.
3. If the priority queue is empty, then it is judged whether the low priority queue is empty, and if it is not empty, then the module takes a request from the low priority queue header to execute.
4. If the low priority queue is empty, then go to step 2-4, until the database stops the service so far.
5. After removing the request to be executed, the counter CO is decremented by one, the accumulator AC subtracts the consistency level of the request, and the threshold is updated.
6. Cycle to implement 1-6 steps, until the database exception or stop the service so far.

## III. Experiment and analysis

### A. Introduce the experimental environment

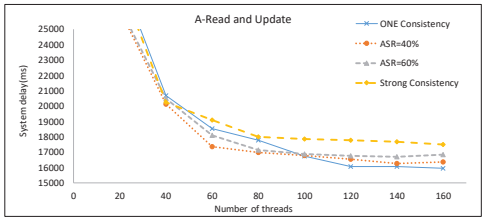
The Python version is 2.7.12+; the JDK version is 1.8.0_111; the Apache Maven version is 3.3.9; the Cassandra database cluster version is the latest version 3.9; the test tool YCSB version is 0.13.0. In the cassandra experimental platform, we set server 2 as a seed node, and server 9 as a client access. The role of seed nodes is that when a new node is added to a cluster, it is necessary to run the gossip process. This experiment is tested using the YCSB tool, and the YCSB is called "Yahoo! Cloud Serving Benchmark". The experiment in this paper uses YCSB's default five loads for experimentation, as shown in Table 1:

TABLE I: Workloads used in the experiment

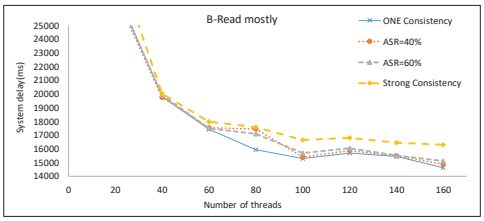
| load | settings | Choose operation | example |
|---|---|---|---|
| A-Read and Update | read50% update50% | Zipfian | Shopping cart online |
| B-Read mostly | read95% update5% | Zipfian | Photo tag |
| C-Read latest | read80% insert20% | Latest | User status read and update |
| D-Read-modify-write | read50% update50% | Zipfian | User profile cache |
| E-Scan short ranges | select95% insert5% | Zipfian / Uniform | Shopping cart online Thread session |

### B. Experimental process and parameter setting

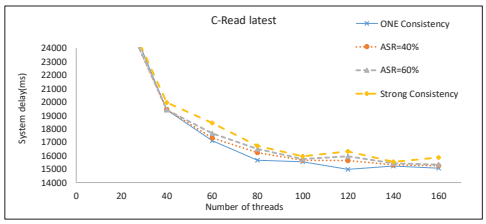
Experiments teste the five workloads in Table 1 and loaded the data into eight Cassandra copies with a total data size of 14.3 GB. Experiments is performed using two different ASR values and were compared using the ONE consistency strategy and strong consistency. The first ASR value is chosen to be 60%. The value means that its consistency requirement is slightly lower and the waiting time is shorter. The second ASR value is chosen to be 40%, which is slightly more stringent than the first ASR value, meaning that the request may read more copies and the consistency requirements are higher.



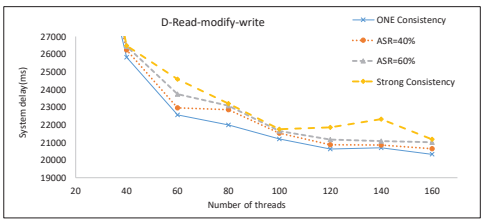

(a) Case (a)



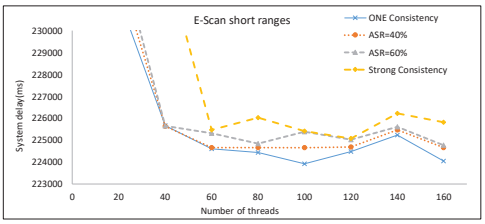

(b) Case (b)




(c) Case (c)




(d) Case (d)




(e) Case (e)

Fig. 3: The relationship between client threads and system delay

### C. results and analysis

*1) Delay experiment:* We conduct experiments separately with the number of threads at 1,20, 40, 60, 80, 100. We teste the delay time for different consistency cases under the five workloads. It can be seen from Figure 3.(a),(b),(c),(d),(e), strong consistency is the longest in any case, since all replicas are required to return the result. The ONE Consistency Policy provides the shortest latency scenario because it requires only one copy of the return value, but this results in a lack of consistency in the large amount of data. The time delay of the tunable consistency strategy proposed in this paper is basically between the ONE consistency strategy and the strong consistency delay. Overall, ASR is inversely proportional to the delay time.

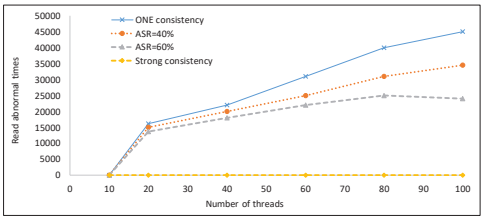


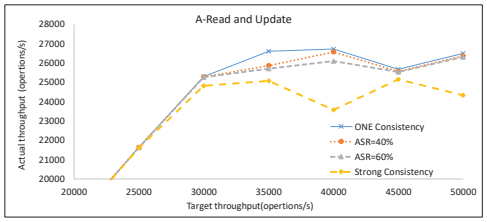

Fig. 4: The relationship between client threads and number of stale reads

*2) Read data exception experiment:* Since the YCSB does not count stale read rate, the experiment first outputs the results of the strong consistency and assumes that all data is correct(ie stale =0). We compared the ASR = 40%, ASR = 60%, and ONE consistency strategies with strong consistency as benchmarks. The experiment uses the workload A-Read and Update.

