Couchbase

vBuckets: The Core Enabling Mechanism for Couchbase Server Data Distribution (aka “Auto-Sharding”)

Table of Contents

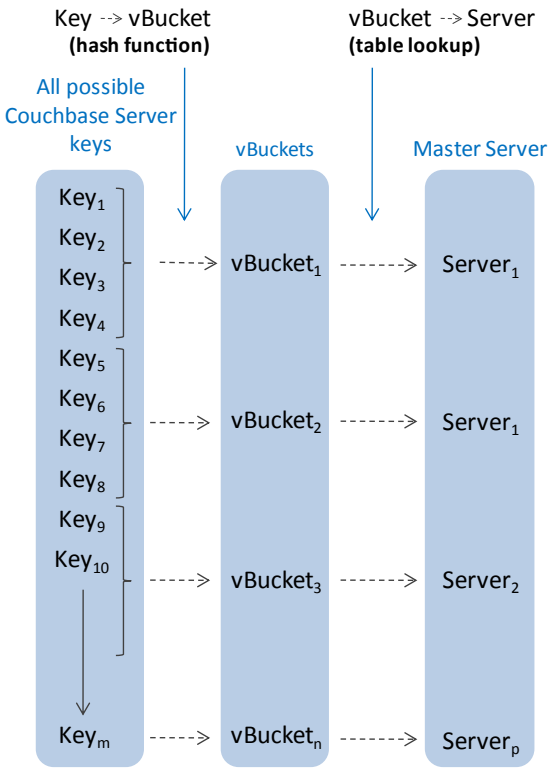
A key design goal for Couchbase Server requires Couchbase Server to support "over-the-counter" ("OTC") memcached clients while also providing data replication, failover and dynamic cluster reconfiguration. The vBucket concept is a foundational mechanism for meeting these seemingly irreconcilable requirements. In this document, we explore the concept of vBuckets in Couchbase Server, covering definitions, key mapping, and deployment options.

Note: For simplicity, in this document we completely ignore Couchbase Server multi-tenancy (the unit of multi-tenancy in Couchbase Server is the "bucket," which represents a "virtual Couchbase Server instance" inside a single Couchbase Server cluster). The bucket and vBucket concepts are not to be confused – they are unrelated. For purposes of this document, a bucket can simply be viewed as synonymous with "a Couchbase Server cluster."

vBuckets defined

A vBucket is defined as the "owner" of a subset of the key space of a Couchbase Server cluster.

Every key "belongs" to a vBucket. A mapping function is used to calculate the vBucket in which a given key belongs. In Couchbase Server, that mapping function is a hashing function that takes a key as input and outputs a vBucket identifier (each Couchbase Server cluster has a fixed number of vBuckets – determined when the cluster is first installed). Once the vBucket identifier has been computed, a table is consulted to lookup the server currently acting as the "master server" for that vBucket. The table contains one row per vBucket, pairing the vBucket to its master server. A server appearing in this table can be (and usually is) responsible for multiple vBuckets.

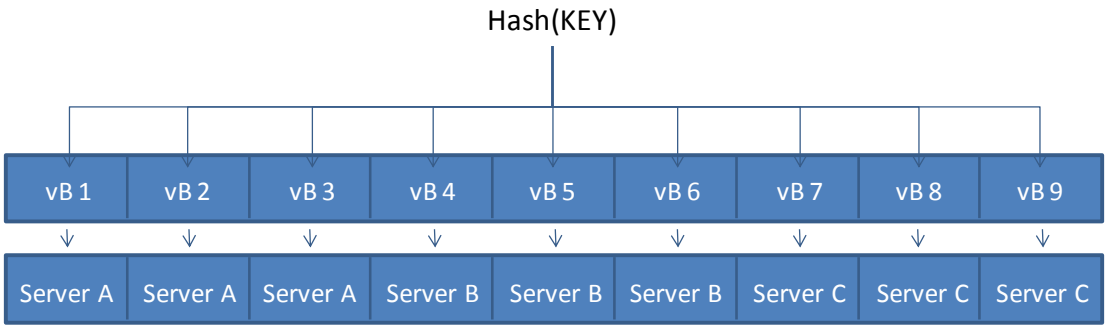


The hashing function used by Couchbase Server to map keys to vBuckets is configurable – both the hashing algorithm and the output space (the total number of vBuckets output by the function). Naturally, if the number of vBuckets in the output space of the hash function is changed, then the table which maps vBuckets to Servers must be resized.

