VERSION 1.0
MARCH 30, 2018

OmniTier

MemStac™: A High Performance, Low Cost Memcached Solution

Table Of Contents

MemStac™ Overview

OmniTier's MemStac™ is a Memcached compatible key-value solution that provides a much higher cache capacity at significantly lower cost than Memcached. It achieves higher capacity by using tiered-memory based on DRAM and NVMe SSD. With an advanced data classification and latency management algorithms, MemStac™ optimally tiers data objects of varying sizes and frequency across a pool of DRAM and NVMe SSDs. The result is a much greater cache capacity that scales at the price point of SSDs and mirrors the performance of DRAM-only solutions like Memcached.

MemStac™ on AWS

MemStac™ can be launched on i3.4xlarge, i3.8xlarge instances to get Memcached compatible capacities of 3800GB and 7600GB per instance, respectively. Any standard Memcached client can interact with MemStac™ without application level or client library level changes. MemStac™ on i3 instances can enable up to 30x higher cache capacity per instance when compared to memory optimized AWS r4 instances (Figure 1).

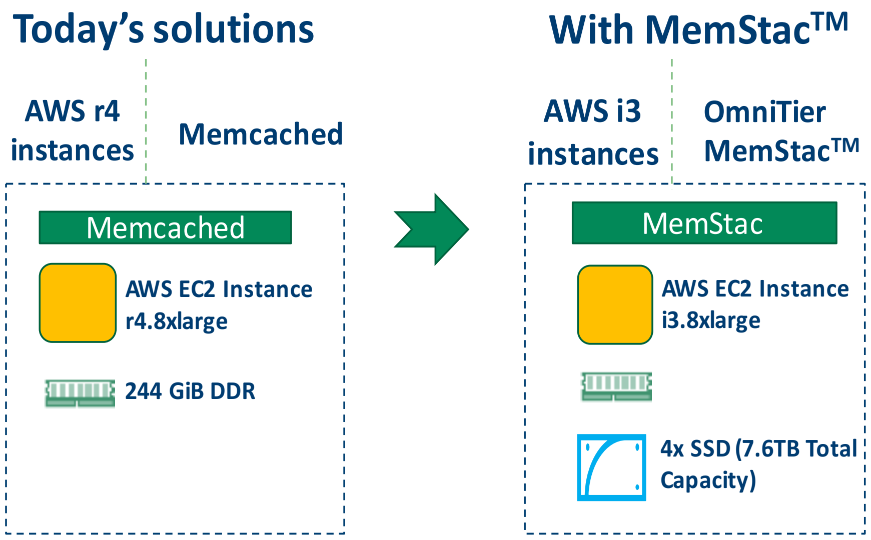

Figure 1 MemStac™ on AWS instances

MemStac™ Caching Strategies

The following section describes caching strategies to fill the cache based on the needs of the application.

Demand-Filled Look-Aside Cache

In this caching strategy, the application fills data into the cache only when the data is requested from the database so that future requests will be found in the cache.

The application first makes a retrieval request to the cache. If available, the application can use it directly. If the data does not exist in the cache, the application must request it from the database and then store it in the cache.

The advantage is that only data which is requested will be loaded into the cache. Data that is written but never read does not get loaded into the cache. The first disadvantage of this strategy is that the first request for a new key will always miss. Another disadvantage is the possibility of stale data in the cache if there is a newer copy in the database. One strategy to address the stale entries would be to issue a delete to the cache whenever a key is set to the database. Another would be to set an expiration for the key, thereby putting a bound on the staleness of the data.

Write-Through

In this approach, the cache is updated along with the database. When a key is set, the application sends storage request to both the cache and the database.

In this way keys that were recently set are already in the cache and do not incur misses for the first retrieval requests to the cache. It also addresses the issue of stale keys in the cache when compared to the database. The disadvantage is that data that is written but seldom read, may unnecessarily occupy cache space.

