1. Expectations :  
  
System/Product Knowledge :  
  
Knowledge of Various Technologies :  
  
Consideration to pick a technology over the other.  
  
Future Growth (Scalable) ?  
  
2. Introduction to Caching :  
  
2.1 High Speed Data Storage Layer. This is fast because you use RAM rather than normal storage. since RAM is faster than disk.  
2.2 It is a transient storage. Generally placed in front of DB's or applications, or in front of places where there are high I/O .  
2.3 Pre computed value are stored in cache and this is hit before the actual system. If there is a miss in the cache then only you go to the main system.  
2.4 Applications : DB, OS, CDN, Web Apps etc.  
2.5 Benefits like improved performance, more IOPS, Reduce DB loads.  
  
3. Specs and features  
  
Specs :-  
These params can help in optimizing the performance of cache.  
  
3.1 Size of cache : ask from interview , google , fb have cache in TBs  
3.2 Query per second : thousands of hits per sec; confirm from interview  
3.3 Cache Validity : validity of the items in cache; it should be in sync with the stored data; should not be inconsistent.  
3.4 Cache Hit Rate : How many times your are getting data from the cache. this means when you hit the cache, you should get output from it rather than going to the actual db/app.  
3.5 Cache Miss : This is when you hit the cache and you didn't find the data over there and then you go to the actual db/app for the data.  
3.6 TTL : Time to Live; how long will your data will reside in your cache.  
  
  
Features :-  
3.7 Cache Access Patterns : this is how you access your cache. Your Query per second would be highly dependent on it.  
3.8 Cache Eviction : decides your TTL, Cache Miss and Cache Hit Rate. Keep highly used frequent items in your cache and remove the low used items.  
3.9 Availability : Should be higly available.  
3.10 Scalability and Consistent Hashing :  
  
  
4. Distributed Cache :  
  
Pool of cache/ram which are capable of working together.They are horizontally scalable.

Diagram

Description automatically generated

5. How can we access Cache :

5.1 Write Through : Request comes you write into the cache and then into the database and then the response is sent back from cache that the data is persisted. When the next request comes either the cache will serve it or the request will go to the db and then persist that data into cache and return the response.

5.2 Write Back ( Lazy Write) : Used by oracle to buffer cache. A little more complez to implement.Faster than Write through. When write request comes Data is persisted into cache and the response is returned. There is a background process where writes the cache into the database. There could be a possibility of dirty write/dirty buffer.There could be challenge where you write into the cache and it goes down but on the contrary it is much faster than write through.

5.3 Write Around (Miss 1st Read): When write request comes, data is persisted into db and when the read request comes to cache we let the first request miss so that it goes to db and then writes it to cache and then returns the response.

Write Back is good for DBs and other 2 are good for webservices. Also cache validity is there in first 2 but for the third there whill be a miss the first time after that it will behave like the rest.

Diagram

Description automatically generated

6. Cache Eviction :

How you will make sure the hit rate is above 90-95 %

6.1 LRU : Least Recently Used.

6.2 LFU : Lease Frequently Used

6.3 Random Replacement

6.4 W Tiny LFU : wrapper in top of LFU in terms of time. If the item is included like 50 mins back then again 30 min back and so on.. so while taking eviction decision the time for which the item has remained in the cache will also be taken into account.

W-TinyLFU uses the Segmented LRU (SLRU) policy for long term retention. An entry starts in the probationary segment and on a subsequent access it is promoted to the protected segment (capped at 80% capacity). When the protected segment is full it evicts into the probationary segment, which may trigger a probationary entry to be discarded. This ensures that entries with a small reuse interval (the hottest) are retained and those that are less often reused (the coldest) become eligible for eviction.

Chart, bar chart

Description automatically generated

7. **Expiration Policy**

Expiration is often implemented as variable per entry and expired entries are evicted lazily due to a capacity constraint. This pollutes the cache with dead items, so sometimes a scavenger thread is used to periodically sweep the cache and reclaim free space. This strategy tends to work better than ordering entries by their expiration time on a O(lg n) priority queue due to hiding the cost from the user instead of incurring a penalty on every read or write operation.

Caffeine takes a different approach by**observing that most often a fixed duration is preferred**. This constraint allows for organizing entries on O(1) time ordered queues. A time to live duration is a write order queue and a time to idle duration is an access order queue. The cache can reuse the eviction policy’s queues and the concurrency mechanism described below, so that expired entries are discarded during the cache’s maintenance phase.

8. Concurrency

Concurrent access to a cache is viewed as a difficult problem because in **most policies every access is a write to some shared state**. The traditional solution is to guard the cache with a single lock. This might then be improved through lock striping by splitting the cache into many smaller independent regions. Unfortunately that tends to have a limited benefit due to hot entries causing some locks to be more contented than others. When contention becomes a bottleneck the next classic step has been to **update only per entry metadata** and use either a [random sampling](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.8469&rep=rep1&type=pdf) or a [FIFO-based](https://en.wikipedia.org/wiki/Page_replacement_algorithm#Second-chance) eviction policy. Those techniques can have great read performance, poor write performance, and difficulty in choosing a good victim.

An [alternative](http://web.cse.ohio-state.edu/hpcs/WWW/HTML/publications/papers/TR-09-1.pdf) is to **borrow an idea from database theory where writes are scaled by using a commit log**. Instead of mutating the data structures immediately, the **updates are written to a log and replayed in asynchronous batches**. This same idea can be applied to a cache by performing the hash table operation, recording the operation to a buffer, and scheduling the replay activity against the policy when deemed necessary. The policy is still guarded by a lock, or a try lock to be more precise, but shifts contention onto appending to the log buffers instead.

