A Caching Mechanism Based on Data Freshness

Yasunari Takatsuka, Hiroya Nagao, Takashi Yaguchi, Masatoshi Hanai, Kazuyuki Shudo
Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku, Tokyo 152-8552 Japan
Email: takatsuka.y.aa@m.titech.ac.jp

***Abstract*—Use of cache is an effective way to improve access to a databases. However, when data in the cache are not updated or invalidated when writing to databases, the possibility increases with time that data in the cache will differ from data in the database. Therefore, we propose to adjust the balance between data freshness and access performance by switching between the use and non–use of a cache based on the load on the databases or on access performance. We also suggest and test a way of implementing the switching.**

***Index Terms*—Data freshness, Cache, Distributed database**

1. Introduction

One general way to improve performance in accessing databases is to replicate the data to locations that can be accessed by clients at high speeds. A typical example is a cache, which replicates data which is accessed once to a location that can be accessed by clients at high speeds and then clients access the replication in subsequent read requests. Specifically, communication latencies can be reduced by replicating data to servers that are geographically close to clients, and read latencies can be reduced by duplicating data to memory to reduce the number and duration of disk accesses. In such techniques, there is the problem of deciding the extent to which we maintain consistency between the original data and replications. Unless the cache is updated or invalidated, cached data will become obsolete when data in the database are replaced, removed, or augmented; so over time, the possibility of clients reading old data gradually increases. In this paper, we refer to this situation as a reduction in data freshness.

For example, if an attempt is made to obtain the most recent data without the use of the external caches even though the database is overloaded, access performance is sacrificed. However, there are applications, such as access counters on a web pages, in which performance is deemed to be more important than reading the most recent data. In such cases, it is possible to maintain access performance by using the cache while allowing a temporary reduction in data freshness.

Therefore, we propose adjusting the balance between data freshness and access performance depending on the load on the database, and we propose a method for controlling the frequency of use of the cache. To evaluate the proposed method, we design and implement a cache mechanism based on the proposed method and perform experiments in combination with a database management system. By the experiments, we confirm that the proposed method can adjust the balance between data freshness and access performance.

The rest of this paper is organized as follows. In Section 2, we define data freshness, and in Section 3 we review related work. In Section 4, we describe the proposed method, and we present the design and implementation of the cache mechanism that is based on the proposed method. In Section 5, we report results from experiments for evaluating the performance of the proposed method. Finally, in Section 6, we summarize conclusions and consider future work.

2. Data freshness

An external cache is sometimes used to compensate for the performance of a database; this might be achieved by using such software as memcached [5] [7] or Redis [6]. If data are saved to an external cache, so long as the data are not updated or invalidated, the cached data will become obsolete over time. In other words, the possibility that data on an external cache differ from the data in the database gradually increases. In this paper, we use *data freshness* to mean the newness of the cached data, or, equivalently, to refer to the probability that consistency is maintained.

We can measure data freshness by using indicators such as the rate at which the latest data are read and the time between successive updates of the cached data.

3. Related work

There are two methods for invalidating old cached data: (a) invalidating the data simultaneously with write operations to the database and (b) invalidating the data by some other timing.

In method (a), it is always possible to get the latest data at the time of reading. In addition, there may be a minimum number of cache updates that are required. However, situations in which this method is available are limited. To implement this method, the write side, or the original database side, must access the cache and be able to manage it. Therefore, it is difficult to update the cache at the time of writing when some application other than the application that manages the cache writes to the database. In addition, cache updates are likely to be costly when the cache is distributed or the output of an application is cached and the output uses data from the database.

When the method of invalidating the cache simultaneously with write operations is not available, method (b) could be used. A typical example is to attach an expiration date to the cache. This approach can be used without restrictions, unlike method (a), because the operation of invalidating is completed in the cache. In this way, minimum data freshness is guaranteed by the restriction of not reading data that are older than the expiration date.

