Couchbase

Dealing with Memcached Challenges

Replacing a Memcached Tier With a Couchbase Cluster

Table of Contents

Summary 3

Introduction 3

Memcached Overview 4

- Memcached Uses – 4
- Hashing and Data Access – 4
- Memcached Deployment Topology – 7
- Problems of Memcached – 7
- Cold Cache – 8
- Lack of Scale-Out Flexibility – 8
- Complex Monitoring and Lack of Operational Effectiveness – 8
- Stale Data Access – 8

Replacing Memcached Tiers with a Couchbase Server Cluster 9

- Benefits of using Couchbase Server and How it Solves Memcached Challenges – 10
- Auto-Sharding – 10
- High-Availability and Failover – 11
- Cluster Management and Monitoring – 13
- Strategies for Replacing a Memcached Tier with Couchbase Server cluster – 14
- Considerations when Replacing a Memcached Tier with Couchbase Server – 15
- Ejection Instead of Eviction – 15
- Memory Pressure and Bulk Loading – 16
- Increased I/O Requirements – 16
- Sizing Changes – 16
- Database Warm Up – 16

The Next Step : Using Couchbase Server as a Primary Data Store 17

Conclusion 17

References 18

About Couchbase 18

Summary

Memcached is an open-source caching technology that is used by 18 of the top 20 most trafficked websites in the world as well as several other high demand online applications. It is primarily used as a performance-enhancing complement to the RDBMS layer and over the past decade has seen extremely wide adoption owing to its two main properties: low latency and ease of programmability. However, there are several data management and operational challenges that still affect memcached users. This white paper describes the issues you might encounter when using memcached along with why these problems occur. Finally, we introduce Couchbase Server, a NoSQL database that can be used as a drop-in replacement for memcached servers to eliminate these problems and highlight some of the considerations when migrating existing memcached-based applications to Couchbase Server.

Introduction

Today's web applications typically use a caching tier to improve online user experience, application responsiveness and the ability to scale to a massive number of users. A memcached tier assists in sharing the workload of the relational database (RDBMS) to improve performance, however, several memcached problems such as cold cache, stale data access, lack of monitoring agility and scale-out flexibility still remain.

This paper focuses on these problems and explains how Couchbase Server, a NoSQL database, can be used to replace memcached, addressing these issues through high-availability, consistent performance and easy linear scalability. If you're already familiar with memcached, you might want to skip the next few sections – which highlight the most common memcached uses and present some typical memcached deployment architectures – and go directly to the section detailing the causes of common data management and operational problems affecting memcached environments. The final section describes how Couchbase Server can eliminate these issues, presents some considerations that you need to keep in mind when migrating an existing memcached tier to Couchbase Server and describes how Couchbase Server can be used as a primary data source to streamline the data tier for your applications.

Memcached Overview

Memcached is an Open Source (BSD licensed), distributed, main-memory-based object caching system that provides a unified layer of memory access across a set of independent nodes over a network.

Memcached Uses

Several internet services use memcached as a caching tier in the data-retrieval hierarchy, either for optimizing disk I/O, where the memcached tier is used to shed load from the RDBMS layer, or for optimizing CPU, where the memcached tier is used to cache expensive-to-compute values. For example:

1. Application user information such as user preferences, profiles, credentials and status.
2. Application object metadata for active media content belonging to users, photo lists, etc.
3. Application session information such as application session state, data passed across different web pages, etc. These values typically have an expiry time associated with them.
4. Non-specific, general-purpose data values such as directly displayable static page elements, tag clouds, etc.

Hashing and Data Access

The memcached API is simple to use and distribution of data is entirely accomplished on the client side. Memcached nodes use the amount RAM you specify and if the memory becomes full, it will evict older data based on LRU. Data in memcached is stored completely in memory and this offers low latency for data access.

Memcached clients have a list of memcached server node addresses (IP address and port) and use a consistent hashing algorithm (ketama) to determine which memcached node caches a key. As illustrated in Figure 1, consistent hashing forms a keyspace called the continuum. The output range of a hash function is treated as a fixed ring. Each node is assigned a random value within this space, which represents its position on the ring. Each data item identified by a key is assigned a node by hashing the data item's key to yield a position on the ring, and then walking the ring in a clockwise manner to find the first node with a position larger than the item's position.

