



# Lecture 11 — OAuth 2.0 & Search AutoComplete Engine



High Level System Design · Continuous Lecture Notes

---



## OAuth 2.0 — First Principle Thinking

Sabse pehle hum ek **modern authentication mechanism** ko samajhte hain jo aaj almost **har real-world product** me use hota hai.



### Core Question

Aaj hum jab kisi website par jaate hain aur dekhte hain:

- “Sign in with Google”
- “Continue with Facebook”

to ye kaam **kaise** hota hai?



Is process ko **OAuth 2.0** kehte hain.

---



## Traditional Sign-Up vs OAuth



### Traditional Sign-Up

User manually deta hai:

- Username
- Password
- Email
- Mobile



Har website ke paas **user ka password store** hota hai  
(Security risk + user fatigue)

---



### OAuth Approach

User bolta hai:

“Main apni Google identity use karunga”



App **password kabhi nahi dekhti**



Google **trust anchor** ban jaata hai

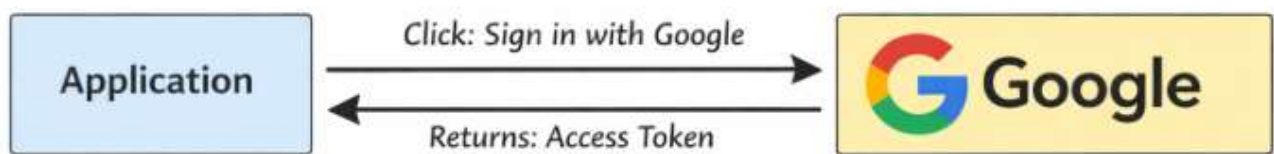
---

## 🧠 OAuth 2.0 — Exact Meaning

OAuth ek authorization framework hai  
jisme ek trusted platform (Google)  
limited access deta hai kisi third-party application ko  
**without sharing a password.**

---

## 🚩 OAuth 2.0 Mechanism



## 🔑 Flow Step-by-Step

- 1 User clicks “**Sign in with Google**”
  - 2 Application Google ke OAuth server par redirect karti hai
  - 3 User Google par **Allow / Deny** karta hai
  - 4 Agar allow:
    - Google app ko **Access Token** deta hai
  - 5 App token ka use karke limited user info leti hai
- 

## 📦 Access Token Ke Andar Kya Hota Hai?

Access Token ke through app ko milta hai:

```
{ email, mobile }
```

### ⚠️ IMPORTANT

- Password **kabhi share nahi hota**
  - Token **limited scope** ka hota hai
  - Google trust boundary ke andar rehta hai
-

## Why OAuth Is Powerful

- User ko baar-baar signup nahi karna
- App ko password handle nahi karna
- Security centralized ho jaati hai
- Trust delegation hota hai

### Industry Standard Authentication

---

## Transition — Auth se Search System

OAuth samajhne ke baad  
ab hum lecture ke **main HLD problem** par aate hain:

---

## Design Search AutoComplete Engine

(Design Top-K Most Searched Queries)

---

## Problem Statement

Jab user Google search bar me type karta hai:

sw


to system turant suggest karta hai:

swing → 1000

swiggy → 800

swan → 700

swim → ...

 Ye suggestions:

- **Instant** aane chahiye
  - **Popularity (frequency)** ke basis par hone chahiye
-

## Performance Goal

 **Response Time:  $O(1)$**

(User ke typing ke saath suggestions change honi chahiye)

---

## Requirements & Constraints (Q&A)

Question	Answer
How many suggestions?	<b>Top 6</b>
How to decide the ranking?	<b>Frequency</b>
Max query length?	<b>100 chars</b>
Language?	<b>English</b>
Sorting?	<b>Yes (High <math>\rightarrow</math> Low)</b>
Case sensitive?	<b>No</b>
DAU?	<b>5 Million</b>

