A novel cache strategy leveraging Redis with filters to speed up queries

Qiankun Su, Xin Gao, Xin Zhang, and Zhihua Wang*
College of Computer Engineering, Jimei University, Xiamen, China
{qiankun.su, 201721122068, 201721121001, edgar_wang}@jmu.edu.cn

ABSTRACT

Query response time is an imperative performance parameter for applications driven by database systems. This paper proposes a novel cache strategy to speed up queries, especially in search-within-result. Redis is implemented as a cache server for Mongo database. We convert multiple query conditions into a set of filters and concatenate the filters to be capable of being keys in Redis. Thus, there is a cache hit if a subset of the filters is cached, not requiring that a query is exactly the same as previous queries. This leads to an increase in cache hit ratio and accelerates queries. Experimental results on a real dataset show the effectiveness of our proposal, achieving a 71.27% improvement of the average query response time.

Keywords: Cache, Filter, Search-within-result, Redis, MongoDB

1. INTRODUCTION

Our world is undergoing a data explosion. According to the report from Statista[1], global data creation is forecast to grow to more than 180 zettabytes up to 2025. In line with the rapid growth of the data volume, the installed base of storage capacity will grow at a compound annual growth rate of 19.2% from 2020 to 2025[1].

Storing data in a database has many advantages over a text file or a flat file, such as scalability, reliability, and fault tolerance. There are lots of different types of modern databases. Databases are widely divided into two major categories, relational databases and NoSQL databases. NoSQL databases have become popular due to their advantages in high capacity, scalability, low latency, etc [2-4]. MongoDB, Memcached, and Redis are the three most popular databases of NoSQL. MongoDB is a document-oriented, disk-based database that uses JSON-like documents. It has the flexibility of document schemas and the effective scalability to distributed collections of documents[5-6].

Query response time is an imperative performance parameter for applications driven by database systems. Caching is a strategy that has been shown to significantly improve the performance of queries. Redis is an in-memory data structure store capable of being used as a cache[7]. It can be implemented as a cache server for Mongo database to optimize queries. There is no cache hit for the query, search-within-result, though the results of the second query are a subset of the cached results of the first query. Inspired by this, this paper proposes a novel cache strategy to look up data from the cached results, rather than from MongoDB. Finally, we evaluate the performance of our cache scheme on a real dataset. Extensive experiments show the validity of our proposal.

This paper is organized as follows. Section 2 summarizes the main related work. The proposed cache strategy is presented in Section 3. Experimental results and their analyses are given in Section 4. Finally, Section 5 concludes this work.