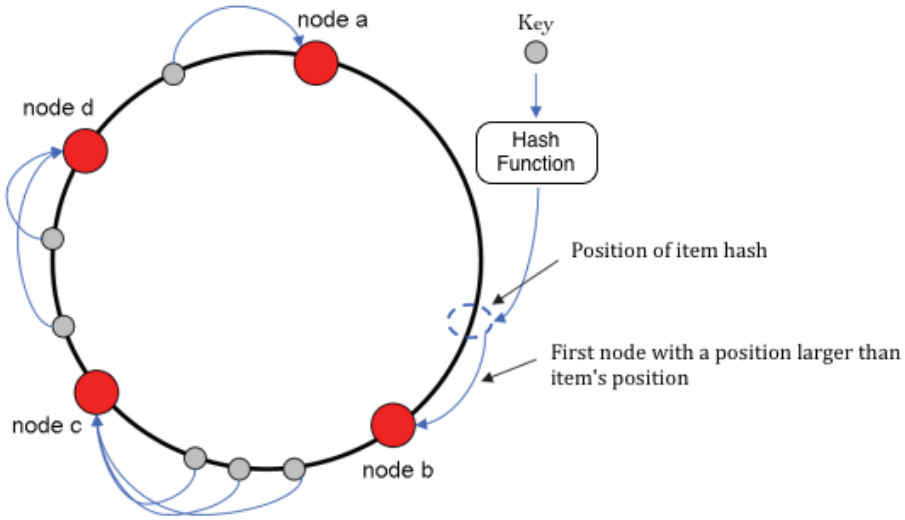

Figure 1: Consistent Hashing Continuum

As shown in the code fragment below (Figure 2), memcached servers are checked for the existence of a data key, which is derived from the requested data. If the data key does not exist on a memcached node, a query is made to the RDBMS layer and the result of the query is stored with the data key in the memcached nodes. Subsequent requests for the same query are serviced through the memcached nodes without impacting the RDBMS layer if the data still exists in the cache.

```
//Fetch the data from the cache
$data = $cache->get("cached_item");

//Check if the data was fetched correctly
if($cache->getResult() == Cache::NotFound) {
  //Cache item not found. So query RDBMS (expensive)
  $data = queryRDBMS();

  //Save the data into the cache, Expires after 30 seconds.
  $cache->set("cached_item", $data, 30);
}

//Display the data
print $data;
```

Figure 2: Data access pattern in applications using memcached and a RDBMS

To modify the data, applications must update the database and also update or delete the data in the memcached nodes. In some cases applications may use expiration times when the application can handle stale data.

When a memcached node fails or restarts, all keys stored on that node are lost and clients may need to be configured to route the request around the dead machine or machines and use the remaining active nodes.

Memcached Deployment Topology

Figure 3 below illustrates the most common deployment topology of memcached, often referred to as a side cache. When memcached is used as a cache, only a portion of the application data is available in the cache and the remaining portion of the data resides in the RDBMS. Data reads are typically serviced through the cache and data writes are handled by the RDBMS. In the case of cache misses (i.e. when data might have been previously evicted from the cache or not fetched into the cache at all) or data writes, the RDBMS layer handles the requests.

