**DBMS**

Redundancy- storing same data at different location

N tier db widely used cz it can be broken into smaller modules.

Normalisation:

Insertion anomaly- where we are not able to insert new data because because some values are not present but needed by table.

Update anomaly- update same info at diff places

Functional dependency- relation that exits between multiple attribute of relation. Trivial and non trivial dependency.

Axioms-

1nf-

**System design**

**Feature, estimation, design goal, api, tolerance**

Before starting a project we need to *collect requirements*.

*Architecture design*- understand components, ui, server (how many students enrolled, leaderboard, problem submission), database, another server to run code…

We use server because I don’t want to use browser (all calculation on server then browser renders) but data store and get from server. Server is both hardware and software. We run the software switch on cpu, software catch the request and return output.

Ui building- client side

Logic building- server side

How many users are going to use? -scalability

If we search google.com on browser, it finds the corresponding ip address. We convert domain name to ip address by going on domain name server DNS.

Individual component/ final analysis

1. **URL shortner**- a-fixed-name/……(I want the dots to be as short as possible). Lets keep minimum 4, if 4 finished then start 5. After how many days it will expire or if it will expire if it doesn’t get clicked for a day. If login id, I can check analytics of how many people clicked. I can create diff url for diff sites and see where I get max clicks. User can decide if random url or readable.

Original url to short or if someone puts short url then redirect to original url. DB will store short and long url corresponding. Where short url=” “. We will put indexing on short url so that search is fast.

UI- 2 types of request on server (long and short) – either get random short url or is short url available depending on user preference.

How to access random short url? Randomly generate.

Cron job- check the url not used in month and delete. Job that is repeated without interference.

**Q:**How many queries per second should the system handle?(Assuming 100 Million new URLs added each month)

**Hint:** Assuming average lifetime of a shortened URL is 2 weeks and 20% of websites creates 80% of the traffic, we see that we'll receive around 1 Billion queries in a month.

400 queries per second in total. 360 read queries and 40 write queries per second.

**Q:**How many URLs will we need to handle in the next 5 years?

**Hint:** Earlier we saw, we would see 100 Million new URLs each month. Assuming same growth rate for next 5 years, total URLs we will need to shorten will be 100 Million \* 12 \* 5 = 6 Billion.

**A:**6 Billion.

**Q:**What is the minimum length of shortened url to represent 6 Billion URLs?

**Hint:** We will use (a-z, A-Z, 0-9) to encode our URLs. If we call x as minimum number of characters to represent 6 Billion total URLs, then will be the smallest integer such that x^62 > 6\*10^9.

**A:**Log (6\*109) to the base 62 = 6

**A:**3 TeraBytes for URLs and 36 GigaBytes for shortened URLs  
Note that for the shortened URL, we will only store the slug(6 chars) and compute the actual URL on the fly.

Data read/written each second?

**Hint:** Data flow for each request would contain a shortened URL and the original URL. Shortened URL as we saw earlier, will take 6 bytes whereas the original URL can be assumed to take atmost 500 bytes.

**A:**Written : 40 \* (500 + 6) bytes, Read : 360 \* (500 + 6) bytes

**Design Goals:**

* **Latency** - Is this problem very latency sensitive (Or in other words, Are requests with high latency and a failing request, equally bad?). For example, search typeahead suggestions are useless if they take more than a second.
* **Consistency** - Does this problem require tight consistency? Or is it okay if things are eventually consistent?
* **Availability** - Does this problem require 100% availability?

**Q:**Is Latency a very important metric for us?

**A:**Yes. Our system is similar to DNS resolution, higher latency on URL shortener is as good as a failure to resolve.

**Q:**Should we choose Consistency or Availability for our service?

**A:**This is a tricky one. Both are extremenly important. However, CAP theorem dictates that we choose one. Do we want a system that always answers correctly but is not available sometimes? Or else, do we want a system which is always available but can sometime say that a URL does not exists even if it does?  
This tradeoff is a product decision around what we are trying to optimize. Let's say, we go with consistency here.

**Q:**Try to list down other design goals?

**A:**URL shortener by definition needs to be as short as possible. Shorter the shortened URL, better it compares to competition.

How should we define our APIs?

