Study and Evaluation of caching mechanisms in web applications

Syed Nisar Hussain Bukhari [1] (Scientist-B), Ashaq Hussain Dar [2] (Scientist-C)

National Institute of Electronics and Information Technology, J & K,

Department of Electronics and IT, Ministry of communication and Information Technology, Govt. of India

E-mail: nisar.bukhari@gmail.com [1], ashaqhussain@yahoo.com [2]

Abstract

Memory is the area in web application development where the most abuse occurs and where the most benefit may be gained.To lower the memory footprint and speed up the application, effective caching strategy is the solution-an optimization technique which improves the performance and responsiveness of an application because it keeps items that have been recently used in memory, anticipating that they will be needed again. But it is a double-edge razor in a sense that if using carefully and with comprehension will make life easier. Playing with it without knowing what you do may ruin your application. In this paper caching architecture of a web application will be presented. The caching strategies in web application like spatial and temporal will be evaluated. These strategies shall be evaluated on a data base driven web application and the performance results of the application against the caching techniques shall be presented.

Keywords: Memory, spatial, caching, temporal, performance.

1. Introduction

Resource sharing and allocation is a major challenge in designing distributed web architecture. Consider a web-based database-driven business application. The web server and the database server are hammered with client requests [1]. Most every data model contains a fair amount of static data, usually implemented in the form of lookup tables. Since this information is static, there s no reason to continually access the database each time this information needs to be displayed [2]. 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 [1]. However, incorrect caching choices and poor caching design can degrade performance and responsiveness. So first decide when to load data into the cache. Load cache if one of the following situations arises:

- You must repeatedly access static data or data that rarely changes.
- Data access is expensive in terms of creation, access, or transportation.
- Data must always be available, even when the source, such as a server, is not available

The underlying caching data structure, cache eviction strategy, and cache utilization policy decide the performance of a caching system. Typically, a hash table with unique hash keys is used to store the cached data. For example the .NET framework 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.

2. Architecture of a simple web based application using caching

A cache is made up of a pool of entries. Each entry has data along with a tag, which specifies the identity of the data in the backing store of which the entry is a copy. When the cache client (a CPU, web browser, operating system) needs to access the data presumed to exist in the backing store, it first checks the cache. If an entry can be found with a tag matching that of the desired data, the data in the entry is used instead. This situation is known as a cache hit. CPUs and hard drives frequently use a cache, as do web browsers and web servers.

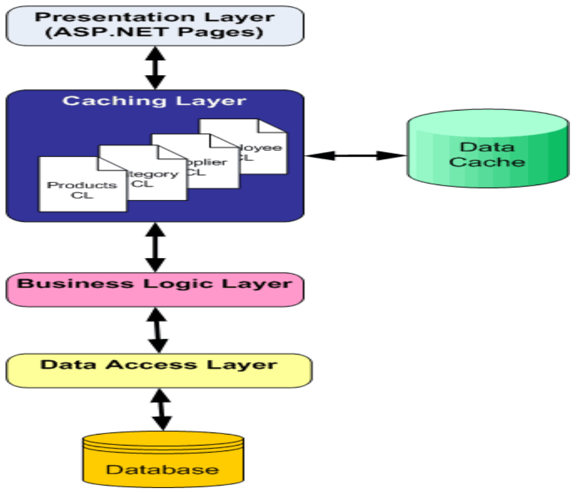


Figure 1: The Caching Layer (CL) is Another Layer

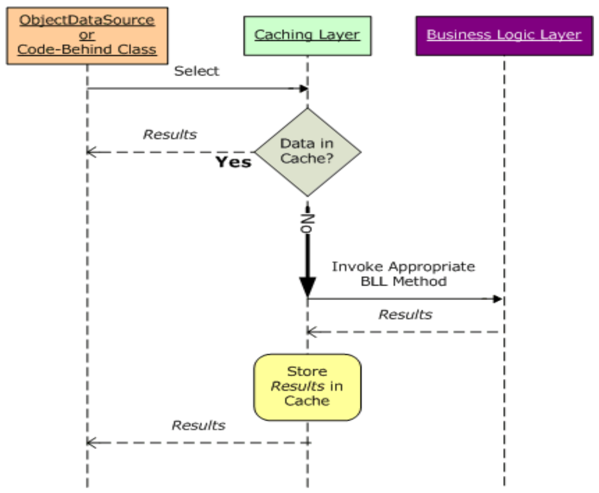


Figure 2: The caching layer's methods returns data in the main architecture from the cache if it is available.