 **Constraints decide architecture**

---

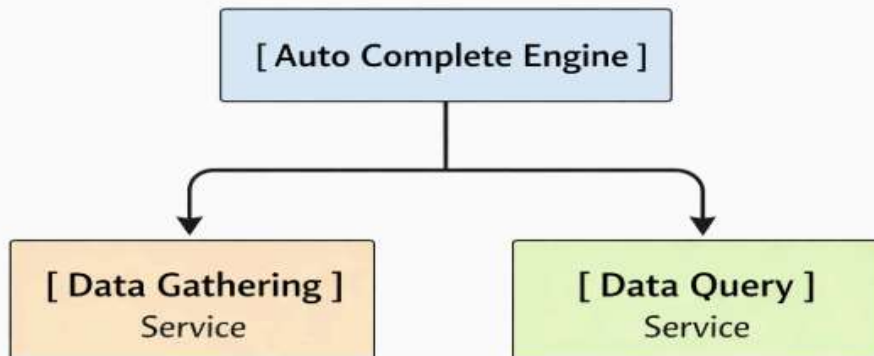
## Yahan Tak Lock Kar Lo

- OAuth = secure delegated login
  - Token = limited, scoped access
  - Autocomplete = **read heavy + low latency system**
  - Requirements clear  $\rightarrow$  design clear
-



## High Level Architecture — First Cut

System ko logically **do independent services** me divide kiya jata hai:



### Why split?

- **Write path** (users kya search kar rahe hain)  $\neq$  **Read path** (suggestions)
- Read traffic bahut zyada hota hai (DAU = 5M)
- Read ko **ultra-fast** banana hai



## Data Gathering Service — Write Path

### Responsibility:

Users ke actual search queries collect karna.

- Source: Search bar analytics logs
- Nature: **Append-only**, not indexed
- Real-time update **✗** (mehenga + unnecessary)



Yeh service **accuracy + completeness** par focus karti hai, speed par nahi.



## Data Query Service — Read Path

### Responsibility:

Prefix diya gaya ho to **Top-6 most searched queries** return karna.

- Read heavy
- Low latency
- User ke typing ke saath response

👉 Yeh service **speed + caching** par focus karti hai.

---

## **Solution 1 — SQL Database (Naive Approach)**

### **Frequency Table (SQL)**

Query	Frequency
-------	-----------

switch	2
--------	---

swing	4
-------	---

swim	7
------	---

swirl	3
-------	---

swan	6
------	---



---

### **Query for prefix = sw**

```
SELECT * FROM frequency
WHERE query LIKE 'sw%'
ORDER BY frequency DESC
LIMIT 6;
```

---

### **First-Principle Analysis**

- Logic  correct
  - Result  correct
-

## ⚠ Performance Reality

- Small dataset → OK
- Millions of rows → ❌ **Very slow**

Why?

- **LIKE 'prefix%'** → full scan
- **ORDER BY** → sorting huge result set

👉 Real-time autocomplete ke liye NOT feasible

---

## 🌳 Solution 2 — Tries (Optimized Approach)

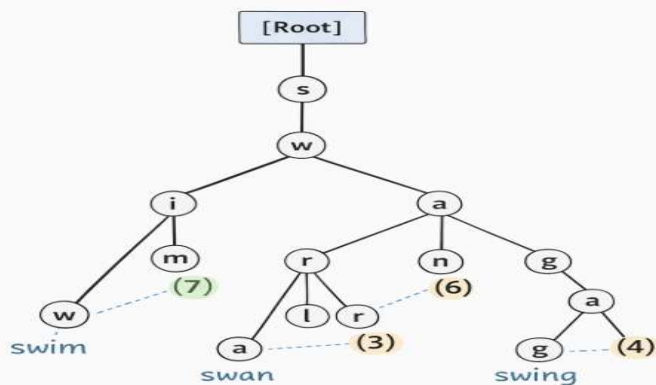
### 🧠 Why Trie?