**A:**

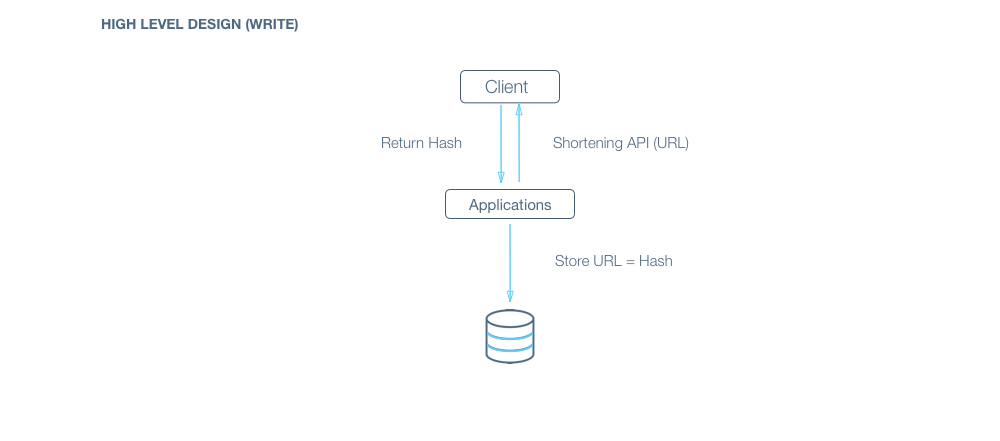
* ShorteningAPI(url) {store the url\_mapping and return hash(url)}
* RedirectionAPI(hash) {redirect\_to url\_mapping[hash]}

Both APIs are very lightweight, their computation will not the bottleneck plus we don't have to store any session information about users.  
Basically, we are trying to build a service which serves as a huge HashMap

How would a typical write query look like?

**A:**Components:

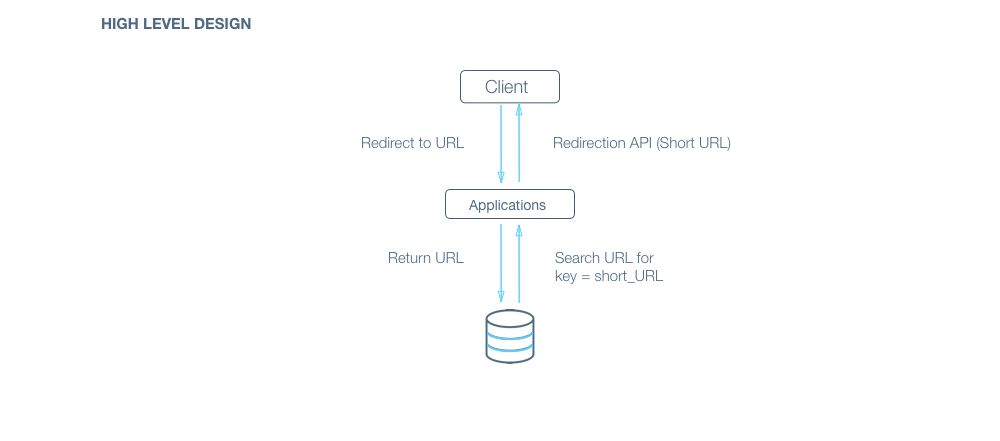
* Client ( Mobile app / Browser, etc ) which calls ShorteningAPI(url)
* Application server which interprets the API call and generates the shortened hash for the url
* Database server which stores the hash => url mapping



**Q:**How would a typical read query look like?

**A:**Components:

* Client ( Mobile app / Browser, etc ) which calls RedirectionAPI(short\_url)
* Application server which interprets the API call, extracts the hash from short\_url, asks database server for url corresponding to hash and returns the url.
* Database server which stores the hash => url mapping



Bottom of Form

***Gotcha:*** How should we handle the case where two separate URL gets shortened to the same URL?

**A:**We can use a list of salts in case of collision.  
For each read request, we can compute all possible shortened URLs using our list of salts and query for them in parallel to save time.

**A: convert\_to\_base\_62(md5(original\_url + salt))[:6](first six characters)**  
Links:[MD5-Wiki](https://en.wikipedia.org/wiki/MD5) [Base 62 Conversion-Stackoverflow](http://stackoverflow.com/questions/1119722/base-62-conversion-in-python)  
***Gotchas:***

* Directly encoding URL to base\_62 will allow a user to check if a URL has been shortened already or not, reverse enginnering can lead the user to the exact hash function used, this should not be allowed. Therefore randomization has to be introduced. Also, if two users shortens the same URL, it should result in two separate shortened URL(for analytics)
* Database ID encoded to base\_62 also won't be suitable for a production environment because it will leak information about the database. For example, patterns can be learnt to compute the growth rate(new URLs added per day) and in the worst case, copy the whole database.

How does our client know which application servers to talk to. How does it know which application servers have gone down and which ones are still working?

**A:**We introduce load balancers. Load balancers are a set of machines (an order of magnitude lower in number) which track the set of application servers which are active ( not gone down ). Client can send request to any of the load balancers who then forward the request to one of the working application servers randomly.