It may also be desirable to fill the cache on retrieval misses when implementing a write-through strategy, otherwise a key that was written a long time ago which suddenly becomes popular may no longer be in the cache when it becomes popular.

MemStac™ Supported Commands and Client Libraries

MemStac™ supports Memcached ASCII protocol. The following section describes Memcached commands and MemStac™ support for each of these commands.

Supported Commands – Memcached ASCII Protocol

Command	Description
SET	Store this data.
ADD	Store this data, only if the key does not exist.
REPLACE	Store this data, but only if the key exists.
APPEND	Add this data after the last byte in an existing item. This does not allow you to extend past the item size limit.
PREPEND	Same as append, but adding new data before existing data.
CAS	Check and Set (or Compare And Swap) operation. An operation that stores data, but only if no one else has updated the data since you read it last.
GET	Command for retrieving data. Takes one or more keys and returns all found items.
GETS	An alternative get command that also returns a CAS identifier (a unique 64bit number) with the item.
DELETE	Removes an item from the cache.
INCR/DECR	Increment or decrement the specified counter.
FLUSH_ALL	Invalidate all existing items immediately (by default) or after the expiration specified.
TOUCH	Updates the expiration time of an existing item without fetching it.
VERBOSITY	Change verbosity of output.
VERSION	Returns version.
QUIT	Close connection to server.
STATS	Return statistics. MemStac™ stats output is different from Memcached due to architectural differences. MemStac™ also supports additional statistics reporting through HTTP. Details are in Metrics and Monitoring section.

TABLE 1: MEMSTAC™ SUPPORTED COMMANDS (MEMCACHED-ASCII PROTOCOL)

Client Libraries Tested For Compatibility

MemStac™ is fully compatible with Memcached ASCII protocol and requires no application level or client library level changes. The following section is a list of client libraries which were tested with MemStac™:

Metric	Description
C/C++	http://libmemcached.org
C#	https://github.com/enyim/EnyimMemcached
Go	https://github.com/bradfitz/gomemcache
Java	https://github.com/couchbase/spymemcached https://github.com/gwhalin/Memcached-Java-Client https://github.com/spotify/folsom
Node.js	https://github.com/3rd-Eden/memcached
PHP	https://github.com/php-memcached-dev/php-memcached
Python	https://github.com/pinterest/pymemcache
Ruby	https://github.com/petergoldstein/dalli

TABLE 2: CLIENT LIBRARIES TESTED WITH MEMSTAC™

Testing MemStac™ with a Client Library

The following section describes how to connect to MemStac™ using a Memcached client library. A PHP client is used as an example to talk to a MemStac™ instance and similar methods can be applied to other programming languages. Before running the following example, make sure PHP and Memcached client libraries are installed on your machine.

Here is a simple PHP script that shows how to connect to MemStac server using PHP Memcached client library.

```
<?php

$mc = new Memcached();
$mc->addServer(”<MemStac  EC2 IP address>”, 11211);

$mc->set("foo", "Hello!");
$mc->set("bar", "World! ");

$arr = array(
   $mc->get("foo"),
   $mc->get("bar")
);
var_dump($arr);
?>
```

The above script stores two keys “foo” and “bar” into MemStac™ and retrieves both keys and displays the values. To verify the behavior, please run:

```
$ php simple.php
```

MemStac™ Cluster Configuration

In a cache setting, data can be too big to fit on a single MemStac™ server. It may therefore be beneficial to have data split across multiple servers. This can be enabled either through a client library or a proxy.

