ISSN:0975-9646

Enhancing the Query Performance of NoSQL Datastores using Caching Framework

Ruchi Nanda[#1], Swati V. Chande[*2], K.S. Sharma[#3]

[#1,# 3] *Department of CS & IT, The IIS University, Jaipur, India*

[*2]*Department of CS, International School of Informatics & Management, Jaipur, India*

Abstract—**The advent of cloud computing technology has facilitated storage of the huge amount of data on the servers. The multi-tenancy feature of cloud-based systems put additional load on database servers and the retrieval of data becomes a major issue. There is need to improve the techniques of data retrieval from cloud databases, so that the query processing time can be further reduced. Caching is one of the prominent techniques to improve the query processing time of SQL based systems. In this paper, a framework based on caching mechanism is proposed, that aims to reduce the query processing time of NoSQL datastore queries. It caches frequently issued queries and results on the database tier of the server. For cache replacement, a simple least recently used policy is implemented. Experiments conducted on HBase show that in order to reduce the query processing time of the NoSQL datastores queries, caching is a viable alternative.**

Keywords— ***Caching, Cloud NoSQL datastores, Query Performance, Query Processing***

I. INTRODUCTION

With the emergence of cloud computing, in recent years, large scale database management systems have been come into sight such as Google's Bigtable [1], Yahoo's PNUTS [2] or Amazon's Dynamo [3]. The performance of these databases is the major area under discussion. The vast amount of data needs to be retrieved in less time.

In cloud environment multiple tenants simultaneously make request to the database, which causes heavy load on the database servers. Hence, the query processing time of the queries increases. Caching is one of the solutions that can improve the performance of databases and maintains the time. Queries and their results can be cached for future retrieval of results for similar or partially-similar queries. It is useful to store the frequently accessed queries and the queries whose re-evaluation in query processing are time-consuming [4].

The proposed technique caches the frequently fired queries and results for fast retrieval in future. The cache-replacement policy, LRU is executed in case the cache is completely occupied and cannot handle more queries. The main contribution of this paper is in explaining the proposed caching framework for fast retrieval of data from cloud-based systems. The experimental evaluation shows the overall performance improvement of database using this framework.

The remainder of the paper is organized as:

The section II briefs the related work done by the researchers in developing caching algorithms. The section III briefs about the important concepts of caching. Section IV explains the working of the proposed framework and the various assumptions taken. The section V explains the experimental setup along with the datasets and the query sets used. The parameters used for evaluation, experiment and the results obtained are also discussed. Finally the section VI explains the conclusions drawn from the work.

II. RELATED WORK

Caching is one of the techniques that has been broadly studied in the area of mobile applications, operating systems, peer-to-peer systems and content delivery networks for reducing the load on the servers as well as the response time of the queries ([5]-[9]).

In cloud environment, the authors believed that data cache should become a cloud service and shared by multiple tenants. In [8] a simple multi-tenant data cache scheme named BLAZE was designed based on the CLOCK replacement algorithm that exploits utility-based cache partitioning between tenants. A cache system for frequently updated data in the cloud (CSFUD) was proposed in [9]. It caches the data by key-value pairs. Their work focused on the index and cache maintenance problem and considered response time of the system. A popularity-based small front-end cache mechanism which caches the most popular items without querying the back-end nodes was presented in [10]. It facilitates even distribution of the load across the back-end nodes. They considered that the node selection is based on the random data partitioning across all cluster nodes. This selection is entirely transparent to the user. They do not consider the systems like BigTable and HBase and the query processing time factor was not considered. In [11] the concepts of database caching were applied to the SQL based systems and discussed that the in-memory mediums provide quicker access time.

The authors worked on any of the three caching mechanisms: static-data caching [15], database-caching [11] or query caching [7]. These studies had made vast contributions in maintaining the cache. The authors vary the parameters based on the environment and application type. These parameters include, cache size, query set size, database size, type of queries system can handle. Generally

the caching techniques are applied to structured databases [11] or applied to cloud but as a cloud service technique [8]. The techniques used [9] were applied to update data queries on document-oriented NoSQL databases, and relational databases like Oracle and DB2. The related work reveals that the query parameters like, cache size, types of queries, cache replacement policies can be further improved to increase the data retrieval performance, by taking into consideration NoSQL datastores.

