UVA | SCHOOL of DATA SCIENCE

Consistent Hashing for Scalable Data Storage

Big Data Systems
School of Data Science
University of Virginia

Agenda

> Horizontal scaling
> Hash function
> Consistent hashing
> Hot spots and vnodes
> uhashring

Why Support Scaling?

As businesses grow,
- data volumes grow
- users increase
- workloads become more complex

Scaling supports this growth

Horizontal Scaling of Data

How can we provide a system that scales with size of dataset (*elastic storage*)?

Consistent Hashing provides a nice solution

Review: Hash Function

A *hash function* is any function that
maps data of arbitrary size to fixed-size values

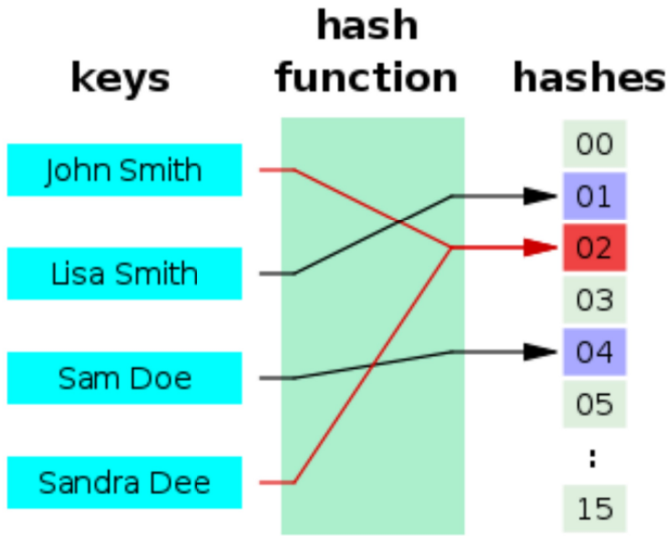
The data is the *key*

Uses *N* buckets

Distributes data uniformly across buckets

Review: Hash Function

One approach is **mod N** hash method:

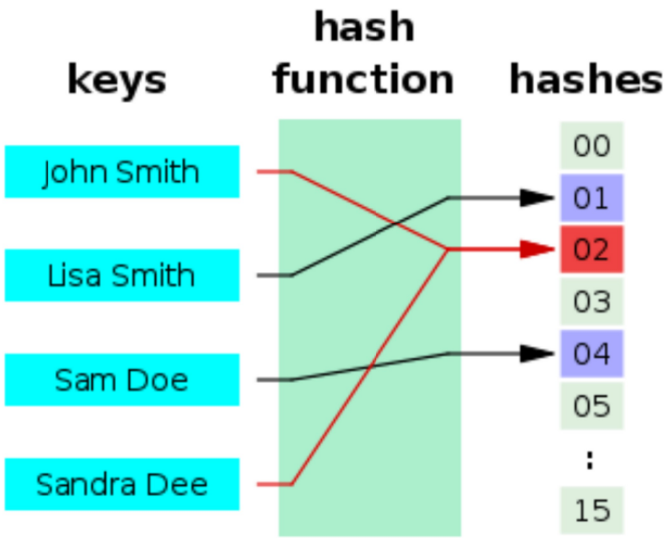


Review: Hash Function

One approach is **mod N** hash method:

server_index = hash(key) mod N

N = 16 in this example



Does mod N hashing work?

Consider data scales by factor of 10

An additional server is added

Server pool is now size N+1 = 17

Does mod N hashing work?

Consider data scales by factor of 10

An additional server is added

Server pool is now size N+1 = 17

Problem: When N changes, the *mod* operation produces new server indexes
… for nearly all data!

This would shuffle data to different servers.

Does mod N hashing work?

