Caching in the Distributed Environment

Abhijit Gadkari
Information Science Dept., Claremont Graduate University, Claremont, CA

Abstract

The impact of cache is well understood in the system-design domain. While the concept of cache is extensively utilized in the von Neumann architecture, the same is not true for the distributed-computing architecture. For example, consider a three-tiered Web-based business application running on a commercial RDBMS. Every time a new Web page loads, many database calls are made to fill the drop down lists on the page. Performance of the application is greatly affected by the unnecessary database calls and the network traffic between the Web server and the database server.

1. Introduction

In production, many applications buckle down because they treat the database as their cache. Web server-based application-level cache can be effectively used to mitigate this problem. An effective caching mechanism is the foundation of any distributed-computing architecture. The focus of this article is to understand the importance of caching in designing effective and efficient distributed architecture. I will discuss the principle of locality of cache, basic caching patterns like temporal and spatial cache, and primed and demand cache, followed by an explanation of the cache replacement algorithms.

ORM technologies are becoming part of the mainstream application design, adding a level of abstraction. Implementing ORM-level cache will improve the performance of a distributed system. I will explain different ORM caching levels such as transactional cache, shared cache, and the details of intercache interaction. I'll also explore the impact of ORM caching on application design.

2. Distributed Systems

A distributed system is a heterogeneous system. Diverse system components are connected to each other via a common network. Applications using TCP/IP-based Internet are examples of open distributed systems.

Figure 1 shows a typical distributed architecture:

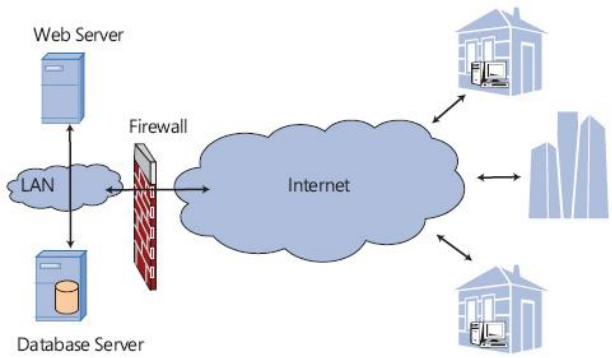

Figure 1. Distributed architecture

In the distributed environment, different activities occur in concurrent fashion. Usually, common resources like the underlying network, Web/application servers, database servers, and cache servers are shared by many clients. Distributing the computing load is the hallmark of distributed systems. Resource sharing and allocation is a major challenge in designing distributed architecture. For example, consider a Web-based database-driven business application. The Web server and the database server are hammered with client requests. Caching, load-balancing, clustering, pooling, and time-sharing strategies improve the system performance and availability. I'll focus on caching in the distributed environment.

Any frequently consumed resource can be cached to augment the application performance. For example, caching a database connection, an external configuration

file, workflow data, user preferences, or frequently accessed Web pages improve the application performance and availability. Many distributed-computing platforms offer out-of-the-box caching infrastructure. Java Caching System (JCS) is a distributed composite caching system. In Microsoft .NET Framework, the **System.Web.Caching** API provides the necessary caching framework. The Microsoft project code-named "Velocity" is a distributed-caching platform [1].

The performance of a caching system depends on the underlying caching data structure, cache eviction strategy, and cache utilization policy. Typically, a hash table with unique hash keys is used to store the cached data; JCS is a collection of hash tables [2]. The .NET cache implementation is based on the Dictionary data structure. The cache eviction policy is implemented in terms of a replacement algorithm. Utilizing different strategies such as temporal, spatial, primed, and demand caching can create an effective caching solution.

3. Cache and the Principle of Locality

The word "cache" comes from the French word meaning "to hide." [3]. Wikipedia defines cache as "a temporary storage area where frequently accessed data can be stored for rapid access." [4] Cached data is stored in the memory. Defining frequently accessed data is a matter of judgment and engineering. We have to answer two fundamental questions in order to define a solid caching strategy. What resource should be stored in the cache? How long should the resource be stored in the cache? The locality principle, which came out of work done on the Atlas System's virtual memory in 1959 [5], provides good guidance on this front, defining temporal and spatial locality. Temporal locality is based on repeatedly referenced resources. Spatial locality states that the data adjacent to recently referenced data will be requested in the near future.