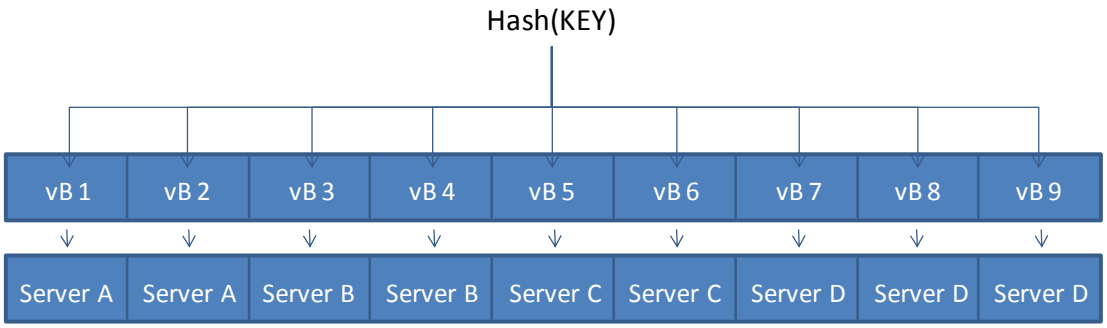
Couchbase key-vBucket-server mapping illustrated

The vBucket mechanism provides a layer of indirection between the hashing algorithm and the server responsible for a given key. This indirection is useful in managing the orderly transition from one cluster configuration to another, whether the transition was planned (e.g. adding new servers to a cluster) or unexpected (e.g. a server failure). Memcached, in contrast, has no intermediary. It uses a hashing function to directly map keys to servers (using a statically-maintained list of servers as the output space). When the server list is changed, the hashing function will remap keys to new servers. Because memcached is a cache, it just drops the data that has been "moved" and it will eventually be re-cached by the application. This doesn't work with a database. The data can't just be "dropped" – it has to be moved.

The diagram below shows how key-server mapping works when using the vBucket construct. There are three servers in the cluster. A client wants to read the value of KEY. The client first hashes the key to calculate the vBucket which owns KEY1. Assume Hash(KEY) = vBucket 8. The client then consults the vBucket-server mapping table and determines Server C is the master server for vBucket 8. The read operation is sent to Server C by the Couchbase client library.



After some period of time, there is a need to add a server to the cluster (e.g. to sustain performance in the face of growing application use). The administrator adds Server D to the cluster and the vBucket Map is updated as follows *(Note: the vBucket-Server map is updated by an internal Couchbase Server algorithm and that updated table is transmitted by Couchbase Server to all cluster participants – servers, clients and proxies)*:



After the addition, a client once again wants to read the value of KEY. Because the hashing algorithm in this case has not changed, Hash(KEY) = vBucket 8, as before. The client examines the vBucket-server mapping table and determines Server D is now the master server for vBucket 8. The read operation is sent to Server D.

vBuckets in a world of OTC memcached clients

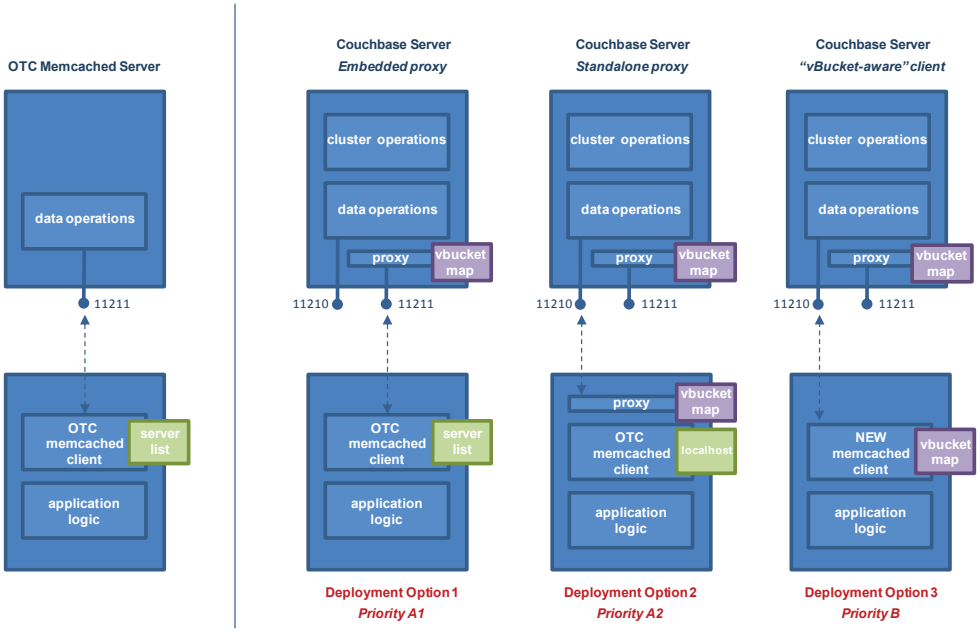
Couchbase Server is designed to be a drop-in replacement for an existing memcached server, while adding persistence, replication, failover and dynamic cluster reconfiguration. Existing applications will likely be using an old memcached client to communicate with an OTC memcached cluster. This client will probably be using a hashing algorithm to directly map keys to servers, as previously described.

