Load Balancing

- It helps to distribute load across multiple resources.
- It also keep track of status of all the resources while distributing requests. If a server is not available, it stops sending traffic.

LB can be added at three places:
  - Between user & web-server
  - Between web-servers & internal servers (Application servers or cache servers)
  - Between Internal platform layer & DB
- Types of load balancers:
  - Smart Clients
    - It'll take a pool of service hosts & balances load, manage them (Detect recovered host, deal with adding new hosts, etc)

Hardware LB:
  - They are hardware which works as LB, but are very expensive.
  - Even big companies use them only as first point of contact & use other machanism for load-balancing.

Software Load balancers:
  - It's hybrid approach. HAProxy is popular open source software LB.
  - Every client request on this port (where HAProxy is running) will be received by proxy & then passed to the backend service in efficient way.
  - HAProxy manages health check & will remove or add machines to those pools.

We should start with these.

Ways of load balancing:

  - Round robin
  - Round robin with weighted server
  - Least Connections
  - Least Response Time
  - Source IP Hash
  - URL Hash

Caching

- Caching works on locality of reference principle: recently requested data is likely to be requested again.
- It's like short-term memory which has limited space but is faster & contains most recently accessed items.
- Cache can be used in almost every layer: hardware, OS, Web browsers, web application, but are often found nearest to the front end.
- Types of cache:
  - **Application server cache**:
    - Placing a cache on request layer node enables the local storage of response data. When a request is made, node will quickly return the cached data, if exists. If not it'll query from disk.
    - But when you've many nodes with load balancer, it randomely distribute across the nodes, the same request will go to different nodes, thus increasing cache misses
    - Two choices are to overcome this: **global cache & distribute cache.**
  - Distribute Cache:
    - In it, each of its nodes own part of cached data.
    - **The cache is divided up using a consistent hashing function,** such that if a request node is looking for a certain piece of data, it can quickly know where to look within distributed cache to check if data is available.
    - Easily we can increase the cache space just by adding nodes to the request pool
  - Global Cache:
    - In this, all nodes use the same single cache space. Each of the request nodes queries the cache in the same way it would a local one.
    - There can two type of global cache:
      - First, when a cached response not found in cache, cache itself becomes responsible for retrieving the missing peice of data.
      - Second, it's the responsibility of request nodes to retrieve any data that is not found. It can be used when low cache hit % would cause the cache buffer with cache misses. In this situation, it helps to have a large % of data set in cache.
  - CDN
    - **It's Content Distribution Network for serving large amount of static media which is common to all.**
    - First request ask the CDN for data, if not cdn will query the back-end servers & then cache it locally
    - If your system is not that big for CDN, you can serve static media from a seperate subdomain using a lightweight HTTP server like Nginx
- **Cache Invalidation**
  - **When data is modified in DB, it should be invalidated in the cache.**

- **write-through cache:**
  - Data is written into the cache & the DB at the same time.
  - This allow cached to be fast
  - Data consitency also there between cache & DB, which make sure nothing get lost in case of power failure
  - This minimizes the risk of data loss, but has **disadvantage of higher latency for write operations.**
- **write-around cache:**
  - Data is written directly to storage, bypassing the cache.
  - This reduces flooded write operations but has disadvantage that a read request for recently written data will create a cache miss & must be read from slower back-end.
- **write-back cache:**
  - Data is written to cache alone & completion is immedialtely confirm to client & write to permanent storage is done after specified intervals.
  - This results in low latency & high throughput, however this spped comes with the risk of data loss in case of crash.

- Cache eviction policy
  - FIFO
  - LIFO
  - LRU (Least Recently Used): Implement using Doubly Linked list & a hash function containing the reference of node in list
  - Most Recently Used
  - Least frequently Used
  - Random Replacement

Sharding or Data Partitioning

- Sharding is a technique to break up a big database into many smaller parts
- Horizontal scaling means adding more machines, which is cheaper & more feasible
- Vertical scaling means improving servers
- Partitioning Methods:
  - Horizontal partitioning:
    - In this we put different rows into different table(db), i.e. rows can be based on location with zip codes, this is also range based sharding
    - Problem is if range is not choosen carefully, it'll lead to unbalanced servers.
  - Vertical Partitioning:
    - In this we divide data to store tables related to a specfic feature to their own servers.i.e. In Instagram, we can have 1 for user profile, 1 for photos, 1 for friend list.
    - Problem is in case of additional growth, its necessary to further partition a feature specific DB across servers.