4. Temporal Cache

Temporal locality is well suited for frequently accessed, relatively nonvolatile data—for example, a drop-down list on a Web page. The data for the drop down list can be stored in the cache at the start of the application on the Web server. For subsequent Web page requests, the drop down list will be populated from the Web server cache and not from the database. This will save unnecessary database calls and will improve application performance.

Figure 2 illustrates a flow chart for this logic:

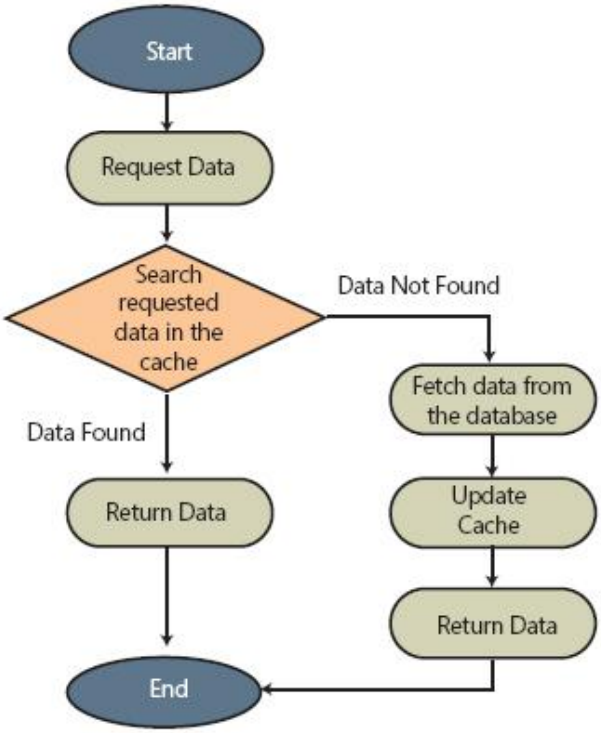

Figure 2. Temporal locality flow chart

When a resource is added to the cache, resource dependencies can be added to the caching policy. Dependencies can be configured in terms of an external file or other objects in the cache. An expiration policy defines the time dependency for the cached resource. Many caching APIs provide a programmatic way to synchronize the cache with the underlying database.

Figure 3 is sample C# code to populate the temporal cache:

```
protected void FillDepartmentList()
    {
      DataTable dtDepartment = (DataTable)Cache["departmentlist"];

      try
      {
        //if the cache is empty, get data from the database
        //and populate the cache
        if (dtDepartment == null)
        {
          dtDepartment = FillDropDownList.FillDepartmentListFromDB();
          Cache["departmentlist"] = dtDepartment;
        }

        //Make sure the cache is not empty
        if (dtDepartment.Rows.Count > 0)
        {
          ddlDepartmentList.DataSource = dtDepartment;
          ddlDepartmentList.DataBind();
        }
        else
        {
          ddlDepartmentList.Items.Add(new ListItem("No Department found!",
string.Empty));
        }
      }
      catch (Exception exce)
      {
        Response.Write(exce.Message);
      }
    }
```

Figure 3. C# code example to populate temporal cache

5. Spatial Cache

Consider an example of tabular data display like a **GridView** or an on-screen report. Implementing efficient paging on such controls requires complex logic. The logic is based on the number of records displayed per page and the total number of matching records in the underlying database table. We can either perform in-memory paging or hit the database every time the user moves to a different page; both are extreme scenarios. A third solution is to exploit the principle of spatial locality to implement an efficient paging solution. For example, consider a **GridView** displaying 10 records per page. For 93 records, we will have 10 pages. Instead of fetching all records in the memory, we can use the spatial cache to optimize this process. A sliding window algorithm can be used to implement the paging. Let's define the data window just wide enough to cover most of the user requests, say 30 records. On page one, we will fetch and cache the first 30 records. This cache entry can be user session specific or applicable across the application. As a user browses to the third page, the cache will be updated by replacing records in the range of 1–10 by 31–40. In reality, most users won't browse beyond the first few pages. The cache will be discarded after five minutes of inactivity, eliminating the possibility of a memory leak. The logic is based on the spatial dependencies in the underlying dataset. This caching strategy works like a charm on a rarely changing static dataset.