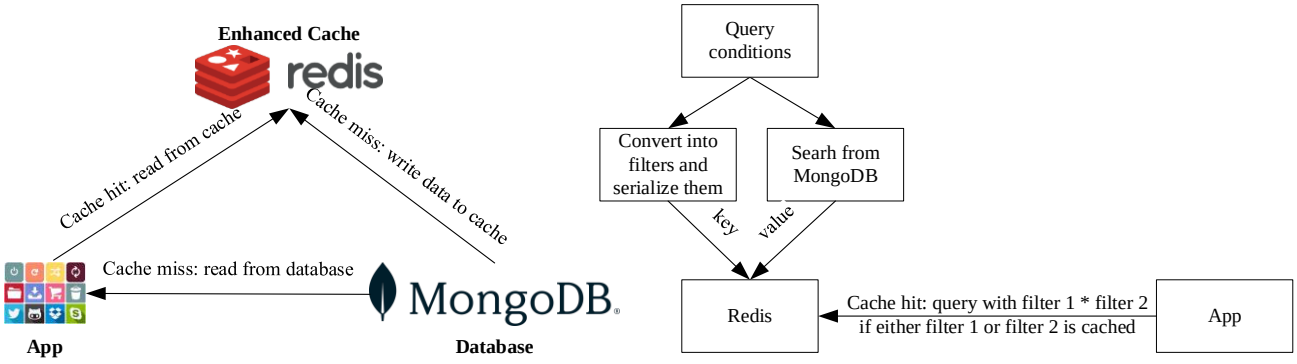
2. RELATED WORK

Database caching dramatically lowers the data retrieval latency, especially in in-memory key-value stores[8-10]. Several cache policies have been proposed to speed up queries for NoSQL databases. For instance, in [11], the authors implement two cache policies for Amazon DynamoDB: i) Redis as a cache; ii) in-memory internal caching in AWS Lambda. The evaluation showed that using Redis cache improves the response time. It is possible to implement Redis as a cache server for MongoDB. In [12], the authors propose a multi-level NoSQL cache architecture leveraging FPGA-based hardware cache together with in-kernel software cache in a complementary style. The authors in [13] implement Redis as a cache layer for Non-volatile memory (NVM).

However, in these cache policies, there is no cache hit for the query, search-within-result, though the results of the second query are a subset of the cached results of the first query. In this paper, we propose a novel cache strategy to look up data from the cached results, rather than from Mongo database.

3. THE PROPOSED CACHE STRATEGY

This section illustrates the proposed cache strategy to speed up MongoDB queries. The diagram of our proposal is depicted in Figure 1.



(a) Redis is implemented as a cache server

(b) Enhanced cache: write to cache and read from cache

Figure 1. Implementation of Redis as a cache server for MongoDB, and an enhanced cache strategy.

3.1 Implementation of Redis as a cache server for MongoDB

Figure 1-(a) illustrates how Redis is implemented as a cache server for Mongo database. When an application issues a query to fetch data from Mongo database, it first goes over to Redis to check if the data is in the cache. The application reads data directly from Redis if we have a cache hit, otherwise the application sends the query to MongoDB and populates the cache with the result.

3.2 Limitations of Redis on search-within-result

Serving as a data ingestion buffer, Redis makes MongoDB much more efficient. Redis is an in-memory non-relational key-value store. A common way to construct a cache key is to concatenate query stings in some way. We concatenate a query string into a cache key with the separator ";". For instance, the key for the query string "network AND year≤ 2019" is "title==network;year<=2020". In this way, swapping the order of query conditions is regarded as two different keys, resulting in no cache hit for the second query in a different order.

There is still room to speed up MongoDB queries, especially in search-within-result. Image a scenario where an application queries the keyword "network" from a bibliographic database, and then searches "year≥ 2020" within the results of the first query. These two search conditions can be combined into a single query "network AND year≥ 2020". We assume the cache is empty before issuing the two queries. There is no cache hit for the second query, though, obviously, the results of the second query (denoted by s_2) is a subset of the results of the first query (denoted by s_1), i.e., $s_2 \subseteq s_1$.

3.3 Our enhanced cache strategy

To construct cache keys capable of reflecting the relationship among keys, we convert a query condition into a **filter**. In functional programming, a filter is a higher-order function that takes a callback function that returns a boolean value. Unwanted elements are removed if the function returns false for an element. We convert a search with multiple conditions into a set of filters and sort them into alphabetical order. Thus, two queries in a different order is the same, increasing the possibility of a cache hit. As Redis stores data based on keys and values, we concatenate the sorted filters with the separator ";" into a string capable of being keys in Redis.

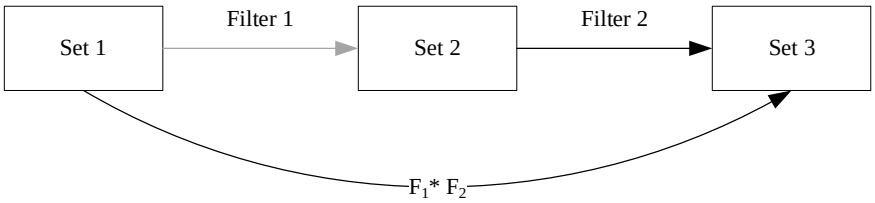


Figure 2. The illustration of search-with-result using filters as cache keys.