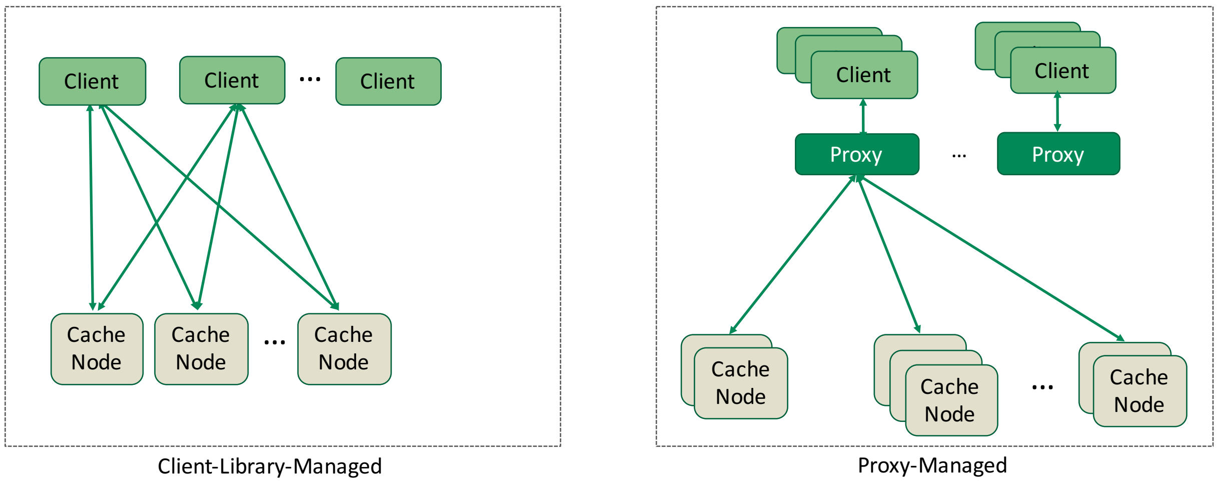

FIGURE 2: CACHE CLUSTER CONFIGURATION

Client-Library-Managed Cache Cluster

Most of the Memcached client libraries support sharding of keys across available MemStac™ instances. Simple way to shard the key is using:

```
MemStac Nodes= { “10.0.0.10:11211”, “10.0.0.11:11211, ““10.0.0.12:11211”}
index = hash(key) % length(MemStac Nodes)
CacheNode = MemStac Nodes[index]
```

Even though this method is simple, it has a severe disadvantage due to the modulo operation. As the number of MemStac™ instances in the cluster changes, most of the keys need to be remapped to match new cluster size. To address this issue, client libraries such as spymemcached and pymemcache support consistent hashing. Consistent hashing creates an internal hash ring with a pre-allocated number of partitions that can hold incoming keys. When server gets added/removed only a portion of keys need to be remapped.

Proxy-Managed Cache Cluster

An alternative to client-library-managed cached cluster is to use a proxy. A proxy, when placed between the client and the cache cluster, handles the distribution of data amongst the cluster. The client requires no modifications; existing client applications speak directly to the proxy as if it's a single MemStac™ instance. However, there is additional setup overhead for the proxy, and a slightly higher latency depending on the deployment configuration.

We will describe two most frequently used proxies and their sample usage with MemStac™: Mcrouter and Twemproxy.

Mcrouter

Mcrouter was originally developed by Facebook and is capable of connection pooling, sharding, replication of data and more. Details of different modes of operation and installation setups can be found at: https://github.com/facebook/mcrouter/wiki

Launching Mcrouter (Sharded Mode)

In most cases, a cache cluster has more than one instance of MemStac™ and the application needs to split the cached data across multiple MemStac™ instances. Mcrouter can send requests to different MemStac™ instances based on the key hash. This allows keys to be evenly distributed across MemStac™ instances and the same key will be served by the same MemStac™ instance.

Here is an example of launching Mcrouter with a single pool of two MemStac™ servers "10.0.0.28:11211" and "10.0.0.30:11211". Mcrouter shards requests to this pool using a consistent hashing.

Configuration File Example (sharded.json):

```
{
  "pools":{
   "A":{
    "servers":[
     "10.0.0.28:11211",
     "10.0.0.30:11211",
    ]
   }
  },
  "route":"PoolRoute|A"
 }
```

Launch command:

```
$ ./mcrouter -p 11211 --num-proxies=1 --config-file=sharded.json
```

Configurations can be dynamically updated to add and/or remove server IPs. Refer to the Mcrouter Wiki page for more information.

