ISSN:0975-9646

Cache-Aside Approach For Cloud Design Pattern

Narendra Babu Pamula[1], K Jairam[2], B Rajesh[3]

[1,2] *ASSISTANT PROFESSOR,DEPT OF CSE ,*
V.KR.,V.N.B & A.G.K COLLEGE OF ENGINNERING,
GUDIVADA,KRISHNA(D.T),ANDHRA PRADESH, INDIA-521301

[3,] *ASSISTANT PROFESSOR,*DEPT OF IT,
VASIREDDY VENKATADRI INSTITUTE OF TECHNOLGY,
NAMBUR,GUNTUR(D.T),ANDHRA PRADESH, INDIA

Abstract: In this paper is represented by showing how each piece can fit into the big picture of cloud application architectures. It also discusses the benefits and considerations for some patterns like Cache-aside pattern, Circuit breaker pattern, Command and query responsibility segregation pattern and Health endpoint monitoring pattern. Most of the features of Windows Azure. However the majority of topics described in this guide are equally relevant to all kinds of distributed systems, whether hosted on Windows Azure or on other cloud platforms.

Keywords: Windows Azure caching, Cache-aside pattern, Evicting Data, Co-Located Topology, Dedicated Topology.

1 INTRODUCTION

In computer networking, cloud computing is a phrase used to describe a variety of computing concepts that involve a large number of computers connected through a communication network such as an Internet. It is very similar to the concept of utility computing. In science, cloud computing is a synonym for distributed computing over a network, and means the ability to run a program or application on many connected computers at the same time. The phrase is often used in reference to network-based services, which appear to be provided by real server hardware, and are in fact served up by virtual hardware, simulated by software running on one or more real machines. Such virtual servers do not physically exist and can therefore be moved around and scaled up or down on the fly without affecting the end user, somewhat like a cloud becoming larger or smaller without being a physical object. A design pattern in architecture and computer - science is a formal way of documenting a solution to a design problem in a particular field of expertise. An organized collection of design patterns that relate to a particular field is called a language. The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment, which describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. The following design patterns are useful in cloud-hosted applications. Each pattern is provided in a common format that describes the context and problem, the solution, issues and considerations for applying the pattern, and an example based on Windows Azure. Each pattern also includes links to other related patterns.

2 RELATED WORK

2.1 Windows Azure Caching

Windows Azure Caching is an in-memory, distributed caching feature designed for Windows Azure application. Caching is available as a part of the Windows Azure SDK.Windows Azure Caching allows a cloud service to host Caching on a Windows Azure role.[1] The cache is distributed across all running instances of that role. Therefore, the amount of available memory in the cache is determined by the number of running instances of the role that hosts Caching and the amount of physical memory reserved for Caching on each instance.

There are two deployment topologies for Caching:

- Dedicated
- Co-located

2.1.1 *Dedicated Topology*

In the dedicated topology, you define a worker role that is dedicated to Caching. This means that all of the worker role's available memory is used for the Caching and operating overhead. The following diagram shows Caching in a dedicated topology. The cloud service shown has three roles: Web1, Worker1, and Cache1. There are two running instances of each role. In this example, the cache is distributed across all instances of the dedicated Cache1 role.

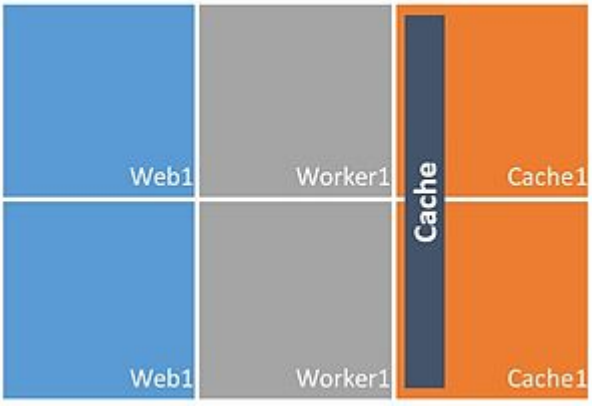

Fig 1: **Dedicated Topology**

A dedicated topology has the advantage of scaling the caching tier independently of any other role in the cloud service. For the best Caching performance, a dedicated topology is recommended because the role instances do not share their resources with other application code and services.

2.1.2 Co-Located Topology

In a co-located topology, you use a percentage of available memory on existing web or worker roles for Caching. The following diagram shows Caching in a co-located topology. The cloud service has two roles: Web1 and Worker1. There are two running instances of each role. In this example, the cache is distributed across all instances of the Web1 role. Because this role also hosts the web front-end for the cloud service, the cache is configured to use only a percentage of the physical memory on each instance of the Web1 role.

