Instructor: https://linktr.ee/agarwal.pragy

Masterclass duration: 3 hours

Approaching a Design problem

Data Structures & Algorithm (DSA) Interviews

- Problem statement is concrete - well defined
- Example input-output pairs
- Idea about the kind of solution that will work
    - input constraints
    - input: array of integers of size N <= 10^6
    - can you use a $O(n^2)$ algo? No. Max $O(n \log n)$

1. Problem Statement
2. Examples - solve it by hand
3. Brute Force
4. **Drawing Observations from the examples you took**
5. Optimized

Design Interviews

- Problem statement is extremely vague
  "Design Typeahead"
- you might not even know what typeahead is
- you often don't know whether you've to do a LLD or HLD
    - LLD:
        - architecture of codebase
        - ER diagram, database schema design, API structure, service layers, code design patterns, ..
        - help you scale your team and your codebase
    - **HLD:**
        - architecture of your infrastructure
        - choosing the correct DB. Design a cache. What to Load Balancer to use, …
        - helps you scale from 100 users to 1 billion users
- interviews are very open ended
    - discussion can go in any direction
    - depends on your expertise
    - depends on the interviewer's expertise
- interviews are still time constrained

    - 45 mins
- you must
    - what you're supposed to build
    - figure out what the interviewer wants
    - build that solution
    - take sufficient complex things so that you can showcase your skills & impress the interviewer

Design Twitter
- whiteboard… starr drawing
- LB, microservices, database..

The 5-step design approach

(20-25 mins)
1. Problem Statement
2. Functional Requirements
3. Non-functional Requirements
4. Scale Estimation

(25-20 mins)
5. **System Design**

(15 mins)
6. Advanced questions

Toy Problem: Design a typeahead system. *High Level Design*

High Level Design (HLD)

Suppose you're building a chat application for your friends. You have 5 close friends.
vs if you're building the app for 2 billion people.

System Design - High Level Design:
- what challenges occur as you scale from 100 users to 1 billion users
- how do you solve those challenges – what are the common patterns & solution

In today's class we will discuss the HLD of "Typeahead".
We will NOT be writing any code whatsoever. We will NOT be deploying system. We be NOT be building apps.

Problem Statement

Interviewer: Design Typeahead
Spooky: Oh, I've never heard of typeahead, can you explain what that is.
Interviewer: It's basically a search feature. When you type something in the search box, it tries to autocomplete your query.

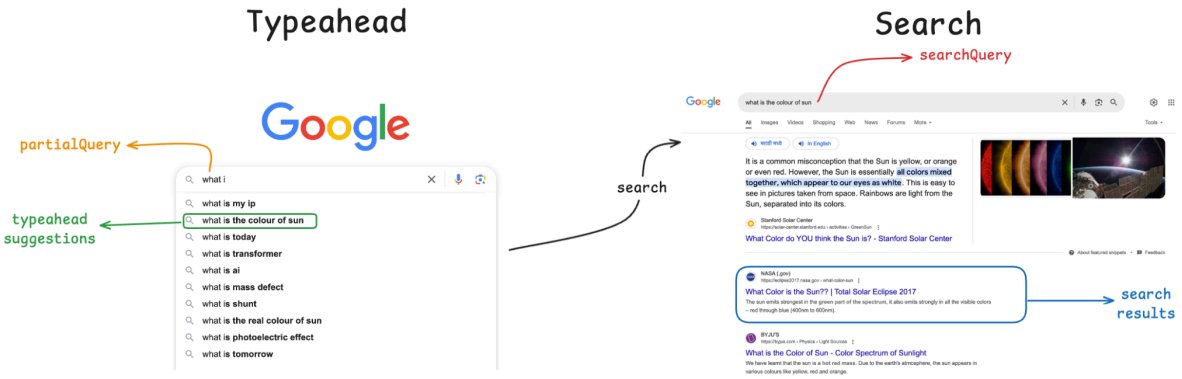
Reason by analogy

Find existing companies / systems that offer a similar product / feature.
Gives you an overview of the umbrella of different scopes that you can consider.

Typeahead is a specific form of autocomplete.
Try to type-ahead of the user

Typeahead is always associated with search

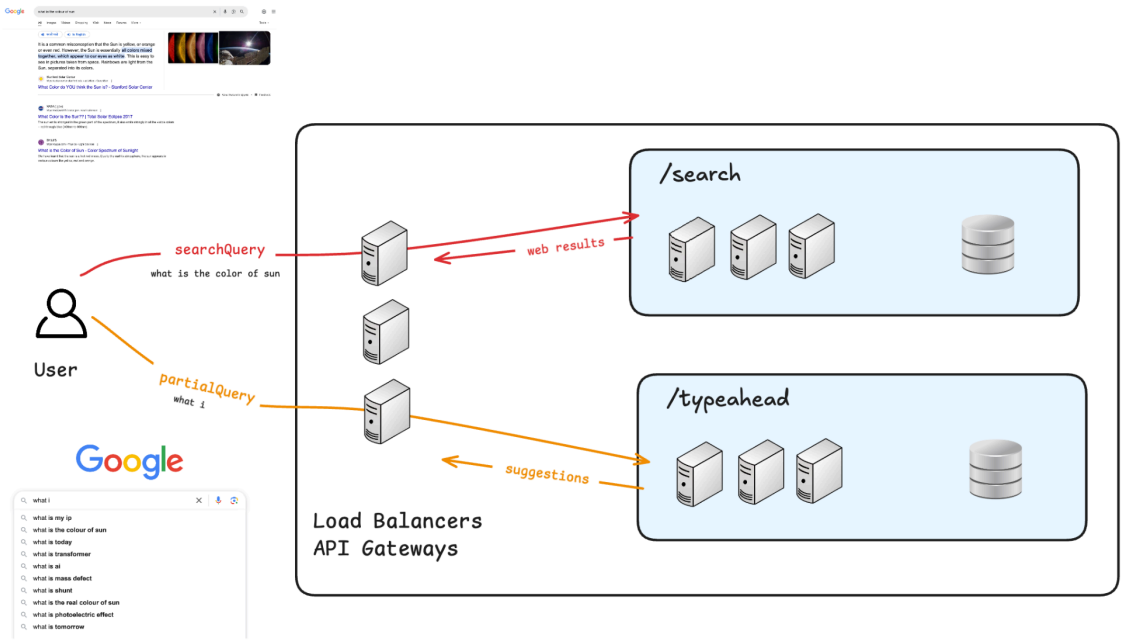
Typeahead in Google



Typeahead in Amazon

Typeahead & Search are different products

but typeahead is always followed by with some form of search.



What not to build

Autocomplete is available in many different settings

1. **Smartphone Keyboards:** word completion / contacts / ..