Consider data scales by factor of 10

An additional server is added

Server pool is now size N+1 = 17

Example when hash value = 20

20 mod 16 = 4
20 mod 17 = 3

Consistent Hashing

A special kind of hashing which uses less shuffling of data

Uses object called a *hash ring*

Hashes are mapped into slots on a line

Source of this example:
https://www.acodersjourney.com/system-design-interview-consistent-hashing/

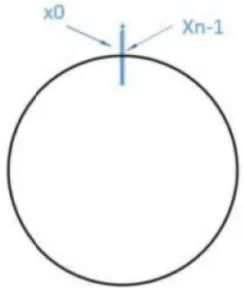
Consistent Hashing

A special kind of hashing which uses less shuffling of data

Uses object called a *hash ring*

Hashes are mapped into slots on a line

For the last value to connect to the first, the space is shaped into a ring



Source of this example:
https://www.acodersjourney.com/system-design-interview-consistent-hashing/

Consistent Hashing Illustration

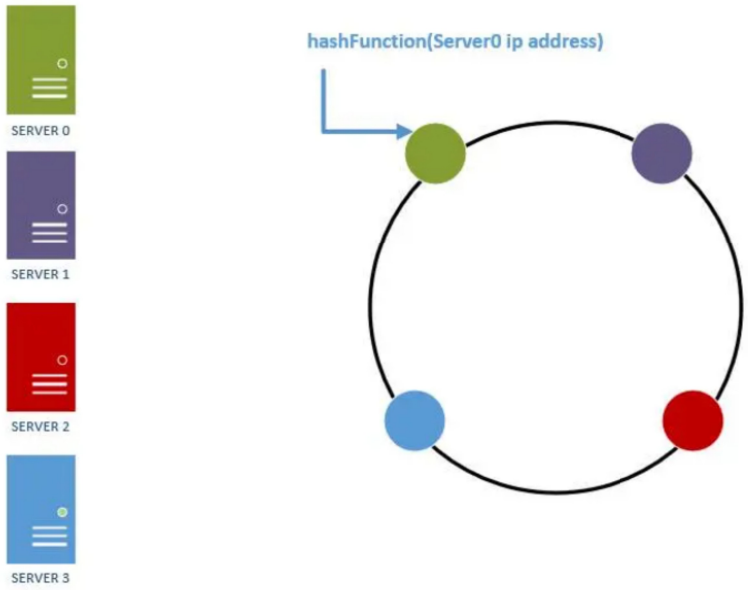
Tasks:

1. Insert key for data storage

2. Search for key (data retrieval)

Consistent Hashing Illustration, contd.

We have four servers to place on a ring

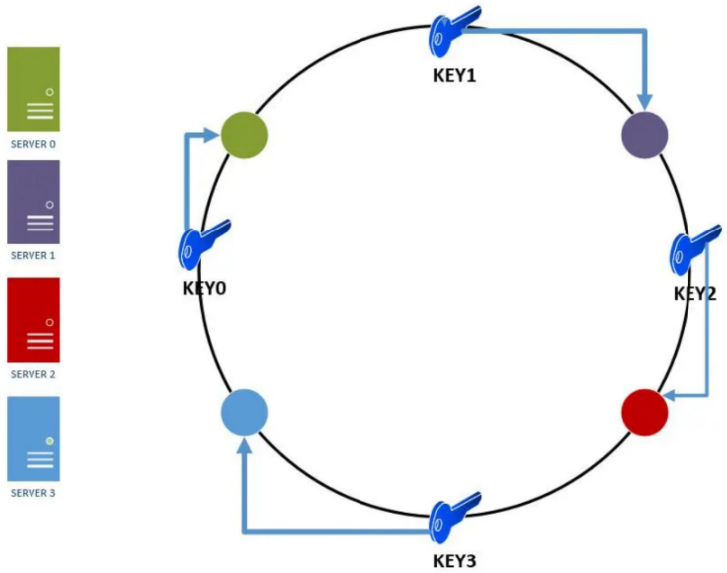


Consistent Hashing Illustration, contd.