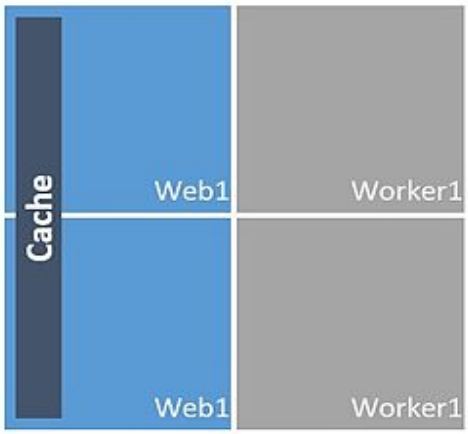

Fig 2: Co-located topology

A co-located cache is a cost-effective way to make use of existing memory on a role within a cloud service. Cloud computing presents a number of management challenges. Companies using public clouds do not have ownership of the equipment hosting the cloud environment, and because the environment is not contained within their own networks, public cloud customers do not have full visibility or control Users of public cloud services must also integrate with an architecture defined by the cloud provider, using its specific parameters for working with cloud components. Integration includes tying into the cloud APIs for configuring IP addresses, subnets, firewalls and data service functions for storage. Because control of these functions is based on the cloud provider's infrastructure and services, public cloud users must integrate with the cloud infrastructure management. Capacity management is a challenge for both public and private cloud environments because end users have the ability to deploy applications using self-service portals. Applications of all sizes may appear in the environment, consume an unpredictable amount of resources, then disappear at any time. Chargeback or, pricing resource use on a granular basis is a challenge for both public and private cloud environments. Chargeback is a challenge for public cloud service providers because they must price their services competitively while still creating profit Users of public cloud services may find chargeback challenging because it is difficult for IT groups to assess actual resource costs on a granular basis due to overlapping resources within an organization that may be paid for by an individual business unit, such as electrical power. For private cloud operators, chargeback is fairly straightforward, but the challenge lies in guessing how to allocate resources as closely as possible to actual resource usage to achieve the greatest operational efficiency. Exceeding budgets can be a risk. Hybrid cloud environments, which combine public and private cloud services, sometimes with traditional infrastructure elements, present their own set of management challenges. These include security concerns if sensitive data lands on public cloud servers, budget concerns around overuse of storage or bandwidth and proliferation of mismanaged images. Managing the information flow in a hybrid cloud environment is also a significant challenge. On-premises clouds must share information with applications hosted off-premises by public cloud providers and this information may change constantly. Hybrid cloud environments also typically include a complex mix of policies, permissions and limits that must be managed consistently across both public and private clouds.

3. Cache-Aside Pattern

Load data on demand into a cache from a data store. This pattern can improve performance and also helps to maintain consistency between data held in the cache and the data in the underlying data store. Applications use a cache to optimize repeated access to information held in a data store. However, it is usually impractical to expect that cached data will always be completely consistent with the data in the data store. Applications should implement a strategy that helps to ensure that the data in the cache is up to date as far as possible, but can also detect and handle situations that arise when the data in the cache has become stale. Many commercial caching systems provide read-through and write-through/write-behind operations. In these systems, an application retrieves data by referencing the cache. If the data is not in the cache, it is transparently retrieved from the data store and added to the cache. Any modifications to data held in the cache are automatically written back to the data store as well. For caches that do not provide this functionality, it is the responsibility of the applications that use the cache to maintain the data in the cache. An application can emulate the functionality of read-through caching by implementing the cache-aside strategy. This strategy effectively loads data into the cache on demand. **Figure 3** summarizes the steps in this process.

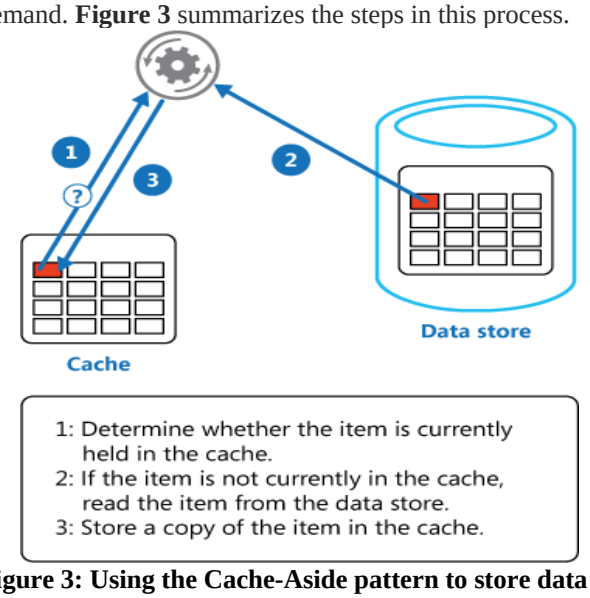

Figure 3: Using the Cache-Aside pattern to store data in the cache

If an application updates information, it can emulate the write-through strategy as follows:

1. Make the modification to the data store
2. Invalidate the corresponding item in the cache.

When the item is next required, using the cache-aside strategy will cause the updated data to be retrieved from the data store and added back into the cache.