Figure 4 illustrates the spatial-cache logic that is used in the **GridView** example:

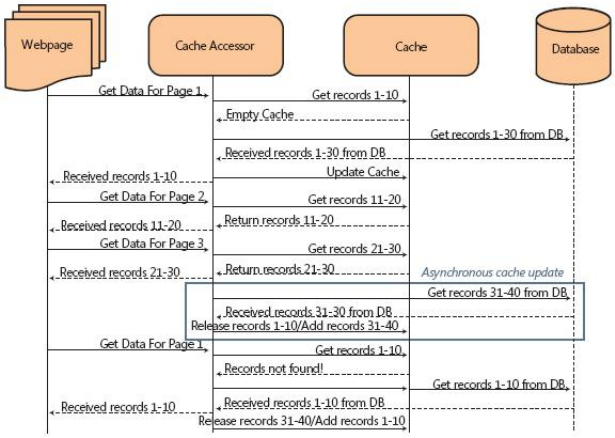

Figure 4. Spatial-cache sequence diagram

The drawback of this logic is the possibility of a stale cache. A stale cache is a result of the application modifying the underlying dataset without refreshing the associated cache, producing inconsistent results. Many caching frameworks provide some sort of cache synchronization mechanism to mitigate this problem. In .NET, the **SqlCacheDependency** class in the **System.Web.Caching** API can be used to monitor a specific table [6]. **SqlCacheDependency** refreshes the associated cache when the underlying dataset is updated.

6. Cache Replacement Algorithms

A second important factor in determining an effective caching strategy is the lifetime of the cached resource. Usually, resources stored in the temporal cache are good for the life of an application. Resources that are stored in the spatial cache are either time-dependent or place-dependent. Time-dependent resources should be purged as per the cache expiration policy. Place-specific resources can be discarded based on the state of the application. In

order to store a new resource in the cache, an existing cached resource will be moved out of the cache to a secondary storage, such as the hard disk. This process is known as paging. Replacement algorithms such as least frequently used resource (LFU), least recently used resource (LRU), and most recently used resource (MRU) can be applied in implementing an effective cache-eviction strategy, which influences the cache predictability [7]. The goal in implementing any replacement algorithm is to minimize paging and maximize the cache hit rate. The cache hit rate is the possibility of finding the requested resource in the cache. In most cases, LRU implementation is a good enough solution. JCS and ASP.NET caching is based on the LRU algorithm. In more complex scenarios, a combination of LRU and LFU algorithms such as the adaptive replacement cache (ARC) can be implemented. The idea in ARC is to replace the least frequently and least recently used cached data. This is achieved by maintaining two additional scoring lists. These lists will store the information regarding the frequency and timestamp of the cached resource. ARC outperforms LRU by dynamically responding to the changing access pattern and continually balancing workload and frequency features [8]. Some applications implement a cost-based eviction policy. For example, in Microsoft SQL Server 2005, zero-cost plans are removed from the cache and the cost of all other cached plans is reduced by half [9]. The cost in SQL Server is calculated based on the memory pressure.

A study of replacement algorithms suggests that a good algorithm should strike a balance between the simplicity of randomness and the complexity inherent in cumulative information [10]. Replacement algorithms play an important role in defining the cache-eviction policy, which directly affects the cache hit-rate and the application performance.

7. Primed Cache vs. Demand Cache: Can We Predict the Future?

Data-usage predictability also influences the caching strategy. The primed-cache pattern is applicable when the cache or part of the cache can be predicted in advance [11]. This pattern is very effective in dealing with static resources. A Web browser cache is an example of primed cache; cached Web pages will load fast and save trips to the Web server. The demand-cache pattern is useful when cache cannot be predicted [12]. A cached copy of user credentials is an example of demand cache. The primed cache is populated at the beginning of the application, whereas the demand cache is populated during the execution of the application.