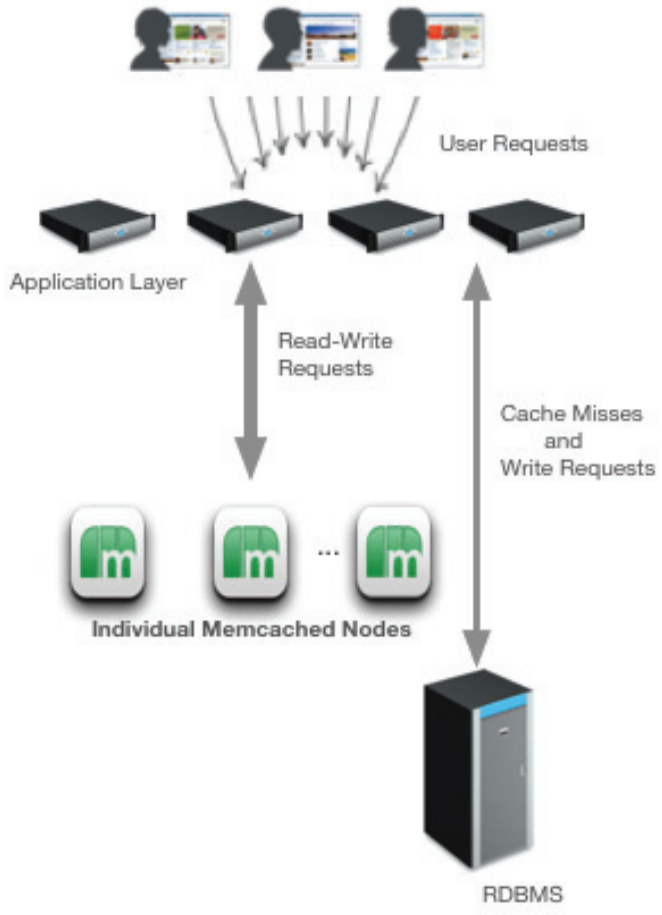

Figure 3: High-level architecture diagram using memcached as a caching layer

Problems of Memcached

In spite of its performance and scalability benefits, there are a few major issues associated with memcached, as follows:

Cold Cache

Sudden failure or offline maintenance of a memcached node causes unpredictable degradation of application performance because of cold cache. Since the new memcached server node is empty at the start, all data requests need to be serviced by the database servers in the RDBMS layers until that cache node warms up. This can significantly overload the RDBMS layer causing slowdown or collapse of the system.

Lack of Scale-Out Flexibility

As the storage needs of an application's caching tier grow, it is necessary to add more capacity. Most commonly, more commodity servers are added to the cluster (scale-out) instead of adding more RAM to the individual servers (scale-up). The process of adding or removing nodes to a memcached tier is complicated and causes unpredictable application performance degradation due to increasing pressure on the RDBMS layer.

If a new node is added to an existing N node memcached tier, around 1/(N+1) th of the keys need to be remapped to different nodes. Additionally, adding or removing nodes can cause incorrect data, or no data at all, to be returned to the clients because if the clients are not updated with the new key remapping locations, the application will possibly query another memcached node that does not have the data or has a stale version of it. This may force the application to send the query to the RDBMS layer, or return incorrect data to the user.

Complex Monitoring and Lack of Operational Effectiveness

Memcached is not a clustered solution because memcached nodes are independent and unaware of the presence or state of other memcached nodes. Additionally, memcached does not provide a built-in monitoring solution that gives a single unified view into the health and effectiveness of all the memcached tier nodes. The lack of consolidated management increases the monitoring complexity and reduces the operational efficiency making it harder to troubleshoot problems in memcached tiers.

Stale Data Access

There is no strict state for where a given key lives in memcached ketama hashing. In the absence of up-to-date key-server remapping info, clients might read or write a key from a wrong memcached server and that will lead to either stale or inconsistent data. For example, if there is any network disruption, and one or more clients decide that a particular memcached server is not available anymore, they will automatically rehash some data into the rest of the nodes even if the original one is still available. If the node eventually returns to service (for example after the network outage is resolved), the data on that node will be out of date and the clients without updated key-server remapping info will read stale data.

Replacing Memcached Tiers with a Couchbase Server Cluster

Couchbase Server is an Apache 2.0 licensed, distributed, high-performance, shared-nothing architecture NoSQL document-oriented database. Several organizations ([3],[4]) have successfully replaced memcached tiers with a Couchbase Server cluster. Replacing a memcached tier with Couchbase Server (as shown in Figure 3) continues to provide low latency, consistent performance, while adding linear scalability and simple monitoring across the cluster ([1],[6]).

