Data Cache with Distributed Cache: A Design Approach

Shah Imran Alam[#1], Samar Wazir[*2], Aqeel khalique[#3], Syed Imtiyaz Hassan[#4]
Department of Computer Science & Engineering
Jamia Hamdard (Hamdard University), New Delhi, India

Abstract

Caching techniques has helped developers to deliver applications that are capable of fast turnaround time which otherwise could have been much slower and under-performed software solutions, less worthy of user's appreciation. Caching can typically be used at both hardware and software levels with the same ultimate goal of either achieving higher throughput or higher latency or both together. Limiting the subject to software level cache, the caching techniques could further be introduced in one of the two categories namely web cache and data cache. While web cache is often defined in the context of a browser which is a client-side application, the data cache is defined in the context of caching needs of a data extensive application. In terms of a database management system, it means a cache provisioned at the database services itself whereas, in the context of the application, it means the cache that spans through layers of the application, more precisely termed as tiers in a multi-tier application that is designed to cache an already queried data. The requirement of frequent data access in high volumes, in distributed applications, drives the need for more capable infrastructure towards building a caching framework. In this paper, we focus our discussion on data cache requirements of a distributed application and the key design factors that distinguish a distributed cache as an elegant cache service provider plugin to such distributed applications. We also propose a simplistic design that could be used to implement the core of a custom distributed cache.

Keywords

Data Cache, Distributed cache, caching strategies, Eviction policy, Custom cache.

I. INTRODUCTION

Databases are usually the most difficult and costly off the shelf components of an application to scale up in the software architecture. Generally, distributed applications itself have a large volume of caching requirements if they are data intrinsic applications. The following are the key factors that make the baseline reasons for the use of caching as a solution in such applications.
Factors that pushes the designers to consider data caching in distributed applications

A. *To improve upon the performance of the application.*

The applications which largely depend on database operations, especially very frequent read operations on data set, based on some access pattern are the ideal candidate to go as cache elements that could save on the query time on next access request. The query time directly from the database largely depends on input-output operations which have a high turnaround time and also depends on the two ways, network communication time between the data access layer and the database.
Normally the data seek time is more expensive part of the communication because of slow input-output speed but in the case of multiple queries that are executed as part of a single transaction, or a complex join operation also could become a performance bottleneck due to the network communication delay.

B. To reduce the database load.

Scaling up the database itself is a costly affair and if there is any such requirement then the data cache could be helpful, as it may limit the number of requests actually being served from the database directly. So, when we consider this factor we are not just optimizing the performance but also on the cost of database scalability, hence gaining a dual advantage.

C. To reduce web-service lookups.

Not only databases are the only components in distributed application that are queried for data access but there could be decoupled services running on the cloud or even a network in case the service provider component is implemented in a heterogeneous technology stack. Typically web-services are used to invoke such operations. In particular, cases where these web-services offer the lookup operations, the result set obtained becomes an important piece of information that could be requested much frequently and hence candidate to be cached.

D. To obtain greater concurrency benefits.

For the read-only operations, that are the fundamental requirements in most of the cases where caching is employed, a greater concurrency as compared the databases, could be achieved and hence add up to the performance benefits.[1]

II. INTRODUCTION TO DISTRIBUTED CACHE

A cache solution generally comes as a pluggable component that could temporarily save data in the memory instead of the file system so that the subsequent requests could be served faster. That data that is normally saved in a cache is either a data object resulting from the direct query to the file system database or could also be a result of computations leading to derived data objects.

