#Caching:

- High speed data storage system that stores the transient data.
- Data stored in RAM.
- Used in compute, storage, CDN to optimize performance.

Caching best practices:

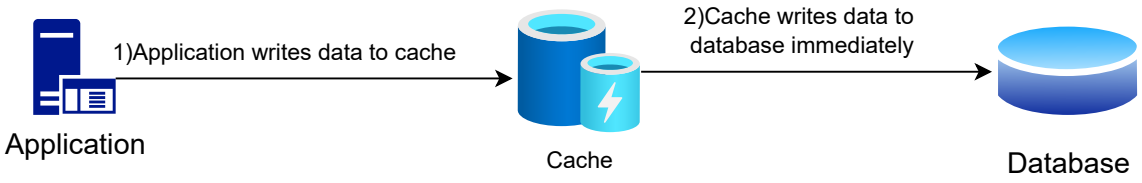
- Validity: How long to store the data in cache.
- High hit rate: Data found in cache.
- Cache Miss: Should be low.
- Time to live (TTL): Time to invalidate data.

Features/Estimation:

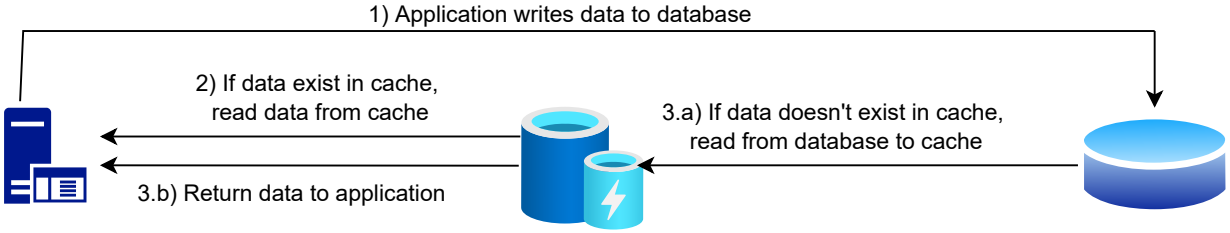
- store data size (Terra Byte)
- 50k to 1M QPS (Query Per Second)
- ~ 1ms latency
- LRU (Eviction)
- 100% availability
- Scalable

Cache Access Pattern:

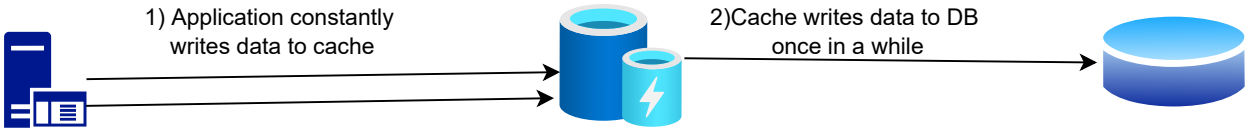
1) Write Through Cache:



2) Write Around Cache



3) Write back cache



Types of Cache:

1. Application Server Cache

In a web application, let's say a web server has a single node. A cache can be added in in-memory alongside the application server. The user's request will be stored in this cache and whenever the same request comes again, it will be returned from the cache. For a new request, data will be fetched from the disk and then it will be returned. Once the new request will be returned from the disk, it will be stored in the same cache for the next time request from the user. Placing cache on the request layer node enables local storage.

Note: When you place your cache in memory the amount of memory in the server is going to be used up by the cache. If the number of results you are working with is really small then you can keep the cache in memory.

The problem arises when you need to scale your system. You add multiple servers in your web application (because one node can not handle a large volume of requests) and you have a load balancer that sends requests to any node. In this scenario, you'll end up with a lot of cache misses because each node will be unaware of the already cached request.

2. Distributed Cache

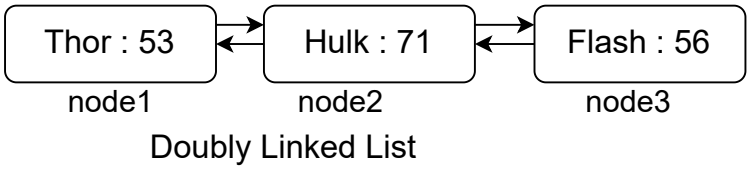
In the distributed cache, each node will have a part of the whole cache space, and then using the consistent hashing function each request can be routed to where the cache request could be found. Let's suppose we have 10 nodes in a distributed system, and we are using a load balancer to route the request then each of its nodes will have a small part of the cached data.To identify which node has which request the cache is divided up using a consistent hashing function each request can be routed to where the cached request could be found. If a requesting node is looking for a certain piece of data, it can quickly know where to look within the distributed cache to check if the data is available.We can easily increase the cache memory by simply adding the new node to the request pool.

#Cache Eviction Policies:

1) FIFO 2) LIFO 3) LRU 4) MRU 5)LFU 6)RR

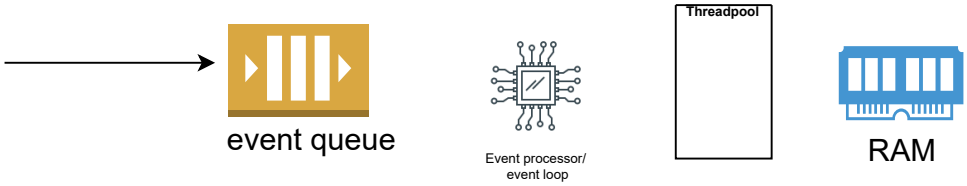
#LRU cache design

HashTable	
Index	**LL node pointer**
100	node1 addr
101	node2 addr
102	node3 addr



- HashTable (to store LL node) + doubly Linked List (to store/update data{key:value})
- start and end node in LL is known
- In case of delete remove entry from hashmap and also remove 1st node from LL
- For a update move the node to end of LL
- For insert add new node at the end of LL

Handling get/put requests:



Fault-tolerant:

- Regular interval snapshot:
- Log reconstruction: Recreate hashtable using log

Availability:

- Sharding
- Replication (master-slave)