2018 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovations

Toward An Efficient Cache Management Framework

1st Xuewei Niu
School of Information Science and Engineering,
University of Jinan
Jinan 250022, China
a@niuxuewei.com

2nd Kun Ma*
Shandong Provincial Key Laboratory of
Network Based Intelligent Computing,
University of Jinan
Jinan 250022, China
ise_mak@ujn.edu.cn

***Abstract*—Nowadays, many companies used the cache clusters to reduce the pressure when network traffic peak is coming. For general cache manage framework, it might cause query bottleneck and lead to the unavailability of services. Therefore, building an efficient cache manage framework is meaningful. In this paper, we purpose an aspect-oriented cache auto load framework with annotation. There are some features for improving performance. Auto load allows the hot-spot keys to be preserved in the memory. And load waiting allows that the data are obtained by a leader who is selected by the framework for the same requests. The result of experiments indicated that our approach is nearly 25% faster than other cache frameworks in the scenario that data size is enormous and concurrency is very high.**

***Index Terms*—Data Cache, Auto Load, Load Wait, Cache Batch Deletion**

I. Introduction

A. Background

Essential features of big data are Volume, Variety and Velocity (3V). It also brings significant challenges for data access and storage [1]. For example, Facebook has become a producer of big data, using the integration of MySQL and NoSQL [2]. In the case of high concurrency, these systems face cache penetration [3].

Cache is a collection of data duplicating original values stored in memory, usually for easier access. Moreover, it can reduce the server response time efficiently [4]. There are many ways to implement the cache, such as the cache based on browser, the cache based on CDN [5] and the cache based on applications [6]. Relative Database Management Systems (RDBMSs) combined with the NoSQL databases is the primary solution for the dynamic data, RDBMSs are used for data persistence, in-memory NoSQL databases are used for caching [7].

There are a lot of cache management systems, but they all have some shortcomings. Firstly, for the data that will be accessed frequently or need a long time to process, the general expired strategy that caches have a fixed expired time rather than lurk in the memory constantly might lead to the server instability [8]. Secondly, if cache miss, there are many requests for the same data in the high-concurrent scenario. RDBMSs need to handle those duplicate requests directly. These inefficient works bring about the service not available eventually. Thirdly, if perform update operation, cache deletion is a way to avoid the inconsistency, but there isn't a effective measure to implement it when the architecture become complex.

B. Contributions

In this paper, we propose an efficient cache management. Contributions of our framework are using the auto load and load wait mechanism for cutting down the read back of source, and using the hfield for reducing the cache deletion complexity.

- Auto load can implement hot-spot data pre-loading. The cache work because the data can be obtained from memory and cut down the read back of source. To further reduce the read back of source, we can make the hot-spot data lurk in memory constantly and update it regularly. Moreover, the data that need to a long time to process also benefit from this strategy, system can load them before peak coming.
- Load waiting can reduce the server pressure when load data from database. Once the cache miss, there are many requests flowing into the database, but for the same data. Load wait strategy select a leader to request the data, others wait until the data are provided by the leader.
- Cache delete in batches. It is inevitable that it leads the inconsistency of physical database and cache data. It is difficult to restore the key of the related cache by programs if the query statement is involved only. Therefore, the precise cache delete in batches are proposed.

C. Organization

The remainder of the paper is organized as follows. The current work of cache framework is discussed in Section II. In Section III, an efficient cache framework is proposed. Some cache storage strategies, such as auto load, load wait, are introduced. In Section IV, the response time, resource consumption and pressure test experiments show that our framework are effective and efficient. Brief conclusions and future research directions are outlined in the last section.

II. Related Work

Caching of data with weak consistency can achieve transparency. Typically, the cache content is periodically refreshed. There are different levels for caching performance, namely the browser cache [9], proxy cache [10], Content Delivery Network (CDN) cache [11], and server level cache [8].