The request scenario is that, if the required data is found in the cache then it is served directly and is termed as a 'cache hit'. Otherwise, it is called as a 'cache miss' and would require the control to direct the fetch routine to reach out to the database and data be fetched from the storage. That means, more the request could be directly fulfilled from the cache less would be the load on the database and also that would result in a lot of time-saving in the network communication and the input-output hence increasing the overall application performance. [2]

In the case of a distributed application, the distributed components run on a different Java virtual machine, assuming the case of a typical java based application. The scenario could be easily mapped in all technologies that support distributed application's development. If the cache is also distributed along with the components, then this cache distribution becomes a baseline for removing performance bottleneck that leads the application to drastically slow down resulting in an extremely poor user experience. The cache distribution could be modelled as an identical copy to gain the highest degree of performance or could be modelled as a partitioned cache to conserve the resource. [3]. The essence, in general, remains unchanged, that is, reuse the data by saving in high-speed memory that could be needed to be accessed frequently. The caching component's distribution along with the distributed component of the application adds substantial complexity, especially to ensure that the distributed cache should remain in sync and updated with the master copy which is saved in the physical storage.

There can be following ways to achieve cache level distribution:

1. ***Notification based distributed caches*** – It adopts master-slave architecture. When the master node gets an update it sends notifies the rest of the slave node. The notification policy is centralized in this case. [4]

2. ***Reconciling distributed caches*** - The synchronization policy mimics the batch update. Based on pre-configured time delay the notes get updated from the master not or reconcile themselves.[5]

III.CACHING STRATEGIES

The techniques that are used to foresee and figure out which data is stored in a faster
Storage that is a distributed cache is termed as caching strategies. These approaches can take up different paths based upon the pattern of data access and how often the data is accessed**.**[6]
Locality is one of those factors which could be used to decide upon the caching strategy. Applications which display high tendency of locality of reference phenomenon are preferred choices for performance optimization with locality based strategies.[7]
Following are the well-known locality based caching strategies.

A. Temporal Cache

Temporal locality is intended to the fetch data or resources in comparatively small temporal gaps. Temporal locality is more fit for frequently fetched and relatively less volatile information. The more is the temporal cache, the less would be the cache miss.

B. Spatial locality

Unlike a temporal cache**, s**patial locality takes the advantage of the use of data elements which are positioned close in the storage. So adjacency is the factor that takes over the charge in this scenario.

C. Sequential Cache

A linear arrangement of the data is a typical requirement to get benefitted from this Sequential locality, which is a special case of spatial locality itself. The fetch time benefit comes because of the traverse is less costly.

D. Static Cache

Once the data values are cached, they are never removed from the cache. Such cache is suited for the applications which do not need to change of data in the entire lifetime. Hence such cache implementation does not need the synchronization abilities. [8]

IV.THE WAY DISTRIBUTED CACHES WORK

A cache is comprised of a pool of data entries. Every data-entry has an information that is a duplicate of similar information in a backup store. At the point when the cache of a CPU, of a web browser or of an operating system needs to get information, probable to exist in the cache, it initially checks the cache. On the off chance that a data-entry can be found in the cache, this coveted information is utilized. This circumstance is known as a 'cache hit'.

The opposite scenario is when the cache is looked up and found not to have the required data value, is typically called as a 'cache miss'. Then, the missing data value retrieved from the physical storage is normally copied into the cache, so that it could be fetched upon the subsequent request.

During a cache miss, the CPU usually ejects some other entry in order to make room for the previously un-cached data value. The set of rules used to pick up an entry to be removed from the cache is called as cache 'replacement policy' or the 'eviction policy'. There are several proposed replacement policies in the literature. One such popular replacement policy, that is 'least recently used' (LRU), which substitute the least recently used data element.[9]A better computation factor could be the frequency as seen against the size of the data element.

Contrary to read operation, When a system writes an information to the cache, it should also write that data to the backing storage either synchronously or asynchronously. This timing decision, of when to write is configured as the 'write policy'. A 'write-through cache', results in a synchronous write whereas, in a 'write-back' or 'write-behind' cache, writes are done at a later time, most suitable in an asynchronous fashion or in a synchronous fashion as batch update routine. This policy facilitates the marking procedure of the cache locations that have been written over and are marked dirty. [10]An explicit notification could also be used to trigger the cache sync to begin to write back the data to the main storage.