The illustration of search within cached results is depicted in Figure 2. An application queries data from Set 1 with filter 1 and returns the result Set 2. To query data with the combined condition filter 1 and filter 2 (can be expressed as $F_1 * F_2$ where the binary operation $*$ represents the combination of two filters), there are two approaches: i) query data with filter 2 from Set 2; ii) query data with $F_1 * F_2$ from Set 1. Apparently, the former one is faster as Set 2 is cached in Redis.

Our enhanced cache strategy can speed up queries. As depicted in Figure 1-(b), when an application issues a query, we first convert the query into a set of filters, sort them into alphabetical order, and concatenate them as a cache key k_1. The application reads data from Redis if the query key is cached. Otherwise, we construct a new key k_2 by removing the last part of k_1, and check if k_2 is cached. If yes, the application queries data from the results returning from Redis, rather than from Mongo database. Clearly, a cache hit doesn't require that a query is exactly the same as previous queries in our proposal. Such an operation goes on until there is only one part of the query key.

4. RESULTS AND ANALYSES

In this section, we evaluate the performance of our cache strategy in terms of cache hit ratio and query response time on a real dataset DBLP. In short, we make the following observations:

i) In section 4.1, we confirm the benefits of our cache strategy in search-within-result, achieving a 86.81% improvement of the average query response time.

ii) In section 4.2, we extend our experiments to a common scenario, and get a similar improvement, achieving a 71.27% improvement.

4.1 Dataset and experimental setup

Datasets. We perform our evaluation on a real dataset, the DBLP computer science bibliography[1]. It provides open bibliographic information on major computer science journals and proceedings. We store the DBLP in MongoDB rather than SQL databases like MySQL, as it provides flexible schema, high performance, and automatic scaling. There are 5,507,988 records, and the total document size is 4.14GB.

Performance metrics. To measure how effective a cache is at fulfilling requests for content, we use two metrics, cache hit ratio and query response time. A cache hit ratio is calculated by dividing the number of cache hits by the total number of cache hits and misses. The query response time is the amount of time between issuing a query request and returning query results.

Experimental setup. We evaluate the performance of our cache strategy on a MacBook Pro released in 2020 with a M1 chip, 16GB RAM, and 256GB SSD.

Configurations. The cache size is set to 1GB. The replacement algorithm we adopted in Redis is allkeys-lru (evict keys by trying to remove the less recently used keys first).

4.2 The benefits of our cache strategy for search-within-result

To evaluate the benefits of our cache strategy for search-within-result, we construct a batch of query conditions. Firstly, we extract titles from all articles, split the titles into words meanwhile remove stop words, and sort the words by the frequency from the highest to the lowest. To avoid returning abundant results with the top frequent words, we select 100 words from the 1000th word to the 1100th word as a set of query strings. The first query string is randomly chosen from the selected set. Secondly, we use publication years between 2020 and 2012 to construct 9 query conditions. For instance, the second query condition is $year \leq 2019$. In this way, we make sure no returning empty value even combining all query conditions.

We perform searching within results and record the query response time for each query for both original and our cache strategy. For instance, the first query is "network". The second operation is to search " year$\leq$ 2020" within the results of the first query, which is actually equivalent to a search with the combined condition "network AND year$\leq$ 2020". The experimental results are plotted in Figure 3. It shows the average query response time over the times of search-within-result. Not surprisingly, with the increase of times of search-within-result, the average response time remains steady using the original cache strategy (labeled as "Baseline"). By contrast, with our proposed cache strategy, the average response

[1] It is available to the public as one big XML file, https://dblp.uni-trier.de/xml/.

time decreases sharply. This is because the query has a cache hit after the last query with our cache policy. Our proposal reduces the average response time by up to 86.81% at the 10 times search-within-result.