- Prefix based search ke liye **perfect data structure**
  - English alphabet size = **26**
- 

## 📐 Trie Structure (Figure)

Example data:

- swim (7)
- swing (4)
- swan (6)
- swirl (3)



---

## Trie Node Structure

```
struct TrieNode {  
    TrieNode* children[26];  
  
    bool eow;    // end of word  
  
    int freq;    // frequency  
};
```

TrieNode	
children[26]	
eow	endOfWord
freq	

---

## Problem with Normal Tries

Top-K (K=6) queries chahiye. Steps:

- 1 Prefix find →  $O(p)$
- 2 Subtree traverse →  $O(c)$
- 3 Sort by freq →  $O(c \log c)$

### Worst Case

User types just "s"

→ almost **entire trie traverse**

👉 Still **slow** for instant UX.

---

## Optimization — Caching (Make it $O(1)$ )

### Goal

Prefix mile → **direct answer**, no traversal.

---

## 🧠 Key Observations

- Max query length = **100**
- Users rarely type very long strings
- **K = 6 (small & constant)**

## 🔺 Optimized Trie with Cache

Har trie node apne paas **Top-6 results cache** karta hai.

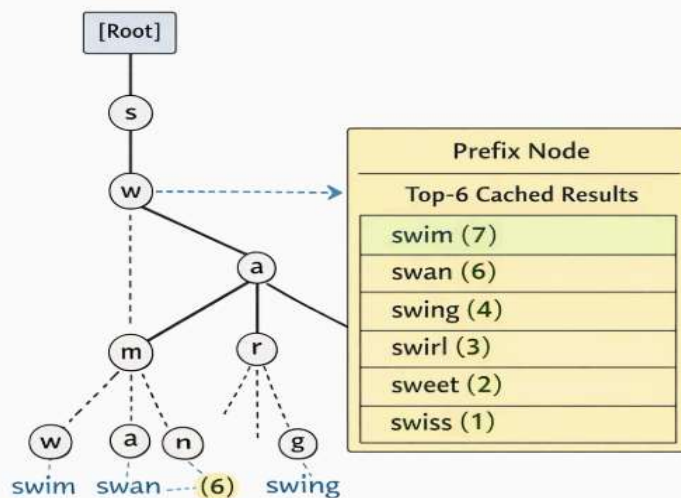
s

|

w ----> [ Cache: swim(7), swan(6), swing(4), swirl(3) ]

/ \

i a



## 🕒 New Complexity

- Find prefix → **O(1)**
- Read cache → **O(1)**
- Already sorted → **O(1)**

✅ **Total =  $O(1)$**

⚠️ **Trade-off:**

Space ↑ to reduce Time ↓

---

## 🧠 **Yahan Tak Lock Kar Lo**

- SQL → correct but slow
  - Trie → correct & fast
  - Cached Trie → **industry-grade solution**
  - Autocomplete = **read-optimized system**
- 

## 🔄 **Data Gathering Architecture — How Cache Is Built**

Autocomplete engine ka sabse important question:

**Data aata kahan se hai aur cache kaise banti hai?**

---

## 🕒 **Real-Time vs Almost Real-Time**

### ❌ **Real-Time Prediction (Not Practical)**

- Har keypress par Trie update ❌
- Massive write load ❌
- Cache constantly invalid ❌

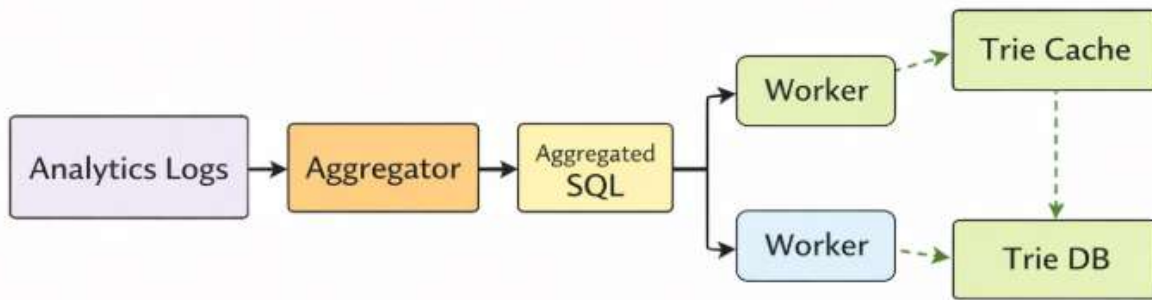
### ✅ **Almost Real-Time (Industry Choice)**