2. **Text editors / Code editors:**
    a. intellisense: complete next token
    b. Github CoPilot: can show entire functions
3. **Grammarly:** sentence completion while prose writing
4. **Gmail / Outlook:**
    a. sentence completion for writing mail
    b. to/cc/bcc: email completion from contacts

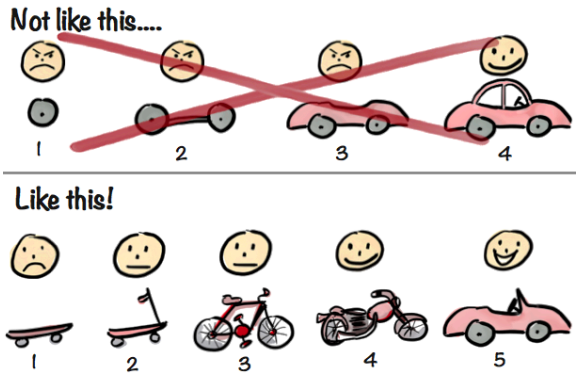
Functional Requirements (FR)

Functional requirements are about the "functionality" that is being offered to the clients/actors

Do NOT go on a feature frenzy — your task is NOT to suggest all the features in the world. Limit yourself to the MVP features — so that we can design them within our limited time

Minimal Viable Product (MVP)

1. **Minimal:** minimal set of features to achieve viability
2. **Viable:** should demonstrate the core features & usability
3. **Product:** solves a genuine problem for someone



Why should you keep the features minimal?

1. Because you've limited time
2. Because you simply cannot dive into everything - it will be too complex.
    a. Google was built during a 45 mins interview?
    b. No! It was built over 30 years by 10,000 devs & product managers & other people
3. Because you need to "impress" the interviewer
    a. you need to "dive" into the complex parts.

Actors

1. **End User:** the person doing the google search

A lot of times you have more than 1 actors

Uber
- drivers
- riders
- backend staff

Amazon
- customers
- sellers
- delivery partners
- support staff

MVP Features

Always keep the "minimal" in your mind. MVP features is not a feature suggestion competition.

“Requirements Gathering” — you’re supposed to figure out the features. The interviewer will NOT tell you all the features. (they might tell you some features)

And you simply can’t ask for the exact features. You can ask clarifying questions — whenever you ask a question, you should also provide a suggestion (your best answer to the question that you’ve asked).

Example feature (for Uber):
- when a customer requests a ride, we should match their request with the available nearby drivers.

MVP features perfect for discussion (v0)	**Future Scope** good to have, but not absolutely necessary (v1+)	**Bad** get yourself rejected in an interview (v-never)
As the user is typing in the search-box, we should auto-complete their search query. *We should show "typeahead" suggestions.* `typeahead(partial_query) => suggestions_list` user-facing	Typeahead results should be personalized 1. geolocation 2. user's browsing/search history	Display a textbox in which the user can type something (frontend)
The suggestions shown in typeahead are popular searches that other people have made in the past → we should keep track of the popularity of searches.	Results should take recency/news factor into consideration *(trending topics)*	Clicking a typeahead suggestion should lead to a search (frontend)

`log_search(search_query) => void` internal — will be invoked by the search microservice		
We will limit to showing only ***top 5***, ***relevant*** suggestions. ● **relevant:** any suggestion should be a *prefix match* with the partial_query ● **top 5:** we will keep track of the number of times a `search_query` has been searched for (globally) — basically keep track of the "popularity" of a search query. Given a `partial_query`, we will show the search_queries that are prefix match with `partial_query`, and have the highest search count (most popular) *We will show the 5 most-popular prefix matches*	Fix typos/Spell-correct the user's partialQuery before showing the typeahead results	While showing the suggestions, the part that has already been typed should be highlighted (frontend)
With every letter that the user types, the typeahead suggestions should change *we will update the typeahead with every letter - send a new API request with every letter typed*	Even when the user has typed `< 3` characters, we should suggest the "trending" global searches / previous recent searches by the user	after giving suggestions, if user press tab button, it should write copy the query in the text box (frontend)
Constraints: We will start showing typeahead suggestions only ● after the user has typed at least `3 letters` ○ otherwise, we will probably show irrelevant results ● if the user's query is at most `50 letters` ○ most likely, they've slept on the keyboard		user age based personalization query: ***why is s*** 5 years => ***why is s****ky blue* 30 years => ***why is s****cotch smooth*
		Grammar-correct the user's partialQuery before showing the typeahead results
		provide suggestions as per the *law of the land* *(what??)*

What to avoid

1. anything that doesn't effect our backend infra / system design
2. frontend features
    a. don't effect our backend infra / system design
    b. purely cosmetic / purely frontend
3. common features that are not the core of the system that we're trying to build
    a. design Uber
        i. we require **authentication**
        ii. is authentication what the interviewer is asking you to design
        no - it's just a dependency
        iii. we require **payments**
        iv. when the interviewer asks us to design Uber, they're not asking us to design payments
        v. we will assume that auth & payments already exist
4. any suggested feature should come in the form of a complete sentence (from the user's perspective)

Hidden features

How to find hidden features?

For every feature that you discuss, think of the following

1. API
2. Data origination & flow

API (internal / user-facing)

```
typeahead(partial_query) => suggestions_list
```
user-facing

```
log_search(search_query) => void
```
internal

~~GET /typeahead?partialQuery={}~~

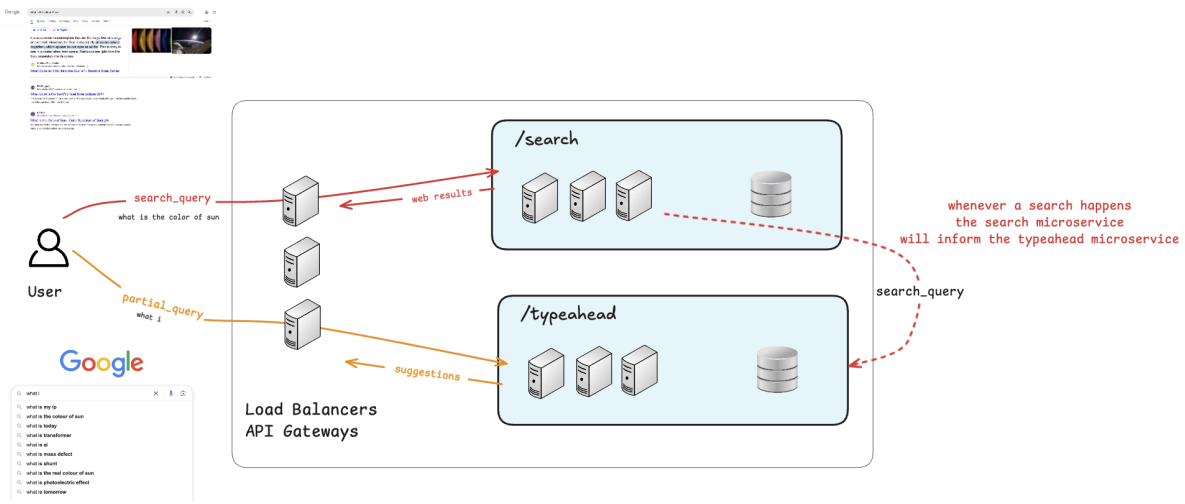
- ~~why REST why not SOAP / RPC / ...~~
- ~~why GET, why not POST~~
- ~~naming convention~~

API: input & output (interface)
Protocol: how the communication happens (REST / ...)