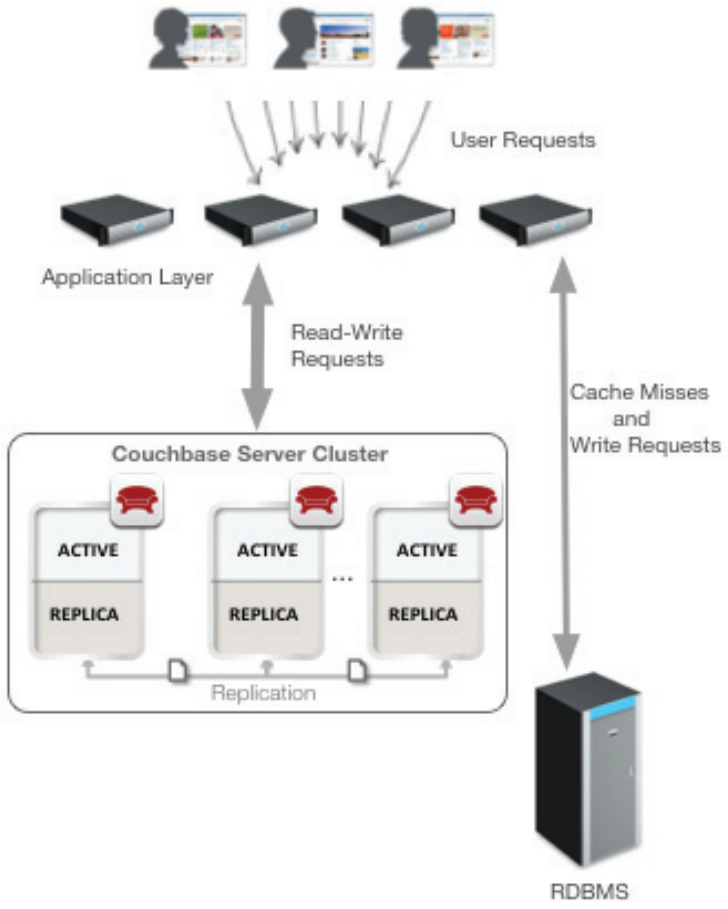

Figure 3: Architecture of using Couchbase Server cluster as a caching layer to replace memcached

Benefits of using Couchbase Server and How it Solves Memcached Challenges

Couchbase Server includes built-in caching technology enabling sub-millisecond response times that matches that of memcached. In addition to the built-in cache and in order to eliminate the challenges with memcached, Couchbase Server also provides auto-sharding, high-availability, failover and built-in cluster management and monitoring that help overcome the challenges of memcached.

Auto-Sharding

The basic unit of data manipulation in Couchbase Server is a document (JSON or binary data). Each document is associated with a key. As illustrated in Figure 4, the key space in Couchbase Server is partitioned based on a dynamically computed hash function (CRC32) into logical storage units called vBuckets. VBuckets are mapped to nodes across a cluster and stored in a lookup structure called the cluster map. Once a vBucket identifier has been calculated, a current copy of the cluster map is consulted within the client library to lookup the Couchbase Server node currently storing the active document associated with the key. The cluster map is shared among all the nodes in the Couchbase cluster as well as the Couchbase clients.

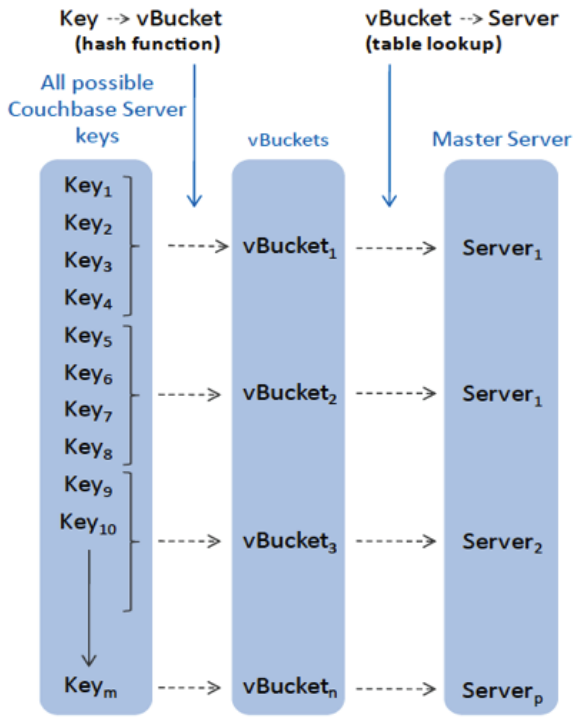

Figure 4: Mapping keys to vBuckets and vBuckets to Couchbase Server nodes

The vBucket mechanism in Couchbase Server provides a layer of indirection between the hashing algorithm and the server responsible for a given key. The number of vBuckets always remains constant (1024 by design) regardless of the server topology.