**A:**If we have only one application server machine, our whole service would become unavailable. Machines will fail and so will network. So, we need to plan for those events. Multiple application server machines along with load balancer is the way to go.

**Q:**What all data should we store?

**A:**Data storage layer: Hash => URL mapping.  
Billions of small sized(1kb) object. There is no relationship between objects.  
We would also need to store data for analytics, for example, how many times was the url opened in the last hour?

Should we choose RDBMS or NOSQL?

**Hint :**Things to consider :

* Are joins required :  
  NoSQL databases are inefficient for joins.  
  In this case, assuming we don’t need analytics, we only need to answer the query “Given a hash, give me the corresponding URL” - a standard key to value lookup. As such, we don’t need any joins here.
* **Size of the DB :**  
  If the size of the data is so small that it fits on a single machine’s main memory, SQL is a clear winner. SQL on a single machine has next to zero maintenance overhead and has great performance with right index built. If your index can fit into RAM, its best to go with a SQL solution. Lets analyze our current case :

*Assumptions*:

* # of writes per month: 100 Million (Refer to estimations section)
* Avg size of URL : 500 bytes
* Provisioning for : 5 years
* Space required : 500 bytes \* 100M \* 12 \* 5 = 3TB

3TB of data can fit on a single machine’s hard disk. However, the index might not. If we store all data on a single machine, our write and read operations would be very slow (Page swaps for indices). Given read latency is critical for us, we can’t store the data on a single machine.  
So, a SQL solution will have a sharding overhead. Most NoSQL solutions however are built with the assumption that the data does not fit on a single machine and hence have sharding builtin.

NoSQL would be a better fit for our case.  
Since we are optimizing for consistency over availability, we will choose a NoSQL DB which is highly consistent like HBase.

**Q:**What would the database schema look like?

**A:**We want to store the mapping from hash -> URL which is ideal for a key value store. In NoSQL domain, we need to be careful when designing the key as entries are indexed by the key.  
In our case, since we will only query by hash which will never update, our key will be hash with the value being the URL.

**Q:**How would we do sharding?

**A:**HBase inherently use consistent hashing for sharding. We explain it in detail at <https://www.interviewbit.com/problems/sharding-a-database/>. Since our total data size is a few TBs, we don’t need to think about sharding across datacentres either.

**2. News feed-** ask ques like when to update real time or after some interval, random or preference for url, what type of content audio video, only followed or recommendation, category based?

User id corresponding to which we need news. We need the data from user’s friends. DB is slow, 5k friends. Cache maintain and change accordingly.

200 copies of my newsfeed. My changes will occur in my friends feed. Mega feed

Main server get data from friend server. Store in memory then 200 copies. Distribute the cache memory. All diff cache will get request from main data for news.

Aggregator- if x,y,z like same story.

Cache means a layer over db. If I need same query from database many times then we create a cache layer between db and server. Cache is inside ram.

Ranking- and then generate top 20 story according to ML

We can use map data structure like which story corresponding to which user. Lazy loading- load when we require, but here we load beforehand.

**3. Search engine**- query-spell character- type predictor in drop down,

Root word extraction ignoring articles and useless word- query reprocessing

Server will return web page top 20 and then well return.

Web crawler- new webpage, content updation

*Scalability*- technology used takes how much time on each request on avg- 1ms

1s-1000request process, db me kitni query, how many processor, how fast, num of user, how many in a day,

* **Q:**How many typeahead suggestions are to be provided?  
  **A:**Let's assume 5 for this case.
* **Q:**Do we need to account for spelling mistakes ?  
  **A:**Example : Should typing *mik* give michael as a suggestion because michael is really popular as a query?
* **Q:**What is the criteria for choosing the 5 suggestions ?  
  **A:**As the question suggests, all suggestions should have the typed phrase/query as the strict prefix. Now amongst those, the most relevant would be the most popular 5. Here, popularity of a query can be determined by the frequency of the query being searched in the past.
* **Q:**Does the system need to be realtime ( For example, recent popular events like “Germany wins the FIFA worldcup” starts showing up in results within minutes ).  
  **A:**Let's assume that it needs to be realtime.
* **Q:**Do we need to support personalization with the suggestions? ( My interests / queries affect the search suggestions shown to me).  
  **A:**Let's assume that we don’t need to support personalization
* **Q:**How many search queries are done per day?
* **A:**Assuming the scale of Google, we can expect around 2-4 Billion queries per day.
* **Q:**How many queries per second should the system handle?
* **A:**We can use the estimation from the last question here.  
  Total Number of queries : 4 Billion  
  Average length of query : 5 words = 25 letters ( Since avg length of english word is 5 letters ).  
  Assuming, every single keystroke results in a typeahead query, we are looking at an upper bound of 4 x 25 = 100 Billion queries per day.