You should NOT worry about the protocol - if you can, that's good - but limit yourself.

Data Source & Flow

Where does the list of suggestions come from? How is the "search query" data getting populated?



Users are constantly searching on Google.
Every time a user makes a search (search_query)

- this search_query goes to the /search microservice
- search microservice will return the relevant web-results
- (async) the search microservice will inform the /typeahead microservice that this search has been made (pass the search_query)
- the /typeahead service can store this search_query to populate its suggestions list

Non-Functional Requirements (NFR)

(design goals)

Functional requirements are the “features” that you provide for your clients (end users / internal systems)
Non-functional requirements are not features – they’re “meta-features” (features about features)

Feature: typeahead suggestions should show up as the user is typing
Meta-feature: typeahead should be super fast (low latency)
Meta-feature: system should be secure
Meta-feature: system should not go down (availability)

PACELC

“In face of a n/w **P**artition, you need to trade-off b/w **A**vailability and **C**onsistency. **E**lse, even if there’s no n/w partition, you still have a trade-off b/w **L**atency & **C**onsistency”

If you want a system which is highly available & has low latency, you’ve to give up on immediate consistency — instead, you will get “eventual consistency”

Consistency vs Availability

Do NOT jump to an answer

Q: What is the data ?

Past search queries and their corresponding counts.

Search Query	Count
why is the sky blue?	49,999
why is water wet?	50,000
what is the color of the ocean	5,000

Q: What does eventual consistency mean in this context ?

When is the data changing?
Everytime someone makes a search

- *the “true” count of some entry “should” change*
- the top-5 relevant results for some typeahead partial_query “should” also change

What if the search counts are stale (outdated)
Then, it is possible for the search-count to reflect in a delayed manner in the top-5 results.
For some time, it is possible that the true count of Query1 > Query2, but we still show Query2 above Query1 in the typeahead suggestions.

Q: Can we afford stale reads in this situation?

Yes, that's okay

1. because the user doesn't really know (or care about) the true search counts
2. we don't necessarily need to maintain a strict order in the typeahead suggestions
    a. it is okay for a few good suggestions to not be shown, as long as the ones shown as also good enough

Q: What does data loss mean in this context ?

Data loss would happen if someone has searched for a query, but for whatever reason, we were not able to increment the count for that query (query didn't get logged).
The counts will be off by a small amount.

Search Query	True Count
why is the sky blue?	49,999
why is water wet?	50,000
what is the color of the ocean	5,000

Search Query	DB Count
why is the sky blue?	49,999
why is water wet?	49,998 (no issues)
what is the color of the ocean	5,000

Q: Can we afford data loss in this situation?

Yes, that's okay.
(same reasons as above)

So out of available vs consistency, we will go for availability — it is okay to have eventual consistency

Consistency vs Latency

What are our latency requirements?

Ultra-low latency — literally competing with the user's typing speed!
each typeahead query to take **< 10ms**
Latency: for how much time does the request have to wait before the request can be processed.

(this doesn't include the round trip - that depends on the user's distance from the servers - don't have much control over that - only solution for that is to have geolocated servers)

Others

- Security: do NOT do this unless you're a security expert
    - 99% of all security issues happen because non-security expert devs try to do security
- Observability (detect issues, diagnose, verify the fix)
- Rate Limits
- Idempotency - can there be duplicates
- Delivery Guarantees - messages lost or messages delivered twice (notifications)
- Order of delivery (of messages)

Mastering HLD

Before you learn HLD - you must first master

- DSA
- LLD
- Computer Networks (deep dive is not necessary - OSI model, and basics of how the internet works, and how DNS works)
- Relational databases & SQL

You MUST invest the time that this topic deserves!

Scaler HLD Curriculum (1.5 months – 2 months)

Lecture	Topics Covered

System Design and Computer Networks 101	• High Level Design (HLD) - what & why • Case Study - bookmarking website • DNS, IP Addresses, Domains (how internet works) • Horizontal vs Vertical Scaling • Load Balancing - what & why • Heartbeat & HealthCheck • Scaling Mindset
Load Balancing and Consistent Hashing	• Load Balancing Techniques • Horizontal vs Vertical Partitioning • Sharding • Routing Algorithms • Round Robin • Consistent Hashing • Stateless Servers
Caching: CDN + Backend Caches, Cache Invalidation	• Memory Hierarchy • In-Browser caches • Content Delivery Networks (CDN) • Backend Caching • Local vs Global Cache • Single vs Distributed Cache • Cache Invalidation & Eviction Algorithms • Write Around / Write Through / Write Back / TTL • Cache Consistency
Case Study: Caching Scaler Code Judge & Contest Leaderboard	• Designing a Cache • Caching testcases in Scaler Code Judge • Global vs Local Cache • LRU Eviction • CPU vs I/O bound processes • Ranklist Computation • Caching Leaderboard in Scaler Contests • Redis Overview
Case Study: Caching Facebook News Feed	• Caching NewsFeed in Facebook • Scale Estimation • Newsfeed Computation • Fan-out reads • Database Sharding • Read Replicas • Cron Job
CAP / PACELC theorem + Master Slave Replication	• Consistency, Availability, Network Partitions & Partition Tolerance • CAP & PALEC theorem • Proof by example • A+P / C+P / A+C systems • Database Replication • Replication vs Sharding • Master-Slave Replication • Eventual Consistency, Immediate Consistency, Data Loss • Quorum, Tunable Consistency • Multi-Master Replication
SQL vs NoSQL + Sharding	• Relational Databases • Pros & Cons of SQL • Normalization • ACID transactions • ACID consistency vs CAP consistency • Database Schema • Unstructured Data • Sharding • Choosing a good Sharding Key • Fanout queries • NoSQL Databases • SQL vs NoSQL • ACID vs BASE • Denormalization & Replication • Key-Value databases (Redis) • Document Databases (Mongodb) • Wide Column Databases (Cassandra) • Large File Storage (S3) • Other database types (graph, vector, ..) • Choosing the correct database • Row wide vs Column wide storage
Database Orchestration & Shard Creation	• Sharding without Replication • Sharding with Master-Slave Replication • Database Orchestrator • Adding / Removing Servers • Extra Replicas • Replication Factor • Seamless Shard Creation (Simulation & Real Phase) • Multi-Master Replication
Case study: Typeahead (Google, Amazon)	• HLD Interviews • 5 step process to Approaching a Design Problem • Problem Statement • Functional Requirements • MVP Features • Features to avoid during interviews • Non-Functional Requirements • Scale Estimation • Pareto Principle (80-20 rule) • Avg. vs Peak Load • Read vs Write heavy systems • Tries • Hashtables • Optimizing Writes - Batching & Sampling • Recency & Personalization
NoSQL Internals - LSM Tree + Bloom Filter + Sparse Index	• Data storage in SQL vs NoSQL • Log Structured Merged (LSM) Trees • Write Ahead Log (WAL) Files • MemTable & Caching • Sorted String (SS) Tables • Visualization of LSM trees • Compaction Process • Compaction Strategies • Sparse Index vs Full Index • Deletion & Tombstones • Bloom Filters • Sequential vs Random reads in HDD
Case Study: Messaging Apps (FB Messenger, Whatsapp, Slack)	• Message Ordering • Automatic Retries & API Idempotency • Scale Estimation • 1-1 messages • Group messages • Illusion of Consistency with High Availability & Low Latency • Database choice • Caching messages