7.1 Primed Cache

The primed cache minimizes the overhead of requesting external resources. It is suitable for the read-only resources frequently shared by many concurrent users.

Figure 5 illustrates the typical primed cache architecture:

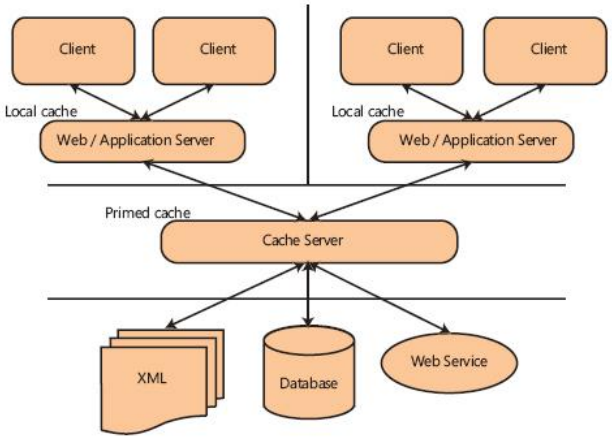

Figure 5. Primed-cache example

The cache server cache is primed in advance, and the individual Web/application server cache is populated from the cache server. Each Web/application server can read, write, update, and delete the cache on the cache server. The cache server in turn is responsible for synchronizing the cache with the resource environment. Because the primed cache is populated in advance, it improves the application response time. For example, reports with static, fixed parameters can be populated and stored in the cache. This way, the reports are available almost instantly. In .NET, the **ICachedReport** interface can be used to store the prepopulated reports in the cache. Updating the primed cache mostly results in updating existing cached resources. The cache is refreshed based on a routine schedule or a predictable event-based

mechanism. The primed-cache results in an almost constant size cache structure [11].

7.2 Demand Cache

The demand cache is suitable when the future resource demand cannot be predicted. The resource environment acquires the resource only when it is needed. This optimizes the cache and achieves a better hit-rate. As soon as the resource is available, it is stored in the demand cache. All subsequent requests for the resource are satisfied by the demand cache. As soon as it is cached, the resource should last long enough to justify the caching cost.

Figure 6 illustrates a class diagram for implementing the demand cache [12]:

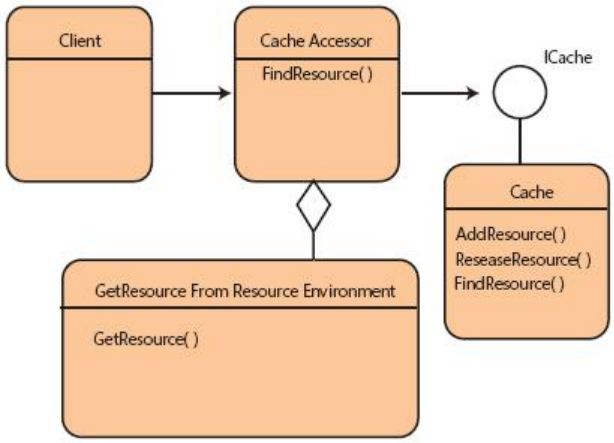

Figure 6. Demand cache

For example, a user can have many roles and one role can have many permissions. Populating the entire permissions domain for all users at the start of an application will unnecessarily overload the cache. The solution is to store the user credentials in the demand cache on a successful log-in. All subsequent authorization requests from the application for already authenticated users will be fulfilled by the demand cache. This way the demand cache will only store a subset of all possible user permissions in the system.

In the absence of a proper eviction policy, the resource will be cached forever. Permanently cached resources will result in memory leak, which degrades the cache performance. For example, as the number of authenticated users grows, the size of the demand cache increases and the performance degrades. One way to avoid this problem is to link resource eviction with resource utilization. In our example, the cache size can be managed by removing the credentials of all logged-off users.

