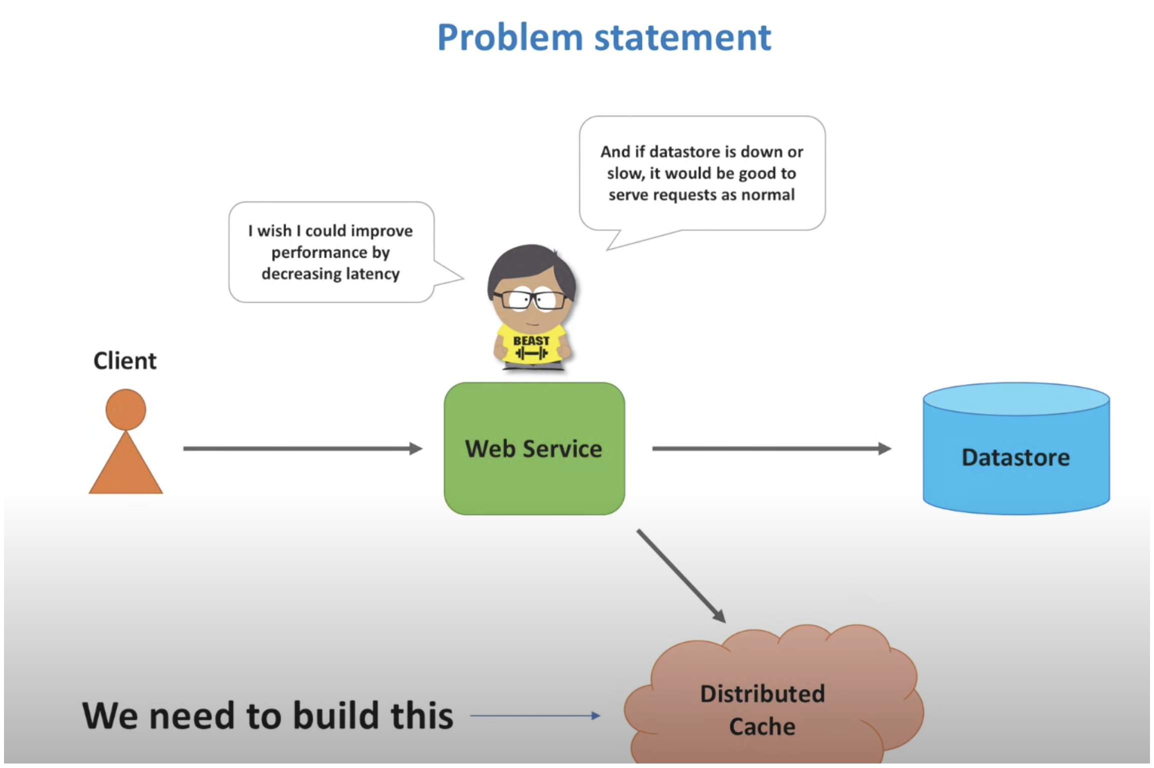
Distributed Cache

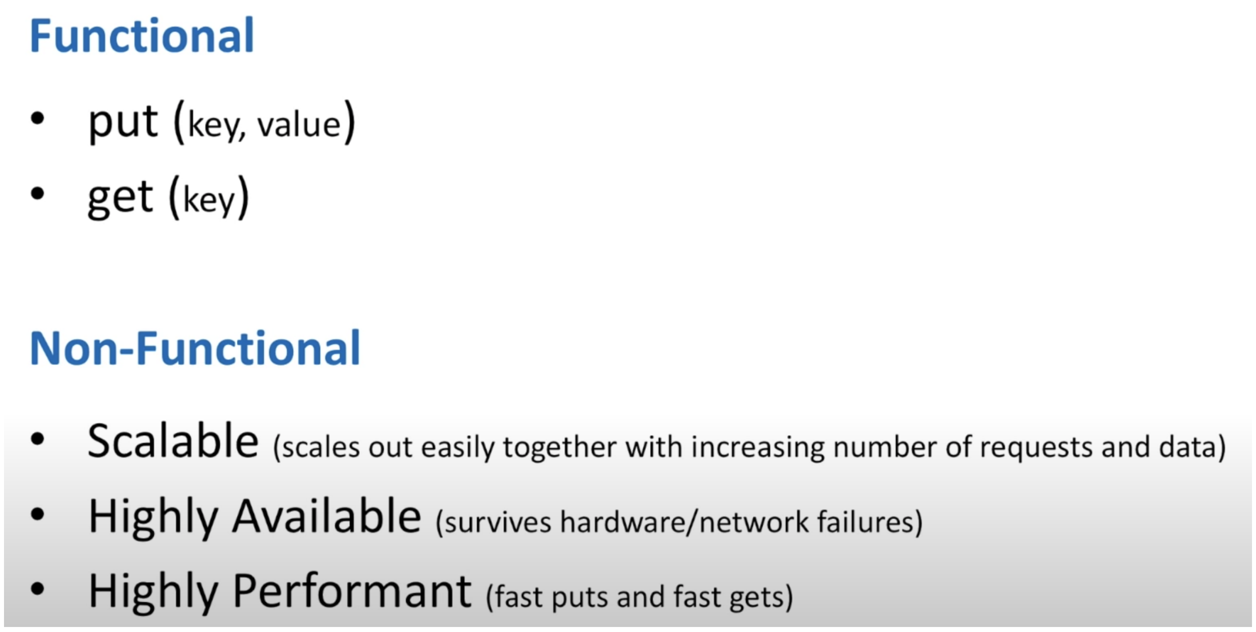


Problem statement

I wish I could improve performance by decreasing latency
And if datastore is down or slow, it would be good to serve requests as normal
BEAST
Client
Web Service
Datastore
Distributed Cache

We need to build this