**Q:**Is Latency a very important metric for us?

**A:**A big Yes. Search typeahead almost competes with typing speed and hence needs to have a really low latency.

**Q:**How important is Consistency for us?

**A:**Not really important. If 2 people see different top 5 suggestions which are on the same scale of popularity, its not the end of the world. I, as a product owner, am happy as long as the results become eventually consistent.

**Q:**How important is Availability for us?

**A:**Very important. If search typeahead is not available, the site would still keep working. However, it will lead to a much degraded experience.

**Q:**What would the API look like for the client?

**A:**

Read: List(string) getTopSuggestions(string currentQuery)  
Write: void updateSuggestions(string searchTerm)

**Q:**What is a good data structure to store my search queries so that I can quickly retrieve the top 5 most popular queries?

**A:**For this question, we need to figure out top queries with another string as strict prefix. If you have dealt with enough string questions, you would realize a prefix tree (or trie) would be a perfect fit here.  
Devil however lies in the details. We will dig deeper into the nitty gritty of this in the next section.

**Q:**How do we handle the case where our application server dies?

**A:**The simplest thing that could be done here is to have multiple application server. They do not store any data (stateless) and all of them behave the exact same way when up. So, if one of them goes down, we still have other application servers who would keep the site running.

**Q:**How does our client know which application servers to talk to. How does it know which application servers have gone down and which ones are still working?

**A:**We introduce load balancers. Load balancers are a set of machines (an order of magnitude lower in number) which track the set of application servers which are active ( not gone down ). Client can send request to any of the load balancers who then forward the request to one of the working application servers randomly.

So, now whenever we get an actual search term, we will traverse down to the node corresponding to it and increase its frequency. But wait, we are not done yet. We store the top 5 queries in each node. Its possible that this particular search query jumped into the top 5 queries of a few other nodes. We need to update the top 5 queries of those nodes then. How do we do it then? Truthfully, we need to know the frequencies of the top 5 queries ( of every node in the path from root to the node ) to decide if this query becomes a part of the top 5.  
There are 2 ways we could achieve this.

* Along with the top 5 on every node, we also store their frequency. Anytime, a node’s frequency gets updated, we traverse back from the node to its parent till we reach the root. For every parent, we check if the current query is part of the top 5. If so, we replace the corresponding frequency with the updated frequency. If not, we check if the current query’s frequency is high enough to be a part of the top 5. If so, we update the top 5 with frequency.
* On every node, we store the top pointer to the end node of the 5 most frequent queries ( pointers instead of the text ). The update process would involve comparing the current query’s frequency with the 5th lowest node’s frequency and update the node pointer with the current query pointer if the new frequency is greater.

**4. Design LRU chache**- least recently used cache- nayi lani, purani hatani, remove one that is used much before.

5. Automatic reference counting in Cpp

6. Youtube

7. Google drive

8. messenger- 1 to 1 or group, size of msg, attachment, notif, msg size limit.

**OS**

Interface between computer hardware and system application programs.

Program execution is called process. Memory distribution:

1. text- compiled program

2. data- global static memory

3. heap-DMA

4. Stack- local variable

Process state- new, ready, running, waiting, terminate

A single program say web browser can have any number of instances running at any time

Modes of OS:

1. kernel mode- direct access to hardware&memory, whole system can hault if any error

2. user mode-program will hault

Types of scheduler:

1. long term scherdular- new-ready me kise bheju, controls degree of multi programming, avg incoming=avg departing

2. CPu/short term- ready-running, dispatcher- allocates memory

3. medium term scheduler- swap in swap out- remove any process

2 types of process: independent and cooprative. Race condition occurs in cooprative process.

Deadlock

Request, use and release request resource.

Arises when mutual exclusion, hold and wait, no premption and circular wait hold simultaneously.

Deadlock handling-

1. prevention/avoidance-

2. deadlock detection/recovery

3. ignore and reboot system- windows/unix

Types of thread:

1. user level- no intervention with kernel, create application by thread application, fast to create and manage thread, block system call then entire process is interepted

2. kernel level thread- through os, slow, multiprocessor

Multithreading models

1. many to one model- many user threads mapped to 1 kernal thread, if one thread stops then entire kernel is blocked.

2. one to one-

3. many to many- multiple user thread mapped to multiple kernel thread. Best model

Process address space- set of logical address that process addresses/refers in code.

Static loading – whole dependency is guaranteed at time of compilation

Dynamic- store reference before compilation and load at time of runtime

Paging- avoid external fragmentation, in physical space we want to store non contiguous allocation