Messaging Queues - Apache Kafka & Zookeeper	• Publisher Subscriber • Message Queue • Decoupling, Load Leveling, Buffering • Async Task Processing • Event Driver Architectures • Pros & Cons of message queues • Apache Kafka • Event Schema • Producers & Consumers • Pull vs Push • Consumer Groups • Topics & Partitions • Message Retention • Message Ordering • Partition Key • Partition Mapping & Consumer Offsets • Consumer Rebalancing • Delivery Guarantees - At most once, At least once, Exactly once • Message Brokers • Fault Tolerance • Apache ZooKeeper • Distributed Configuration Management • KRaft • Persistent vs Ephemeral nodes • Master election via Zookeeper
Case Study: ElasticSearch (Full Text Search)	• Full Text Search • Why SQL is suboptimal for text search • Usecases • Inverted Index • Text Preprocessing pipeline - Parsing, Tokenization, Normalization, Stop Word Removal, Reduction, Stemming, Lemmatization • Indexing • Term Frequence, Document Frequency, TF-IDF score • Apache Lucene • Sharding by Key, by Document ID • Elasticsearch • Fan-out reads • Sharding & Replication in ElasticSearch • B+Trees • Binary vs Ternary vs k-ary
Case Study: S3, HDFS (Large File Storage)	• Large File Storage (S3, HDFS) • Log Aggregation • Multimedia uploads • File chunking • Ideal Chunk Size • Name Nodes & Data Nodes • Hadoop Distributed File System • Replication • Rack aware algorithms • File uploads & downloads • Bittorrent
Case Study: Uber (Nearest Neighbor Search)	• k-Nearest Neighbor (kNN) search • Grid based approach • Dynamic Grid Sizing • QuadTrees • Nearest Restaurant in Zomato/Swiggy • Space Requirements for storing Quadtrees • Nearest Drivers in Uber • Ride booking request flow • Optimizing Quadtree writes • Optimizing movement within node boundary • Quadtree location vs structure updates • Notification Systems crash course • Uber H3 (Hexagonal Hierarchical Spatial Index)
Rate Limiter + Unique ID Generator	• ID generation @ Scale • Universal Uniqueness • Strictly Increasing, Lexicographically Sortable • Roughly Sortable • Enumeration Attacks • Randomness & Sparsity • Decentralization • Autoincrement integers in SQL • UUIDv4 • Timestamp • Network Time Protocol (NTP) • Twitter Snowflake • ULID, UUIDv6
Case Study: Hotstar, Google Meet, Netflix/Youtube, Scaler Live Classes, Whatsapp Video/Voice calls (Video Streaming)	• Challenges with Video Streaming • Adaptive Bitrate Streaming (ABS) • Resolution vs FPS • User Agent Detection • Video buffering • Chunking • Just in Time downloads • HTTP Live Streaming (HLS) • m3u8 format • Streaming Pre-recorded videos (youtube, netflix, jio cinema) • CDNs for video serving • Streaming Live Videos (hotstar, twitch) • Video Streaming Latency • Auto Scaling challenges • Ideal Chunk Size • Scaler Live Classes • Google Meet • Whatsapp video calls
Microservices - 1 - Monolith vs Microservices vs SOA	• Types of Scale • Monolithic Architecture • Scaling a Monolith • Pros & Cons of Monoliths • Modular Monolith • Microservices Architecture • Pros & Cons of Microservices • Typical Backend Architecture • Untrusted vs Trusted Layer • Service vs Server • API Gateway • Reverse Proxy • Load Balancers • Trusted Virtual Private Cloud (VPC) • Authentication vs Authorization • When to use a Monolith vs Microservices • Breaking a Monolith into Microservices • Hybrid Architecture • Handling Cross Cutting Dependencies • Service Oriented Architecture

Microservices - 2 - Communication, Observability, Resilience	• Monolith Communication • Client to Microservices Communication • API Gateway • GraphQL • Backend for Frontend • Microservice to Microservice communication • Synchronous Communication • RESTful APIs • SOAP • Remote Procedure Calls (gRPC) • Inter Process Communication (IPC) • Asynchronous Communication • Message Queues, Event Driven, Event Streaming • Observability • Metrics & Monitoring • Alerts • Logging • ELK Stack • Distributed Tracing • Traces & Spans • Correlation ID • Resilience • Common Failure Types • Cascading Failures • Thundering Herd • Basic Defenses • Circuit Breaker • Bulkhead • Graceful Degradation • Chaos Engineering • Sidecar
Microservices - 3 - Data Management, Consistency & Distributed Transactions	• Managing Data • Command Query Responsibility Segregation (CQRS) • API Composition / Aggregation • Consistency in Microservices • Two Phase Commit (2PC) • Saga Patterns • Orchestration • Choreography • When to choose what • Event Sourcing

Software salaries in India go up to 3 Cr (base) + stock on top
This is not silicon valley, or Seattle, or London..
This 3Cr package is tier-1 companies (Uber, Netflix, Amazon, Flipkart, Google, PhonePe) for SDE3+ (10+ years of experience)
These salaries are in India - Hyderabad, Bengaluru, Mumbai, Chennai, Gurgaon

Why on earth would a company pay this amount of money for a single person?
Because that person knows

- how to design systems at scale (HLD)
- how to ensure that the design doesn't have to change when the requirements change (LLD)

Scale Estimation

The scale will dictate our design choices

If you're building typeahead for 10 users, then how will you design it?
I would not even write a software. I will hire a person to manually tell the users what they can search.