To make this work, a proxy is required. Note that the optimal solution is to replace the client library with a client that implements the vBucket concept directly (though a proxy will continue to be desirable in some environments). There are vBucket-aware clients for Java, .NET, Ruby, PHP, Python and C/C++. But in this example, we assume an application is already running and that a client change is undesired.

Couchbase Server TCP ports

Couchbase Server listens for data operations on two configurable ports. TCP ports 11210 and 11211 (see figure below) are the defaults. Both ports are “memcapable,” supporting the memcached ASCII and Binary protocols (binary only on 11210).

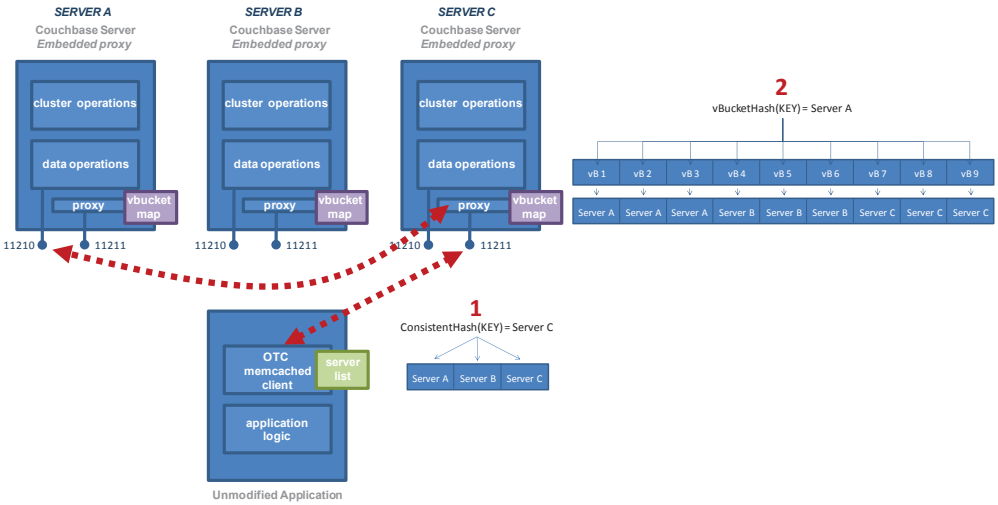
> **Port 11211** is the port on which an embedded proxy listens (11211 the traditional standard memcached port). It can receive, and successfully process, data operations for keys that are owned by vBuckets not hosted by this server. The proxy will forward the request to the right server then return the result to the client.
>
> **Port 11210** is the port on which the Couchbase Server database server listens. It will reject data operations for keys owned by vBuckets not hosted by this server. The client sends the vBucket number in the request. The vBucket is then compared with the list of vBuckets hosted by this server.



Deployment Option 1 – Using Couchbase Server embedded proxy

The first deployment option is to communicate with the embedded proxy in Couchbase Server over port 11211. This option allows you to install Couchbase Server and begin using it with an existing application, via an OTC memcached client, without also installing another piece of proxy software. The tradeoff is a potential performance impact, though Couchbase Server attempts to minimize latency and throughput degradation.

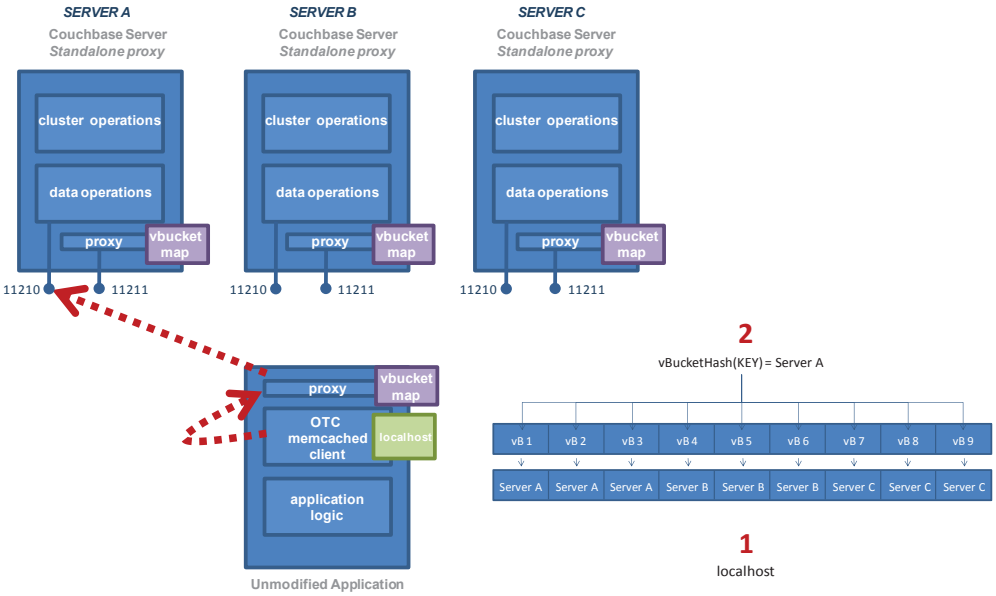
In this deployment option (as shown in detail below) versus an OTC memcached deployment, *in a worst case scenario*, server mapping will happen twice (e.g. using direct hashing to a server list on the OTC client, then using vBucket hashing and server mapping on the proxy) with an additional round trip network hop introduced.