Moreover, system components, other than the cache itself may also change the data elements in the storage, which also makes the data stale in the cache. The communication protocols employed by to keep the data up-to-date are termed as 'coherency protocols'.[11]

V. SCENARIO OR THE USE CASES IN WHICH DISTRIBUTED CACHE WORKS

A. To improve performance by replacing expensive reads

An application which is dependent highly on frequent database access will have frequent read calls. Such calls are expensive due to communication lag as well as input-output latency which is quite high as compared to CPU operation's time and depends on a on the business logic and data retrieval patterns, for example, locality.

B. Partitioned distributed cache

Another use case of a distributed cache is a partitioned Distributed cache, in which the part of data lives in one node while the other parts of the data are distributed among other nodes, sometimes also with a backup copy of the data. This could be useful if the amount of data is very large.

C. Complex queries and calculations result-set

In case data value comes from a combination of database processing and complex calculations or just complex computation itself, such results can be cached, and requests coming again for the same operational result can be fetched from the cache.

D. Session caching

Session Caching is the technique used to handle failover. In this case, the session is cached and duplicated across the whole of the cluster. So, if one node fails, the control is transferred to another working node and because the replicated session is there to pick up and keep the communication alive.

E. Design improvement considerations

Once it has been decided that data caching is an integral part of the architecture, choosing the right caching solution can prove to be difficult. There is always an option to implement a caching solution from scratch. Another solution is to choose one of the open-source caching products.

VI. IMPORTANT DESIGN ASPECTS

While choosing a caching solution or designing a solution from scratch, the following aspects should be considered

1) Does caching solution provide easy integration with an ORM product?

It should be easy to integrate the caching product with some of the popular ORM products such as Hibernate and Toplink, to name a few. All ORM tools map database entities to domain objects typically modeled as simple objects with attributes and corresponding getters and setters. Caching solution should be capable of being used to cache such objects returned by ORM.[12]

2) Does caching solution allow storage of objects on disk in an efficient way?

In case the memory capacity is full, the cache product should evict objects to a local disk. A typical implementation may use plain serialization whereas a good solution may use a more efficient implementation that could probably use a format that may need less memory space. The mechanism employed could also provide a technology agnostic solution as against serialization which works only for Java object. These criteria could only be considered if the possibility of failover is high and that the cache data is large. This is when the solution would require persisting cache data on a frequent basis on the disk. Such caches empowers the persistence capability

3) Is it easy to use?

A cache product should expose minimum API for the client to use. For example, a good solution may provide a single API to fetch the data from all possible integrated sources of data storage, whether it is from the local cache or from a distributed partition. The more is the complexity hidden, more are the chances of developer community adopting it fast.

Another API simplification approach could be providing a single function for putting the data in the cache and updating the data store instead of two separate functions. This typically models the façade design pattern. The caching solution may support many topologies for clustered data management by using a single API, The movement from on topology to the other remains simple. A consistent logical view is always a preferred design.

4) Is it easy to maintain?

The cache product should have proper logging facilities in order to debug the code. It should also provide a good monitoring tool that could collect and present the statics in a usable form. The monitoring system should be pluggable to custom built views and should have the capability to easily build reports.

5) Does it adhere to standards?

Standards force the solution providing vendors to follow single specification and hence the choice of migration from one vendor to the other is always open to the customers. For example, JSR107 is one of standards for cache implementation.[13]

VII. LIMITATIONS OF EXISTING SOLUTION THAT MAY MOTIVATE THE CUSTOM IMPLEMENTATION.

1. The caching solution is specific to database entities. Another caching mechanism like web caching is missing.

2. ORM based cache loader is not available for the ORM framework used in the project. It could provide a clean design and more maintainable code.

3. Eviction policies could be implemented with much fine-grained control and could be mapped to production behavior which could be different because of the nature of the application.

4. Monitoring capabilities may not have the capability to provide the type of insight required.

5. A better design consideration is needed to minimize the complexity of APIs used to integrate the application. For example for database content caching, an in-memory database could be used instead of a hashmap as it will provide optimized solutions with inherent advantages like indexing. It also provides SQL queries to be forwarded to the in-memory database in case an ORM framework is not used or we need to integrate caching solution with a legacy system.

