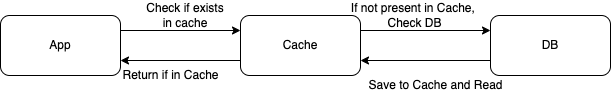
**Design Cache**

****

**Features:**

Amount of Data we will be Caching :Approx. Few TBs

Eviction Strategy:

Few picks : FIFO , LRU (Least Recently Used) , LFU ( Least Frequently Used )

Let’s pick LRU (default eviction strategy for Cache)

Access Pattern:

Few picks: Write through, Write around and Write back.

Write through Cache: This is a caching system where writes go through the cache and the write is considered success only if the writes to DB and Cache both succeed. Useful in applications which write and reread information quickly. However, write latency is higher as there are writes to 2 separate systems.

Write around cache: This is a caching system where write directly goes into DB. The cache system reads information from DB in case of a miss. While this ensures lower write load to cache and faster writes, this can lead to higher lead latency in case of applications which write and reread information quickly.

Write back cache: Write is done to the caching layer first and the write is confirmed as soon as the write to cache completes. The cache then asynchronously syncs this write to the DB. This ensures quick write but it’s an in-memory write / not persistent - there is a chance of loss of data in case of crash/ caching layer dies. We can solve this problem by introducing replicas.

Let’s pick the write around cache (more [details](https://www.computerweekly.com/feature/Write-through-write-around-write-back-Cache-explained)).

**Estimations:**

Total Cache: 30 TB

QPS: 10 M QPS

Total Number of Machines: 1 Machin has 72 G RAM, so 30 TB/ 72 G = 30000 G/ 72 G = 400+ Machines

**Design Goals:**

Latency, Consistency and Availability

We cannot have all 3, so we need to prioritize 2 of them.

Is Latency important ? Yes, the whole point of caching is low latency.

Consistency vs Availability: Unavailability means the caching machine goes down. Since we are thinking of a system like Google/Twitter, when a user views a Timeline , it’s okay to miss a few latest tweets, but unavailability could lead to higher latency and load on DB. So, prioritizing availability over consistency. It’s okay to have a system which is eventually consistent.

Deep Dive:

LRU Cache

Initially : A Simple HashMap

With eviction strategies: LRU Cache code:

We need a data structure which at any given instance - can give the least recently used object in order. Let’s say we maintain a linked list to do it, we try to keep the list ordered by the order in which they are used.

Whenever a get happens, we will need to move the object from a certain position to the front of the list, which means a delete followed by an insert at the beginning. Inserting at the beginning of the list is trivial. How do we achieve erase of an object from a random position at the least amount of time possible ? How about we maintain a map that stores the values corresponding to the linked list node.

Ok, now we know the node, we need to know the previous and next node to enable deletion of the node - easiest way is to use a Doubly Linked list.

public class Solution {

private Hashtable<Integer, Node> cache = new Hashtable<>();

private int count;

private int capacity;

private Node head, tail;

class Node {

int key;

int value;

Node prev;

Node next;

}

public Solution(int capacity) {

this.capacity = capacity;

this.count = 0;

head = new Node ();

tail = new Node ();

head.prev = null;

head.next = tail;

tail.prev = head;

tail.next = null;

}

public int get(int key) {

Node node = cache.get(key);

if ( node == null ){

return -1;

}

this.moveToHead(node);

return node.value;

}

private void moveToHead(Node node){

this.removeNode(node);

this.addNode(node);

}

private void removeNode (Node node){

Node prev = node.prev;

Node next = node.next;

prev.next = next;

next.prev = prev;

}

private void addNode(Node node ){

node.prev = head;

node.next = head.next;

head.next.prev = node;

head.next = node;

}

public void set(int key, int value) {

Node node = cache.get(key);

if ( node == null ){

Node newNode = new Node();

newNode.key = key;

newNode.value = value;

this.cache.put(key, newNode);

this.addNode(newNode);

++count;

if (count > capacity){

Node toBeDeleted = this.popTail();

this.cache.remove(toBeDeleted.key);

--count;

}

} else {

node.value = value;

this.moveToHead(node);

}

}

private Node popTail(){

Node result = tail.prev;

this.removeNode(result);

return result;

}

}