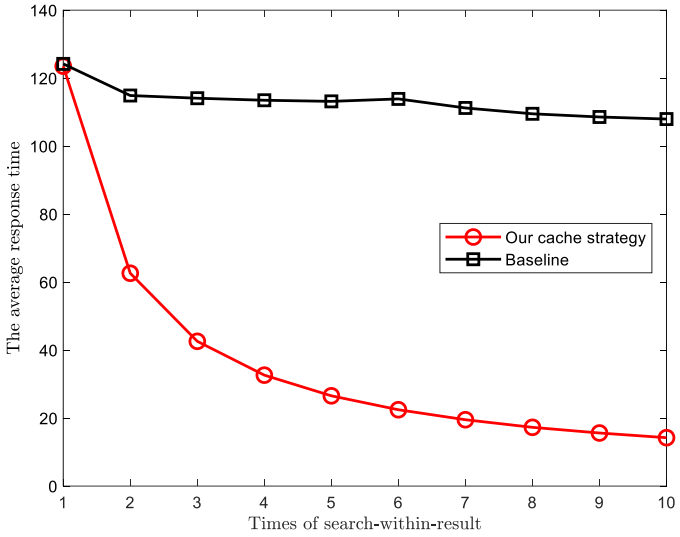


Figure 3. The average response time (unit: second) over times of search-within-result in original and our cache strategy.

4.3 The benefits of our cache strategy in a common scenario

We have confirmed the benefits of our cache strategy in the case of search-within-result. In practice, some users perform searching within results, and others may not. Some users happen to query data based on another's query. To investigate the benefits of our cache policy in a common scenario, we assume users carry on search with a query string, search-within-result once, search-within-result twice evenly. Thus, we construct three sets of query conditions: i) a set of 100 frequent words as the same as described in Section 4.2; ii) a set of 21 query conditions of publication year between 2000 and 2020, e.g., "year≤ 2001"; iii) a set of 1000 query conditions of publishers randomly chosen from the DBLP database. We perform 6000 queries and record the query response time for each query. In each query, we generate a random number from [1, 2, 3], representing search with a query string, search-within-result once, search-within-result twice respectively.

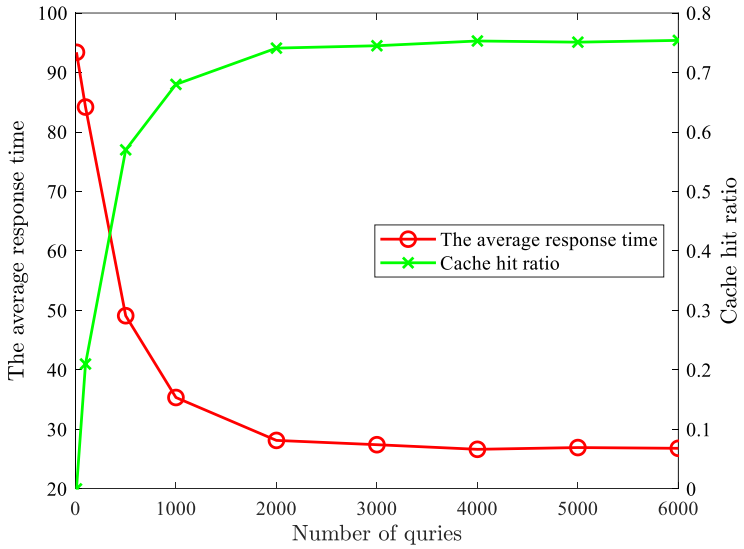


Figure 4. The average response time (unit: second) and cache hit ratio over the number of queries with our cache strategy.

Take the generated number 3 as an example, we randomly select three query conditions from the above-mentioned sets respectively. The combined query condition is something like "network (a query string from set 1) AND year≤ 2001 (from set 2) AND publisher=ACM (from set 3)".