Launching Mcrouter (Replication Mode)

In large scale deployments, there is a risk that a server may go down or be inaccessible (e.g. network partition). In these scenarios, application performance could suffer since all requests serviced by the failed node will be routed to the backend data store. To avoid this scenario, along with sharding, Mcrouter can be configured for replication. All data writes to Mcrouter are replicated to each MemStac™ instance within the pool. Reads are retrieved from one of the MemStac™ instance within the pool.

The example below deploys two pairs of nodes that are pooled together. All data writes to Mcrouter are replicated and sent to all nodes within the pool (10.0.0.28,10.0.0.30). When reads are requested, Mcrouter will retrieve data from only one of the nodes within the pool since data should be present on all nodes.

Configuration File Example (replication.json):

```
{
  "pools":{
    "A":{
      "servers":[
      "10.0.0.28:11211",
      "10.0.0.30:11211",
      ]
    }
  },
  "route":{
   "type":"OperationSelectorRoute",
   "operation_policies":{
     "add":"AllSyncRoute|Pool|A",
     "delete":"AllSyncRoute|Pool|A",
     "get":"LatestRoute|Pool|A",
     "set":"AllSyncRoute|Pool|A"
   }
  }
}
```

Launch Command:

```
$ ./mcrouter -p 11211 --num-proxies=12 –config-file=replication.json
```

Cache Warm Up With Mcrouter

When a new server is added or an existing server is power cycled, the MemStac™ server, Like Memcached, would have no valid keys. Every request to this server will end up with a cache-miss and this can degrade the overall application performance because each miss will trigger a read to the backend data store. To avoid this, Mcrouter can be configured to warm up the cold cache without impacting the performance. More info on Mcrouter be found at: https://github.com/facebook/mcrouter/wiki

Twemproxy

Developed by Twitter, Twemproxy is another widely-used proxy for cache deployment. The proxy supports consistent hashing, connection pooling (reducing the total number of connections that the server side sees), and command pipelining (accumulating requests before sending to the server side to reduce network overhead).

Details of different modes of operation and configuration setups can be found here: https://github.com/twitter/twemproxy

MemStac™ is compatible with Twemproxy and it can be used to setup a MemStac™ cluster. After the installation of Twemproxy, launching the proxy (named nutcracker) results in a single CPU driven instance. If limited by a single CPU performance, multiple instances of Twemproxy can be launched to interface with MemStac™.

Launching Twemproxy

Configuration File Example:

```
omega:
 auto_eject_hosts:"false"
 distribution:ketama
 hash:hsieh
 listen:127.0.0.1:11211
 redis:"false"
 server_connections:1
 servers:
 -10.0.0.28:11211:1
 - 10.0.0.28:11212:1
```

Launch command:

```
$ ./nutcracker -v 0 --stats-port=22222 -c  nutcracker_10.0.0.25_11211.yml
```

Metrics and Monitoring

MemStac™ Metrics Service:

MemStac™ provides a service endpoint delivering a snapshot of a single node's basic metrics data. The service endpoint is <host IP address>:6060/metrics. The following table describes details of the available metrics:

Metric	Description
TotalNumConnections	Total number of connections since the node has been up
CurrentNumConnections	Current number of open connections against the node
PerOpcodeMetrics	For each operation type, the following are reported: - average latency, - operation count, - number of misses
NumItems	Total number of keys in the system

NumEvictedItems	Total number of keys evicted from the system
DiskErrors	Disk errors encountered
SmartLogs	NVME SMART logs
CpuUsage	CPU usage metrics
MemUsage	Memory usage metrics

TABLE 3: EXPORTED METRICS FROM MEMSTAC™

Performance improvements with MemStac™

Modern databases are several orders of magnitude slower than either MemStac™ or Memcached – tens of thousands of requests per second compared to millions. Since overall system performance is determined not only by the speed of the cache but the percentage of requests serviced by the cache (hit rate), insufficient cache capacity can cause the database to be the primary performance bottleneck. Due to its inherent capacity limitation, Memcached may only be able to service a small fraction of the system workload. This causes most of the requests to be directed to the database, leading to poor system performance.