- Suggestions frequently change nahi hoti
- Weekly / Daily update sufficient
- Stability + performance balance

👉 **Google-scale systems Almost-Real-Time follow karte hain**

---

## 🧠 The Data Pipeline (End-to-End)



### ✿ Step-1: Analytics Logs (Raw Data)

- Append-only table
- No indexing
- Sirf user behavior capture hota hai

Example:

swing (Aug 11 2025 22:48:03)

swan (Aug 11 2025 22:48:05)

swing (Aug 11 2025 22:48:34)

👉 Yeh data **directly user-facing nahi** hota

---

### ✿ Step-2: Aggregator Service

- Raw logs ko process karta hai
- Format normalize karta hai
- Frequency count banata hai

#### 🕒 Frequency of run:

- Normal system → Weekly
  - More fresh system → Daily
-

### ✖ Step-3: Aggregated SQL Table

Query	Time	Freq
-------	------	------

swing	–	2
-------	---	---

swim	–	4
------	---	---

👉 Ab data **ready** hai **Trie** banane ke liye

---

### ✖ Step-4: Workers (Async Builders)

- Multiple worker servers
- Aggregated data se Trie build karte hain
- CPU-heavy task (offline)

👉 Users par koi latency impact nahi

---

### ✖ Step-5: Storage Layer

#### 🧠 Trie Cache (In-Memory)

- Fastest reads
- Weekly / Daily updated
- User-facing requests yahin hit karti hain

#### 📁 Trie DB (Persistent Storage)

- NoSQL (MongoDB / Cassandra)
  - Crash ke baad recovery ke liye
  - Source of truth
- 

## 🔒 Serialization — Trie to DB

Trie ko directly DB me store nahi kar sakte

👉 isliye **serialize** karte hain.

## Trie Serialization (Key–Value)

K (Prefix)	V (Top Results)
sw →	swim, swan, swing
swi →	swing, swim

### 🔑 Key-Value Mapping

Key (Prefix)    Value (Top-6 Results)

sw                swim, swan, swing, swirl

swi               swing, swirl, swim

swim              swim

👉 Har prefix = ek DB key

---

## 📊 Back of the Envelope — First Run

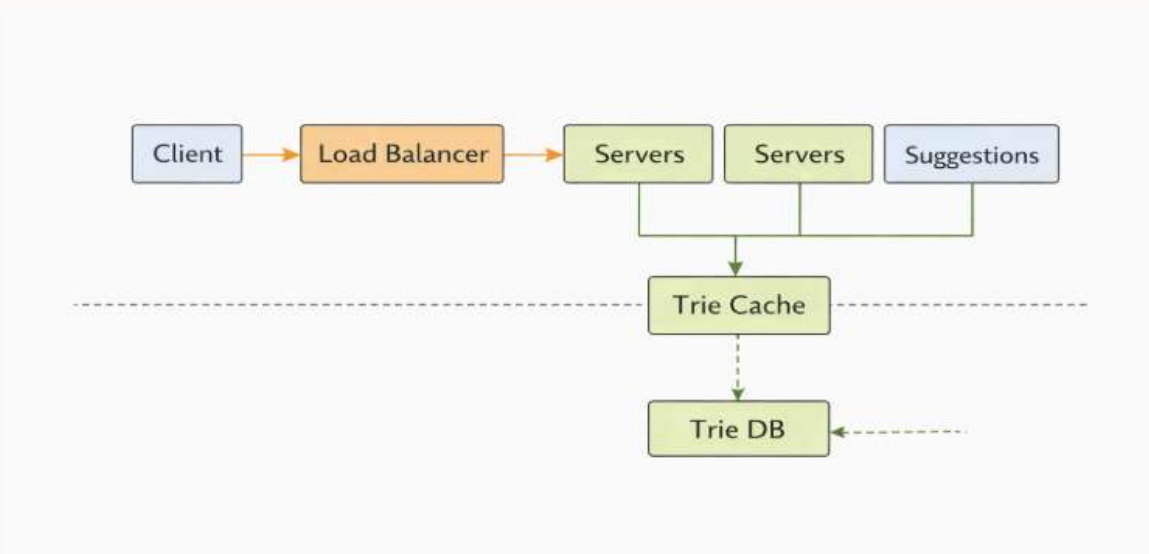
Agar system **scratch se start** ho:

- 1 lakh analytics logs / week
- Aggregator process karega
- ~10,000 unique queries milengi
- Trie build hoga
- Trie Cache + Trie DB me push

👉 One-time heavy cost, baad me smooth reads

---

## Client Request Flow



[Client]

|

v

[ LB ]

|

v

[Servers] -----> /search?query=sw

|

|

[Trie Cache] <-----> [Trie DB]

### Runtime Logic

- 1 Request LB par aati hai
- 2 Server Trie Cache check karta hai
- 3 Cache hit → Direct response
- 4 Cache miss → DB se load → cache replenish

👉 User ko hamesha  $O(1)$  response

---

## ⚡ Further Optimizations

### 🌐 Client-Side Optimization (AJAX)

- ❌ Page refresh on every keypress
- ✅ AJAX calls without refresh

/search?query=sw

/search?query=swi

👉 Smooth UX + less server load

---

### 🧠 Browser-Side Caching

- Same prefix repeat ho raha hai?
- Browser cache se serve karo

👉 Network calls kam ho jaati hain

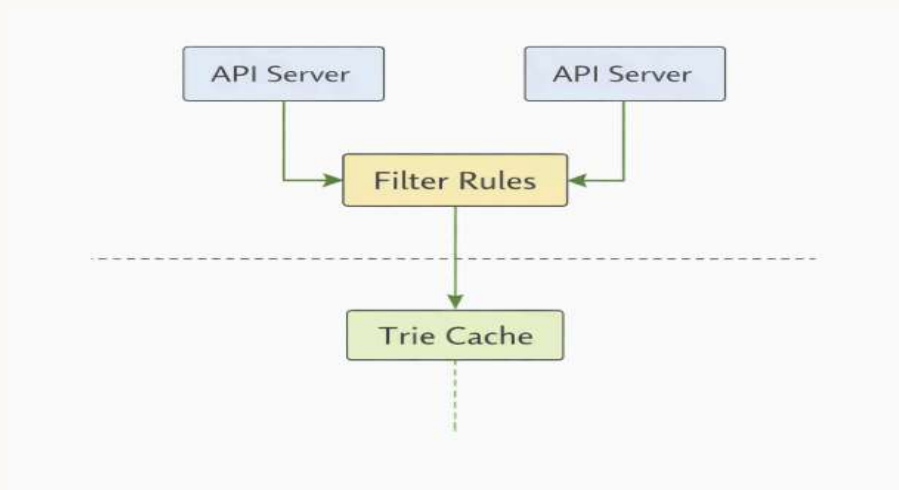
---

### 🚫 Filtering Layer

Offensive / abusive words aa sakte hain.