Browser cache is closest to the end user and can speed up page loads by storing the static files, such as a single page, on user's local storage media [12]. CachePortal [13] relies on timestamps and HTTP logs to conservatively determine which pages to invalidate. It has a unique form of weak, time-lagged consistency. Proxy caches are situated at network access points for web users [10]. Consequently, proxy caches can store documents and directly serve requests for them in the network, thereby avoiding repeated traffic to web servers. CDN is similar to proxy caching [11]. But the difference is that the servers of CDN are distributed around the users, data is obtained from the server with least delay. The disadvantage of CDN cache is that excessive consumption of resources due to continuous analysis. Proxy cache and CDN solves the problem of the local cache cannot be shared. Although some of these products implement dynamic page cache, the effects are not good. In particular, server data processing efficiency is not effectively improved. Therefore, the present mainstream of cache is server level cache.

The combination of RDBMSs and in-memory NoSQL is the mainstream architecture of server cache. Several solusions are proposed such as Redis [14], Memcached [15], and EhCache. RDBMSs are used to persist data, and in-memory databases are used to store high-frequency data [8]. Redis performs better for high concurrency and memory consumption. As an improvement, Memcache pool is used to duplicate high-frequency data into a dedicated data pool by cache access frequency and cache expired cost [3]. This feature is achieved by the time of the cache analysis by some dedicated servers.

III. ACALFA: AN EFFICIENT CACHE MANAGEMENT FRAMEWORK

A. Architecture

There are 4 main components to implement it:

- CacheHandler: It is charged with process control rather than specific business. It parses cache configurations - are loaded from $@Cache$ Annotation - by expression parser. The crucial cache configurations are as follow:
  - CacheKey is a cache identification expression, which can generate cache key dynamically according to the parameters.
  - ExpireTime is a value to set up the cache expire time, the default value is 120(s).
  - IsCacheable is a expression that return true or false to determined whether the cache can be cachable.
  - IsAutoload is a expression that return true or false to determined whether the cache should be auto loading.
- AutoLoadHandler: It, maintaining multiple internal tasks and queues, is seen as auto load executor. The cache that is going to expire update by loading asynchronous. The details will be described in following section.
- DataLoader: It works with DAO(Data Access Object) for fetching data from database and prevents the excessive traffic from coming into database. Therefore, the load wait strategy is implemented in this component.
- CacheHelper: It encapsulates functions of operating cache, such as get, set.

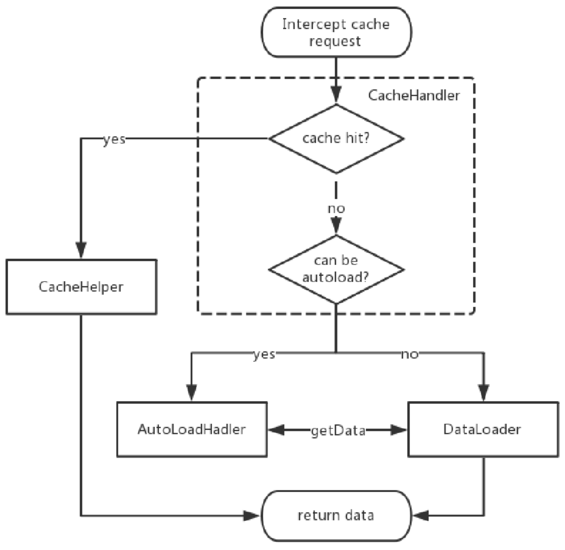

Fig. 1. Executive process of ACALFA

Figure 1 gives the simplified architecture of our proposed cache management framework that is used for reducing the read back of sources. When interceptor intercepts cache request, CacheHelper start to handle it on the basic of cache configurations. First of all, CacheHandler should judge the cache state, cache hit, indicates that cache of corresponding data exists in the cache, or cache miss, means the system should fetch data from physical database, by CacheKey. And then, CacheHandler should decide whether the cache need to be added to auto load queue according to the IsAutoload and IsCacheable. Finally, the query data need be synchronized to the cache asynchronously. If the IsCacheable is true, the data will be saved to cache and return, if not, only return.

B. A Pre-Loading Strategy: Auto Load

Generally, each cache have its own expired time, which it's not suitable for high-frequent cache or time-consuming data in in-memory cache. Once expired, the system has to fetch those data from data source, which may consume lots of resources and make response slow. Therefore, those types of data should reside in the memory until they don't take too many resources when fetching them.