Keys are placed on the ring

1. Key is either in same location as server or

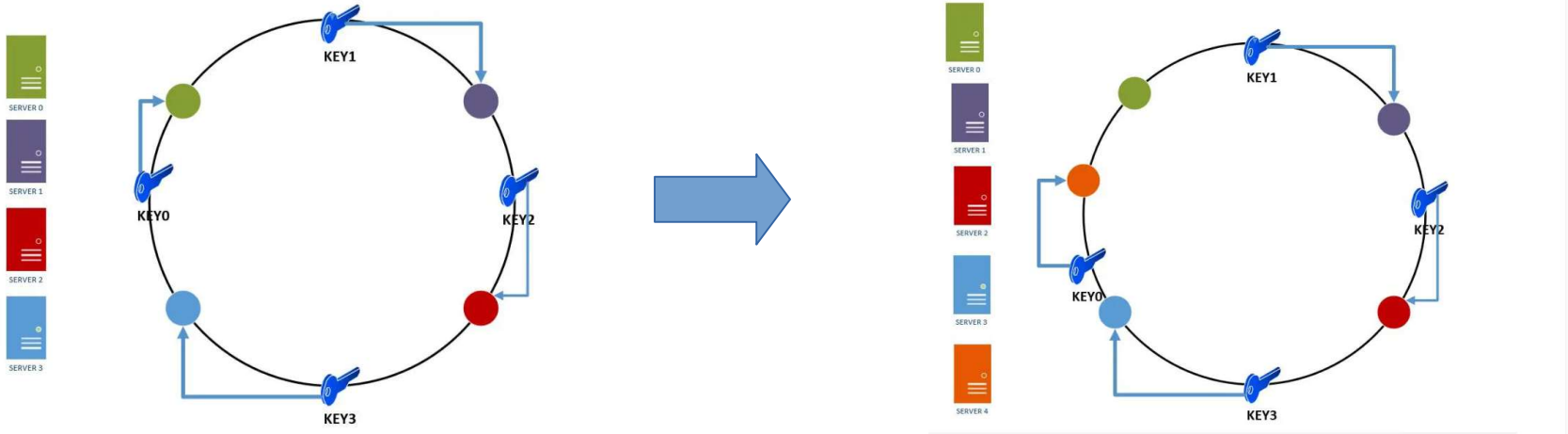
2. Look up server by moving clockwise from key



Consistent Hashing: Adding a Server

Adding a fifth server affects the ring:
key0 will be remapped from server0 to server4

