A Study on Cache Consistency Maintenance in Cloud Computing Databases

Dr.C.Inbam
Assistant Professor
PG Dept of Mathematics
Govt Arts College
Melur, Madurai Dt.

Abstract
The increasing use of cloud computing systems improves the development of high quality cloud based applications. It provides inexpensive computing resources on the basis of pay-as-you-use. More and more users migrate their applications to cloud environments as it provides many data services like database as a service (DaaS). Today, we are increasingly more hooked into critical data stored in cloud data centers across the globe. To deliver high-availability and augmented performance, different replication schemes are needed to maintain consistency among replicas. With classical consistency models, performance is necessarily degraded, and thus most highly-scalable cloud data centers sacrifice to some extent consistency in exchange of lower latencies to end-users. To tackle this inherent and well-studied trade-off between availability and consistency, the utilization of CCM (Cache Consistency Maintenance) Scheme, a completely unique consistency model for replicated data across data centers with framework and library support to enforce increasing degrees of consistency for various sorts of data is studied in detail.

Keywords: Cache, Consistency, Cloud, DaaS.

Introduction
The great success of Internet achieved during the last decade has brought along the proliferation of web applications which, with economies of scale (e.g., Google, Facebook, and Microsoft), can be served by thousands of computers in data centers to millions of users worldwide. These very dynamic applications got to achieve higher-scalability so as to supply high availability and performance. Such scalability is usually realized through the replication of knowledge across several geographic locations (preferably on the brink of the clients), reducing application server and database bottlenecks while also offering increased reliability and durability. Highly-scalable cloud-like systems running round the world often comprise several levels of replication, specially among servers, clusters, inter-data centers, or maybe among cloud systems. More so, the advantages of geo-distributed micro-data centers over singular mega-data centers have been gaining significant attention (e.g., [1]), as, among other reasons, network latency is reduced to end users and reliability is improved (e.g., in case of a fire or a natural catastrophe, or simply network outage).
With this current trend that brings higher number of replicas, more and more data needs to be properly synchronized, carrying out the need of having smart schemes to manage consistency while not degrading performance. In replication, the consistency among replicas of an object has been handled through both traditional pessimistic (lock-based) and optimistic approaches [2]. Pessimistic strategies provide better consistency but cause reduced performance, lack of availability, and do not scale well. Where optimistic approaches rely on eventual consistency, allowing some temporary divergence among the state of the replicas to favor availability and performance.
Since it is not possible to fully have the best of both approaches in a distributed environment with arbitrary message loss, as stated in the CAP theorem [3], it is envisioned that more are often done to approximate consistency from availability. One path not yet significantly explored, which is addressed in this work, consists of wisely and dynamically strengthen and weaken cache consistency in accordance to the importance of the data being replicated.

Distributed Cache and its Maintenance
It represents an in-memory, transactional and distributed database cache for: i) temporary storing frequently accessed database items; and ii) keep tracking of which items need to be replicated Caching: The CCM (Cache Consistency Maintenance) framework comprises a distributed and transactional in memory cache system to be used at the application-level. It has two main purposes:
Keep track of the items waiting for being fully replicated and
Temporarily store both frequently used database items and items within the same locality group (i.e., pre-fetch columns of a hot row), in order to improve read performance.

Specifically, this type of cache stores partial database tables, with associated QoS constraints, that are very similar from the ones in the underlying database, but with tables containing many less number of rows.