Architecture:

[API Servers] → [Filter Rules] → [Trie Cache]

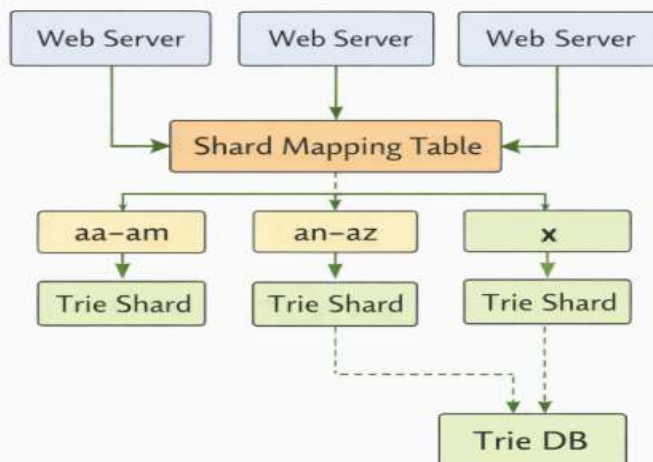


## DB Scaling — Sharding

### Naive Sharding

- a → heavy load
- x → very less load

### Shard Mapping Table



- 👉 Heavy alphabets ko further split karo
- 👉 Light alphabets single server par

## Multi-Language & Geography

### Multiple Languages

- Hinglish
- Unicode support (global encoding standard)

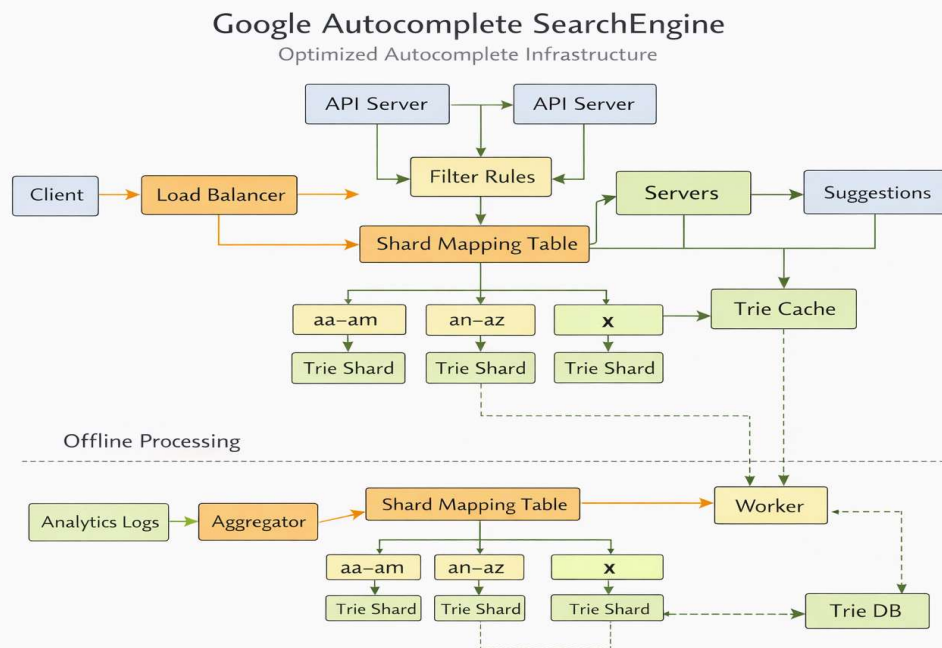
### Geography-Aware Results

- Country-specific trending queries
- Regional Trie Cache

- Faster global response
- Edge-level caching

## Final Lock — What This Lecture Teaches

- OAuth = secure delegated authentication
- Autocomplete = **read-heavy, low-latency system**
- Trie + Cache =  $O(1)$  UX
- Almost-real-time > Real-time
- Space–Time trade-off = conscious decision



## OAuth 2.0 — One-Line Truth

OAuth 2.0 ek secure authorization framework hai jisme user apna password share kiye bina kisi trusted platform (Google) ke through third-party app ko limited access deta hai.