Remaining keys stay on same servers,
which is a large benefit of the approach

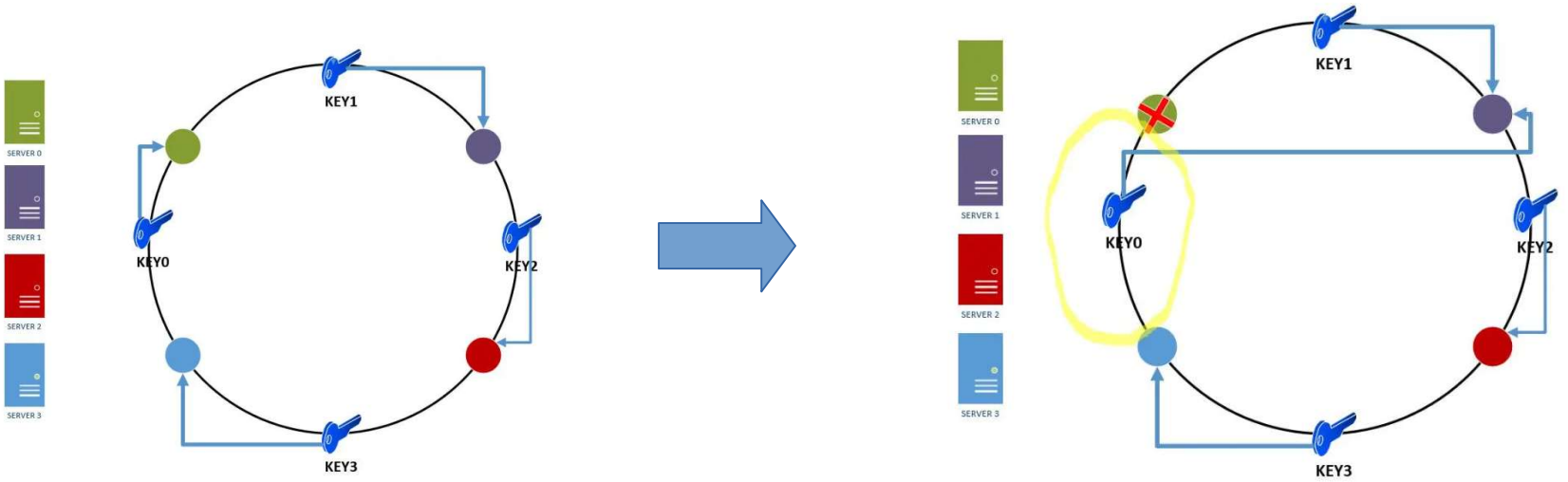


Consistent Hashing: Server Failure

In this illustration, there were 4 servers and one failed

key0 is mapped to the next available server

Remaining keys don’t move



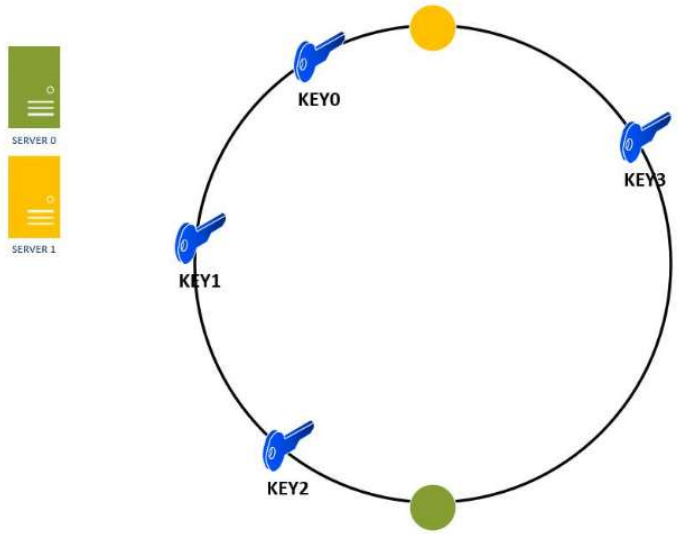
Hot Spots

The situation may arise where data is not uniformly distributed across servers

For example if a server fails, its data may be remapped to next server

In this figure, most keys will map to server 1

Server 1 is a **hot spot**



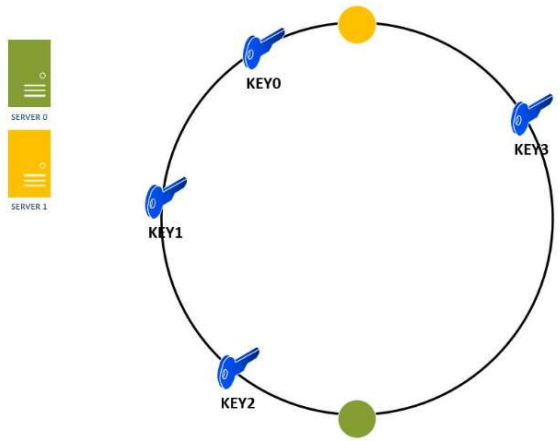
Hot Spots: Handled with Virtual Nodes

We can introduce replicas of each server on the ring

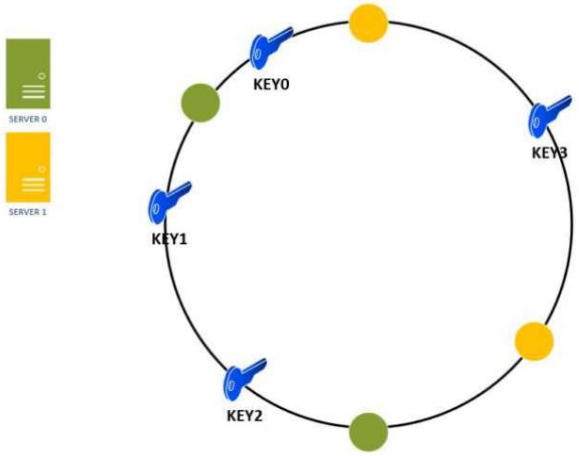
Replicas also called *virtual nodes* or *vnodes*