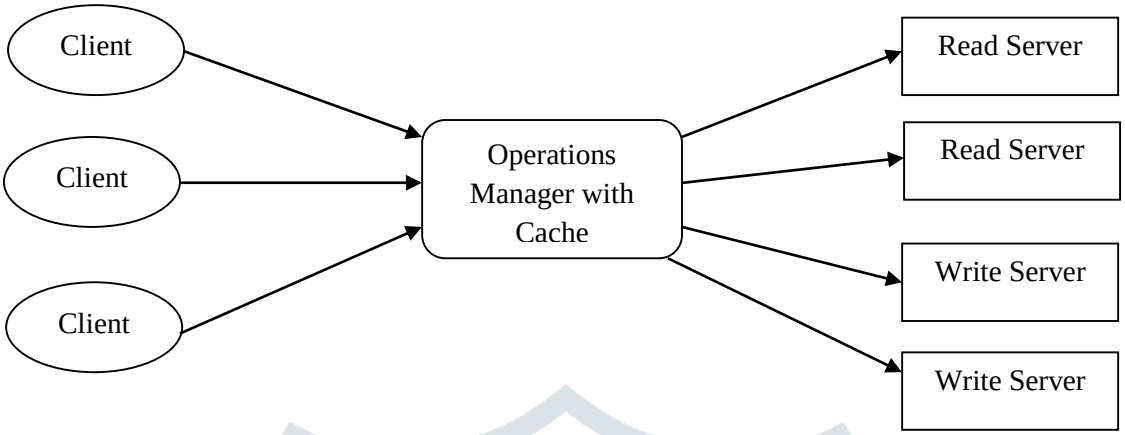

Figure 1. Operations Manager with Cache

Moreover, the CCM cache is completely transparent to applications; it guarantees transactional consistency with the underlying database and the data is automatically maintained and invalidated, relieving developers from a very error-prone task. This cache can be spanned across multiple application servers within the same data center, so that it can grow in size and take advantage of the spare memory available in the commodity servers. Although the work is distributed, it still gives a logical view of a single cache. The partition of knowledge follows an horizontal approach, meaning that rows are divided across servers and therefore the hash of their identifiers works as key to locate the servers in which they should be stored. Hence, this cache is optimized for our target database systems, since rows constitute indexes within the multi-dimensional map and each query must contain a row identifier (apart from scans).

Furthermore, each running instance of the CCM cache knows all others running on neighbor nodes. Cache is a part of Operations Manager and its existence is transparent to the Read and Write servers. The processor issues Read and Write requests within the same manner. If the required data is available in the cache, it is called a Read or Write hit. If the data is obtained from the cache, it is called Read Hit. Cache has a selective replica (frequently used data items) of the contents of the Write server. The contents of the cache and the Write Server may be updated simultaneously. This is the write-due protocol. Update the contents of the cache, and mark it as updated by setting a touch referred to as the dirty bit or modified bit. The contents of the most memory are updated when this block is replaced. This is write-rear protocol.

Context cache

Context users request context to the context agent about a few particular entity and scope by sending a ContextML encoded query. The agent forwards the query to an appropriate context provider which can satisfy this request. When the query-satisfying context information is given there , the provider sends the context response to the agent. If there is no caching facility, the agent simply forwards the query to the querying user. The Context Provisioning Architecture utilizes a caching component that caches recently received contextual data in response to context queries, additionally to forwarding the response to the querying user. The context data remains within the cache for the validity period unless it's replaced by newer context of an equivalent scope/entity or has got to be removed to free the cache thanks to cache size limits. The query processing and notification operations from the context agent's point of view are described in Algorithms 2 and a couple of respectively.

Context Agent Query Algorithm

Context agent query processing

WHERE $P = \{P_1, P_2, \ldots, P_3\}$, P is the set of all providers in the system

WHERE a query $Q = \{I_q, I_e, I_s, I_{CxC}, Q_p\}$

I_q is the query ID, I_e is the entity ID, I_s is the scope ID, I_{CxC} is the user component's ID and Q_p consists of other query parameters.

WHERE $T_q = \{I_{q_i}, I_{CxC_i}\}$

T_q is a table where the agent stores the query ID to user ID mappings of the form $\{I_q, I_{CxC}\}$

subscribe(Q) # Query arrives at the agent *record*(T_q,I_q,I_{CxC}) # Query is recorded in the queries table

CXT_f=*searchCache*(Q,I_e,I_s) # See if cache can satisfy the query

if CXT_f then

notify(I_{CxC},CXT_f) # Notify the user in case of cache hit

incrementUseCount(CXT_f) # Increment the use count of the particular item

else

P_s=*lookup*(P,I_e,I_s) # Agent looks up an appropriate provider

query(Q,P) # And forwards the query to that provider

end if

Context Agent Notification Algorithm

Context agent notification processing

WHERE $T_q = \{I_{q_i}, I_{CxC_i}\}$ # T_q is a table where the agent stores the query ID to user ID mappings of the form $\{I_q, I_{CxC}\}$

WHERE T_{Ins} is the cached item insertion time

WHERE T_{Exp} is the cached item's validity expiry time

WHERE CXT_{in} is the cached item's use count *publish*(I_q,CXT_p) # Context response arrives from the provider

storeInCache(CXT_p,T_{Exp},T_{Ins}) # Store the context item in the cache

I_{CxC}=*resolve*(T_q,I_q) # Find out which user requested this context item

notify(I_{CxC},CXT_p) # Notify the user

3.3. Context Cache Insertion and Replacement Algorithm

Context cache insertion and replacement procedure (storeInCache)

WHERE T_{Ins} is the cached item's insertion time

WHERE T_{Exp} is the cached item's validity expiry time

WHERE CXT_{in} is the cached item's use count

WHERE *policy*=`*L U* ″∨`*O F* ″∨`*S E* ″# The cache replacement policy

if *filledSpace*<*maxSpace* then

$insert(\mathcal{MD}5(I_e||I_s), CXT, T_{Exp})$

else if *policy*==`*S E* ″then CXT_{Rem}=*Minimum*(T_{Exp}) # Select the item with the soonest reaching *expiry* time*remove*(CXT_{Rem})end if

else if *policy*==`*O F* ″then CXT_{Rem}=*Maximum*(T_{Ins}) # Select the item with the oldest insertion time *remove*(CXT_{Rem})end if

else if *policy*==`*L U* ″then $CXT_{Rem} = Minimum(CXT_{in})$ # Select the item with the least usage count

end if

end if

Evaluation of Cache Performance

The cache performance is evaluated with different workloads performing 1,00,000 operations each: The cache size was 30% of the size of each workload.