Figure 2 shows the executive process of auto load. $hashTable$, which providing a method called $putIfAbsent$ to avoid duplicate tasks and controlling the number of auto load tasks, is a storage for auto load assignments. $taskQueue$ is taken from $hashTable$ by a unified strategy, like LRU. In distributed scenario, the standardized sequence of tasks help

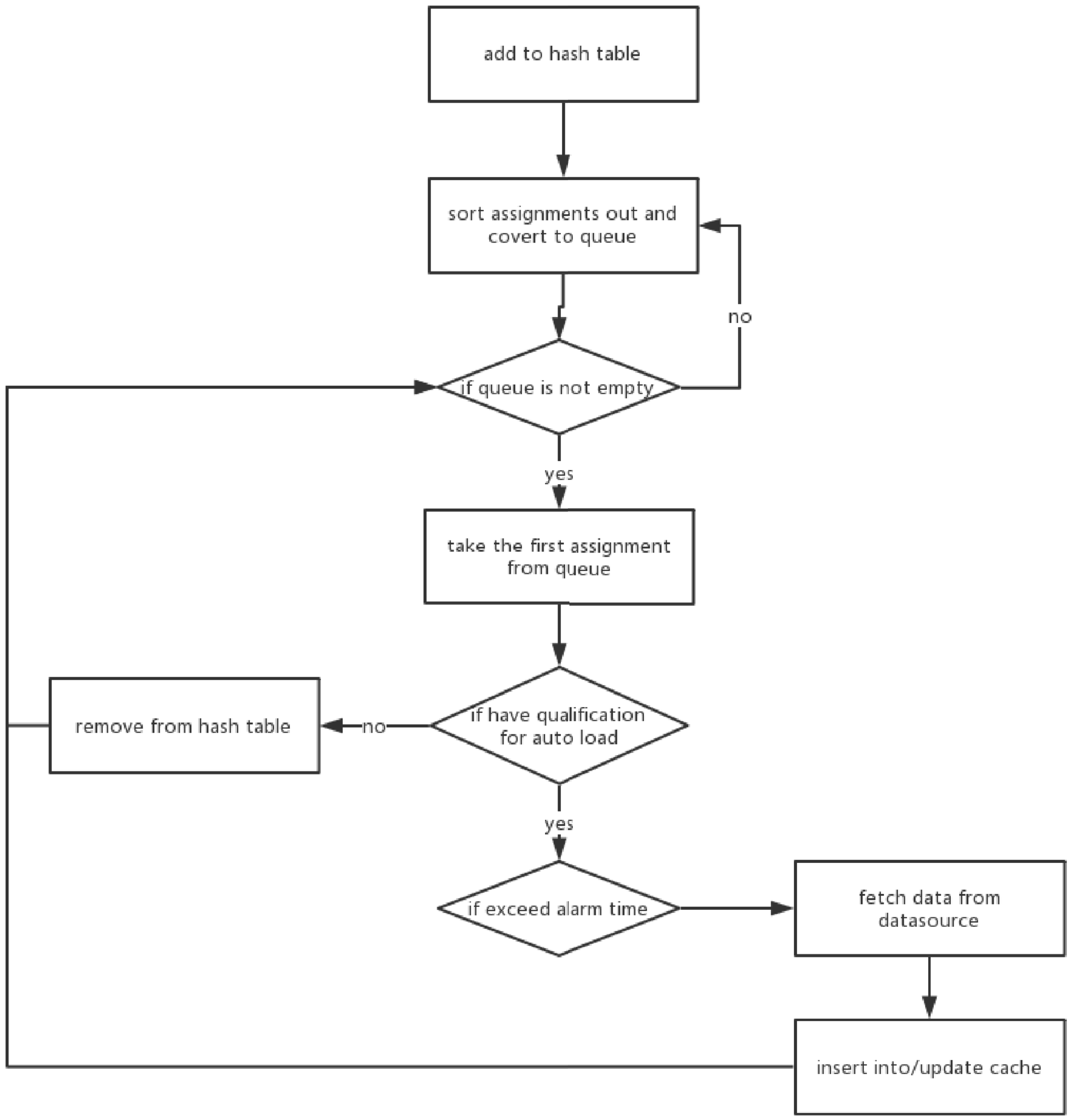
add to hash table
sort assignments out and covert to queue
no
if queue is not empty
yes
take the first assignment from queue
remove from hash table
no
if have qualification for auto load
yes
if exceed alarm time
fetch data from datasource
insert into/update cache