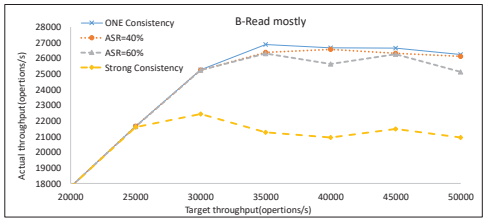
It can be seen from Figure 4, the number of the read data anomalies proposed in this paper is less than that of the ONE consistency strategy. And the number of the read data abnormalies when the ASR becomes smaller. When the number of threads is greater than 40, the number of abnormalities in this strategy has a tendency to decrease, and the gap between the ONE consistency strategy is also increased.

*3) Throughput experiment:* In YCSB, the delay and throughput can be output at the same time when the workload is run. So the throughput experiment and the delay experiment actually need only once.

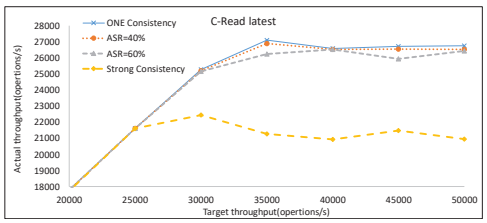
It can be seen from Figure 5.(a),(b),(c),(d),(e) that the actual throughput increases as the target throughput increases, but when the target throughput reaches a certain amount, the increase in actual throughput tends to be stable or even reduced because the throughput has reached the limits of the system. The throughput of the strategy proposed in this paper is greater than the strong consistency and less than the ONE coherency policy, since the higher the consistency requirement, the longer



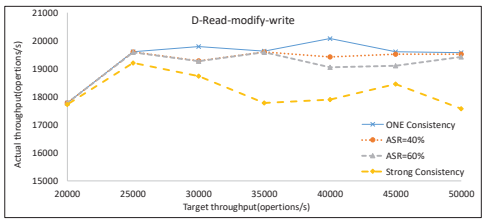

(a) Case (a)



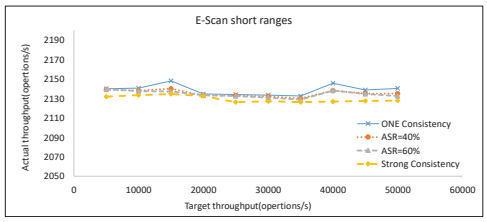

(b) Case (b)




(c) Case (c)




(d) Case (d)




(e) Case (e)

Fig. 5: The relationship between target throughput and actual throughput

the time it takes to process a single request. So the number of requests per unit time can be reduced. Regardless of whether the ASR is set to 40% or 60%, the throughput is similar to the ONE consistency strategy in workloads A, B, D, and E. But in most cases ASR = 60% throughput is still less than ASR = 40% throughput.

## IV. Conclusion

The delay and throughput of this strategy are between two kinds of traditional consistency, and can effectively reduce the number of times the system reads the data anomaly. Users only need to enter the overall application of reading data anomaly needs, and no longer need to make a consistent request for each request, greatly saving the user's time. This strategy is more suitable for a variety of types of applications, users can provide more consistent with the needs of user's consistent strategy.

Therefore, the strategy can dynamically coordinate the consistency level of the system and ensure the security of its data.

## References

[1] F. Wang and G. Zhong, "Research on technology of drds distributed database," in *International Conference on Information Technology and Management Innovation*, 2015.

[2] H. E. Chihoub, S. Ibrahim, Y. Li, G. Antoniu, M. S. Perez, and L. Bouge, "Exploring energy-consistency trade-offs in cassandra cloud storage system," in *International Symposium on Computer Architecture and High PERFORMANCE Computing*, 2015, pp. 146–153.

[3] N. Zaza and N. Nystrom, "Data-centric consistency policies: A programming model for distributed applications with tunable consistency," in *The Workshop on Programming MODELS and Languages for Distributed Computing*, 2016, p. 3.

[4] A. Dey, A. Fekete, R. Nambiar, and U. Rohm, "Ycsb+t: Benchmarking web-scale transactional databases," in *IEEE International Conference on Data Engineering Workshops*, 2014, pp. 223–230.

[5] D. Vohra, *Using the Java Client*. Apress, 2015.

[6] H. Wang, J. Li, H. Zhang, and Y. Zhou, *Benchmarking Replication and Consistency Strategies in Cloud Serving Databases: HBase and Cassandra*, 2014.

[7] H. Liang, J. Wei, Y. Xing, and J. Wu, "A feature-based replica consistency strategy for spatial data in distributed gis," in *International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2010, pp. 11–16.

[8] H. E. Chihoub, S. Ibrahim, G. Antoniu, and M. Prez, "Consistency management in cloud storage systems," *Large Scale and Big Data - Processing and Management*, 2014.

[9] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective," in *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, 2011, pp. 134–143.

[10] A. T. Tai and J. F. Meyer, "Performability management in distributed database systems: An adaptive concurrency control protocol," in *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, 1996, p. 212.