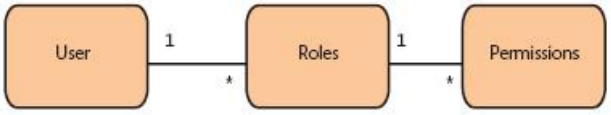

Figure 7. Demand-cache example

Predicting the future is a difficult business. In a dynamic environment, adaptive caching strategies represent a powerful solution, based on some sort of application usage heuristics. However, adaptive caching strategies are beyond the scope of this article.

8. Caching in the ORM World!

Object relational mapping is a way to bridge the impedance mismatch between object-oriented programming (OOP) and relational database management systems (RDBMS). Many commercial and open-source ORM implementations are becoming an integral part of the contemporary distributed architecture. For example, Microsoft Entity Framework and Language Integrated Query (LINQ), Java Data Objects (JDO), TopLink, Hibernate, NHibernate, and iBATIS are all popular ORM implementations. The ORM manager populates the data stored in persistent storages like a database in the form of an object graph. An object graph is a good caching candidate.

The layering principle, based on the explicit separation of responsibilities, is used extensively in the von Neumann architecture to optimize system performance. N-tier application architecture is an example of the layering principle. Similar layering architecture can be used in implementing the ORM caching solution. The ORM cache can be layered into two different categories: the read-only shared cache used across processes, applications, or machines and the updateable write-enabled transactional cache for coordinating the unit of work [13].

Cache layering is prevalent in many ORM solutions—for example, Hibernate's two-level caching architecture [14]. In a layered-caching framework, the first layer represents the transactional cache and the second layer is the shared cache designed as a process or clustered cache.

Figure 8 illustrates the layered-cache architecture:

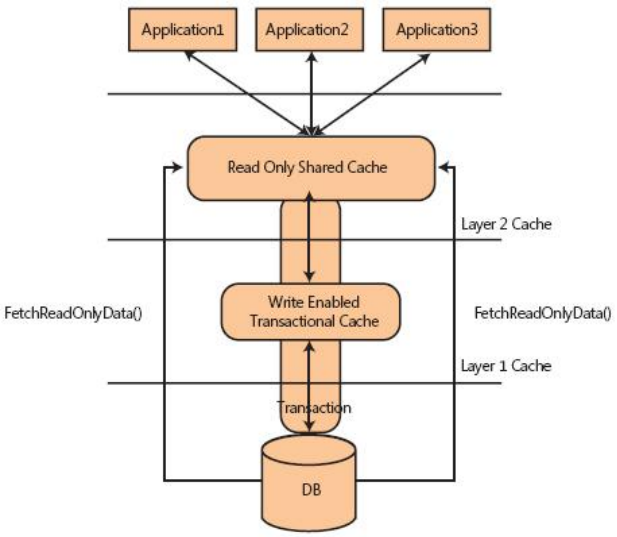

Figure 8. Layered-cache architecture

9. Transactional Cache

Objects formed in a valid state and participating in a transaction can be stored in the transactional cache. Transactions are characterized by their ACID (Atomicity, Consistency, Isolation, and Durability) properties. Transactional cache demonstrates the same ACID behavior. Transactions are atomic in nature; each transaction will either be committed or rolled back. When a transaction is committed, the associated transactional cache will be updated. If a transaction is rolled back, all participating objects in the transactional cache will be restored to their pretransaction state [15]. You can implement this behavior by using the unit of work pattern [13].

Thrashing, cache corruption, and caching conflicts should be strictly avoided in implementing the transactional cache. Many caching implementations offer a prepackaged transactional cache solution, including the TreeCache implementation in JBoss. TreeCache is a tree structured, replicated, transactional cache based on the pessimistic locking scheme [16].