Fig. 2. Executive process of auto load

the system reduce the read back of source, because others can fetch data from cache rather than database after first server loaded. Auto load, however, is a double-edged, the system may become unstable if the amount of auto load tasks were too much or update frequency were too high. Accordingly, a strict inspection should be applied. We assume that t_n(ms) is current time, t_{fr}(ms) is the first request time, t_{lr}(ms) is last request time, t_a(ms) is the average of fetching time, c is the times of cache usage, t_{to} is a preset parameter determining cache time out. There are three main aspects as follow:

- Remove if the targeting cache is not requested for a long time. The time of not be requested is get from $t_{nr} = t_n - t_{lr}$, if $t_{nr} > t_{to}$, the task will be discarded.
- Remove if efficiency of fetching data from database is high. This may be confused. The intention of auto load is pre-load the data that cost a lot of time and resources and be accessed frequently. Hence, the data that obtaining efficiency is high should not occupy the space of auto load. The standard of determination is $c > 100$ and $t_a < 10$.
- Remove if usage rate is low. If the cache, which is requested only once within the preset timeout, were added to auto load queue, that will go against the purposes of auto load. Therefore, only removing the task that the

targeting cache is not requested for a long time is not enough. The usage rate can be get from $r = \frac{c}{\frac{t_n - t_{fr}}{3600000}}$, the meaning of r is how much the cache is requested per hours. If the task has been running for more than 1 hour and the average loading time is less than 1000 ms and r is less than 60, the task will be removed.

The last part of auto load is when to refresh the cache. As a pre-loading schema, the cache should be refreshed before expired. For this reason, setting alarm time is valuable. Experiments prove that the default alarm time should be set $t_e - 120$ if $t_e \geq 600$ or $t_e - 60$ if $t_e < 600$, which t_e is a preset parameter determining cache expired time.

C. Load Wait Strategy

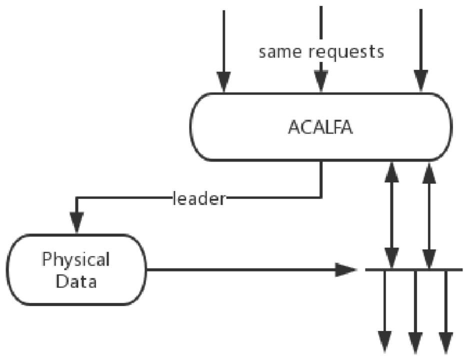

Fig. 3. Process of load waiting strategy.

Load wait strategy works when the cache missed. Once hot-spot data miss and there are mass of requests flowing into database within a very short period time. After analyzed those requests, we found the proliferation of requests are same and can be optimized for alleviating the pressure on the database. The principle is to select a leader to fetch data from database, others, requesting for data with the same key, will be suspended until the leader processed. Figure 3 shows the process of load waiting strategy.

The framework identify the same requests by the cacheKey, which is a unique identification for the data. Processing is a hash map storing the different processing requests. A request want to fetch the data from database, DataLoader will search whether the same request is being processed. The framework try to get request from Processing hash map by cacheKey. If the result is null, it indicates that the request is a new request. In the concurrency, there are mass of the requests, the framework will select the first request as a leader to fetch data from physical data, others are waiting until the leader notify them and bring the data back. The specic load waiting strategy algorithm is as Algorithm 1.

Algorithm 1 Load waiting strategy algorithm *loadwaiting*

```
Require:
    String $cacheKey$;
    Map $hashMap$;
Ensure:
 1: Processing processing = hashMap.get(cacheKey);
 2: if processing == null then
 3:     Processing newProcessing = new Processing();
 4:     Processing _processing = HashMap.putIfAbsent(cacheKey, newProcessing);
 5:     if null != _processing then
 6:         processing = _processing
 7:     end if
 8: end if
 9: if processing == null then
10:     doRequest(newProcessing)
11:     newProcessing.setFirstFinish(TRUE);
12:     hashMap.remove(cacheKey);
13:     newProcessing.notifyAll();
14: else
15:     waitUntillLeaderGotData(processing);
16: end if
17: return
```