If you're building it for 5 billion people — the design will change, even though the requirements are same.

Staff Engineer @ Google - HLD Interview Question

Question: Given a bunch of strings, you have to sort them in the lexicographic (dictionary) order.

Input: `cat, bottle, apple, design, system, doctor, trie, camel`
Output: `apple, bottle, camel, cat, design, doctor, system, trie`

```
with open('words.txt') as f:
     lines = f.readlines()
     print(sorted(lines)
```

Catch: there is 50 PetaBytes of data.

1 bit → unit of information
4 bits → nibble
8 bits → byte
10^3 = 1000 bytes → kilobyte
1024 bytes → Kibbibyte (KiB)
10^6 bytes → MegaByte
10^9 → Giga
10^{12} → Tera
10^{15} → Peta

50 Petabytes is 50,000,000 GB of data

Will this fit on your RAM/HDD?

No!

- distribute across datacenters
- prevent data loss: replicate the data (have backups)
- servers will go down
- network will go down
- programmer errors
- hard disks will crash
- network is slow
- different servers will perform differently (heterogenous)

Simple problems can become extremely complex at scale.

Goals

1. Amount of Data *(in bytes)*
    - Is sharding needed?
2. Amount of Load
    - Requests per second *(for each API)*
3. Peak Load (vs avg load)

4. Read-heavy / Write-heavy
    - guide our database/cache choices

All these things will help us guide our choice of database, cache, & sharding

Does accuracy matter?

No. Back-of-the-envelope estimates / approximations / guesstimations

being off by a factor of 2 or 3 is okay.
Actual value: 100
You suggest: 50 okay
You suggest: 200 okay
You suggest: 10 not okay (off by a factor of 10)

However, any estimates that you make should be justified. Don't pull out numbers from thin air. You will have to make some assumptions - each assumption should be clearly stated as such.

For example, “how many footballs can fit in this room”?
- Let’s assume that a football is 1 feet in diameter
- Let’s estimate how large the room is
    - Length:
        - assume that the floor tile is 2ft x 2ft
        - count the number of floor tiles 8x7 tiles
        - 16ft x 14ft

On our universe it is always better to make assumptions, feed that into a model, and get the answer from the model.

How to start

Start by estimating the number of Daily Active Users (DAU).

Total number of users / Monthly Active Users (MAU)

World Population: `~8.5 billion`
Number of Internet Users: `~5 billion`
India's Population: `~1.5 billion`
USA's Population: `~350 million`

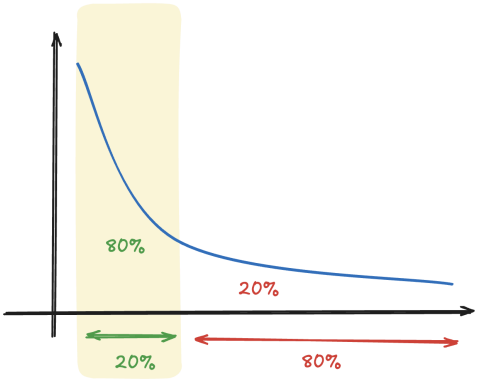
Pareto Principle (the 80/20 rule)

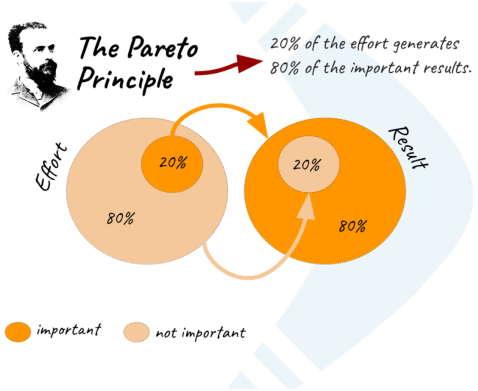
Only 20% of your users will be active.

Can be extended to the 80-20-1 principle for social media

- 80% are just passively browsing (remaining 20% will not log in on a particular day)
- 20% are interacting (comments, likes, shares)
- 1% are content creators (posts)

Can you apply this rule for whatsapp? Only 20% of whatsapp users are active daily? NO! For whatsapp you should assume ~80% of the users are active daily.





Fun stuff — Pareto principle & Zipf's Law: *The Zipf Mystery*

Estimates

assumption: Google has `5 billion` users.

Daily Active Users (DAU): `20% (assumptions)`
`= 20% of 5 billion users`
`= 1 billion users`

assumption: *(average)* number of searches per active-user per day: `20`

Total number of searches / day
`= (20 searches / ~~user~~ / day) * (1 billion ~~users~~)`
`= 20 searches / day * 1 billion`
`= 20 billion searches / day`

Avg. searches / second
`= 20 billion searches / day`
`= 20 * 10^9 searches / (10^5 seconds)` (1 day = 86,400 seconds ~ 10^5 seconds)
`= 200,000 searches / second`

Note: searches are going to search API, not the typeahead API

assumption: Number of typeahead queries per search query:

assumption: avg size of a search query is `10 letters`
we will start showing suggestions after the first 3 letters
avg. number of typeaheads per search would be `(10 - 3) = 7 ~ 10`

`typeahead(...)` API: requests / second

Everytime a user is searching for something, they will first type the query, and then they will search.

search_query: "why is the sky blue"

1. typeahead("why")
2. typeahead("why i")
3. typeahead("why is")
4. typeahead("why is t")
5. typeahead("why is th")
6. typeahead("why is the")
7. typeahead("why is the s")
8. ...
9. typeahead("why is the sky blue")

`= (200,000 ~~searches~~ / second) * (10 typeaheads / ~~search~~)`
`= 2 million typeaheads / second on average`

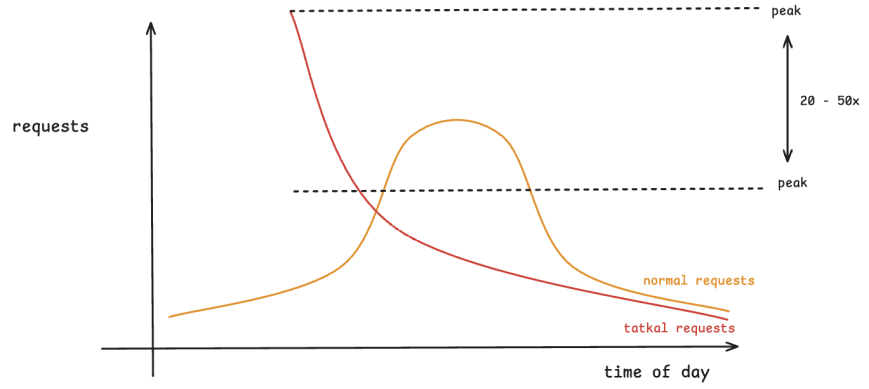
`log_search(...)` API: requests / second