III. CACHING

Caching refers to the storage of processed data in the cache memory for fast and easy access. The data that can be cached includes a query, intermediate results, final results or part of the database. The purpose of caching is, to avoid repeated processing of the same query and to get the previously processed results. It can be done at client-tier, application-tier and database-tier. The caching of static data is done at client-tier. The database tier can cache both the static and dynamic data.

It is useful to store frequently accessed queries and the queries whose evaluation in query processing time takes much time. For example, the join operation is considered to be time-consuming in SQL based-systems. The re-evaluation of those queries needs to be done again and again, in case, similar queries approach the database. The queries which involve join operator can be cached for future retrieval so that the query processing time is reduced.

The cache efficacy is calculated in terms of cache hits and misses. Cache hit occurs when the query is found in cache and cache miss occurs if the query is not present in the cache. However, the data needs to be updated in the cache if the source data in the database gets changed. The outdated data can lead to inconsistent results.

The data that has been recently used by the application are likely to be used again. Hence the location of the data needs to be changed such that it can be accessed easily and reflects the most recently used data item. It is differentiated from the data which has not been used for long and can be evicted in case of any requirement.

When the cache exceeds its maximum size, and in case, cache-miss query approaches, the query which has not been used for long is replaced by a new query and result. The page replacement algorithm is implemented which decides which query and its results has to be evicted.

The page replacement algorithms help for better operation of cache. The Least Recently Used (LRU) policy is used generally, as it is very simple. It keeps the query in the cache according to the time it has been accessed, so that the queries that have not been used for long are considered for eviction from the cache. The new queries, in this case, can be simply appended in the cache.

IV. PROPOSED FRAMEWORK

The purpose of the framework is, to, improve the query processing time of the cloud queries by caching the queries and their results on the database tier of the server. The caching is done so that, in future, if the same query occurs, then it will not be again processed by the query processor and can be directly accessed by the cache.

All the clients hence, share the same cache. If the cache becomes full, the least recently used queries and their results are evicted from the cache.

Clients directly access this framework for their query and results. In case cache-hit occurs, the result is retrieved from the cache. The cache maintains this frequently accessed data for easy access. In case of cache-miss, the query results are obtained from the database, and the results are then appended in the cache. If the cache is full, the framework evicts the least recently used query and its results, so that new query and its results can be cached.

Fig. 1 shows the flowchart describing the working of the framework. In flowchart, Q represents query and R represents Result.

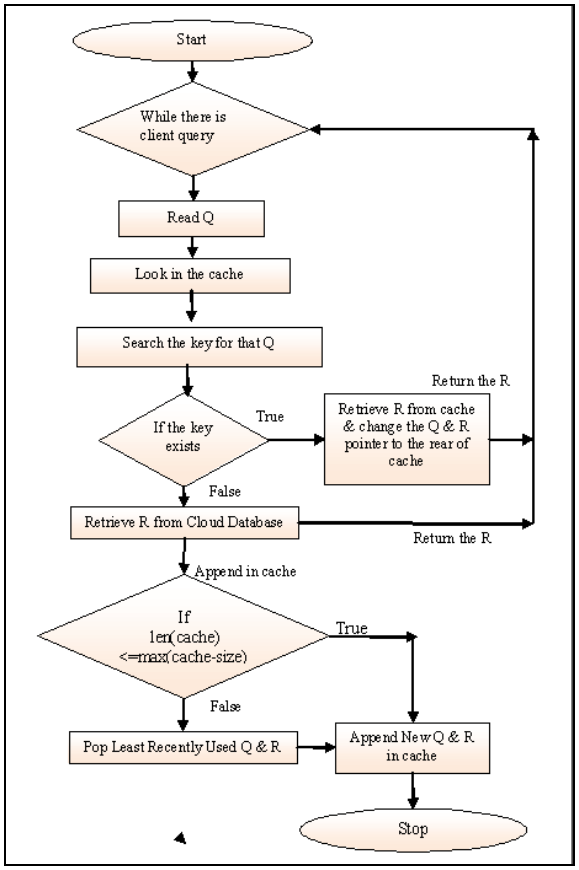

Fig. 1 Flowchart of the Proposed Framework

This framework helps to decrease the number of interactions directly with the cloud database. The working of the framework is given below:

The server receives the queries from the client. This framework search the query in the cache, if the query is cache-hit, then the result is retrieved from the cache and submitted to the user. The query and result pointer changes to the rear of the cache. This is done, so that, it becomes the most recently used query.