3.1 Implementation

Consider the following points when deciding how to implement this pattern:

3.1.1 *Lifetime of Cached Data*.

Many caches implement an expiration policy that causes data to be invalidated and removed from the cache if it is not accessed for a specified period. For cache-aside to be effective, ensure that the expiration policy matches the pattern of access for applications that use the data. Do not make the expiration period too short because this can cause applications to continually retrieve data from the data store and add it to the cache. Similarly, do not make the expiration period so long that the cached data is likely to become stale. Remember that caching is most effective for relatively static data, or data that is read frequently.

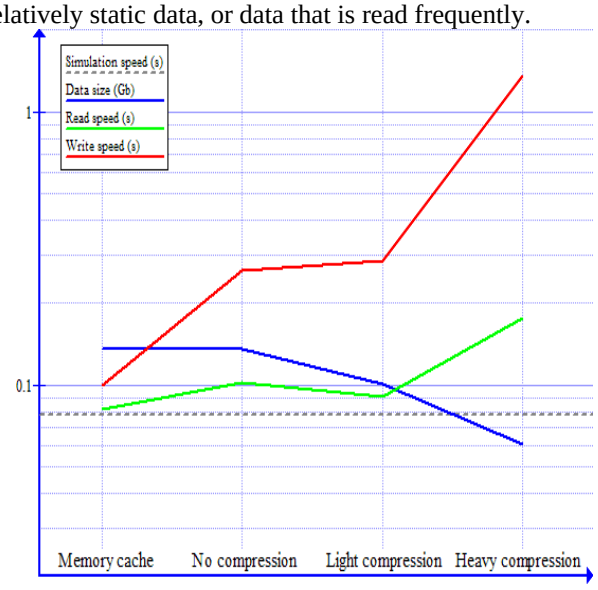

Figure 4: Life time of Cache comparison

3.1.2 *Evicting Data*.

Most caches have only a limited size compared to the data store from where the data originates, and they will evict data if necessary. Most caches adopt a least-recently-used policy for selecting items to evict, but this may be customizable. Configure the global expiration property and other properties of the cache, and the expiration property of each cached item, to help ensure that the cache is cost effective. It may not always be appropriate to apply a global eviction policy to every item in the cache. For example, if a cached item is very expensive to retrieve from the data store, it may be beneficial to retain this item in cache at the expense of more frequently accessed but less costly items.

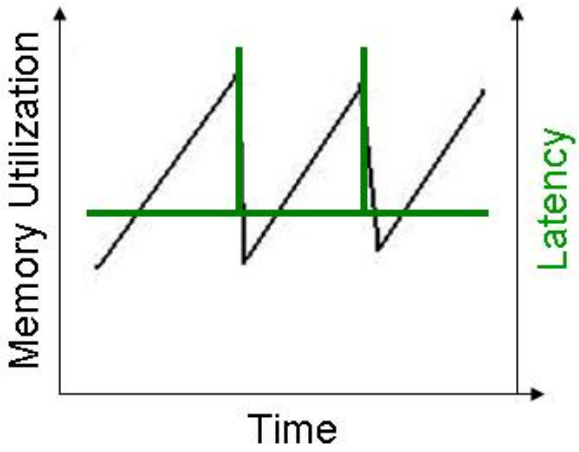

Figure 5: Eviction data

3.1.3 *Priming the cache*.

Many solutions prepopulate the cache with the data that an application is likely to need as part of the startup processing. The Cache-Aside pattern may still be useful if some of this data expires or is evicted. Implementing the Cache-Aside pattern does not guarantee consistency between the data store and the cache. An item in the data store may be changed at any time by an external process, and this change might not be reflected in the cache until the next time the item is loaded into the cache. In a system that replicates data across data stores, this problem may become especially acute if synchronization occurs very frequently.