We present here a basic Low-level design of a distributed caching solution that could be used as a guideline for the custom distributed cache implementation

A. Class Diagram:

The class diagram in figure 1 presents the structural view of core of a distributed cache solution. Some of the important entities of the class diagram are detailed as below. The names could mean a reference to entities shown in the class diagram as presented in figure 1 as well as the functionality indicator.

1) CacheConsumer

The cache consumers provide the read-only capability from the Cache. There is one cache consumer for every entity. A cache consumer has an instance of cache loader.

Example: For Account entity, we may have AccountCacheLoader which implements CacheLoader. So AccountCacheConsumer may have a reference to AccountCacheLoader injected in it.

2) Cache Group

A cache group is a collection of cache consumers and all the cache Consumer in a cache group share a common transaction. It could be modeled as an attribute of CacheManager and hence have a ‘HAS-A’ relationship with CacheManager. This is similar to the concept of regions in many known based caches. Every such cache group has its own settings that could be configured for example reconciliation time for a cache group is one such parameter.

The cache group is defined in the configuration and contains members in it which are either cache consumers or cache managers or a combination of both. It moreover also contains topic names to which it sends the events and from which it listens for updates Every entity extends CachebleObject and inherits these properties

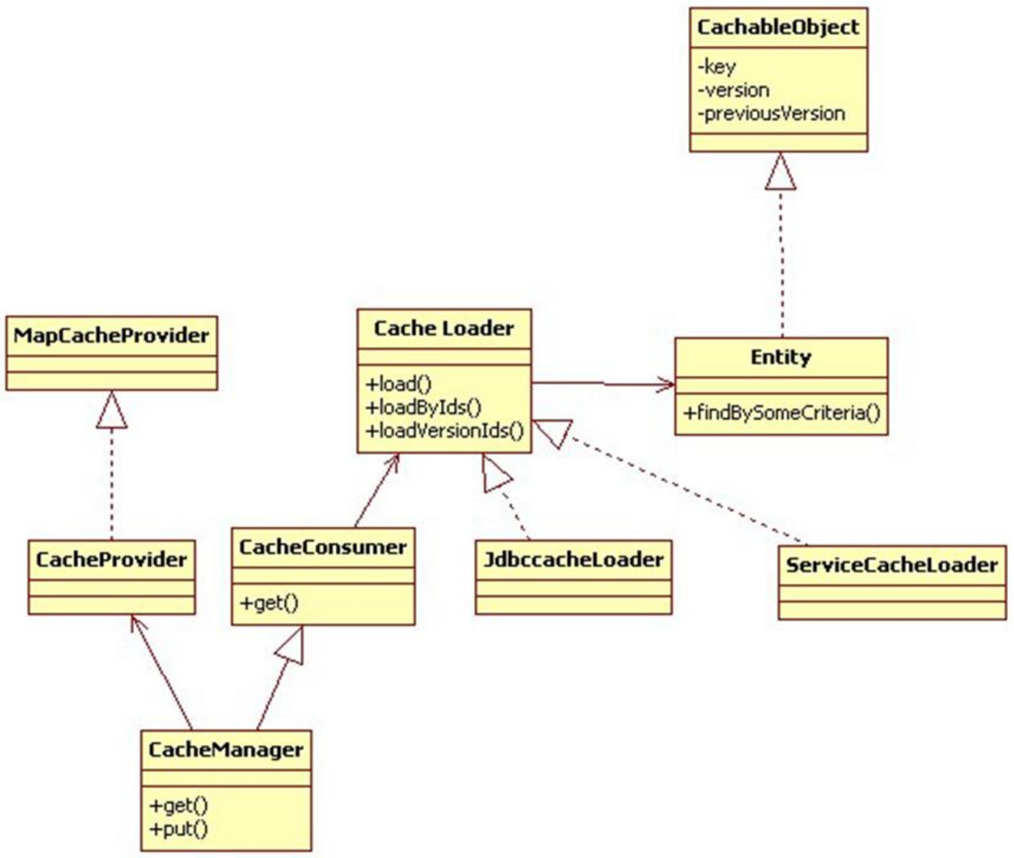
3) Cache Loader

Cache loader is the entity that connects to the database and initializes the cache. Different implementations are shown in the class diagram, each with specialized storage mechanism in the context.