D. Cache Delete in Batches

It's difficult to restore or obtain the cache key that need to delete when the query statements are complex, active deletion will hardly complete. In order to manage that, "namespace + hfield + cachekey" schema is proposed. The cache is added string, indicating its namespace, before the cache key, aiming to address conflict in the cache cluster. For active deletion, the key point is that establish a hash table to store the caches that need to be removed together. "hfield" is to identify a unique hash table. Developers can design the deletion granularity on need basis. The process of cache delete in batches is ACALFA intercepts the delete request and inspect whether hfield is set. If hfield is set, ACALFA will delete all the cache that store in corresponding hash table. If not, ACALFA will only delete that cache.

IV. Experiments

A. Experiment Setup

Three experiments are designed to illustrate the ACALFA performance in high-concurrency scenario. The metrics are response time, resource consumption and stability. Response time is for validating the optimization of auto load, especially for time-consuming tasks and high-frequent business. Resource consumption is for validating the load wait performance when handling the same requests. Stability testing uses multiple threads to access applications simultaneously to simulate high concurrency. The server with ACALFA is an Intel Core i7 @ 3.40GHz CPU, 16GB memory, and 100Mbps bandwidth, and this server runs a 64-bit Windows 10 with a Java 1.8 64-bit server JVM. Another server with MySQL database runs on Ali ECS Cloud Server: 2 core 2.60GHz Intel Xeon E5-2650 CPU machines with 4 GB RAM, 40GB SSD and 100 Mbps Ethernet. The system is CentOS 6.8 x64. Spring Cache, a well-known cache management framework at j2ee, will be compared with ACALFA for performance. The dataset, the structure is shown in Table I, is selected 4,000 records randomly from a voting system.

TABLE I
VOTING TABLE STRUCTURE.

Field	Type	Description
id	INT	primary key
candidate_id	INT	candidate identification
voter_id	INT	voter identification
score	INT	candidate scores between 1, 100

B. Response Time

We set up six experiments to test the ACALFA effect. Cache expired time we set was 120s and enabled auto load and load wait for ACALFA. Exp1, exp3 and exp5 used ACALFA, on the contrary, others used Spring Cache. The experiment sets are as follow:

- Exp1 & Exp2: The resources that will be requested were not in cache. Therefore, the frameworks should fetch data from database.
- Exp3 & Exp4: The response time when cache hit.
- Exp5 & Exp6: After exp1 and exp2 120s, in other words, cache expired, we send requests again for validating the effect of auto load.

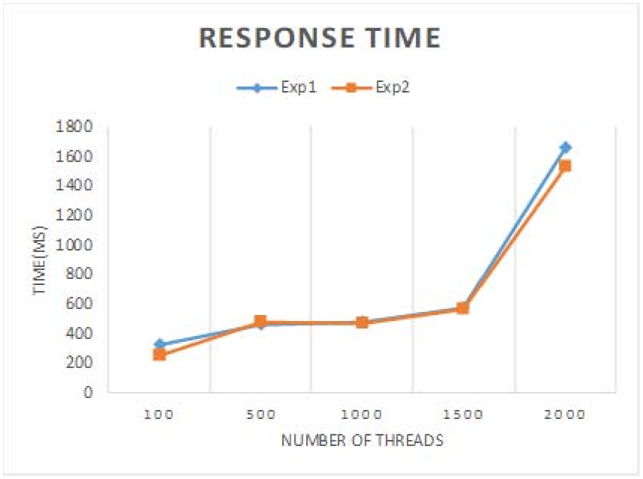

Fig. 4. Response time of Exp1 and Exp2

We use the 90th percentile response time for performance measurement. Figure 4 and figure 5 shows the response time is increasing with the increase of number of threads. The performance of two cache management systems when cache miss or cache hit is same roughly. Figure 6, however, shows vast difference between ACALFA with auto load and Spring Cache. The reason is obviously: ACALFA will reload data when the cache is going to expire, but Spring Cache can't. According to this range of experiments, we found auto load is suited to the hot-spot data and has better cache effects for the more time-consuming or more frequent data.