Every time a search is made, the `/search` microservice pings the `/typeahead` microservice
`= 200,000 requests / second`

Peak Load

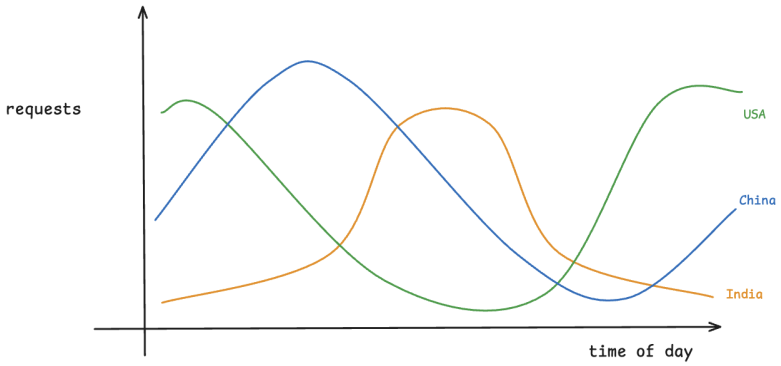
assumption: peak load is `5x` the average load
`= 5 * (2 million typeaheads / second)`
`= 10 million typeaheads / second`

This system has “crazy” scale!



day-day operation
peak can be very large (50x)

IRCTC



day-day operation
peak & avg are very close (2x)

world-events (covid/earthquake)
peak can spike (5x)

Google Typeahead

Amount of Data

What's the data?
Past search queries & their counts

Search Query (10 chars = 10 bytes)	**True Count** (8 bytes)
why is the sky blue?	49,999
why is water wet?	50,000
what is the color of the ocean	5,000
2+2	10 billion

1 character takes 1 bytes (ASCII encoding)
1 character takes 2 bytes (Unicode encoding)

Avg. size of search query: `10 letters` (made this assumption earlier)
Avg. size of each entry = 10b + 8 b = 18b **~ 20b**

why 8 bytes for storing the count?
int: 4 bytes (32 bits)
long/bigint: 8 bytes (64 bits)

Gangnam Style was the first video on youtube to cross 1 billion views.
Youtube used a 4 byte counter to store the number of views for a video.
Range for a signed 4 byte integer:
`approx -2 billion … 2 billion`
When *Gangnam Style* crossed 2 billion views, for a few days, Youtube showed a -ve view count because the integer had overflown!

Range for a signed 8 byte integer:
`approx -8 quadrillion … 8 quadrillion`

How many entries in this table?

Should we say that for each search that happens, we will have an entry in this table?

No. Because for new searches, we're adding entries
But for queries that have already been searched earlier, we're just updating the counts

The number of entries in the table will depend on the number of "unique" (never seen before in the history of Google) search queries that happen

assumption: % of search queries are new (never seen before) = 10%

number of new entries / day
= 10 % of (20 billion searches / day)
= 2 billion **new** entries per day

amount of data / day
= (20 bytes / ~~entry~~) * (2 billion ~~entries~~ / day)
= 40 GB / day

total amount of data (over 20 years)
= (40 GB / day) * (20 years)
= (40 GB / ~~day~~) * (20 * 400 ~~days~~) (365 ~= 400)
= 32 * 10^4 GB
= 320 TB

1 bit = unit of information
1 nibble = 4 bits (a small byte)
1 byte = 8 bits

1 kilo byte = 1,000 bytes (10^3) 1 kibbi-byte (KiB) = 1,024 bytes
1 mega byte = 1 million bytes (10^6)
1GB = 10^9
1 TB = 10^{12}
1 PB = 10^{15}
1 EB = 10^{18}
1 YT = 10^{21}
For example, the entire internet is ~ 10 YB in size!

Can this amount of data fit on a single server?

Probably yes, with modern storage.
But, will a single server be able to handle 10 million requests / second?
Absolutely no!

We need **sharding**!

Read Heavy / Write Heavy?

`typeahead` ⇒ **read** request (show suggestions)
`10 million requests / second peak load`

`log_search` ⇒ **write** query (increment count)
`1 million requests / second peak load`

reads are 10x more than writes

We will call this **read heavy**, because even though the system has to handle lots of writes too, it is **dominated by reads**.

Whenever you've read heavy systems & the writes are also significant

- you can absorb your reads in a cache
- optimize your DB for writes

No database system is optimized for both reads & writes (that's impossible)

Tunable consistency allows you to optimize the database for either reads or writes, but not both.

System Design

Typical Components

- Microservices
- Load Balancer
- Database choice
    - SQL
    - NoSQL
        - justify
- Cache choice
    - Need for caching
    - Type of cache
        - Local vs Global
        - (if global) Single vs Distributed
    - Invalidation
    - Eviction
- Communication
    - user - backend
    - backend - backend
- CDN
- Observability

Q: What's the ideal database to store the query-counts data?

Types of Databases

1. **SQL (Relational Databases)**
    a. PostgreSQL, MySQL, Oracle DB, IBM DB2, Sqlite, MSSQL, ...
    b. Strengths
        i. ACID transactions
        ii. Relations & Joins (complex schema)
        iii. Rigid Schema (structured data)
        iv. Indexing
        v. Rich & Complex querying
    c. Weaknesses
        i. Do not scale well
            1. no built-in support for sharding
            2. sharding negates most of the strengths
        ii. Difficulty with unstructured data

2. **(no-sql) Key-Value**
    a. Redis, Firestore, DynamoDB, Memcached
    b. Strengths
        i. very fast (usually in-memory, with optional disk persistence)
        ii. auto-sharding - scale well
        iii. very simple

c. **Weaknesses**
    i. no rigid schema
    ii. no joins / relations
    iii. no complex query (get/set)
    iv. no search
    v. no indexes
    vi. no transactions

3. **(no-sql) Document**
    a. MongoDB, ElasticSearch, CouchDB, CockroachDB
    b. **Strengths**
        i. support semi/unstructured data
        ii. support full text search
        iii. index on any top level attribute
    c. **Weaknesses**
        i. no relations / joins
        ii. only local indexes (not global)
        iii. no transactions

4. **Column Family**
    a. Cassandra, ScyllaDB, HBase, BigTable
    b. **Strengths**
        i. fast aggregate queries over a single column
        ii. semi-structured data
        iii. very fast inserts
        iv. support time based pagination efficiently
    c. **Weaknesses**
        i. no joins
        ii. reading a row is slow (because column major storage)
        iii. design upfront

5. Graph (rarely used)
6. Vector (rarely used)

Ideal Database for storing query-counts

Search Query	Count
what is the color of the sky	5000
what is the day today	1000
what is 2 + 2	10000
what does the fox say	2000

what does a fox eat?	1900
how to kill someone	2000
how to cook eggs	1500
how to sing	500

How will the queries work?