3. Caching Strategies

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 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 [1].

3.1 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 3 illustrates a flow chart for this logic

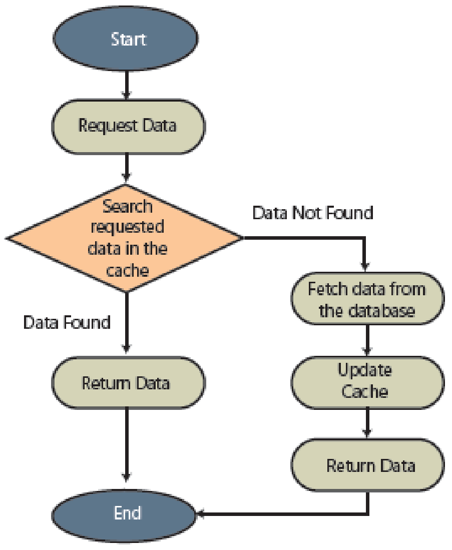
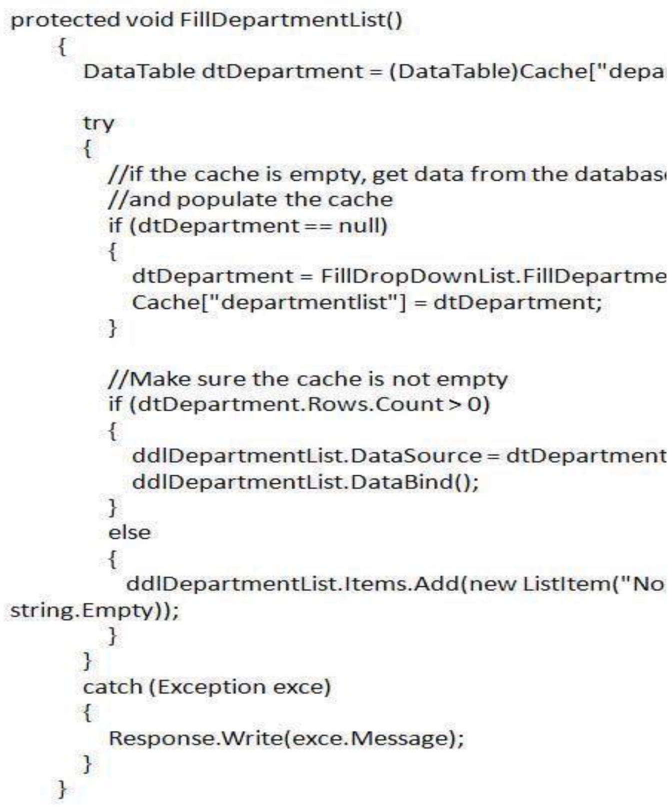


Figure 3: Temporal locality flowchart -source [1]

Below is the sample C# code to populate the temporal cache.[1]



```
protected void FillDepartmentList()
    {
      DataTable dtDepartment = (DataTable)Cache["depa

      try
      {
        //if the cache is empty, get data from the databas
        //and populate the cache
        if (dtDepartment == null)
        {
          dtDepartment = FillDropDownList.FillDepartme
          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
string.Empty));
        }
      }
      catch (Exception exce)
      {
        Response.Write(exce.Message);
      }
    }
```

3.2 Spatial Cache

Consider an example of tabular data display like a Grid View 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 Grid View displaying 10 records per page. For 93 records, we will have 10 pages. Rather than 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 used in the Grid View example. 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 provides 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. SqlCacheDependency refreshes the associated cache when the underlying dataset is updated. [1]