When the number of nodes in the cluster changes (scaling-out or due to failures), redistribution of vBuckets is required to ensure that data is evenly distributed throughout the cluster and that application access to the data is load-balanced evenly across all the cluster nodes. This is triggered through a rebalance operation. When a rebalance operation is initiated, the orchestrator node computes the new cluster map that has the balanced distribution of the vBuckets and sends this cluster map to all nodes and clients of the cluster. Data is then moved via vBucket migration directly between the source and the destination server nodes of the cluster. As each vbucket is moved from one location to another, an atomic and consistent switchover takes place, and each connected client library is updated with the change. This consistent switchover enforces strict states for the location of the data, which is enforced by the server and obeyed by the client library to guarantee data consistency. Auto-sharding in Couchbase Server allows for online growth or reduction of system capacity without disrupting operations or suspending data access. The incremental behavior allows new capacity to begin taking load nearly immediately and allows for the rebalance operation to be paused and restarted without having to undo or redo any of the completed movements.

High-Availability and Failover

High availability in the Couchbase Server cluster is achieved through replication at the vBucket level. Replication of data and failover in a Couchbase Server cluster provides consistent performance eliminating cold cache and sudden load shifts to the RDBMS layer.

Couchbase maintains multiple copies of data within the cluster by replicating the active vBuckets (represented by A) on one node to its replica vBuckets on other nodes (represented by R). For example, in Figure 5, the first active vBucket on node S1 has replicas on nodes S2 and S3 respectively. The number of replicas and the failover policy is configurable.

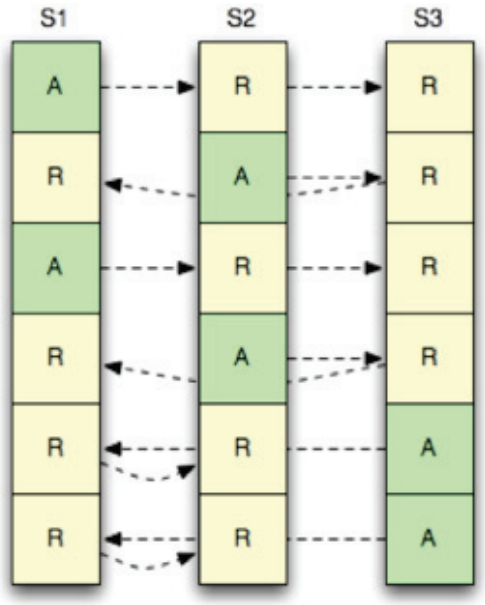

Figure 5: Replication in Couchbase Server

If a machine in the cluster crashes or becomes unavailable, the current cluster orchestrator will notify all the other machines in the cluster and, the replica vBuckets corresponding to the vBuckets on the down server node are made active. The cluster map is then updated on all the cluster nodes and the clients. This process of activating the replicas is known as failover, and is completed immediately once initiated. The amount of time between when a server is actually unavailable and when the failover is initiated will vary based upon whether the cluster is configured for auto- or manual failover. Additionally, the failover can be triggered by an external monitoring script via the REST API.

If the orchestrator node crashes, existing nodes will detect that it is no longer available and will elect a new orchestrator immediately so that the cluster continues to operate without any disruption.

Cluster Management and Monitoring

The Couchbase Server administration web console (in Figure 6) provides a unified view of the cluster health as well as the ability to perform granular drill-down analysis of operational statistics and information. Additionally, programmatic monitoring is possible using the REST API and a suite of command-line tools supporting integration capabilities with external monitoring systems. This simplifies your system management, monitoring and operations, enabling you to focus on the differentiating parts of your application.