Requirement:



Functional

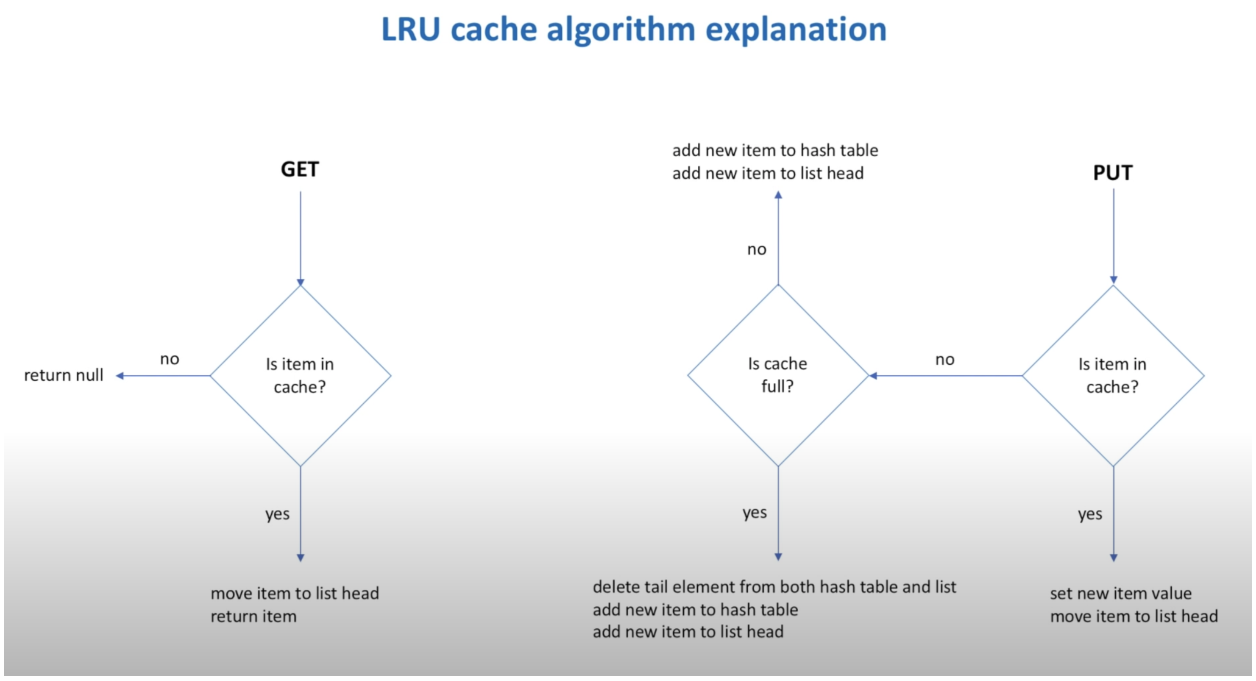
- put (key, value)
- get (key)

Non-Functional

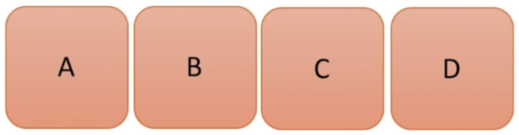
- Scalable (scales out easily together with increasing number of requests and data)
- Highly Available (survives hardware/network failures)
- Highly Performant (fast puts and fast gets)