Cache Loader contains these operations

- ***a.*** ***Load() :*** Loads every entity from the database.
- ***b.*** ***loadByIds(Collection keys)*****:** Loads entities whose keys are in the collection parameter.
- ***c.*** ***loadVersionIds()*****:** Loads only the version id to compare with the cache version id**.**

Figure 1



4) CacheManager

A Cache Manager is extended from Cache Consumer and provides both read and write functionality. Like Cache consumer, there is one Cache Manager for every entity. To retrieves the data from the cache it uses the inherited methods from cache consumer. whereas it implements its own methods for putting the data into the cache. To put the data into the cache it uses the instance of Cache Provider. An implementation to Cache Provider could be Map Provider which saves the data into a hash map.

VIII. DISTRIBUTED CUSTOM CACHE DEPLOYMENT

In figure 2 below we present the deployment architecture for our custom distributed caching solution. Central Server is a component that contains all the entities in its cache. The Server could be replicated to handle failover. The client nodes (subsystem) are distributed and contain a subset of the entities from Central Manager Node that depicts the cacheConsumers who have the read operations only. For example, a subsystem that is a trading application will contain only 20 relevant entities out of say 50 entities in the central manager, whereas another subsystem says portfolio application may contain only 15 entities out of those 50 entities in the cachegroup<manager> node.

Another use case that is supported by the caching mechanism is event-based processing when the data changes in the cache. Once the changed entity is published to an event queue, the EventListners which are registered to those event queues perform the synchronization. A typical example would be a ValidationEventListener which extends itself from Event Listener. ValidationEventListener contains the validation logic.