As the number of replicas increases, the key distribution becomes more uniform

Without vnodes



With vnodes

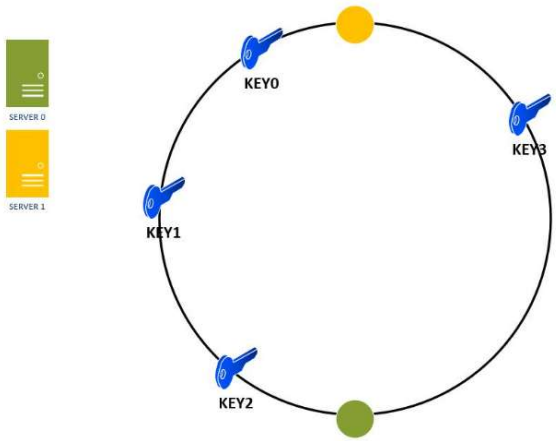


Hot Spots: Handled with Virtual Nodes

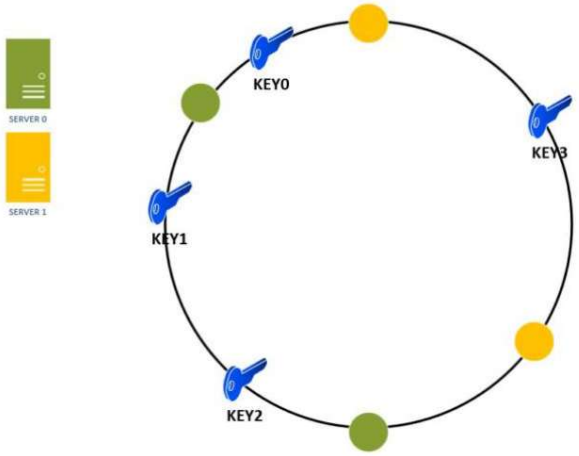
Virtual nodes are hashed independently and land at random point on ring

Vnodes map back to same physical machine

Without vnodes



With vnodes



Summary

> Consistent hashing solves horizontal scalability
  - Only a small fraction of keys are rearranged

> To mitigate hot spots, vnodes can be used to balance data
  - Each vnode maps back to same physical machine
  - As number of vnodes increases, data becomes more balanced

> In practice, # vnodes chosen to balance uniformity vs lookup overhead

Example Implementation

README.rst

uhashring

pypi v2.3 CI passing

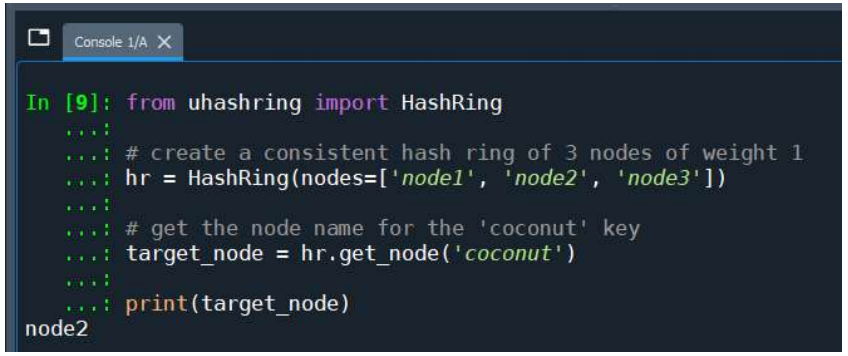
uhashring implements **consistent hashing** in pure Python.

Consistent hashing is mostly used on distributed systems/caches/databases as this avoid the total reshuffling of your key-node mappings when adding or removing a node in your ring (called continuum on libketama). More information and details about this can be found in the *literature* section.

This full featured implementation offers:

- a lot of **convenient methods** to use your consistent hash ring in real world applications.
- simple **integration** with other libs such as memcache through monkey patching.
- a full ketama compatibility if you need to use it (see important mention below).
- all the missing functions in the libketama C python binding (which is not even available on pypi) for ketama users.
- possibility to **use your own weight and hash functions** if you don't care about the ketama compatibility.
- **instance-oriented usage** so you can use your consistent hash ring object directly in your code (see advanced usage).
- native **pypy support**, since this is a pure python library.
- tests of implementation, key distribution and ketama compatibility.

Basic usage



Source: https://github.com/ultrabug/uhashring

Lab: Dive into the Repo

Basic usage is defined here:
https://github.com/ultrabug/uhashring/blob/master/uhashring/ring.py

1. Review and experiment with this file to understand how it works
2. Where is the hash created?
3. Build rings with different numbers of nodes and notice how *get_node()* output changes

Some useful methods:

```
def _get_pos(self, key):
    """Get the index of the given key in the sorted key list.

    We return the position with the nearest hash based on
    the provided key unless we reach the end of the continuum/ring
    in which case we return the 0 (beginning) index position.

    :param key: the key to hash and look for.
    """
```

```
def _get(self, key, what):
    """Generic getter magic method.

    The node with the nearest but not less hash value is returned.

    :param key: the key to look for.
    :param what: the information to look for in, allowed values:
        - instance (default): associated node instance
        - nodename: node name
        - pos: index of the given key in the ring
        - tuple: ketama compatible (pos, name) tuple
        - weight: node weight
    """
```