Note : Three important points to consider are Scalability, Availability and Performance
If you consider CAP theorem then Availability can be replaced with Consistency

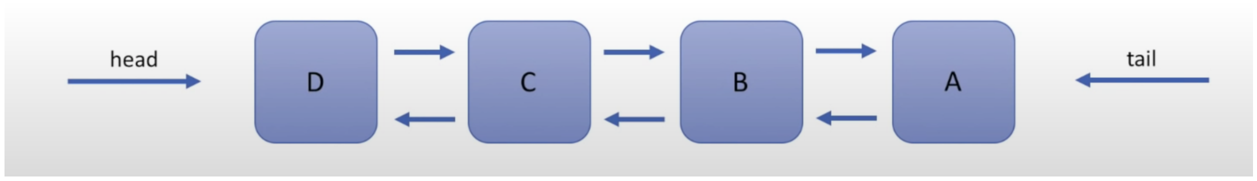
- When Cache is full then it needs to be clearer for further use. In this case a replacement policy is used. Most probably LRU policy.
- Hashtable doesn't track which entry has been used recently. So we need someother data structure

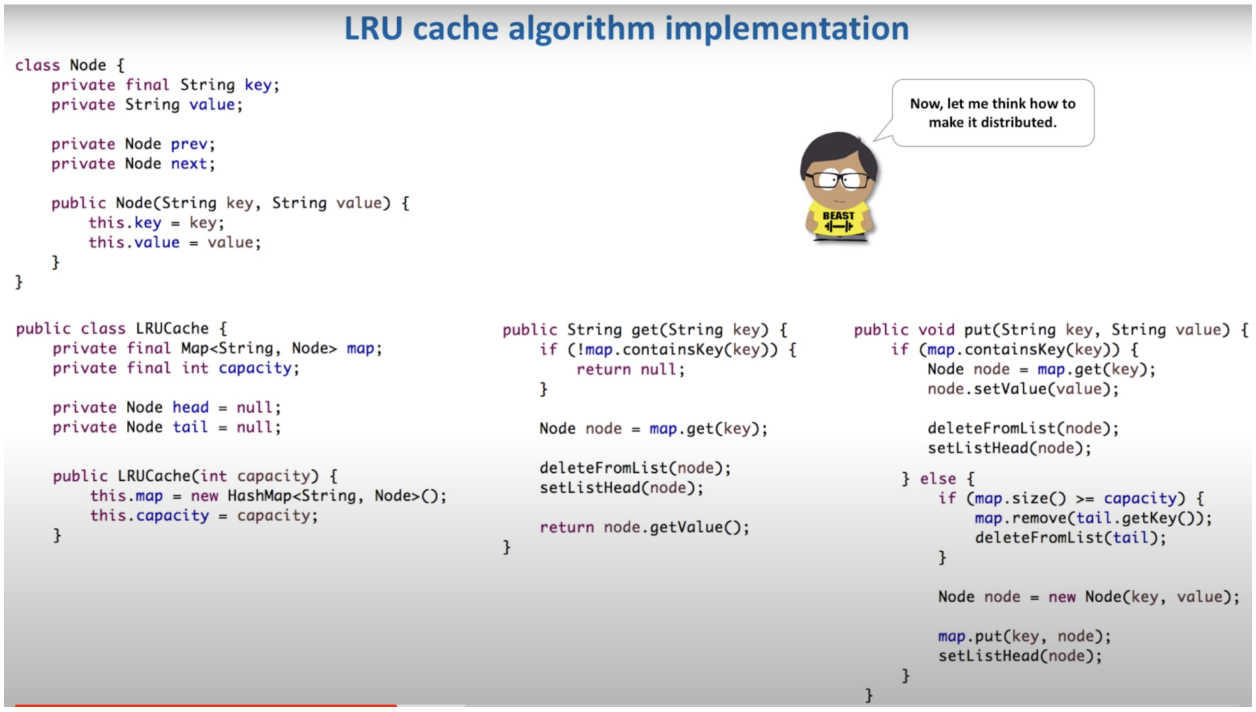

LRU cache algorithm explanation
GET
Is item in cache?
no
return null
yes
move item to list head
return item
PUT
Is item in cache?
yes
set new item value
move item to list head
no
Is cache full?
no
add new item to hash table
add new item to list head
yes
delete tail element from both hash table and list
add new item to hash table
add new item to list head