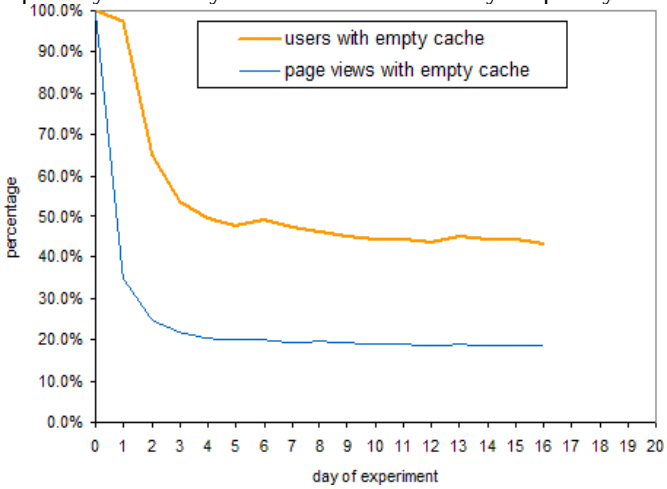

Figure 6: Priming the cache

3.1.4 *Local (in-memory) caching*.

A cache could be local to an application instance and stored in-memory. Cache-aside can be useful in this environment if an application repeatedly accesses the same data. However, a local cache is private and so different application instances could each have a copy of the same cached data. This data could quickly become inconsistent between caches, so it may be necessary to expire data held in a private cache and refresh it more frequently. In these scenarios it may be appropriate to investigate the use of a shared or a distributed caching mechanism.

4. APPLICATION

- A cache does not provide native read-through and write-through operations.
- Resource demand is unpredictable. This pattern enables applications to load data on demand. It makes no assumptions about which data an application will require in advance.

5. FUTURE WORK

When the cached data set is static. If the data will fit into the available cache space, prime the cache with the data on startup and apply a policy that prevents the data from expiring. For caching session state information in a web application hosted in a web farm. In this environment, you should avoid introducing dependencies based on client-server affinity. Most applications will include diagnostics features that generate custom monitoring and debugging information, especially when an error occurs. This is referred to as instrumentation, and is usually implemented by adding event and error handling code to the application. The process of gathering remote information that is collected by instrumentation is usually referred to as telemetry.

6. CONCLUSION

In Windows Azure you can use Windows Azure Cache to create a distributed cache that can be shared by multiple instances of an application. The GetMyEntityAsync method in the following code example shows an implementation of the Cache-aside pattern based on Windows Azure Cache. This method retrieves an object from the cache using the read-though approach. An object is identified by using an integer ID as the key. The GetMyEntityAsync method generates a string value based on this key (the Windows Azure Cache API uses strings for key values) and attempts to retrieve an item with this key from the cache. If a matching item is found, it is returned. If there is no match in the cache, the GetMyEntityAsync method retrieves the object from a data store, adds it to the cache, and then returns it (the code that actually retrieves the data from the data store has been omitted because it is data store dependent). Note that the cached item is configured to expire in order to prevent it from becoming stale if it is updated elsewhere.

REFERENCES

1. MSDN Library. Microsoft. Retrieved 12 February 2013.
2. "Capacity Planning Considerations for Windows Azure Caching". MSDN Library. Microsoft. Retrieved 13 February 2013.
3. "Windows Azure Caching on Dedicated Roles". MSDN Library. Microsoft. Retrieved 13 February 2013.
4. "Windows Azure Caching on Existing Roles". MSDN Library. Microsoft. Retrieved 13 February 2013.
5. "About Windows Azure Caching, Cache Cluster". MSDN Library. Microsoft. Retrieved 13 February 2013.
6. "How to Use Windows Azure Caching". Windows Azure Website. Microsoft. Retrieved 13 February 2013.
7. "Windows Azure Caching Role Configuration Settings (ServiceConfiguration.cscfg)". MSDN Library. Microsoft. Retrieved 13 February 2013.
8. "About Windows Azure Caching, Named Caches". MSDN Library. Microsoft. Retrieved 13 February 2013.
9. "Getting Started with Development for Windows Azure Caching, Configure the Clients". MSDN Library. Microsoft. Retrieved 13 February 2013.
10. "Windows Azure Caching Client Configuration Settings (Web.config), dataCacheClients". MSDN Library. Microsoft. Retrieved 13 February 2013.
11. "About Windows Azure Shared Caching". MSDN Library. Microsoft. Retrieved 13 February 2013.
12. "Understanding Quotas for Windows Azure Shared Caching". MSDN Library. Microsoft. Retrieved 13 February 2013.
13. "Windows Azure Shared Caching FAQ". MSDN Library. Microsoft. Retrieved 13 February 2013.
14. "Differences between Caching On-Premises and in the Cloud". MSDN Library. Microsoft. Retrieved 13 February 2013.
15. "Introducing the Windows Azure Caching Service". MSDN Magazine. Microsoft. Retrieved 13 February 2013.
16. "Windows Azure Caching Release Notes (October 2012)". MSDN Library. Microsoft. Retrieved 13 February 2013.