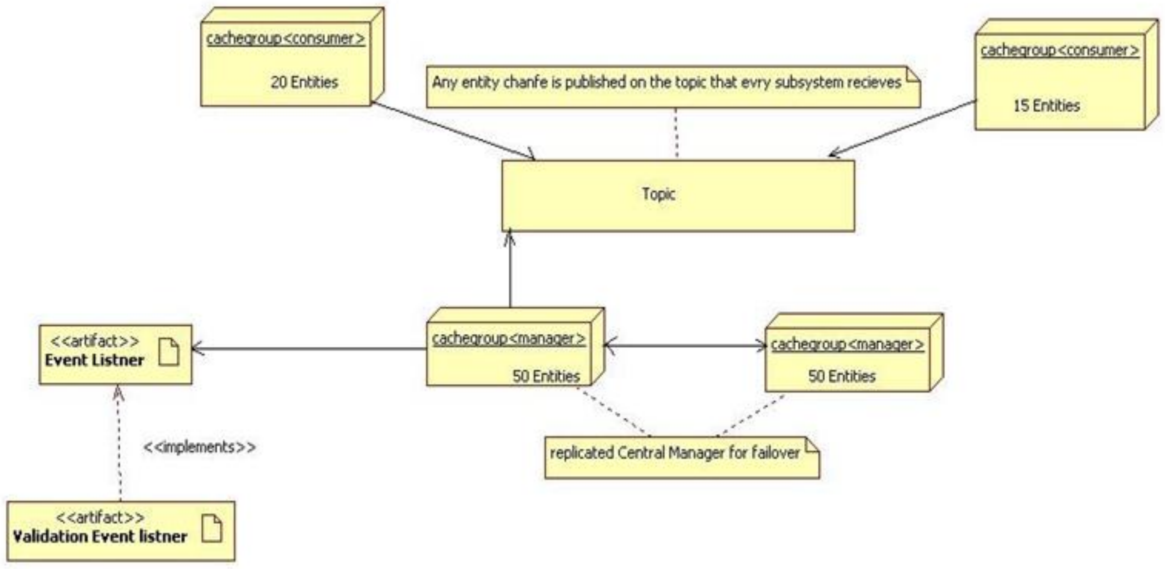


Figure 2

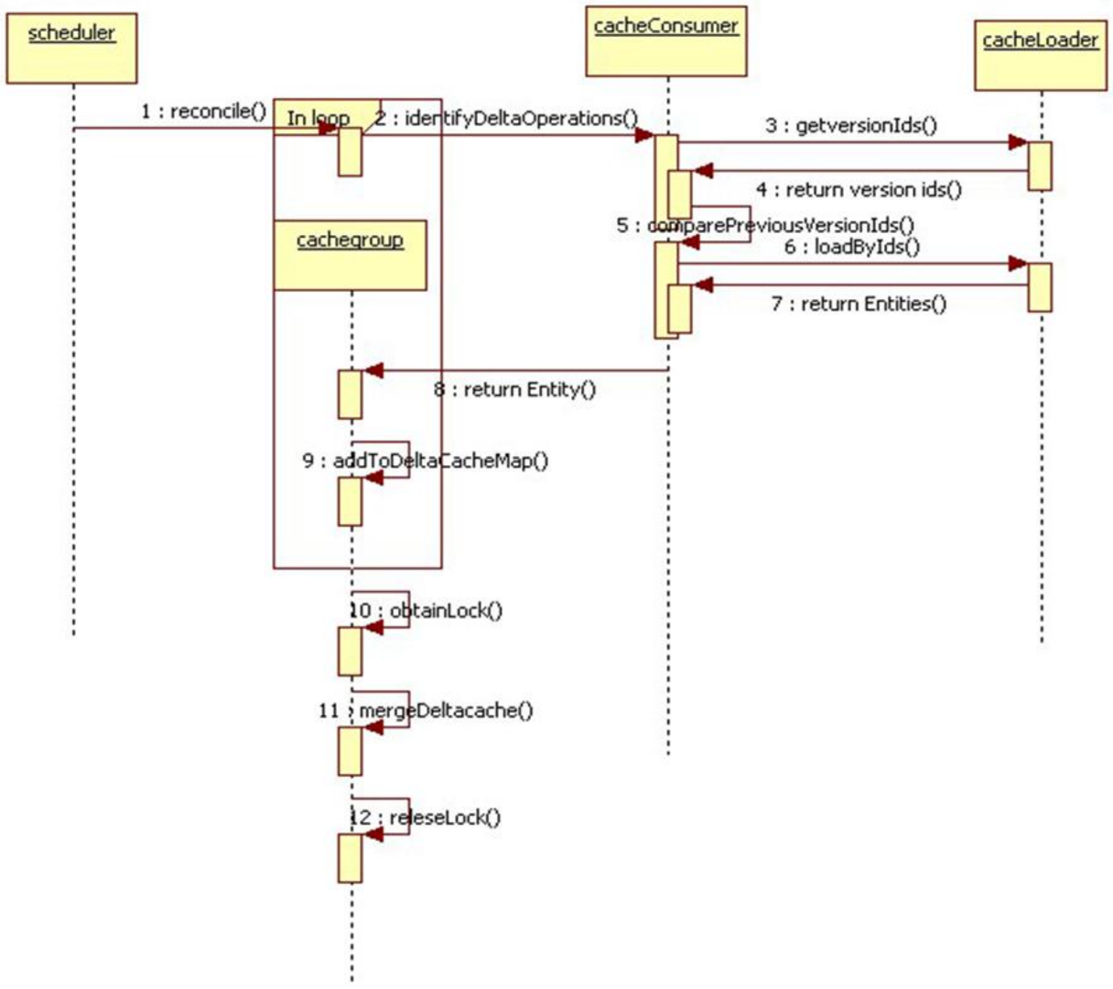


Figure 3

IX. INDEXING IN CACHE AND CRITERIA DESIGN ASPECTS.

The cache implementation is map based that is cache elements are stored in the form of <key, value> pair. Where key is the id and value are the entity. When required to query on some other field we need to create criteria. Criteria object contains only one public method: evaluate (), that fetches the result based on the criteria.