Hash table



Doubly linked list





LRU cache algorithm implementation

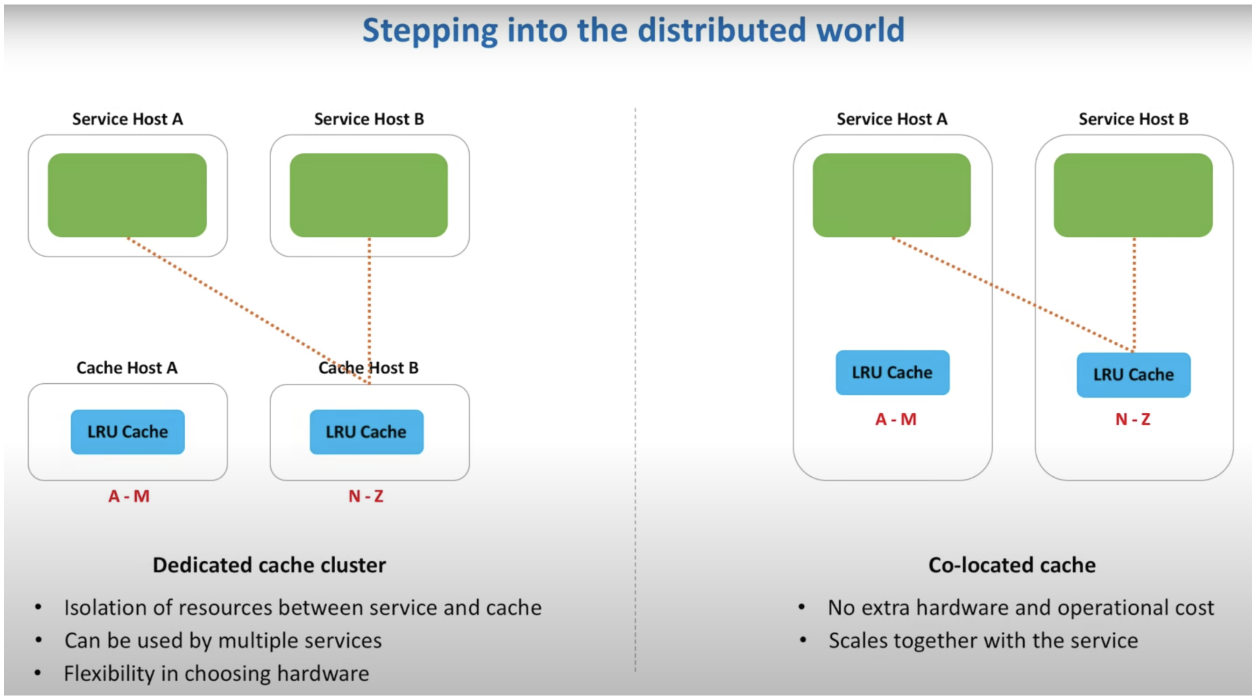
```
class Node {
    private final String key;
    private String value;

    private Node prev;
    private Node next;

    public Node(String key, String value) {
        this.key = key;
        this.value = value;
    }
}
```

```
public class LRUCache {
    private final Map<String, Node> map;
    private final int capacity;

    private Node head = null;
    private Node tail = null;


    public LRUCache(int capacity) {
        this.map = new HashMap<String, Node>();
        this.capacity = capacity;
    }
```

```
public String get(String key) {
    if (!map.containsKey(key)) {
        return null;
    }

    Node node = map.get(key);

    deleteFromList(node);
    setListHead(node);

    return node.getValue();
}
```

```
public void put(String key, String value) {
    if (map.containsKey(key)) {
        Node node = map.get(key);
        node.setValue(value);

        deleteFromList(node);
        setListHead(node);

    } else {
        if (map.size() >= capacity) {
            map.remove(tail.getKey());
            deleteFromList(tail);
        }

        Node node = new Node(key, value);

        map.put(key, node);
        setListHead(node);
    }
}
```