- Directory based Partitioning:
  - In this, create a lookup service which knows your current partitioning scheme & get it from DB access code.
  - To find out a data entry, we query directory server that holds the mapping between each tuple key to its DB server.
  - This is good to perform tasks like adding servers to the DB pool or change our partitioning scheme without impacting application.
- Partitioning Criteria:
  - Hash-based partitioning:
    - In this we apply a hash function to some key attribute of the entity we're storing, that gives partition number.
    - Make sure to ensure uniform allocation of data among servers
    - Problem is it effectively fixes total number of DB servers, since adding a new server means changing the hash function, which would require redistribution of data & downtime, the workaround is Consistent Hashing.
  - List Partitioning:
    - In this each partition (DB server) is assigned a list of values, i.e. APAC, EMEA, US region has respective partition.
  - Round-robin Partitioning:
    - One by one assign which ensure uniform data distribution
  - Composite partitioning:
    - Combining any of above partitioning schemas to devise a new scheme. i.e First list partitioning with hash partitioning in each.
- Common problems with Sharding:
  - Joins & Denormalisation:
    - Performing joins on a database which is running on one server is straightforward, but if DB is partitioned & spread across multiple machines, it's not feasible to perform joins that span database shards.
    - These joins won't be performance efficient.
    - Workaround is to denormalise the DB so that queries that required joins can be performed on single table.
  - Referential integrity:
    - Trying to enforce data integrity constraint suh as foreign keys in a sharded DB can be extremely difficult. Most of RDMS do not support this.
  - Rebalancing:
    - When data distribution is not uniform.
    - When there is lot of load on a particular Shard.

Indexes

- Very useful in database to improve the speed of data retrieval.
- Index makes trade-off of increased storage overhead, slower writes for the benefit of faster read.

- Index is a data structure of table of contents that points us to the location where actual data lives, so when we create an index on a column of a table, we store that column & a pointer to the whole row in the index.
- Indexing is mainly done in two ways:
  - Ordered Indexing: Column is sorted as per ascending order
  - Hash Indexing: Indexing is as per Hash function & Hash table

Proxy

- Proxy server is intermediary hardware/software that sits between the client & server
- They are used to filter requests, log requests, transform request by adding/ removing headers, encrypting/decrypting or compression
- It's cache can serve a lot of requests.
- It can coordinate requests from multiple servers & can be used to optimize request traffic.
- It can merge same request from multiple requests into one.
- It collapse requests for data that is spatially close together in the storage, which'll descrease request latency.
- Proxies are useful under high load situations or when we have limited caching.

Message Queue

Asynchronous service to service communication used in server-less and micro-service architecture.

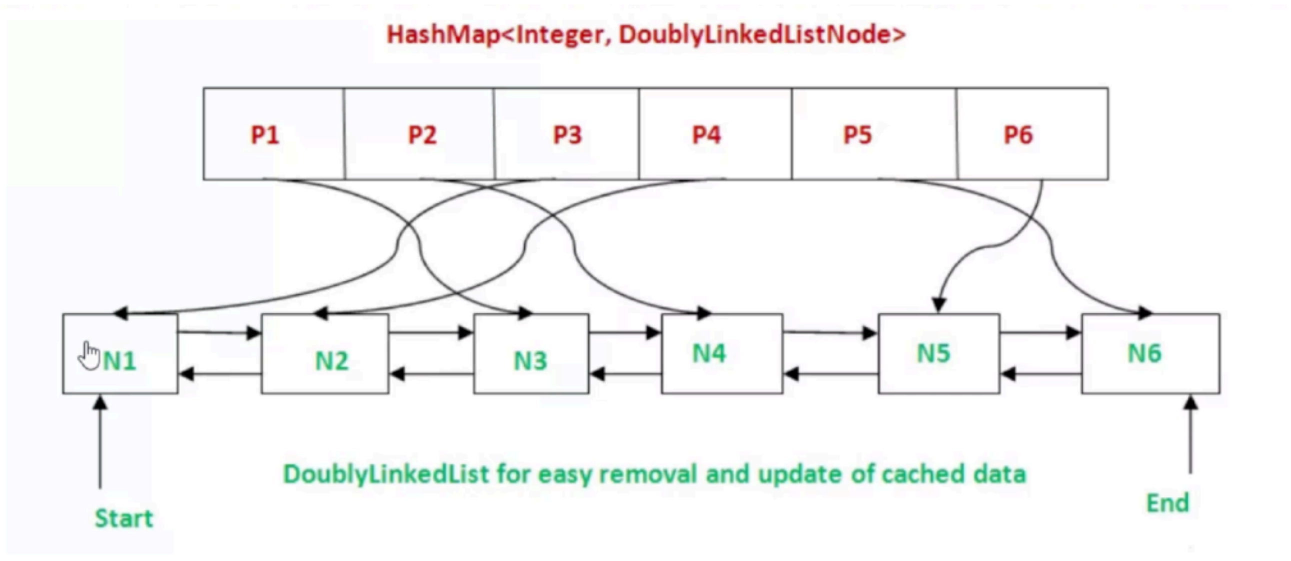
LRU Cache

Typically LRU cache is implemented using a doubly linked list and a hash map.
Doubly Linked List is used to store list of pages with most recently used page at the start of the list. So, as more pages are added to the list, least recently used pages are moved to the end of the list with page at tail being the least recently used page in the list.
Hash Map (key: page number, value: page) is used for O(1) access to pages in cache

When a page is accessed, there can be 2 cases:
1. Page is present in the cache - If the page is already present in the cache, we move the page to the start of the list.
2. Page is not present in the cache - If the page is not present in the cache, we add the page to the list.

How to add a page to the list:
  a. If the cache is not full, add the new page to the start of the list.

b. If the cache is full, remove the last node of the linked list and move the new page to the start of the list.