The results are plotted in Figure 4. It shows that the average response time and cache hit ratio over the number of queries using our cache strategy. In the beginning where the number of queries equals 10, the average response time is quite high because of no cache hit. With the increase of the number of queries, the probability of cache hit increases greatly, leading to a sharp decrease in the average response time. The average response time finally remains steady, and the cache hit ratio is also stable at 75% because of the limitation of cache size. The improvement reaches 71.27% at 6000 queries, compared with 10 queries. It is worth pointing out that the improvements might be more impressive as applications would benefit greatly from the principle of locality.

5. CONCLUSION

In this paper, we propose a novel cache strategy by converting queries into a set of filters for Redis. The key novelty of our design is that there is a cache hit in the case of search-within-result. It can improve the cache hit ratio, leading to a great decrease in the average response time. Extensive experiments show the validity of our proposal.

ACKNOWLEDGMENTS

This work is supported by Jimei University (no. ZQ2018008), the Education Department of Fujian Province (CN) (no. JAT200273), and the Natural Science Foundation of Fujian Province (CN) (no. 2019J05099). The authors would like to thank the editor and anonymous reviewers for their insightful feedback on improving the manuscript quality.

REFERENCES

[1] Statista. 2021. Amount of data created, consumed, and stored 2010-2025. (June 2021).
[2] Ivan Veinhardt Latták. 2021. Schema Inference for NoSQL Databases. (2021).
[3] S Uyanga, N Munkhtsetseg, S Batbayar, and Sh Bat-Ulzii. A Comparative Study of NoSQL and Relational Database. In Advances in Intelligent Information Hiding and Multimedia Signal Processing. Springer, 116–122. (2021).
[4] Benymol Jose and Sajimon Abraham. Performance analysis of NoSQL and relational databases with MongoDB and MySQL. Materials Today: Proceedings 24 (2020), 2036–2043. https://doi.org/10.1016/j.matpr.2020.03.634. (2020).
[5] Pranoy Ranjan Bhowmick. Overview of NOSQL Databases. (2021).
[6] Cristina Evangelista. Performance modelling of NoSQL DBMS. In Proceedings of the 2020 OMI Seminars (PROMIS 2020), Vol. 1. Universität Ulm, 6–1. (2021).
[7] Rob Reagan. Redis Cache. In Web Applications on Azure. Springer, 257–300. (2018).
[8] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast key-value stores: An idea whose time has come and gone. In Proceedings of the Workshop on Hot Topics in Operating Systems. 113–119.
[9] Lucas Lersch, Ivan Schreter, Ismail Oukid, andWolfgang Lehner. Enabling low tail latency on multicore key-value stores. Proceedings of the VLDB Endowment 13, 7, 1091–1104. (2020)
[10] Kefei Wang, Jian Liu, and Feng Chen. Put an elephant into a fridge: optimizing cache efficiency for in-memory key-value stores. Proceedings of the VLDB Endowment 13, 9. (2020).
[11] Bishakh Chandra Ghosh, Sourav Kanti Addya, Nishant Baranwal Somy, Shubha Brata Nath, Sandip Chakraborty, and Soumya K Ghosh. Caching Techniques to Improve Latency in Serverless Architectures. In 2020 International Conference on COMmunication Systems NETworkS (COMSNETS). 666–669. https://doi.org/10.1109/COMSNETS48256.2020.9027427. (2020).
[12] Yuta Tokusashi and Hiroki Matsutani. Multilevel NoSQL Cache Combining In-NIC and In-Kernel Approaches. IEEE Micro 37, 5 (2017), 44–51. https://doi.org/10.1109/MM.2017.3711653. (2017).

[13] Jiaqiao Zhang, Zhili Yao, and Jianlin Feng. 2021. NCRedis: An NVM-Optimized Redis with Memory Caching. In Database and Expert Systems Applications, Christine Strauss, Gabriele Kotsis, A. Min Tjoa, and Ismail Khalil (Eds.). Springer International Publishing, Cham, 70–76. (2021).