If the query is cache-miss query, then the result is retrieved from the database and submitted to the client. The query and result are appended in the cache. Before appending, framework checks the current size of the cache. If the current cache size is smaller than the maximum allocated cache-size, the query and result is directly appended. But if the cache attains its maximum size, then

least recently used algorithm is executed, that pop-up the query and result from the front-end of the cache. The new query and result is appended and the pointer points to the rear of the cache.

The basic assumptions for the implementation of the framework are:

1) The cache is of enough speed and never becomes the performance problem for the system.
2) The cost consideration is assumed to be uniform, regardless of the type of queries.
3) The static databases are considered, as we will confine our concentration to reduce the query processing time of the queries and hence, ignore the updates.
4) The scan queries in HBase having filter conditions 'SingleColumnValueFilter' are considered as these are considered as the most time-consuming.
5) If the cache attains its maximum size, the least recently used query and result is evicted, so that the new query and result can be appended.
6) The cache size is of 15% of the database-size, as it is being used by previous researchers. The optimal buffer cache size is between 10% and 15% of the database size and in some cases up to 20% of the database size [12].

V. EXPERIMENT & RESULTS

A. Experimental Set-up

The experiment is performed on a customized IaaS cloud server having processor (Intel(R) Core (™) 2 Duo CPU P9400 @ 2.40GHz; 4 GB RAM). Apache HBase database is used to check the performance of our framework. The framework is implemented in Python 2.7, on a system having specification shown in Table I. Python applications connect to HBase using Happybase [13].

HappyBase is a Python library that offers application developers a rich set of APIs that can be used to interact with HBase. The experiments were run on a steady environment, where the system had a GNOME text editor (gedit) and one virtual machine running. For the experiments, three runs have been performed in order to minimize the environmental factors [14].

TABLE I

SPECIFICATION OF HOST MACHINE

Processor	Intel(R) Core(™)2 Duo CPU P9400 @ 2.40GHz ; 4 GB RAM
Platform	Red Linux Enterprise Edition 6 (RHEL6)
Hypervisor	KVM
Language used	Python Interpreter 2.7
Database	Apache HBase-0.94
API	HappyBase-0.3
Interface used for HBase	Thrift server

B. Datsets and Query Sets Used

In order to evaluate the framework, BlogInfo repository has been used. It contains the information of the blogs that includes title, author and the date on which the blog is created. It also contains a short description of the contents of the blog.

The query sets are prepared such that they match in 100%, 60% and 20% of exact cache-hit and the remaining in cache-miss.

C. Evaluation Parameters

To check the performance of the proposed framework, the query processing time in case of cache-hit and cache-miss queries are evaluated. The results are compared with the direct HBase system; hence the overall time taken by the system is also calculated. The following parameters are evaluated in case of cache-hit query:

1) The query processing time (QPT) in case of cache-hit query is the time needed to retrieve the data from the cache.
2) The system time (ST) is summation of all the phases. It includes the time taken to process the query i.e cache retrieval time, time taken to change the pointer and make the query as the recently used item and sent the result to the client.
3) The cache background processing time is the time that cache takes to process the query and keep it as the recently used item. It works as a background process.
4) Time taken to send is the time to build the result and sent to the client. It does not include any network time.

The following parameters are evaluated in case of cache-miss query:

1) The query processing time in case of cache-miss query is the time needed to retrieve the data from the database.
2) The system time is summation of all the phases. It includes the time taken to process the query i.e database retrieval time, time taken to insert query and result in the cache and time taken to build and sent the result to the client.
3) The database retrieval time is the time taken to get the result from the database.
4) The cache background insertion time is the time taken to append the cache-miss query and its result in the cache. It works as a background process.

D. Experiments

An experiment is conducted to check the impact of varying percentage of cache-hit queries on the query processing time. The purpose of this is, to check whether caching of queries and result, results in the reduction of query processing time in NoSQL based datastores.

The cache is warmed by chosen 5 queries. The experiment is conducted using small sets of queries of different percentage of exact cache-hit. The percentages taken are 100%, 60% and 20%. One query set is prepared for 100% cache-hit queries, since the cache is warmed by 5

queries. 3 query sets are prepared for 60% and 20% cache-hit percentage, so that the experiment is run 3 times for each cache-hit percentage and the possibilities of any variation caused by environmental factors are neglected. Each query set contains 5 queries. In all, 7 query sets have been prepared and 35 queries are executed by the framework. In each run of the experiment, the processing time of each query and the time taken by the system to process the query are calculated and the averages are calculated. Table II shows the average query processing and system times for different percentages of cache-hit queries. The same query sets are run in the system having direct HBase, for the comparison purpose. Table also depicts the performance improvement of cache in percentage.