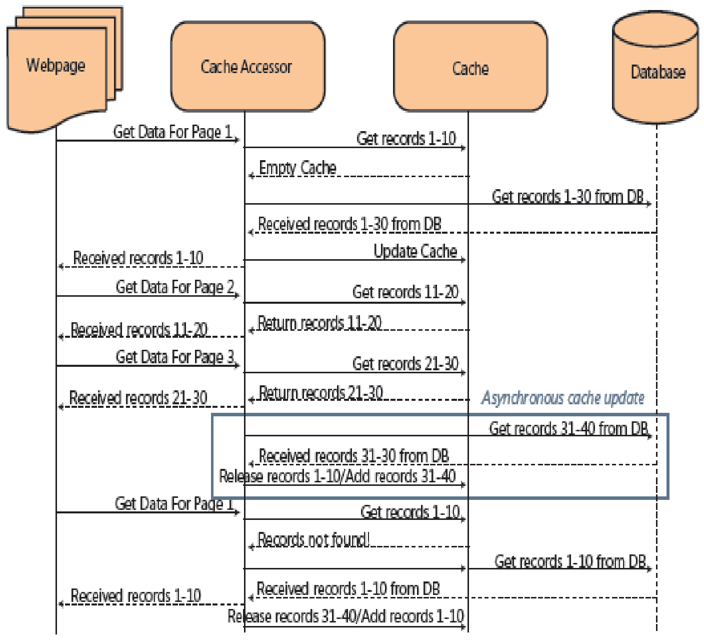


Figure 4: spatial cache sequence diagram-source [1]

One good approach to implement the above scenario is to use stored procedures. SQL script below shows a sample stored procedure that pages through the Orders table in the Northwind database. In a nutshell, all we're doing here is passing in the page index and the page size. The appropriate resultset is calculated and then returned.

```
CREATE PROCEDURE northwind_OrdersPaged ( @PageIndex int, @PageSize int )
AS BEGIN
DECLARE @PageLowerBound int
DECLARE @PageUpperBound int
DECLARE @RowsToReturn int
-- First set the rowcount
SET @RowsToReturn = @PageSize * (@PageIndex + 1)
SET ROWCOUNT @RowsToReturn -- Set the page bounds
SET @PageLowerBound = @PageSize * @PageIndex
SET @PageUpperBound = @PageLowerBound + @PageSize + 1
-- Create a temp table to store the select results
CREATE TABLE #PageIndex ( IndexId int IDENTITY (1, 1) NOT NULL, OrderID int )
-- Insert into the temp table
INSERT INTO #PageIndex (OrderID) SELECT OrderID FROM Orders ORDER BY OrderID D
-- Return total count
SELECT COUNT(OrderID) FROM Orders
-- Return paged results
SELECT O.* FROM Orders O, #PageIndex PageIndex
WHERE O.OrderID = PageIndex.OrderID AND
PageIndex.IndexID > @PageLowerBound AND PageIndex.IndexID < @PageUpperBound
          ORDER BY PageIndex.IndexID END
```

The total number of records returned can vary depending on the query being executed. For example, a WHERE clause can be used to constrain the data returned. The total number of records to be returned must be known in order to calculate the total pages to be displayed in the paging UI. For example, if there are 1,000,000 total records and a WHERE clause is used that filters this to 1,000 records, the paging logic needs to be aware of the total number of records to properly render the paging UI [6].

4. 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 stored in the spatial cache are time-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. The goal in implementing any replacement algorithm is to minimize paging and maximize the cache hit rate. In most cases, LRU implementation is a good enough solution. 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. Some applications implement a cost based eviction policy. For example, in SQL Server 2005, zero cost plans are removed from the cache and the cost of all other cached plans is reduced by half. 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. Replacement algorithms play an important role in defining the cache-eviction policy, which directly affects the cache hit-rate and the application performance. [1]