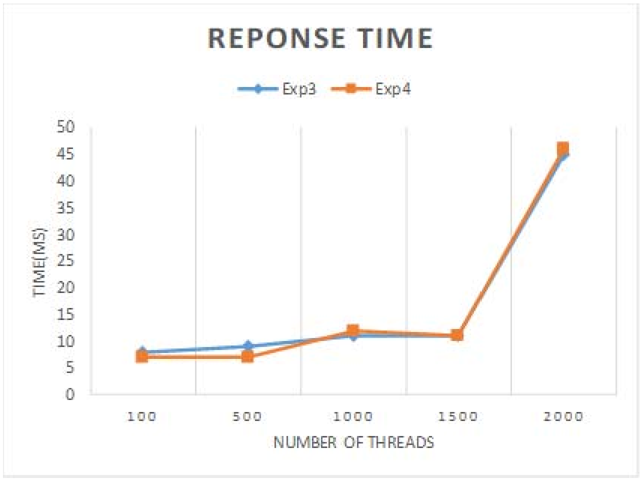

Fig. 5. Response time of Exp1 and Exp2

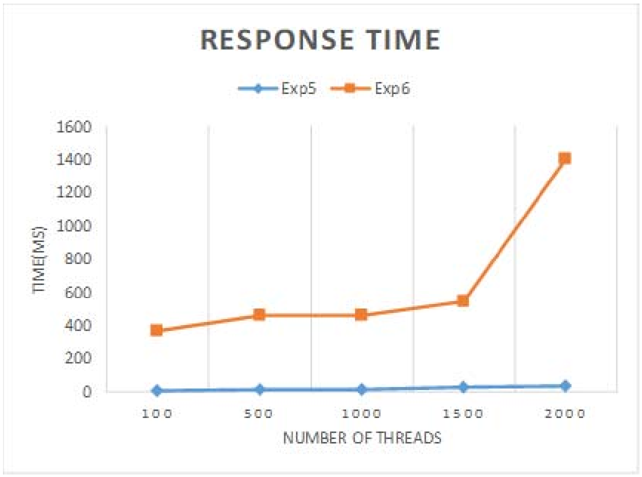

Fig. 6. Response time of Exp1 and Exp2

C. Resource Consumption

Resource consumption experiment is set for validating the load wait performance. Therefore, the cache is disabled and the query statement is changed to join two tables for increasing cpu usage. There are 30 threads and request 10 times repeatedly. The CPU usage is shown in figure 7. From results, CPU usage of non-load-wait lasted for a long time in 100%. However, ACALFA selected a leader to fetch data in each round of request to reduce the CPU consumption. According to the results, we found wait-load strategy is effective in cache miss situation. Coordinate with auto-load, ACALFA can deal with complex and high-concurrent scenario well.

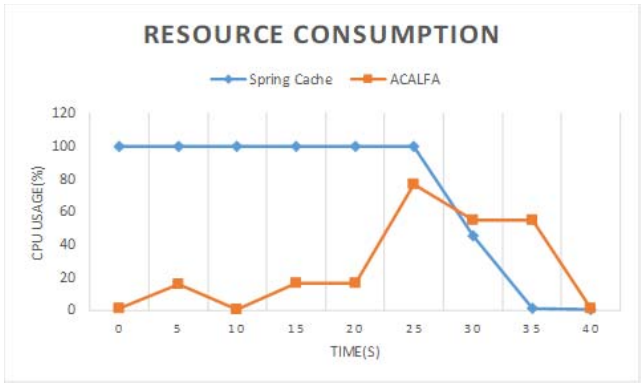

Fig. 7. CPU usage of performing the select statement

D. Pressure Test

The pressure test is getting access to a page repeatedly within one minute in order to test the stability of ACALFA in high concurrency. We used 100 threads to simulate 100 users for requesting the targeting page simultaneously. We can infer the system availability and response speed from the report. The query statement is same as resource consumption experiment. We can learn ACALFA response is 25% faster than Spring Cache, processing speed is 48% faster and stability has improved 36% from above figure 8.

V. Conclusions

The purpose of a cache is to duplicate frequently accessed or important data in such a way that it can be accessed very fast and close to where it is needed. In the big data era, caching generally moves data from a low cost. This paper introduces ACALFA - an efficient cache management system. Several experiments are illustrated that our cache framework has a good effect on the complex business with high concurrency.