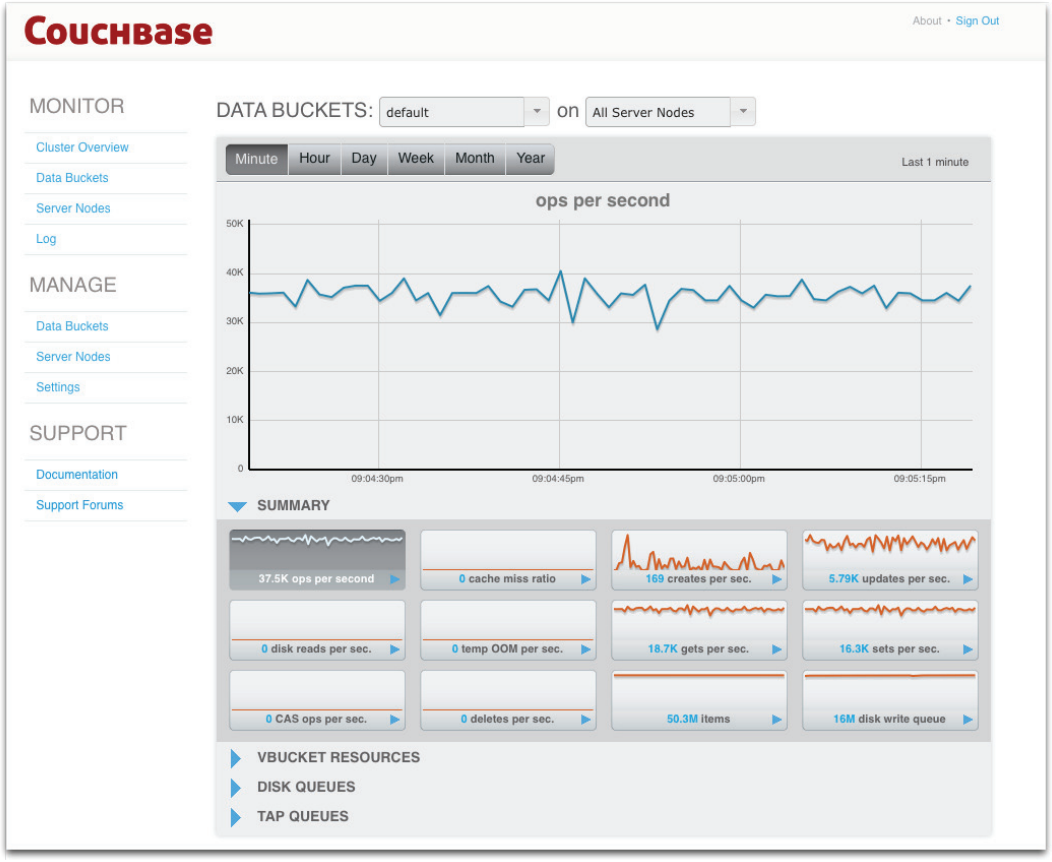

Figure 6: Couchbase Monitoring Administration Web Console

Strategies for Replacing a Memcached Tier with Couchbase Server cluster

Couchbase Server can be used as a drop-in replacement for memcached while adding replication, failover, auto-sharding and persistence.

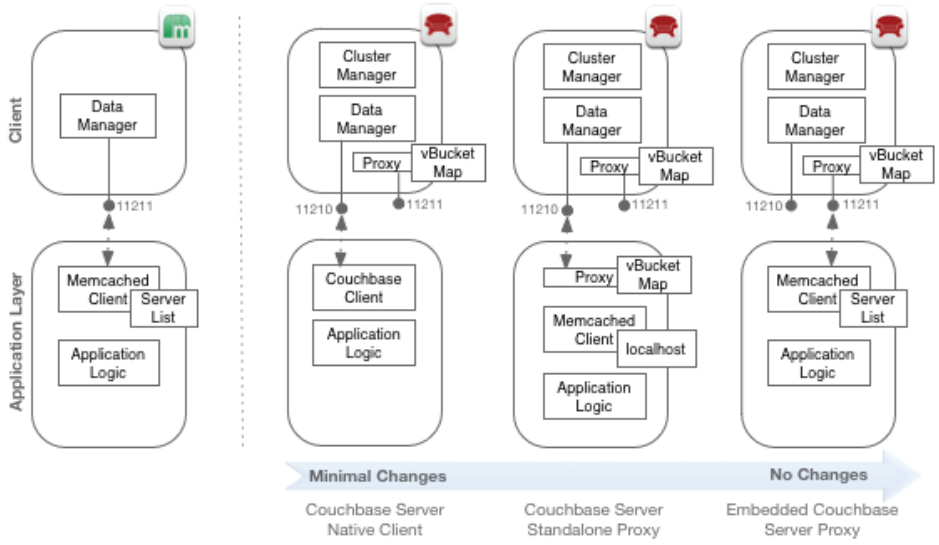

Figure 7: Deployment strategies for replacing memcached with Couchbase Server

As illustrated in Figure 7, depending on your organization's software deployment policies, following are some of the recommended best practice strategies you could choose for replacing a memcached tier with Couchbase Server cluster:

1. Using a native Couchbase client

   In this deployment strategy, since the client is vBucket aware, the client or server side proxy is not leveraged. The client directly forwards a request to the appropriate Couchbase Server cluster node that hosts the particular vBucket. Where there is flexibility to replace the client driver for an existing application or for a new application, this option should be used.

   As a best practice, we recommend using the native Couchbase client option for replacing a memcached tier with Couchbase Server. Most of the native Couchbase clients are API compatible with their language-specific memcached counterparts and thus minimal application changes (if any) are required.

2. Using a standalone proxy installed on each application server

   In this deployment strategy there are no changes to the application code but a client-side standalone proxy [7] needs to be installed to provide valuable services such as mapping memcached hashes to vBuckets and client-side connection pooling. The existing memcached client server list is configured to just have one server (localhost) so that all operations are forwarded to the localhost:11211 port that is serviced by the local proxy. The local proxy hashes the key to a vBucket, looks up the server in the vBucket map and then sends the operation to the appropriate Couchbase Server node via port 11210. It should be noted that many deployments using this strategy see increased performance over traditional memcached deployments as the proxy pools persistent connections on behalf of the application.

3. Using the embedded proxy in Couchbase Server

   In this deployment strategy there are no changes to the application code and no additional proxy software [7] needs to be installed on the client-side. Existing applications use port 11211 to communicate with the proxy that is embedded inside Couchbase Server. This proxy maps existing memcached hashes to vBuckets on the appropriate Couchbase Server cluster node.

Considerations when Replacing a Memcached Tier with Couchbase Server

While Couchbase is API compatible with the memcached protocol (including support for expiration/TTL), following are the few behavioral differences to keep in mind when replacing a memcached tier with Couchbase Server.

Ejection Instead of Eviction

As more active and frequently used data is loaded into RAM, memory pressure in the system builds up due limited amount of available memory. Couchbase Server responds to memory pressure by ejecting already persisted data from RAM. This process is called ejection. Ejection in Couchbase Server is automatic and operates in conjunction with the disk persistence system to ensure the data in RAM has been persisted to disk and can be safely ejected from the working set in the managed cache. Because the data is still available on disk, it can be served to the client when requested. The drawback to this is that instead of getting the standard 'Key Not Found' error as seen in the case of memcached evictions, you will receive a slightly higher latency response with the data itself. To reduce the response latency, additional resources may be dynamically added to the cluster. To keep SLAs within requirements when adding resources is not possible, application code may make use of timeouts on operations to take a different action when the data is not available within the required timeframe.

Memory Pressure and Bulk Loading

When you bulk load data to Couchbase Server or when Couchbase Server is under memory pressure, you can accidentally overwhelm available memory in the Couchbase cluster before it can store data on disk. In such situations, Couchbase Server may be forced to return a 'temporarily out of memory' error message instead of simply throwing away data to make room for more (as memcached does). However, this is typically a transient error condition and can be mitigated by introducing an exponential back-off as part of your bulk load [9].

Increased I/O Requirements

Due to the persistent nature of Couchbase Server, a Couchbase node usually uses much more disk I/O compared to memcached nodes. Since disk I/O in Couchbase Server is asynchronous [8], it is not in the critical path and thus there is minimal impact on applications interfacing with Couchbase Server. Couchbase will also automatically de-duplicate repeated writes to the same item in a short period of time so each write doesn't necessary correspond to disk I/O. Additionally, due to replication, network traffic is increased but since the number of replicas is configurable bandwidth usage can be optimized.

Sizing Changes

Couchbase Server includes a built-in caching technology enabling sub-millisecond response times that matches that of memcached. In addition, you get benefits of auto-sharding, replication and persistence. To provide these benefits, Couchbase Server requires about 150 bytes of meta-data per item stored (memcached requires about 80). Additionally, the optional addition of replica copies of the data means RAM being taken up to store them as well. It might be useful to take a look at the Couchbase Server sizing guide [2] to understand how your application workload might map into physical resources.