In method (b), the "expiration date" is the parameter that adjusts the balance between data freshness and performance. Although the minimum freshness specified by the expiration date is secure, it is not possible to guarantee performance as a result of cache utilization. Therefore, if the application wants to guarantee the worst–case response time of the database, this poses a problem that can greatly compromise data freshness because it is necessary to set the expiration date to a value longer than necessary.

In our proposed method, we use "performance" as a parameter to adjust the balance between data freshness and performance. For example, by setting the desired communication latency to the minimum guaranteed for "performance," it is possible to maintain high data freshness while maintaining the average communication latency.

Other studies of changing the service level depending on load and access performance of databases have been reported. The method of Terry et al. [3] is an example. That method sets levels on the consistency guaranteed by the database and changes consistency levels depending on the read latency from the database. In contrast, our proposed method does not limit the configuration of the database, their method presupposes implementation to a database management system itself under a master-slave configuration. In such configurations, it is difficult to achieve performance for a number of machines since writing is necessarily performed on the master.

4. Proposed method

Typical database management systems ensure that they can read the latest data. In other words, they do not allow reductions in data freshness. In contrast, by sacrificing the minimum data freshness, we intend to achieve the target access performance.

If a cache is used, data freshness is reduced when access performance is improved. Therefore, our method adjusts the balance between data freshness and access performance by switching between use and non–use of a cache. Specifically, the method monitors the load on the database and switches to use a cache only when the load is high. For example, the method monitors read latency and switches to use a cache when the latency exceeds a target value that is specified in advance. This enables data freshness to be maintained under low loads and access performance to be improved under high loads at the expense of data freshness.

Our proposed method does not limit the function of a cache itself, so it can be combined with another techniques like "expiration date" and any replacement policies for data on a cache.

4.1. Implementation

To evaluate the proposed method, we design and implement a prototype of the cache mechanism. The cache mechanism is implemented using the application program interface of an existing database management system; the cache mechanism serves as a reverse proxy of a database. In this paper, we call the cache mechanism a freshness–aware reverse proxy (hereafter, FARP).

4.1.1. Switching between use and non–use of a cache. The implemented cache mechanism measures read latency between FARP and the database, and it switches between use and non–use of the cache to read data from the cache only when the measured latency exceeds a target value.

Sometimes, read latency increases suddenly because individual read latency varies somewhat according to the status of the database. For example, if the data to be read is present in the memory of the database, the read latency is small; otherwise, the read latency is large. Therefore, the accuracy of the latency control can fall if the switch between use and non–use of a cache is judged every time a read request is issued. Accordingly, we use an average read latency to decide when to switch between use and non–use of the cache. The interval at which to judge for switching between use and non–use (hereafter, judgeInterval) is set by specifying the number of reads used to calculate the average latency. The switch between use and non–use is judged every time the number of reads reaches the value of judgeInterval.

4.1.2. Dynamic adjustment of judgeInterval. As mentioned in Section 4.1.1, the interval to judge for switching between use and non–use of the cache (judgeInterval) is equivalent to the interval for calculating the average read latency. Then, the average latency becomes more accurate as the value of judgeInterval is increased because the number of samples used to calculate the average latency is increased. However, if judgeInterval is too large, switching may not take place when switching is really required (e.g. when latencies become increased). Conversely, if judgeInterval is small, finer control over switching is attained, but we increase the possibility of switching when no switching is needed (e.g. when a latency sharply increases by chance). This can occur because the number of samples used to calculate the average latency is reduced, so the estimate for the average latency is less accurate.

Therefore, we implemented FARP to dynamically change judgeInterval according to the measured average latency. FARP calculates the ratio of the average latency to the target value that FARP tries to maintain every time the average latency is calculated; when the ratio is large, a correspondingly large change in judgeInterval imposed. However, an upper bound is also imposed on judgeInterval to prevent it from becoming too large.

Changes in judgeInterval are done by the following equation. The initial value and upper limit of judgeInterval can be set separately when the cache is used and when it is not used. We call the initial value when using a cache initialIntervalWhenCaching (iIWC) and the initial value when not using a cache initialIntervalWhenNotCaching (iIWNC).

