Research on A Tunable Consistency Strategy of Distributed Database

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.

1 Introduce

In the context of massive data, the use of distributed database system has become an inevitable choice. The requirements of the traditional strong consistency brought a problem of delayed improvement. 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. Cassandra allows users to use different conformance.

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 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 consistency.

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 1 introduces the tunable consistency strategy. Section 2 presents the experiment. Section 3 gives the experimental results. Section 4 summarizes the conclusion.

2 Tunable Consistency Strategy

2.1 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.



Fig. 2.1 The comparison of the structure before and after adding the tunable consistency strategy.

The request schedules layer added in the strategy are the core architectures that achieve tunable consistency. The request scheduling layer is a consistent control layer and is 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.



Fig.2.2 System Module diagram

As shown in Figure 2.2, the request scheduling layer in this 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 module. While the module is responsible for passing the read data to the interface layer.

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

2.2 Realization of the strategy

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

Algorithm 2: Tunable consistency strategy

QueHigh = new List();

QueMiddle = new List();

QueLow = new List();

Xn = Calconsistence(Tp, app\_atale\_rate, λr, λw); // get consistency level based on Alg.2

Th = Calthreshold(); // calculate threshold

PushQue(Th,Xn);

If(time >Δ T1){// prevent starvation

Pop(QueMiddle,N1);

}else if(time>Δ T2){// prevent starvation

Pop(QueLow,N2);

}else{ DealQue(); // process queue normally}

Xn is the number of replication, Tp is the time required to write or update the request to propagate to all copies, app\_atale\_rate is need of the application for consistency, λr and λw are the parameters of the exponential distribution followed by the random variable read time and write time, Th is the threshold, N is the number of replications, Δ T is the interval of the initialization time.

2.2.1 Consistency level processing

In the adaptive consistency module, we introduce the replica factor number adaptive algorithm[48] to get the consistency level of the request. This algorithm uses stale read rate 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 >=θstale . θstale is the stale read rate. The main algorithm of Consistency level processing is as follows:

Algorithm 2: Consistency level processing

if app\_stale\_rate >=θstale then

Choose eventuall consistency (Consistency Level = One)

else Compute Xn the number of always consistent replicas necessary to have app\_stale\_rate >=θstale

Choose consistency level based on Xn

end if

(1) Calculate stale read rate

The process of arriving is often seen as a poisson distrib ution, which is used in both literatures [50] and [51] to describe transactional access. The formula of stale read rate is as follows:

 (2.1)

(2) Calculate the number of replications

In order to meet the consistency requirements of the application(ie app\_stale\_rate >=θstale ), We need to calculate the number of replications for the read requests(ie Xn). Xn satisfy the following inequality:

 (2.2)

ASR is the app\_stale\_rate.

2.2.2 Priority processing queue

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) 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) The request processing module read Xn replications from Cassandra database according to the order.

(6) The system loop through steps 2-6 until the database exception or stop the service.

2.2.3 Request distribution and request dispatch processing

The formula for the threshold of the priority queue:

 (2.3)  (2.4)

Thlow is the threshold of the current low-priority queue, Thhiah is the threshold of the current high-priority queue, err\_vali 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 Thlow  and Thhiah, 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 Thlow, 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 Thhiah, and greater than or equal Thlow, 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 Thhiah, 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.

2.2.4 Request 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.

3 Experiment and analysis

3.1 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 3.1:

Tab. 3.1Workloads used in the experiment

|  |  |  |  |
| --- | --- | --- | --- |
| load | settings | Choose operation | example |
| A-Read and Update | read：50%  update：50% | Zipfian | Shopping cart online |
| B-Read mostly | read：95%  update：5% | Zipfian | Photo tag |
| C-Read latest | read：80%  insert：20% | Latest | User status read and update |
| D-Read-modify-write | read:50%  update：50% | Zipfian | User profile cache |
| E-Scan short ranges | select：95%  insert：5% | Zipfian/Uniform | Thread session |

3.2 Experimental process and parameter setting

Experiments teste the five workloads in Table 3.1 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.

3.3 results and analysis

3.3.1 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.

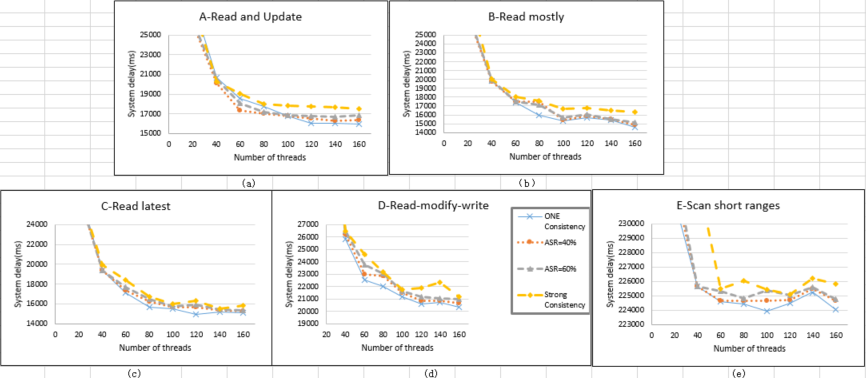


Fig. 3.1The relationship between client threads and system delay

It can be seen from Figure 3.1, 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.

3.3.2 Throughput experiment and Read data exception 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 3.2(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 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.

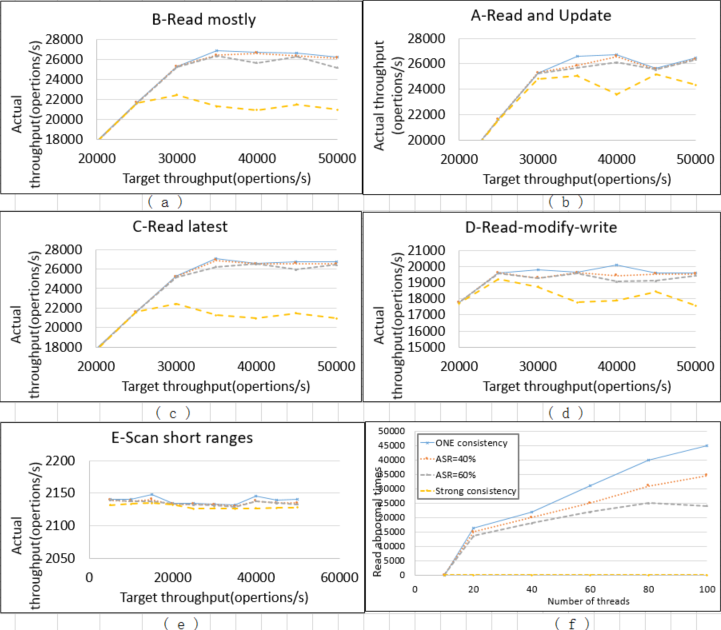


Fig. 3.2 The relationship between target throughput and actual throughput

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 θstale =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 3.3(f), 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 abnormal 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.

As can be seen from Figure 3.3 (f), the number of readout data anomalies proposed in this paper is less than the number of ONE consistent policies. And when the ASR requires more stringent, number of readout data anomalies will be reduced. With the increase of the number of threads, the number of abnormalities of the reading data has a tendency to decrease, and the gap with ONE consistency strategy is also increased.

4 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.