10. Shared Cache

The shared cache can be implemented as a process cache or clustered cache [14]. A process cache is shared by all concurrently running threads in the same process. A clustered cache is shared by multiple processes on the same machine or by different machines. Distributed-caching solutions implement the clustered cache; for example, the project code-named "Velocity" is a distributed-caching API [1]. The clustered shared cache introduces resource replication overhead. Replication keeps the cache in a consistent state on all the participating machines. A safe failover mechanism is implemented in the distributed-caching platform; in case of a failure, the cached data can be populated from other participating nodes.

Objects stored in the transactional cache are useful in optimizing the transaction. As soon as the transaction is over, they can be moved into the shared cache. All read-only requests for the same resource can be fulfilled by the shared cache; and, because the shared cache is read-only, all cache coherency problems are easily avoided. The shared cache can be effectively implemented as an Identity Map [13].

As shown in Figure 9, requests for the same shared-cache resource result in returning the same object:

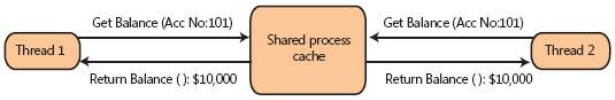

Figure 9. Shared-cache example

You can use different coordination techniques to manage the interaction between the shared and transactional cache [15]. These techniques are explained in the following section on intercache interaction.

11. Managing the Interaction

The interaction between the shared cache and the transactional cache depends on the nature of the cached

data. Read-only cached data will result in infrequent cache communication. There are many ways to optimize intercache communication [15].

One solution is to populate the object graph simultaneously in the shared and transactional cache. This saves the overhead of moving objects from one cache to the other. On completion of the transaction, an updated copy of the object in the transactional cache will refresh the shared cache instance of the object. The drawback of this strategy is the possibility of a rarely used transactional cache in the case of frequent read-only operations.

Another solution is to use the just-in-time copy strategy. The object will be moved from the shared cache to the transactional cache at the beginning of a transaction and will be locked for the duration of the transaction. This way no other thread, application or machine can use the locked object. The lock is released on completion of the transaction and the object is moved back to the shared cache (see Figure 10).

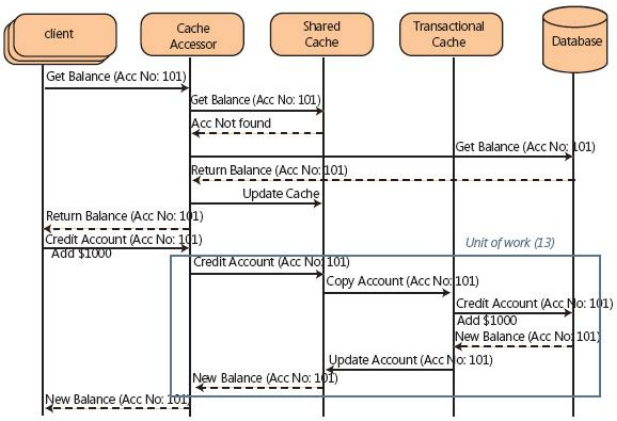

Figure 10. Intercache interaction

It is important to minimize the possibility of a stale or corrupt cache and maximize resource utilization. The data copying between the transactional and shared cache should also be minimized in order to increase the cache hit rate. Because locks are effectively managed in the database, there are some concerns in implementing the same at the application level. This discussion is important but beyond the scope of this article.

Caching domain-specific dependencies is an essential but difficult task. As illustrated in Figure 7, caching the combination of all roles and corresponding permissions for the logged-on user will populate a large object graph. Application patterns like Domain Model and Lazy Load can be effectively applied in caching such domain dependencies [13]. One important consideration in designing a caching strategy is the cache size.

12. Chasing the Right Size Cache

There is no definite rule regarding the size of the cache. Cache size depends on the available memory and the underlying hardware viz. 32/64 bit and single-core/multicore architecture. An effective caching strategy is based on the Pareto principle (that is, the 80–20 rule). For example, on the ecommerce book portal, 80 percent of the book requests might be related to the top 10,000 books. The application's performance will greatly improve if the list of top 10,000 books is cached. Always remember the principle of diminishing returns and the bell-shaped graph in deciding cache size. (See Figure 11.)