Database Warm Up

Since Couchbase Server is a NoSQL database, it provides persistence of data. If a Couchbase Server node is starting up for the first time, it will create whatever DB files are necessary and begin serving data immediately. However, if there is already data on disk (likely because the node rebooted or the service restarted) the node needs to read the meta-data for this database from disk before it can begin serving data. This is called warm up. As an optimization, Couchbase Server will also load a portion of the data into memory while scanning for metadata, thereby avoiding a cold cache once metadata is loaded. Depending on the amount of data, this can take some time but it can be monitored [5].

The Next Step : Using Couchbase Server as a Primary Data Store

Once you have started using Couchbase Server as a replacement for your memcached tier, you can further streamline your data tier by using a Couchbase Server cluster as the primary data store, without an RDBMS. Getting rid of the RDBMS and having one less component in the data access tier reduces the operational complexity. Additionally, the high-availability, auto-sharding, replication and simple monitoring capabilities provided by Couchbase Server make it an ideal NoSQL database to take your data tier to the next level. In cases where querying and indexing capabilities are required, Couchbase Server 2.0 [10] can be used to efficiently sort data and query data ranges, just like an RDBMS.

Conclusion

Couchbase Server NoSQL database can be used as a drop-in replacement for your memcached tier, delivering low latency and high availability for consistent application performance, while simplifying scalability and management. By replacing the memcached tier with a Couchbase Server cluster, you can leverage the high-availability, replication and simple monitoring capabilities to eliminate typical memcached challenges like cold cache, stale data access, lack of scale-out flexibility and complex cache monitoring. Additionally, you can also use Couchbase Server as a primary data store to streamline your data tier and take it to the next level.

References

[1] Understanding Cisco SolarFlare Performance Benchmark of Couchbase Server. Retrieved 2012, from http://blog.couchbase.com/understanding-performance-benchmark-published-cisco-and-solarflare-using-couchbase-server

[2] Couchbase Server 1.8 Sizing Best Practices. Retrieved 2012, from http://www.couchbase.com/docs/couchbase-manual-1.8/couchbase-bestpractice-sizing.html

[3] Case Study – Nami Media Uses Couchbase to Grow their Business. Retrieved 2012, from http://www.couchbase.com/case-studies/namimedia

[4] Case Study – Concur Scales Market-Leading SaaS application with Couchbase Server. Retrieved 2012, from http://www.couchbase.com/case-studies/concur

[5] Monitoring Couchbase Server 1.8 Warmup. Retrieved 2012, from http://www.couchbase.com/docs/couchbase-manual-1.8/couchbase-monitoring-startup.html

[6] Couchbase Server Overview. Retrieved 2012, from http://www.couchbase.com/couchbase-server/overview

[7] Moxi Proxy Manual, Retrieved 2012, from http://www.couchbase.com/docs/moxi-manual-1.8/

[8] Disk Write Queue in Couchbase Server 1.8, Retrieved 2012, from http://www.couchbase.com/docs/couchbase-manual-1.8/couchbase-monitoring-diskwritequeue.html

[9] Bulk Load and Exponential Back-off using Java SDK, Retrieved 2012, from http://www.couchbase.com/docs/couchbase-sdk-java-1.0/java-sdk-bulk-load-and-backoff.html

[10] Introducing Couchbase Server 2.0, Retrieved 2012, from http://www.couchbase.com/couchbase-server/next

About Couchbase

Couchbase is the NoSQL database leader, with production deployments at AOL, Deutsche Post, NTT Docomo, Orbitz, Salesforce.com, Turner Broadcasting Systems, Zynga and hundreds of other household names worldwide. Couchbase Server is the simple, fast, elastic NoSQL database that delivers a more scalable, high-performance, and cost-effective approach to data management than relational database technology. It is particularly well suited for web applications deployed on virtualized or cloud infrastructures, and for applications requiring real-time data synchronization between mobile devices and the cloud. Couchbase, the privately held company behind the Couchbase open source project, is funded by Accel Partners, Ignition Partners, Mayfield Fund and North Bridge Venture Partners. www.couchbase.com