MemStac™, on the other hand, with its significantly larger cache size, can economically cache a majority of the application's requests and reduce the load on the backend database, resulting in a system-level performance improvement of up to 80x. MemStac™ on AWS i3 instances can provide up to 30x higher capacity at 30x lower $/GB/Hour compared to Memcached running EC2 r4 instances.

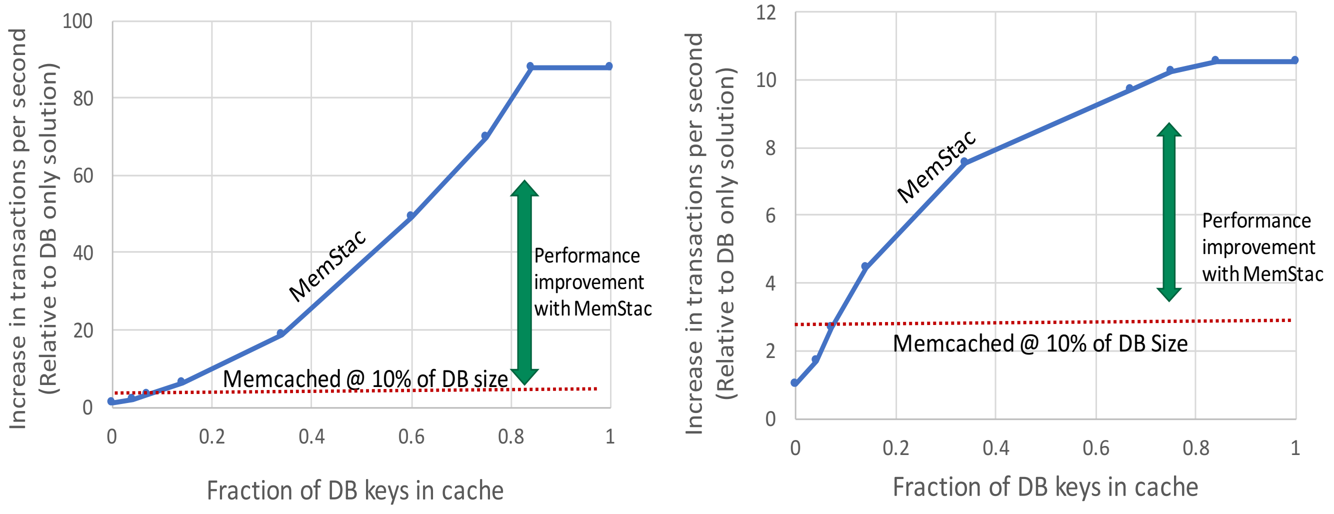

FIGURE 3: INCREASE IN TRANSACTIONS PER SECOND RELATIVE TO DATABASE ONLY SOLUTION FOR YCSB WORKLOAD-A AND WORKLOAD-B

Figure 3 summarizes the projected performance improvements with MemStac™ relative to capacity constrained Memcached (with capacity at 10% of DB size) for industry standard Yahoo Cloud Serving Benchmark workloads (YCSB) of 50% GET (workload-A) and 95% GET (workload-B) when AWS RDS/MySQL is used as the backend database.

About OmniTier

OmniTier Inc., founded in 2015, develops and supports integrated software solutions for memory-centric infrastructure applications, including high performance object caching, scientific analysis for machine learning, AI, and genomics. Its leadership team has a track record of delivering many industry firsts in data storage and access across different media types. The company has offices in Milpitas, California, and Rochester, Minnesota