TABLE III

AVERAGE QUERY PROCESSING AND SYSTEM TIME IN EACH RUN

Run-I			
Cache-hit %	**100%**	**60%**	**20%**
QPT (sec)	0.0000	0.0116	0.0192
ST: Cache (sec)	0.0005	0.0120	0.0197
ST: HBase (sec)	0.0558	0.0434	0.0356
Run-II			
Cache-hit %	**100%**	**60%**	**20%**
QPT (sec)	0.0000	0.0134	0.0187
ST: Cache (sec)	0.0004	0.0136	0.0190
ST: HBase (sec)	0.0558	0.0418	0.0395
Run-III			
Cache-hit %	**100%**	**60%**	**20%**
QPT (sec)	0.0000	0.0122	0.0197
ST: Cache (sec)	0.0005	0.0127	0.0199
ST: HBase (sec)	0.0558	0.0434	0.0388
Average of 3 Runs			
Cache-hit %	**100%**	**60%**	**20%**
QPT (sec)	0.0000	0.0124	0.0193
ST: Cache (sec)	0.0005	0.0128	0.0195
ST: HBase (sec)	0.0558	0.0426	0.0363
% improvement (using cache)	-99.10	-69.95	-46.24

Figure 2 shows the graphical representation of the query processing time in each run of the experiment. The x-axis shows the cache-hit percentage and y-axis shows the average query processing time of the queries.

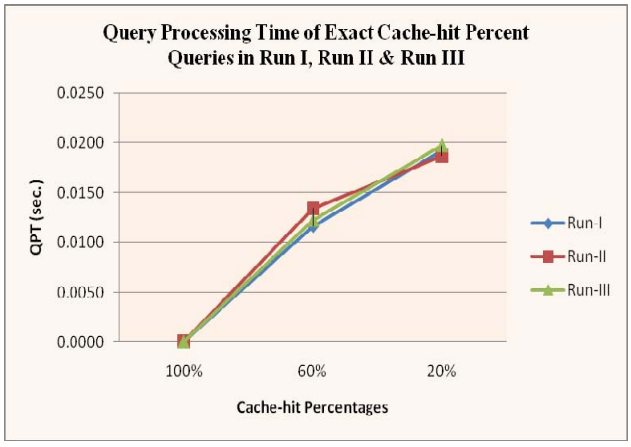

Fig. 2 Query Processing Time of Exact Cache-hit Percent Queries in Run I, Run II & Run III

E. Results & Discussion

It is observed that the average query processing and system time is almost negligible in case of 100% cache-hit queries. It increases with the decrease in the percentage of exact cache-hit queries. The average ST of the queries executed directly on HBase decreases from 100% to 20%. This reduction in the system time is due to the partial caching in database system and operating system [11].

It is seen that the performance improvement in the system using cache framework over the system having HBase at 100%, 60% and 20% are 99.10%, 69.95% and 46.24% respectively.

These observations in average query processing time are visible in Fig 3. The x-axis shows the cache-hit percentage and the y-axis shows the average query processing time in seconds.

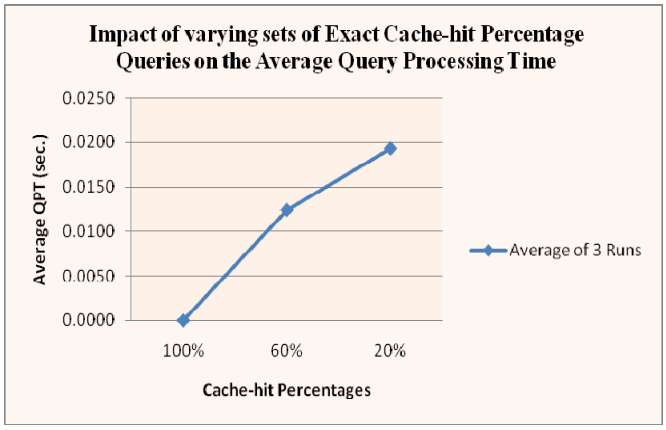

Fig. 3 Impact of set of exact cache-hit percentage queries on the Average Query Processing Time

The observations in the system time are shown graphically in Figure 4. The x-axis represents the cache-hit percentage and the y-axis shows the average system time in seconds.