Assume there is an existing application, with an OTC memcached client, with a server list of three servers (Servers A, B and C). Couchbase Server is installed in place of the memcached server software on each of these three servers. As shown in the figure above, when the application wants to Get(KEY), it will call a function in the OTC client library. The client library will hash(KEY) [see 1] and be directed, based on the server list and hashing function, to Server C. The Get operation is sent to Server C, port 11211 (the proxy). When it arrives to the Couchbase Server proxy port [see 2], the Key is hashed again to determine its vBucket and server mapping. This time, the result is Server A. The proxy will contact Server A on port 11210, perform the read operation and return the result to the client.

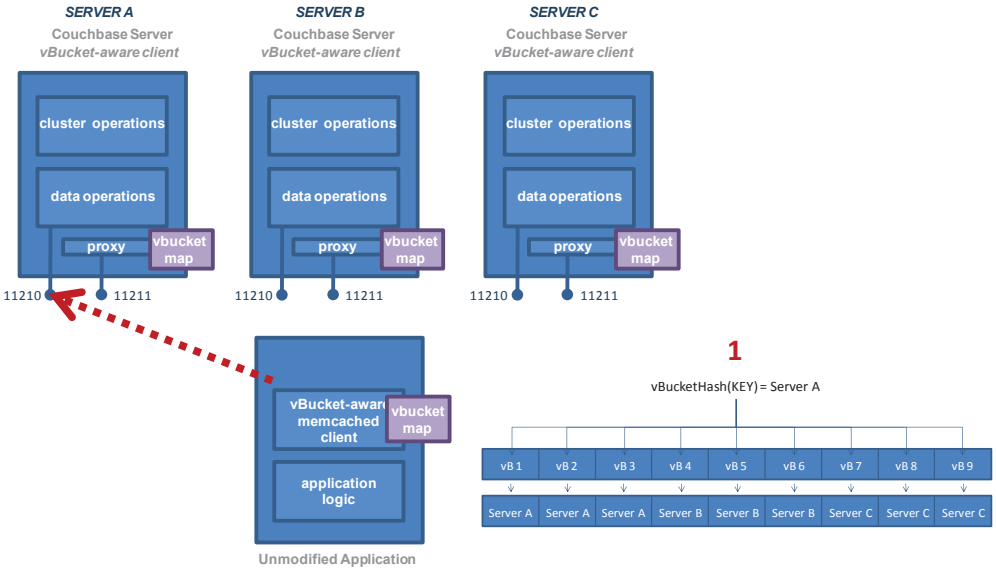
Deployment Option 2 – Standalone proxy installed on each application server

The second option is to deploy a standalone proxy, which performs substantially the same way as the embedded proxy, but potentially eliminating a network hop. A standalone proxy deployed on a client may also be able to provide valuable services, such as connection pooling. The diagram below shows the flow with a standalone proxy (the Couchbase Server proxy is called "moxi") installed on the application server. The memcached OTC client is configured to have just one server in its server list (localhost), so all operations are forwarded to localhost:11211 – a port serviced by the proxy. The proxy hashes the key to a vBucket, looks up the host server in the vBucket table, and then sends the operation to the appropriate Couchbase Server (Server A in this case) on port 11210.



Deployment Option 3 – "vBucket aware" client

In the final case, no proxy is installed anywhere in the data flow. The client has been updated and performs server selection directly via the vBucket mechanism. Where there is flexibility to replace client technology on an existing application, or for new application development, this is the highest performance option.



For more information on building and deploying applications with Couchbase Server, visit www.couchbase.com.