9. Data Structure for Cache

Hash Table used for faster access and linked list to implement LRU.Basically a set is used to maintain the entries for the linked list. When an element comes it could be modulus by 10 and then a location in the set is identified and then in case there are more elements with same address a doubly linked list is maintained which will also tell us that which is the Least Recently Used element which would be the first one, in case the LL will be full the first element will be removed and the latest element will be added in the last.

Diagram

Description automatically generated

When we access any element from the set, it will be removed from DLL and will be put in the end to show that it has been recently used. Since we are using a DLL removing and adding will be done in O(1).

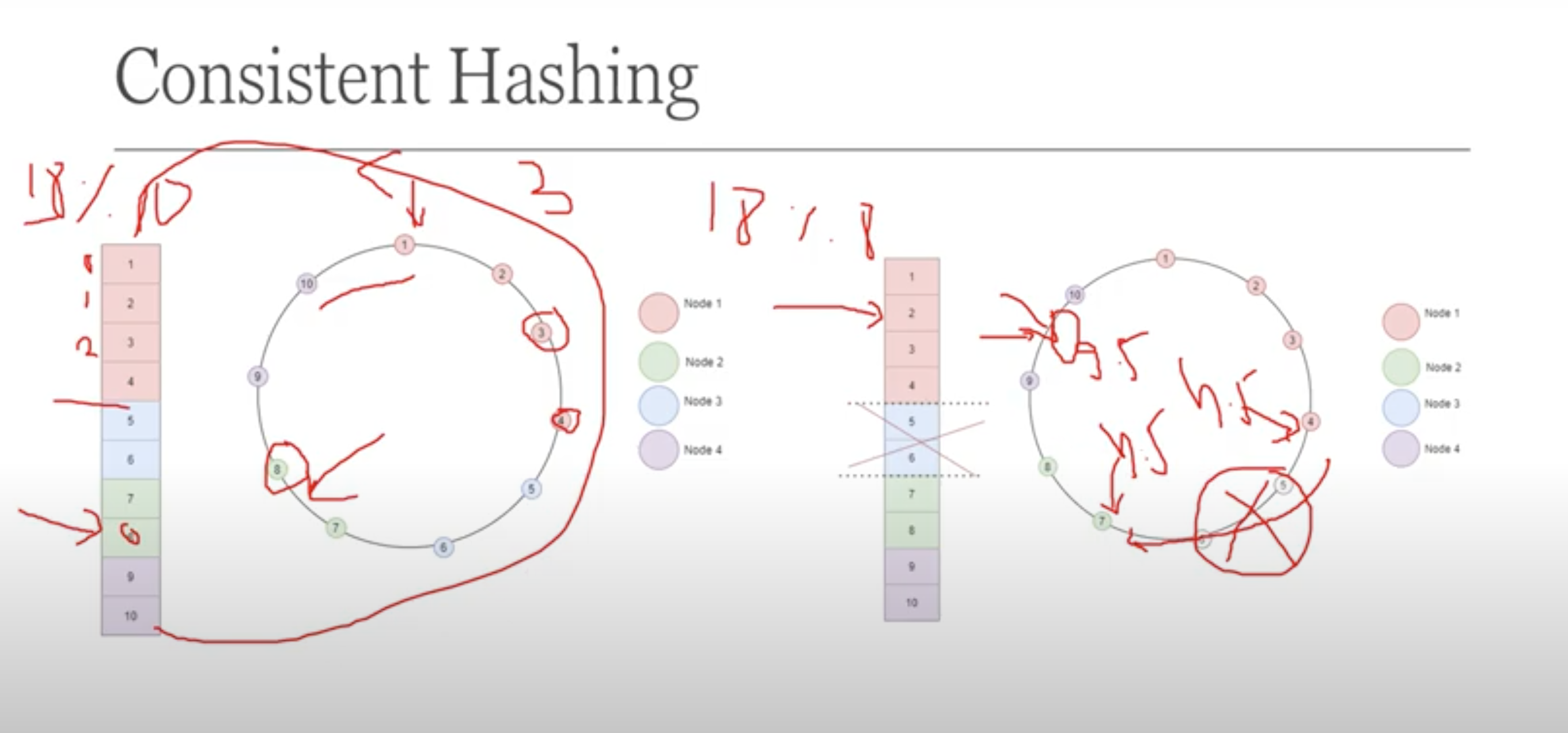
A picture containing table

Description automatically generated

10. Consistent Hashing

Rather than placing things in a hasset you place it circularly ie the last element is pointing to the first element so that if any node goes down in middle then there is a more probability in finding the correct node.

It is also helpful in scaling in a distributed environment.



Code :

|  |
| --- |
| import java.util.Deque; |
|  | import java.util.HashSet; |
|  | import java.util.Iterator; |
|  | import java.util.LinkedList; |
|  |  |
|  | public class LRUImplementation { |
|  | HashSet<Integer> hSet = new HashSet<>(); |
|  | Deque<Integer> q = new LinkedList<>(); |
|  | private final int cacheSize=100; |
|  | private void updateCache(int val){ |
|  | if(!hSet.contains(val)){ |
|  | if(q.size()==cacheSize){ |
|  | int last = q.removeLast(); |
|  | hSet.remove(last); |
|  | }  }else{ |
|  | q.removeLast(); |
|  | } |
|  | q.push(val); |
|  | hSet.add(val); |
|  | } |
|  |  |
|  |  |
|  | private void displayQueue(){ |
|  | Iterator<Integer> i = q.iterator(); |
|  | while(i.hasNext()){ |
|  | System.out.println("Value is: "+i.next()); |
|  | } |
|  | } |
|  | } |

<https://www.geeksforgeeks.org/lru-cache-implementation/>