I) 50% reads and 50% and writes (sessions recording recent actions)

II) 95% reads 5% write mix (photo tagging);

III) 100% reads;

IV) read latest workload (user status updates on social networks);

V) read-modify-write (user database activity).

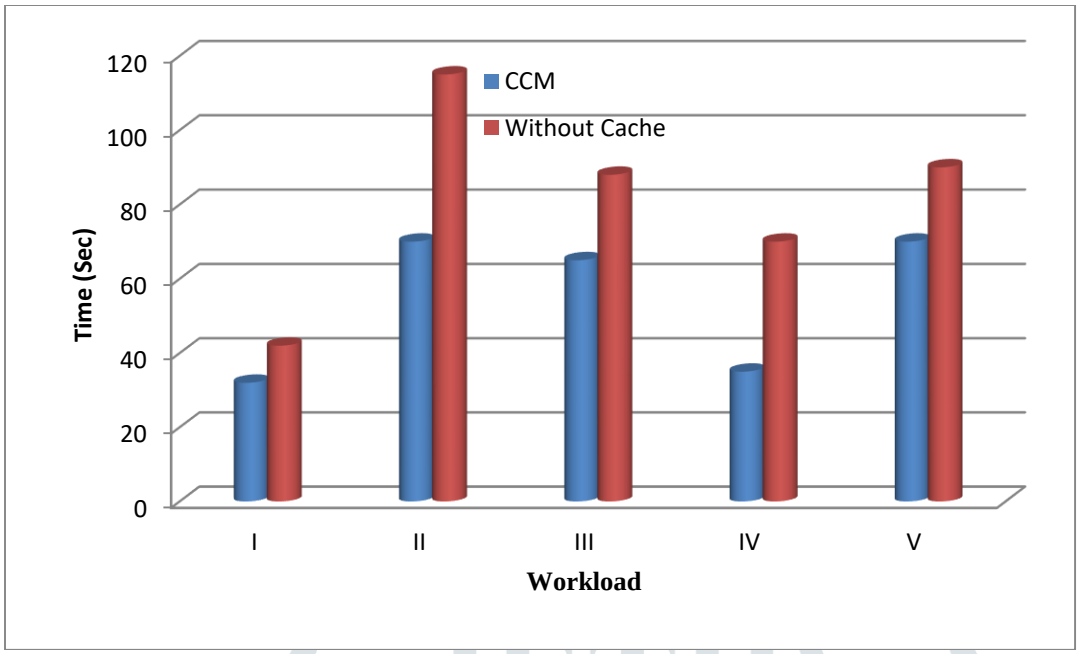

Figure 2. With and Without Cache Performance

Figure 2 shows that CCM mechanism is effective and reduces latency and communication hops. The workload IV shows best results since the most recently inserted records are read and LRU is the cache discharge policy. For the other workloads, the results were good, between 23-35%. I and II gives extreme results. It is important to note that the cache size and discharge policy will impact the results; Hence, the cache size and discharge policy are the important parameters that must be adjusted to better suit the jobs. The average cache hit rate for the experimented workloads was about 51%, revealing that CCM cache can significantly improve performance (77% when compared to others). It avoids expensive searches to the DB memory for a set of developments.

Conclusion

This paper studied an important cache consistency technique capable of enforcing different degrees of cache hit rate, according to the semantics of the data in cloud database. CCM framework is implemented and evaluated as architecture. It is useful for improving QoS, thereby reducing latency for a set of workloads. To the best of the studies, database replication offers a flexible control of consistency and provides high-availability without compromising performance. CCM improves cache hit rate and reduces latency.

REFERENCES

1. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, pp. 143–154. ACM, New York (2010)
2. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (2006)
3. Lu, Y., Lu, Y., Jiang, H.: Adaptive consistency guarantees for large-scale replicated services. In: Proceedings of the 2008 International Conference on Networking, Architecture, and Storage, pp. 89–96. IEEE Computer Society, Washington, DC (2008)
4. Yu, H., Vahdat, A.: Design and evaluation of a continuous consistency model for replicated services. In: Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation, OSDI 2000, p. 21. USENIX Association, Berkeley (2000)
5. Gao, L., Dahlin, M., Nayate, A., Zheng, J., Iyengar, A.: Application specific data replication for edge services. In: Proceedings of the 12th International Conference on World Wide Web, WWW 2003, pp. 449–460. ACM, New York (2003)
6. Kraska, T., Hentschel, M., Alonso, G., Kossmann, D.: Consistency rationing in the cloud: Pay only when it matters. PVLDB 2, 253–264 (2009)
7. Sivasubramanian, S., Pierre, G., van Steen, M., Alonso, G.: GlobeCBC: Contentblind result caching for dynamic web applications. Technical Report IR-CS-022, Vrije Universiteit, Amsterdam, The Netherlands (2006)