- When using a cache
  judgeInterval = iIWC $\times$ (average latency / target value)
- When not using a cache
  judgeInterval = iIWNC / (average latency / target value)

When using the cache, the interval between calculations of the average latency becomes longer (by the number of reads from the cache) than when not using the cache. Therefore, when using the cache, judgeInterval and its upper bound should be set to smaller values than those when not using the cache.

5. Experiments

Based on experiments using FARP, we now evaluate the proposed method from two points of view. First, we check whether FARP can maintain the read latency when the load on the database is increased by increasing the number of read/write processes per unit time. Second, we examine data freshness for that time. We use Apache Cassandra [1] [4] as the database management system.

In this experiment, we imitate clients by using the YCSB [2]. YCSB, which is a benchmarking tool for NoSQL, can specify the ratio of read/update, the access distribution to the database, and the target value for processing number per second. However, YCSB does not have a function for measuring data freshness, although it does have a function for aggregating latency. Therefore, we modified the YCSB to measure data freshness.

In this experiment, we use the rate at which the latest data are read as the measure of data freshness. It is calculated by

$$DataFreshness = \frac{\mathrm{readCountOfTheMostRecentData}}{\mathrm{totalReadCount}}$$

The total read count can be specified by YCSB. Therefore, to calculate data freshness by the above formula, it is necessary to measure the number of reads of the latest data. Accordingly, we modified YCSB to measure it. YCSB stores the contents of writes, and sequentially checks whether there is an inconsistency between the data read and data stored.

5.1. Experimental environments and parameters

Table 1 shows the machine configuration used in the experiments. We used a solid–state drive (SSD) as the storage device.

TABLE 1. Server configuration for experiments

OS	Ubuntu 12.04.3 LTS
CPU	2.40 GHz Xeon(R) E5620 $\times$ 2
Memory	8 GB RAM
SSD	Crucial Real SSD C300 128 GB
Java	Java SE 7 Update 4

TABLE 2. Value of judgeInterval

judgeInterval	initial value	upper limit
when using cache	10	1000
when not using cache	100	10000

The number of data records stored on one server was ten million records. The capacity of one record was 1 KB because one record had 10 columns and one column was 100 bytes. Therefore, the overall data size was 10 GB, which is further increased by information of schema; thus, it was impossible for all data to be stored in 8 GB memory.

In the experiments, from the workloads that YCSB provides, we used the update–heavy workload whose ratio of reads to writes is one to one. We used a Zipf distribution as the access distribution to the database. The Zipf distribution imitates the access distribution of an application in which the access frequency for data is determined regardless of the freshness of the data.

YCSB gradually increases the number of accesses to database per second (throughput). In this way, we gradually increase the load on the database. In the experiment, we measured the average read latencies and data freshness for each throughput.

We used LRU as the cache algorithm for FARP, and set the cache size to be one million records. The initial value and upper limit of judgeInterval were as shown in Table 2.

We set the target latencies for FARP to be 5, 10 and 15 ms. For comparison, we also tested the case of always using cache, even when the latency was small and the case of directly accessing the database without using FARP.

5.2. Experimental results

We experimented with one database node for Apache Cassandra. Results for the update–heavy workload are shown in Figures 1 and 2. Figure 1 shows read latencies, and Figure 2 shows data freshness. Figure 1 shows that read latencies were controlled to the target value. In figure 2, we can see that data freshness was higher as the target value of the latency increased. This is because, when the target value is large, it is possible to further increase the rate of access to the original data.

We also experimented on the read–heavy workload whose ratio of reads to writes is ninety five to five, and we confirmed that the tendencies in changes of read latencies and data freshness were similar to the experimental results on the update–heavy workload.

5.3. Discussion

Data freshness began to decrease when the direct access latencies began to exceed the target value. In other words, the cache began to be used around that time. Even when the load was increased in succession, read latencies were maintained at the target value, while data freshness decreased as the sacrifice. From this result, although the load with which the read latencies can be maintained at the target value were limited, we can confirm that FARP, which is implemented based on the proposed method, can adjust the balance between read performance and data freshness.