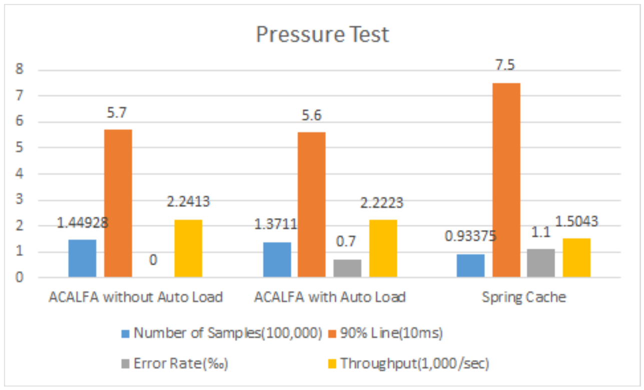

Fig. 8. Pressure with different methods

In the future work, memory resource cost will be optimized to increase cache hit ratio.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (61772231 & 61702217 & 61702216), the Shandong Provincial Natural Science Foundation (ZR2017MF025 & ZR2014FQ029), the Shandong Provincial Key R&D Program of China (2015GGX106007 & 2016ZDJS01A12 & 2017CXGC0701 & 2018CXGC0706), the Science and Technology Program of University of Jinan (XKY1734 & XKY1828). Kun Ma is the corresponding author.

References

[1] S. Kaisler, F. Armour, J. A. Espinosa, and W. Money, “Big data: Issues and challenges moving forward,” in *System Sciences (HICSS), 2013 46th Hawaii International Conference on*. IEEE, 2013, pp. 995–1004.
[2] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li, “Ripq: Advanced photo caching on flash for facebook.” in *FAST*, 2015, pp. 373–386.
[3] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, “Scaling memcache at facebook.” in *nsdi*, vol. 13, 2013, pp. 385–398.
[4] K. Ma, B. Yang, Z. Yang, and Z. Yu, “Segment access-aware dynamic semantic cache in cloud computing environment,” *Journal of Parallel and Distributed Computing*, vol. 110, pp. 42–51, 2017.
[5] A. T. Davis, J. Parikh, S. Pichai, E. Ruvinsky, D. Stodolsky, M. Tsimelzon, and W. E. Weihl, “Java application framework for use in a content delivery network (cdn),” Mar. 5 2013, uS Patent 8,392,912.
[6] C.-J. Wu and M. Martonosi, “Characterization and dynamic mitigation of intra-application cache interference,” in *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 2–11.
[7] K. Ma and Y. Bo, “Stream-based live data replication approach of in-memory cache,” *Concurrency and Computation: Practice and Experience*, vol. 29, no. 11, 2017.
[8] K. Ma and B. Yang, “Column access-aware in-stream data cache with stream processing framework,” *Journal of Signal Processing Systems*, vol. 86, no. 2-3, pp. 191–205, 2017.
[9] B. D. Davison, “A web caching primer,” *IEEE internet computing*, vol. 5, no. 4, pp. 38–45, 2001.
[10] J. A. Uruburo, C. E. Gomez, C. A. Candela, and L. E. Sepulveda, “Methodology applied in the construction of a proxy-cache server,” in *Computing Colombian Conference (9CCC), 2014 9th*. IEEE, 2014, pp. 156–161.
[11] A. Vakali and G. Pallis, “Content delivery networks: Status and trends,” *IEEE Internet Computing*, vol. 7, no. 6, pp. 68–74, 2003.
[12] V. S. Mookerjee and Y. Tan, “Analysis of a least recently used cache management policy for web browsers,” *Operations Research*, vol. 50, no. 2, pp. 345–357, 2002.
[13] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal, “Enabling dynamic content caching for database-driven web sites,” in *ACM SIGMOD Record*, vol. 30, no. 2. ACM, 2001, pp. 532–543.
[14] J. Zawodny, “Redis: Lightweight key/value store that goes the extra mile,” *Linux Magazine*, vol. 79, 2009.
[15] U. U. Hafeez, D. Male, S. K. Naeni, M. Wajahat, and A. Gandhi, “Realizing an elastic memcached via cached data migration,” *Restoration*, vol. 23, no. 24, p. 25, 2017.