## 🔑 Key Takeaways

- Password **kabhi share nahi hota**
- Access **token-based** hota hai
- Token ke andar **limited info + scope**
- Google / Facebook = **Trust Authority**

## 👉 Industry-standard login mechanism

---

## 🔍 Search Autocomplete — Core Problem

User type karta hai →  
system ko **real-time me top-K (K=6)** suggestions dene hain  
based on **popularity (frequency)**

### 🎯 Goal:

- Ultra-fast response
  - Typing ke saath suggestions
  - Time Complexity  $\approx O(1)$
- 

## 🏠 High-Level Design Philosophy

System ko logically **do alag concerns** me todna:

1 Data Gathering (Write Heavy)

2 Data Query (Read Heavy)

👉 Read aur write ko separate karna = **scalable design**

---

## ❌ SQL Naive Approach — Why It Fails

- `LIKE 'prefix%'` → full table scan
- `ORDER BY frequency` → heavy sorting
- Millions of rows → ❌ slow

👉 Correct logic ≠ correct system

---

## Trie Data Structure — Right Tool

### Why Trie?

- Prefix-based search ke liye perfect
- Alphabet size fixed (26)
- Search predictable

### Problem:

- Prefix find  $\rightarrow O(p)$
- Subtree traversal  $\rightarrow O(c)$
- Sorting  $\rightarrow O(c \log c)$

👉 Still **not fast enough**

---

## Cached Trie — Industry Grade Solution

### Core Optimization

- Har Trie node par **Top-6 results** cache
- Prefix length limited ( $\leq 100$ )
- $K = \text{constant } (6)$

### Final Complexity

Prefix lookup  $\rightarrow O(1)$

Cache read  $\rightarrow O(1)$

Sorting  $\rightarrow O(1)$

✅ **Total =  $O(1)$**

👉 Space badhao, time girao = **Conscious trade-off**

---

## Data Pipeline — How System Lives

- 1 **Analytics Logs** (raw, append-only)
- 2 **Aggregator Service** (weekly/daily)
- 3 **Aggregated SQL Table**
- 4 **Workers** (async Trie builders)
- 5 **Trie Cache (RAM) + Trie DB (NoSQL)**

👉 Users par **zero impact**, background me heavy kaam

---

## Trie Serialization — DB Ready Design

- Trie ko **key-value** format me store
- **Key = prefix**
- **Value = top-6 results**

Example:

sw → swim, swan, swing

swi → swim, swing

👉 Fast recovery + scalable storage

---

## Client Request Flow (Runtime)

- 1 Client → LB
- 2 Server → Trie Cache
- 3 Cache hit → return instantly
- 4 Cache miss → DB → cache refill

👉 User ko hamesha fast response

---

## Optimizations — Production Ready Touch

### Client Side

- AJAX calls (no page refresh)
- Smooth typing experience

## Browser Cache

- Same prefix → cached result
- Server load kam

## Filtering

- Offensive words filter
  - Rules before Trie cache
- 

## Scaling Strategy (DB Sharding)

✗ Alphabet-based naive sharding (a heavy, x light)

### ✓ Shard Mapping Table

- Heavy prefixes → further split
- Light prefixes → single node

👉 **Balanced load, scalable growth**

---

## Global System Considerations

- **Multiple languages** → Unicode support
  - **Geography aware results**
  - **Country-specific trending queries**
  - **CDN** for faster global response
- 

## Final Lock — What You Actually Learned

- OAuth = **secure delegated authentication**
- Autocomplete = **read-heavy, low-latency problem**
- Trie + Cache = **industry standard**
- Real-time ≠ practical, **Almost-real-time wins**
- HLD = **trade-offs + clarity**

---

## Interview Gold Lines (Yaad Rakho)

- “We separate read and write paths.”
- “We cache top-K results at each Trie node.”
- “Space-time trade-off gives us  $O(1)$  latency.”
- “Almost real-time is chosen intentionally.”

 Bol diya = **impression locked**

---

---

 **Made By @Harshal Chauhan** 

---