How to make it distributed cache



Stepping into the distributed world

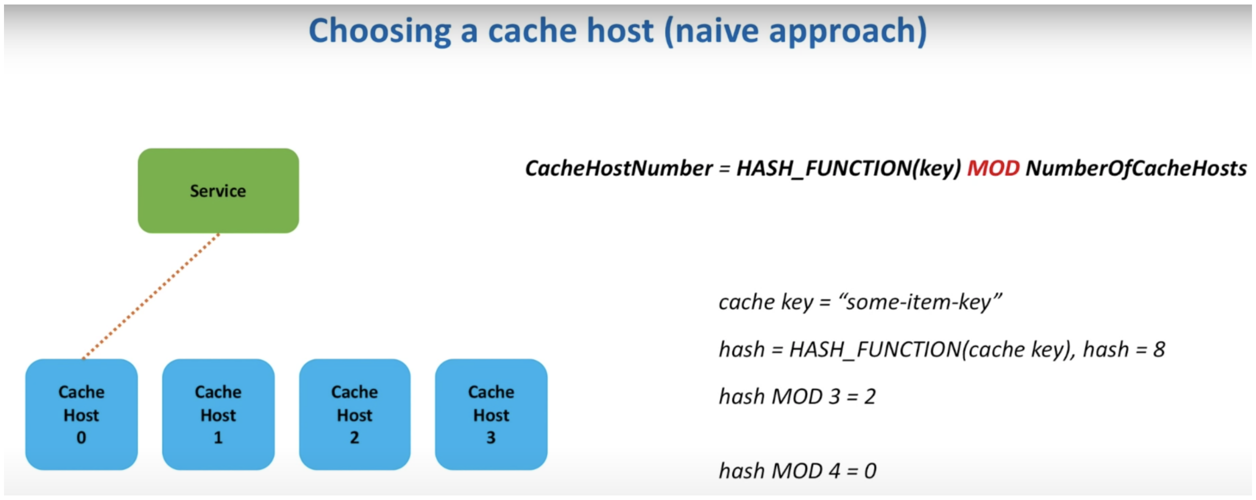
Dedicated cache cluster

- Isolation of resources between service and cache
- Can be used by multiple services
- Flexibility in choosing hardware

Co-located cache

- No extra hardware and operational cost
- Scales together with the service

Choosing the cache host



Choosing a cache host (naive approach)

CacheHostNumber = HASH_FUNCTION(key) MOD NumberOfCacheHosts

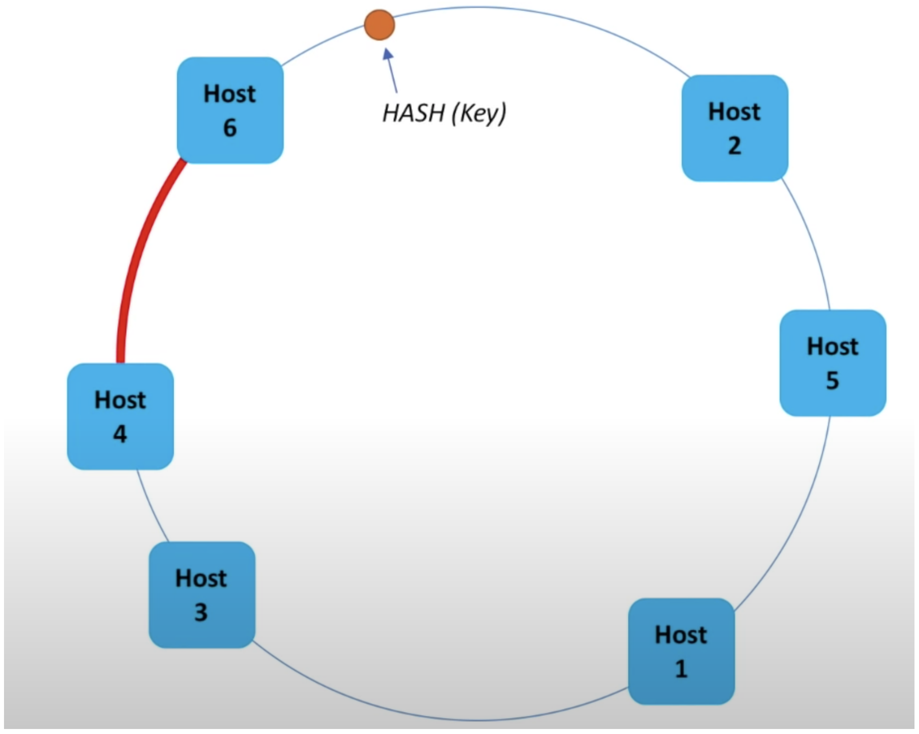
cache key = "some-item-key"

hash = HASH_FUNCTION(cache key), hash = 8

hash MOD 3 = 2

hash MOD 4 = 0

Consistent Hashing



- Adding a new host will minimize the number of keys we need to rehash
-