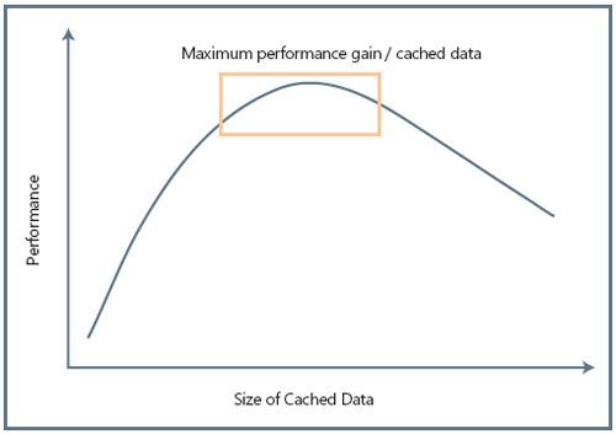

Figure 11. Cached data versus performance

How much data should be cached depends on many factors such as processing load patterns, the number of concurrent connections/requests, and the type of application (real-time versus batch processing). The goal of any caching strategy is to maximize the application performance and availability.

13. Conclusion

Small caching efforts can pay huge dividends in terms of performance. Two or more caching strategies and design patterns like GOF [17], PEAA [13], and Pattern of Enterprise Integration (PEI) can be clubbed together to implement a solid caching platform. For example, shared demand cache coupled with a strict time-based eviction policy can be very effective in optimizing the performance of a read-heavy distributed system like the enterprise reporting solution.

Forces like software transactional memory (STM), multicore memory architecture such as NUMA (Non-Uniform Memory Access), SMP (symmetric multiprocessing architectures), and concurrent programming will influence the future of caching platforms. In the era of cloud computing, caching will play a pivotal role in the design of distributed systems. An efficient caching strategy will differentiate a great distributed architecture from the good. Let your next design be a great one.

Acknowledgments

I would like to thank Gita Gadgil for reading the draft material and providing invaluable feedback and suggestions.

References

[1] http://code.msdn.microsoft.com/velocity
[2] http://jakarta.apache.org/jcs/getting_started/intro.html
[3] http://dictionary.reference.com/browse/cache
[4] http://en.wikipedia.org/wiki/Cache
[5] Peter J. Denning, "The Locality Principle, Communications of the ACM," July 2005, Vol 48, No 7.
[6]http://msdn.microsoft.com/en-us/library/system.web.caching.sqlcachedependency.aspx
[7] Michael Kircher and Prashant Jain, "Caching," EuroPloP 2003.
[8] Nimrod Megiddo and Dharmendra S. Modha, "Outperforming LRU with an Adaptive Replacement Cache Algorithm," IEEE Computer, April 2004.
[9] Kalen Delaney, Inside Microsoft SQL Server 2005: Query Tuning and Optimization, Microsoft Press, 2007.
[10] L.A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," IBM Systems J. 5, 2 (1966), 78–101.
[11] Octavian Paul Rotaru, "Caching Patterns and Implementation," Leonardo Journal of Sciences LJS: 5:8, January-June 2006.
[12] Clifton Nock, Data Access Patterns: Database Interactions in Object-Oriented Applications, Addison-Wesley, 2003.
[13] Martin Fowler, Pattern of Enterprise Application Architecture (P of EAA), Addison-Wesley, 2002.
[14] Christian Bauer and Gavin King, Java Persistence with Hibernate, Manning Publications, 2006.
[15] Michael Keith and Randy Stafford, "Exposing the ORM Cache," ACM Queue, Vol 6, No 3, May/June 2008.
[16]TreeCache(http://www.jboss.org/file-access/default/members/jbosscache/freezone/docs/1.4.0/TreeCache/en/html_single/index.html#introduction)
[17] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

Abhijit Gadkari is an Enterprise Architect with AMG-SIU. Abhijit's background includes an M.S. in IS from Claremont Graduate University, and post graduation in computer science from DOE, India. He has 10 years of software design and development experience. His work and research focuses on building SaaS-based IT infrastructure.