We can add multiple conditions in the query by appending many criteria. The criteria object is passed to the CriteriaUtil class that returns us the final criteria which contain an array of all such criteria. To support multiple conditions there are many available Criteria like,

a. AndCriteria
b. OrCriteria
c. NotCriteria
d. NullCriteria

Indexed fields are specified while creating cache consumers for that entity. The cache
contains a reference to an indexed map which again contains <key, value> pair and where the key is the indexed field and value is the reference to the entities in the parent map. Because the entities in the indexed map are only the reference and not the copy of actual objects, so the only overhead is the data structure but the search is faster.

X. RECONCILIATION SEQUENCE

A sequence diagram in figure 3 above illustrates the behavioural aspect of the reconciliation operation which is the focused functionality of our discussion and the design.

To reconcile from the database only the version ids are fetched and compared with the existing previous version ids in the cache group. If the version ids are changed then that entity is loaded from the database and is added to a temporary map that is delta cache map. The lock is obtained on the whole cache group for the merge to happen. Once the merge of the delta cache map with the previous cache is complete, the lock is removed.

The put() operation:

The put() operation in cache Manager in the normal context in any cache framework is not the same as in this implementation. In normal context, a put() operation is performed after a cache miss occurs. The element is retrieved from the database and before returning it back to the calling function it is put into the cache via put() operation, for future use. Because we get the whole copy of the database in the cache, it means that the database is no longer in the picture after loading. We get the entity from the cache and we also update the entity in the cache. The update happens through a call to put(). This operation is, in turn, is responsible for updating the database.

XI. CONCLUSION

In this paper, we have reviewed the purpose of cache and its extension to a distributed application in as distributed cache. We have also discussed various design aspects which add value to the design of a distributed cache. We have put an emphasis on the fact that although various proprietary and open source solutions are available, certain specialized needs of the application may need a custom implementation. Moreover, the architecture of the application itself may not be open for easy integration with the available solutions. The cost of acquiring the solution or support may itself be a reason to consider a custom solution which is tailored to the in-house needs of the application and is light weighted. We conclude our discussion with a simplistic design of the core component that could be used to realize a working implementation.

REFERENCES

[1] Wang Y, Rowe LA (1991) Consistency and Concurrency Control. 367–376.

[2] Smith AJ (1982) Cache Memories. ACM Comput Surv 14:473–530. doi: 10.1145/356887.356892

[3] Cited R, City O, Data RU-A (2003) (12) United States Patent. 1:0–4. doi: 10.1016/j.(73)

[4] KyleBrown Messaging To Update Distributed Caches. http://wiki.c2.com/?MessagingToUpdateDistributedCaches. Accessed 19 Jun 2017

[5] Borst S, Gupta V, Walid A, et al (2010) Distributed Caching Algorithms for Content Distribution Networks.

[6] Cao P, Karlin AR, Li K A study of Integrated Prefetching and Caching Strategies. 188–197.

[7] González A, Aliagas C, Valero M (1995) A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. Ics 338–347. doi: 10.1145/224538.224622

[8] Sahuquillo J, Pont A (2000) Splitting the data cache: A survey. IEEE Concurr 8:30–35. doi: 10.1109/4434.865890

[9] Podlipnig S, Böszörmenyi L (2003) A survey of Web cache replacement strategies. ACM Comput Surv 35:374–398. doi: 10.1145/954339.954341

[10] Koller R, Marmol L, Rangaswami R, Sundararaman S (2013) Write Policies for Host-side Flash Caches. Proc 11th USENIX Conf File Storage Technol 45–58.(2005) Cache coherence protocol.

[11] Gupta P, Zeldovich N, Madden S (2011) A trigger-based middleware cache for ORMs. Lect Notes Comput Sci (including Subser Lect Notes Artif Intell Lect Notes Bioinformatics) 7049 LNCS:329–349. doi: 10.1007/978-3-642-25821-3_17

[12] JSR107Specification.odt.pdf.