You cannot possible go through all the billions entries that match your prefix, and sort, and find the top 5 for each request!

So you need to pre-compute this and store it in a cache!

Search Frequency DB

for each actual `searchQuery` it will store the count

Search Query	Count
what is the color of the sky	5000
what is the day today	1000
what is 2 + 2	10000
what does the fox say	2000
what does a fox eat?	1900
how to kill someone	2000
how to cook eggs	1500
how to sing	500

Top Suggestions DB (cache)

for each possible `prefix` *(for which our system will show suggestions)*, it will store the top k suggestions

Prefix (partial query)	**Top k `(=3)` suggestions**

wha	`[`
	`  what is 2+2 => 10000`
	`  what is the color of the sky => 5000`
	`  what does the fox say => 2000`
	`]`

what	`[`
	`  what is 2+2 => 10000`
	`  what is the color of the sky => 5000`
	`  what does the fox say => 2000`
	`]`

...	

what i	`[`
	`  what is 2+2 => 10000`
	`  what is the color of the sky => 5000`
	`  what is the day today => 1000`
	`]`

what d	`[` `  what does the fox say => 2000`

	what does a fox eat? => 1500]
how	[how to kill someone => 2000 how to cook eggs => 1500 how to sing => 500]
...	
how to k	[how to kill someone => 2000]

Queries

`typeahead(partial_query)`

If someone types `"what i"` then we can just go to the suggestions db, and look up that partialQuery.

This will be very fast

1. hashmaps have `O(1)` lookup
2. we don't need any computation / processing - we just have to get the value for the given key from the key-value DB

We needed **low latency** (< 10ms), and, **high read throughput** (10 million reads / second at peak)

- Redis lookup has latency of <= 1ms
- A single redis server can easily handle 100,000+ reads/writes per second!
    - just need ~100 servers
      (google has over 10 million servers - as of 2020)

`log_search(search_query)`

Whenever someone searches for a query `"what does the fox say"`

1. I need to update the count in the search frequency database
    - super simple - just call `redis.inc(searchQuery)`
2. I need to update the cache (top suggestions database)
    - what all entries do I need to update?

i. **If I'm updating the count for `what does the fox say` then is it possible for the suggestions of the prefix `"how"` to get changed?**
No.

ii. **What prefixes will get effected?**
only the prefixes of the search query

```
wha
what
what d
...
what does the fox
what does the fox s
what does the fox sa
what does the fox say
```

iii. **How many such prefixes (on average) need to be updated?**
the average query is 10 letters. So ~10 prefixes on average need to be updated.
but this now means, that for each update, I must do `1 + 10` writes
total number of writes in redis

```
= (1 million log_search / sec) * 11 writes / log_search
= 11 million writes / second
```

this makes my system both read & write heavy!
no db in the world that is optimized for both reads & writes

Search Frequency DB

Search Query	Count
what is the color of the sky	5000
what is the day today	1000
what is 2 + 2	10000
what does the fox say	~~2000~~ 2001
what does a fox eat?	1900
how to kill someone	2000
how to cook eggs	1500
how to sing	500

Top Suggestions DB
(cache)

Prefix (partial query)	Top 3 suggestions
wha	[what is 2+2 => 10000 what is the color of the sky => 5000 what does the fox say => ~~2000~~ 2001]
what	[what is 2+2 => 10000 what is the color of the sky => 5000 ~~what does Shubadeep say => 2000~~ what does the fox say => 2001]
...	
what i	[what is 2+2 => 10000 what is the color of the sky => 5000 what is the day today => 1000

	]
what d	[what does the fox say => ~~2000~~ **2001** what does a fox eat? => 1500]
what do	[what does the fox say => ~~2000~~ **2001** what does a fox eat? => 1500]
how	[how to kill someone => 2000 how to cook eggs => 1500 how to sing => 500]
...	
how to k	[how to kill someone => 2000]

Sharding

Sharding is automatic - based on the `hash(key)`

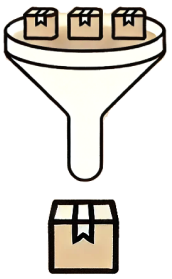
After all this, when you search for something, you have update multiple entries in the cache – because the rankings have potentially changes for each prefix.
The number of writes has increased by approx 10x

The Advanced stuff

Optimizing Writes

We can reduce the number of writes significantly, because, we can afford eventual consistency & data loss (non-functional requirements)

Batching / Batch Processing



Instead of doing each task 1 by 1, you wait for a lot of tasks to pile up. And then you do all of them in 1 go as a batch.

Whenever we get a `log_search(search_query)` request, instead of updating the suggestions db immediately, we wait for the count to increase by a fixed amount (say 1000)

Earlier

```
fn log_search(search_query):        // 1 million qps
    frequency_db.inc(search_query) // 1 million qps
    update_prefixes(search_query)

fn update_prefixes(search_query):
    for i = 3 ... len(search_query) - 1
        prefix = search_query[:i]
        ... logic to update the suggestions list  // 10 million qps
            in the suggestions db
```

Batching

```
fn log_search(search_query):        // 1 million qps
    updated_count = frequency_db.inc(search_query) // 1 million qps
    if updated_count % 1000 == 0:
          // update the prefixes only if the count
          // has changed by the batch size
         update_prefixes(search_query)

fn update_prefixes(search_query):
    ...                                  // 10 million / 1000 qps
```

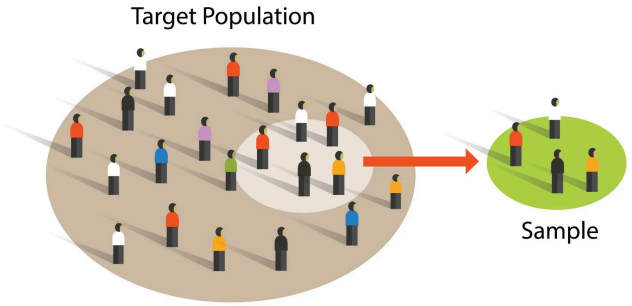
Effectively, we will only increase the counts in multiples of 1000.

Thanks to batching, instead of having `1 million + 10 million` writes/second, we now come back to just `1 million + (10 million / 1000)` = `1,010,000` writes / second

Note: we cannot make the number of writes less than 1 million per second, because that's the number of times we've the update at least the counts in the frequency database.

Note: batching doesn't cause data loss (freq data is always up-to-date) — it just causes delays (stale reads) in the suggestions updates.