HashMap<Integer, DoublyLinkedListNode>
P1
P2
P3
P4
P5
P6
N1
N2
N3
N4
N5
N6
Start
DoublyLinkedList for easy removal and update of cached data
End

Points to consider for System Design:

- Requirements
- API Creation
- Database Design (Table creation)
- Logic to solve Problem
- System Workflow
- Load Balancer
- Caching
- Sharding
- Indexes
- Messaging Queue (Kafka)
- Hashing (Consistent Hashing)
- LRU
- Hadoop (HDFS)
- Cassandra
- Microservices

URL Shortener

Requirements

- User should be able to create Shorten Url from Original URL
- When clicked on short url, it should redirect to Original URL
- User can give expiration time for a url
- User can give custom shorten url

Extended:
- User should not exhaust the api (should limit the url creation)
- Exposing the service through REST APIs
- Analytics: Recording different parameters
- Able to create private/ public shorten url

API Creation

http://shorturl.com/api/v1/shorturl

POST
Create_url(original_url, user_id, expiration_time, custom_url)

GET
Get_url(short_url)

Database Design - Tables

User Table

User_id
name
Email
Contact
Num_url

URL

Short_url
Original_url
Expiration_time
Custom_url
User_id

Logic to create Short URL

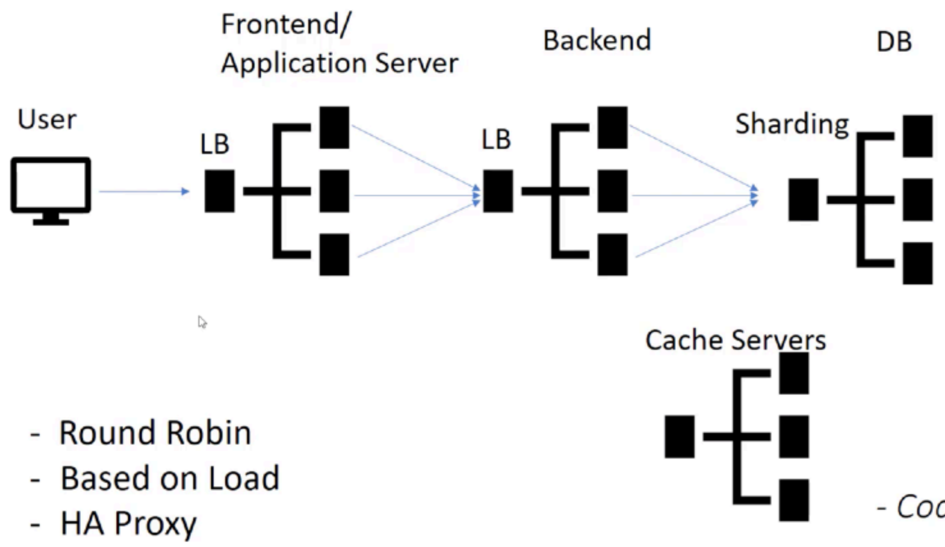
Encoding URL:

- We can compute unique hash via MD5 or SHA256 from original url
- Encode the hash using base64 (a-z, A-Z, 0-9, -, .) or base32(a-z, 0-9)
  If we want 6 character encoded value, it would generate following number of urls
  Base32 = 32 ^ 6
  Base64 = 64 ^ 6 = ~ 68 Billion Strings

- MD5 generates 128-bit hash, & if we use base64, it'll generate in more than 21 character

  base64 – Each character encode 6 bit of hash, so to accommodate 128-bit,
  atleast 21 characters
- Pick starting 6 characters out of 21 characters

- Load balancers

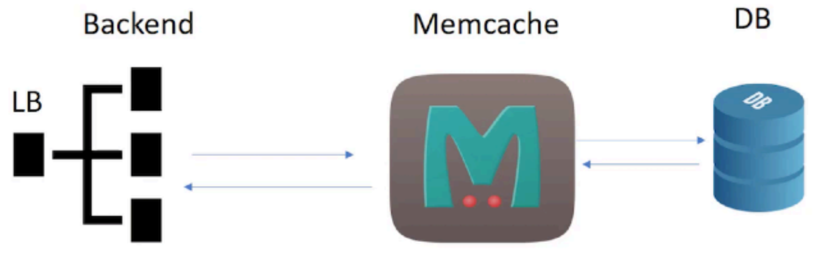


- Round Robin
- Based on Load
- HA Proxy

- Coding Simplified

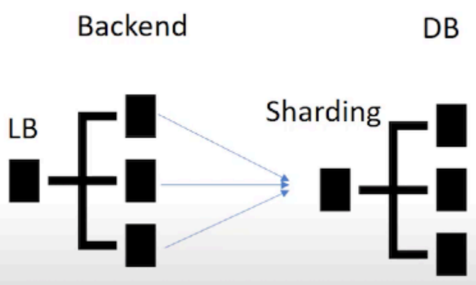
Caching

- Memcache
- LRU



Data Partitioning/ Sharding

- Range Based Sharding
- Hash Based (Consistent Hashing)


Backend
DB
LB
Sharding