5. Results

Use caching of data for a small period of time and avoid caching for the whole application lifecycle. Try to cache the data that is likely to not be changed very often (e.g., dictionary elements). Data retrieving from a repository can be quite a "heavy" task from a performance point of view, especially when the data repository is located far from the application server (e.g., web service call, RPC call, Remoting etc.) or some specific data is accessed very often. So, in order to reduce the workload and time for data retrieving, you can use a caching functionality.[3] Your data access layer gets a whole lot of code that deals with caching objects and collection, updating cache when objects change or get deleted, expire collections when a contained object changes or gets deleted and so on. The more code you write, the more maintenance overhead you add. [4] Besides the obvious goals, data caching has some pitfalls (all of them are about potential situations when cached data can expire and application uses inconsistent data):

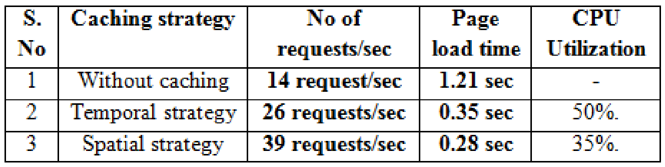
- If the application is going to be scaled to a distributed environment (web farm, application cluster), then every machine will have its own copy of cached data. So, one of the machines can modify the data at any time.
- Several applications can access the same data repository.[3]
- Entries in the cache might be removed for reasons other than that they've expired. For example, the web server might temporarily run low on memory, and one way it can reclaim memory is by throwing entries out of the cache [5]

Results of temporal and spatial cache mechanisms on a test web application: The tests were conducted on a SQL Server 2008 driven web application in asp.net and results were noted on three scenarios below:

a) One of the application I tested without caching can only serve about **14 request/sec** with 10 concurrent users on a dual core 64 bit PC. The average page response time noted was **1.21 sec.**

b) One of the page named TemporalDemo.aspx was rendering some news items and a grid view with 120 records. After implementing cache on this page in a temporal way, it became significantly faster, around **26 requests/sec**.Page load time decreased significantly as well to **0.35 sec** only. During the load test, CPU utilization was around 50%.

c) Another page named SpatialDemo.aspx was rendering the same news items as on TemporalDemo.aspx and a grid view .But this time at the time of page load only 10 records were fetched from the database (temporal cache)instead of 120 in one go at the time of load as in case scenario a above . It became even more faster, around **39 requests/sec**. Page load time decreased significantly as well to **0.28 sec** only. During the load test, CPU utilization was around 35%.



S. No	Caching strategy	No of requests/sec	Page load time	CPU Utilization
1	Without caching	**14 request/sec**	**1.21 sec**	-
2	Temporal strategy	**26 requests/sec**	**0.35 sec**	50%.
3	Spatial strategy	**39 requests/sec**	**0.28 sec**	35%.

Table 1: Results showing different parameter values using spatial and temporal caching strategy

It shows clearly the significant difference it can make to your application based on the caching policy used (temporal or spatial).

6. Conclusion

In terms of advantage, Cache concept can make or break the performance of your computer system. Smart buffer algorithm will make the site loading faster than traditional approach. Cache takes the heavy load / execution from the server for the repeated operations. By doing so, Servers can be efficiently used. Caching can provide huge performance benefits to applications, and should therefore be considered when an application is being designed as well as when it is being performance tested. Choosing a caching strategy is important. The locality principle 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. Spatial strategy can have an added advantage over temporal but it again depends upon the requirement.

References:

[1] Abhijit Gadkari, Caching in the Distributed Environment, The Architecture Journal-Microsoft

[2] Scott Mitchell , Caching Data at Application Startup,http://www.asp.net/webforms/tutorials/data-access/caching-data/caching-data-at-application-startup-cs

[3] Code project, http://www.codeproject.com/

[4] Simple Way to Cache Objects and Collections for Greater Performance and Scalability,http://www.codeproject.com/Articles/43434/A-Simple-Way-to-Cache-Objects-and-Collections-for

[5] Caching data for better performance,http://www.asp.net/web-pages/tutorials/performance-and-traffic/15-caching-to-improve-the-performance-of-your-website

[6] Microsoft MSDN, http://msdn.microsoft.com/en-us/magazine/cc163854.aspx#S3