Sampling



Exit polls during elections

1. country-wide election: 10 crore voters
2. the actual counting will take several days
3. News channels want to predict the winner with high accuracy before the actual results are announced
4. The news channels will ask various people "who did you vote for" and they will create their own voting list & winners
5. Can the news channels talk to all the 10 crore voters?
   No.
6. They stand in front of a small fraction of the polling booths
   As the people exit the booths, they will ask a few random people "who did you vote for"
7. They will only use this small subset/sample of data to "estimate" the winner
8. if the news channel is unbiased their exit poll results will almost perfectly match the actual election results

If you draw an unbiased sample from a population, then any trends that hold within the population will also hold within the sample.

Typically a good way of getting unbiased samples is to just sample uniformly at random.

Note: Sampling can cause data loss of individual data points. But it won't lose the overall trends. (use it only when data loss is okay)

Earlier

search service

```
fn search(query):
    ...
```

```
    make an API call to typeahead service's log_search endpoint
```

typeahead service

```
fn log_search(search_query):
    frequency_db.inc(search_query)
    update_prefixes(search_query)

fn update_prefixes(search_query):
    for i = 3 ... len(search_query) - 1
        prefix = search_query[:i]
        ... logic to update the suggestions list
            in the suggestions db
```

Sampling

search service

```
fn search(query):
    ...
    if rand() < 0.001:
        // with a probability of 1/1000, make the following call
        make an API call to typeahead service's log_search endpoint
```

typeahead service

```
    ...
```

For 99.9% of the searches, we're not even updating the counts (just ignore them)
For a random 0.1% of the searches, we're calling the log_search and we will update both the counts & the suggestions.

Thanks to sampling, instead of having `1 million + 10 million` writes/second, we now come back to just `(1 million + 10 million) / 1000` = `10,000` **writes / second**

Q: Won't the counts be inaccurate?

Yes. Effectively, each count is being divided by 1000 (but not exactly - it's random)

Q: Won't the suggestions be not exactly in order?

Yes.

But none of that matters. Because still, the suggestions shown to the users will continue to be highly relevant!

Q: If we use sampling, then won't infrequent queries have very low counts / might completely be missed out from the database?

Yes, that can happen! That's a feature, not a bug!
A rare query will never be a part of typeahead suggestions anyway (because it is not popular)

Effectively, sampling has automatically reduced your data massively by removing the majority (99.9%) of the bad queries 🙂

Recency Factor

Problem Statement

Suppose there's a big event that happens

- COVID lockdown gets declared
- Election results are out
- Some high profile person got arrested
- Federer won the Wimbledon
- ...

This is a trending event.

Even though `"who has control over nukes?"` might have a higher overall count, but since the query `"who won the Wimbledon?"` is trending *(recent + popular)*, we should rank that higher when someone types `"who"`

Goal

give more weightage to the recent counts

Solution

Simply decay the historical counts after every fixed period of time.
After each day, decrease the total count for each query by 10%

```
"who has control over nukes?"
[1000, 1000, 1000, 1000, ...]   => total is 100,000
day 1: 1000
day 2: 90% of 1000 + 1000 => 1900
day 3: 90% of 1900 + 1000 => 2710
day 4: 90% of 2710 + 1000 => 3439
...
day 100:  => 1000 + 1000 * 0.9 + 1000 * 0.9^2 + ...
          => 1000 * (1 + 0.9 + 0.9^2 + 0.9^3 + ...)
          => 1000 * 1 / (1 - 0.9)
```

```
          => 10,000

"who won the wimbledon"
[0, 0, 0, ... , 5000, 5000, 5000] => total is just 15000
day 997: 0
day 998: 5000
day 999: 5000 * 0.9 + 5000 => 9,500
day 1000: 9500 * 0.9 + 5000 => 13,550
```

If the count after decay reduces below a threshold (say 0) then we can remove that entry.

Geolocation based personalization

Given the user's request, find out their location based on ip address.
Build a separate database for each location => shard the DB by country_id
Now an Indian user's request will only go to the Indian shard.

User based personalization

Do this purely on the client side.
The browser stores the user's search/browsing history.

1. browser will create typeahead suggestions from the user's local data
2. browser will initiate a backend request to get the global typeahead suggestions
3. browser will merge these two lists

Handling Typos — Spelling correction

You can simply have a dictionary and you can calculate the “edit-distance” b/w the user’s words and the words in the dictionary.
Peter Norvig — How to Write a Spelling Corrector

Resources

Reference Book	"Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems" by Martin Kleppmann

Good Resources	1. Alex Xu: https://www.youtube.com/@ByteByteGo 2. Arpit Bhayani: https://www.youtube.com/@AsliEngineering 3. Hussein Nasser: https://www.youtube.com/@hnasr 4. Martin Kleppmann: https://www.youtube.com/@kleppmann 5. Microservices Reference: https://microservices.io/ 6. Engineering blogs by companies - Amazon: https://aws.amazon.com/blogs/architecture/ - Meta: https://engineering.fb.com - Netflix: https://netflixtechblog.com
Resources to avoid	- Medium articles by random authors (go to engineering blogs of companies instead of engineering blogs of people) - geeksforgeeks (poor quality) - random youtube videos by *bhaiyas* and *didis*
Drawing HLD Diagrams	- I prefer https://excalidraw.com/ - A lot of companies use https://draw.io

Math Prerequisites (bite-sized lessons)	- powers: https://www.khanacademy.org/math/algebra-basics/alg-basics-expressions-with-exponents - logarithms: https://www.khanacademy.org/math/algebra2/x2ec2f6f830c9fb89:logs - sequences / series - https://www.khanacademy.org/math/algebra/x2f8bb11595b61c86:sequences - https://www.khanacademy.org/math/precalculus/x9e81a4f98389efdf:series - combinatorics *(not needed, but good to know)*: https://www.khanacademy.org/math/statistics-probability/counting-permutations-and-combinations
CS Fundas Prerequisites (Full length courses)	*Please note that these are full-length courses. It will take a long time to complete & master these subjects* 😁 *It is not recommended to go through these while you're doing an interview driven prep. However, these courses should be in your to-do list in the long run* 1. Operating Systems - UC Berkley: https://www.youtube.com/playlist?list=PLF2K2xZjNEf97A_uBCwEl61sdxWVP7VWC - Scaler: https://www.scaler.com/topics/course/free-operating-system-course/ 2. Computer Networks - Northwestern University: https://www.youtube.com/playlist?list=PLWl7jvxH18r3nnotitKkyAjq268PQGc0- - Scaler: https://www.scaler.com/topics/course/free-computer-networks-course/ 3. Databases - CMU: https://www.youtube.com/playlist?list=PLSE8ODhjZXjbj8BMuIrRcacnQh20hmY9g - Scaler: https://www.scaler.com/topics/course/sql-using-mysql-course/ 4. Distributed Systems: https://www.youtube.com/playlist?list=PLrw6a1wE39_tb2fErI4-WkMbsvGQk9_UB

QnA