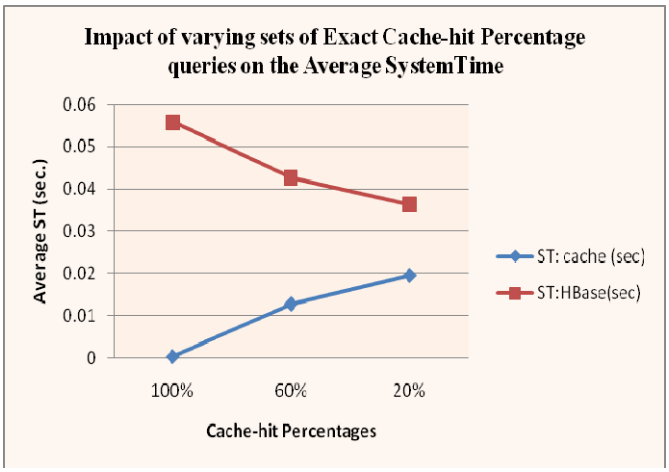

Fig. 4 Impact of set of exact cache-hit percentage queries on the Average System Time

VI. CONCLUSIONS AND FUTURE WORK

The framework proposed in the study is based on caching mechanism. In cloud systems, it caches the frequently accessed queries and results on the database tier. The purpose of the framework is to reduce the query processing time of multiple queries on cloud database. The applications that access static databases or the databases in which the updations are rarely done are the best candidates for using this framework. It has been evaluated systematically and the experiment proves that caching is a viable alternative for NoSQL datastores. The query processing and system time is drastically reduced for the exact cache-hit queries. It presents an optimistic direction for future research. The work can be extended by working on the parameters that affect the caching mechanism. The parameters that need consideration are:

- Cache-size that can be calculated by experimentation
- Increase in the database-size and query set size
- Type of queries to be processed
- Type of page replacement algorithm.

REFERENCES

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26(2), p.4, 2008.

[2] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," in *Proc. VLDB Endowment*, 2008, vol. 1(2), pp.1277-1288.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store" *ACM SIGOPS Operating Systems Review*, vol. 41(6), pp. 205-220, 2007.

[4] D. Wessels, Web *caching*, O'Reilly Media, Inc. 2001.

[5] K. Raichura, N. Padhariya, and K. Atkotiya, "Cache-Based Query Optimization In Mobile Ad-Hoc Networks," *International Journal of Technology Enhancements and Emerging Engineering Research*, vol. *3*(2), pp.226-232, 2014.

[6] O. D. Sahin, A. Gupta, D. Agrawal, and A. El Abbadi, "A peer-to-peer framework for caching range queries," in Proc. *Data Engineering,2004.* IEEE, pp. 165-176.

[7] H. Ding, A. Yalamanchi, R. Kothuri, S. Ravada, and P. Scheuermann, "QACHE: query caching in location-based services," *Progress in Spatial Data Handling.* Berlin, Heidelberg: Springer, 2006, pp. 99-116.

[8] G. Chockler, G., Laden, and Y. Vigfusson, "Data caching as a cloud service," in *Proc. of the 4th International Workshop on Large Scale Distributed Systems and Middleware ACM, 2010*, pp. 18-21.

[9] F. Dong, K. Ma, and B. Yang, "Cache system for frequently updated data in the cloud," *WSEAS Transactions on Computers*, vol. 14, pp. 163-170, 2015.

[10] B. Fan, H. Lim, D. G Andersen, and M. Kaminsky, "Small cache, big effect: Provable load balancing for randomly partitioned cluster services," in *Proc. of the 2nd ACM Symposium on Cloud Computing, 2011*, p. 23.

[11] B. J. Sandmann, "Implementation of a Segmented, Transactional Database Caching System," *Journal of Undergraduate Research at Minnesota State University, Mankato*, vol. *6*(1), pp. 21, 2014.

[12] A. N. Packer, *Configuring and tuning databases on the Solaris platform*, Prentice Hall PTR, 2001.

[13] Userguide HappyBase [Online]. Available: http://happybase.readthedocs.io/en/latest/user.html

[14] M. Perrin, "T*ime-, Energy-, and Monetary Cost-Aware Cache Design for a Mobile-Cloud Database System*", Doctoral dissertation, University of Okalahoma, 2015.

[15] Y. Bu, B. Howe, M. Balazinska, and M.D. Ernst, "The HaLoop approach to large-scale iterative data analysis," *The VLDB Journal—The International Journal on Very Large Data Bases*, vol. *21*(2), pp.169-190, 2012.