5.3.1. Read Latencies. Until use of the cache began, that is, while data freshness was at 100%, read latencies with FARP were slightly greater than those for direct access in both workloads. This was due to the communication delay through FARP and the processing time of FARP itself.

The throughput with which read latencies could be maintained to the target value was limited. Since, in this implementation, FARP used memory to store data, the amount of data that FARP could hold was smaller than the amount of data stored in the database. In addition, when the cache was not hit, FARP accesses the database to retrieve the data to be returned to the client. For these reasons, as the load increased, the cache miss ratio per unit time increased, and the database access per unit time also increased. In other words, even when the cache always used, there was still a limit to the reduction of read accesses to the database. As shown in Figures 1, read latencies increased to an extent that is beyond the capacity of FARP to retain access performance. This problem can be solved by increasing the memory capacity used for the cache.

5.3.2. Data freshness. Data freshness decreased when the load was high. This was because, when the load is high, it is necessary to use the cache frequently to maintain read latencies. In addition, when the target latency is low, the cache must be accessed more frequently. Therefore, at low target latencies, data freshness decreased.

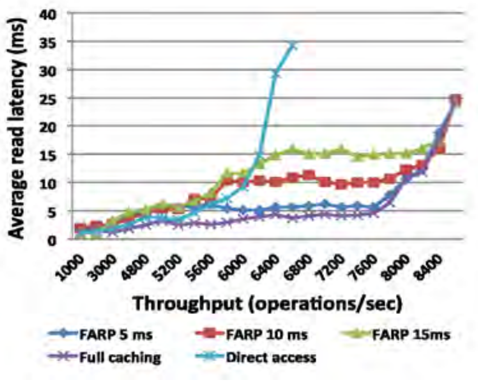

Figure 1. Read latencies with update–heavy workload

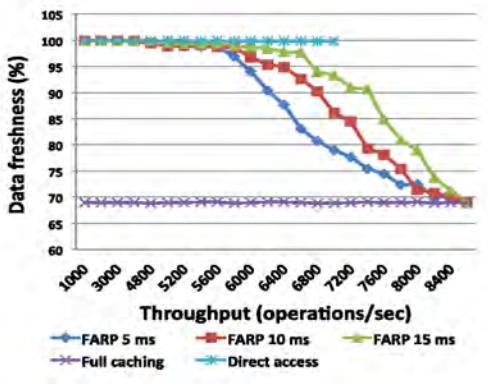

Figure 2. Data freshness with update–heavy workload

6. Conclusions and future work

We have proposed a method for adjusting the balance between access performance and data freshness according to the load placed on the database, by controlling the frequency of use of a cache. Based on the proposed method, we designed and implemented a cache mechanism (FARP) that ensures access performance can be maintained, and we evaluated read latency and data freshness.

In future work, we intend to compare the proposed method with a method that invalidates old data on a cache based on expiration date. When the proposed cache switching method is used, it is possible to maintain an average value for read latencies, but it is not possible to maintain each individual read latency at or below a threshold. Therefore, in future work, we intend to maintain read latency at the 99 percentile by adjusting cache usage.

Acknowledgment

This work is supported by JSPS KAKENHI Grant Numbers 25700008 and 26540161.

References

[1] Apache Software Foundation: Apache Cassandra, http://cassandra.apache.org/.

[2] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears.: Benchmarking Cloud Serving Systems with YCSB, Proc. SOCC '10, pp. 143–154, 2010.

[3] Douglas Terry, Vijayan Prabhakaran, Rama Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh.: Consistency-based Service Level Agreements for Cloud Strage, Proc. SOSP '13, pp. 309–324, 2013.

[4] Avinash Lakshman, and Prashant Malik.: Cassandra - A Decentralized Structured Storage System, Proc. LADIS '09, 2009.

[5] Brad Fitzpatrick.: Distributed Caching with Memcached, Linux Journal, vol. 2004, no. 124, p. 5, 2004.

[6] Redis: Redis, http://redis.io/.

[7] Jure Petrovic.: Using Memcached for Data Distribution in Industrial Environment, Proc. IEEE Computer Society, pp. 368–372, 2008.