Designing Search Autocomplete

High-Level Overview: Autocomplete System

Autocomplete has **two main parts**:

1) Data Gathering Service (Offline / Backend)
2) Data Query Service (Online / Real-time)

1.Data Gathering Service (Offline Pipeline)

Purpose

Collect and prepare data for autocomplete suggestions.

What Data is Collected?

- User search queries
- Click logs
- Trending topics
- Popular products
- Dictionary words
- User behavior logs

Example:

"iphone 15"

"iphone charger"

"iphone case"

Steps in Data Gathering

Step 1: Logging

Store all user searches and clicks.

Step 2: Cleaning

Remove:

- Typos
- Spam
- Rare useless queries

Step 3: Ranking / Scoring

Assign popularity score:

score = frequency + recency + CTR

Step 4: Build Index

Store in fast data structure:

- Trie
- Sorted List
- Elasticsearch Index
- Vector DB

Output of Data Gathering

A **prepared autocomplete index** like:

app → apple (1000)

app → application (800)

app → app store (500)

2.Data Query Service (Real-Time System)

Purpose

Return suggestions when user types.

When User Types:

a → ap → app

Steps in Query Service

Step 1: Receive Prefix

Example:

prefix = "app"

Step 2: Search Index

Use Trie / Search Engine to find prefix matches.

Step 3: Rank Results

Sort by:

- Popularity
- Personalization
- Freshness

Step 4: Return Top K Results

Return top 5 or 10 suggestions in **< 50 ms**.

High-Level Architecture Diagram (Explain Like This)

Users → API Gateway → Query Service → Autocomplete Index

↑

Data Gathering Pipeline

↑

Logs / Analytics / Batch Jobs

Key Differences

Feature	Data Gathering Service	Data Query Service
Type	Offline / Batch	Online / Real-time
Speed	Not critical	Must be ultra fast
Tech	Hadoop, Spark, Kafka	Trie, Redis, Elasticsearch
Updates	Hourly/Daily	Instant lookup

Autocomplete system has an offline data gathering pipeline that collects and ranks query data, and an online query service that retrieves top prefix matches with low latency using indexes like Trie or search engines.

Data Gathering:

- Batch processing
- Feature engineering
- Popularity scoring

- Index building

Query Service:

- Low latency (< 50 ms)
- Caching
- Trie or Elasticsearch
- Personalization layer

Approach 1: Use SQL Database for Autocomplete Query Service

Store all words/queries in a SQL table and use **LIKE / prefix search**.

Example SQL Table

```
CREATE TABLE suggestions (
    id INT,
    query VARCHAR(255),
    score INT
);
```

Query Example

```
SELECT query
FROM suggestions
WHERE query LIKE 'app%'
ORDER BY score DESC
LIMIT 10;
```

Advantages of Using SQL DB

1.Simple and Easy to Implement

- Everyone knows SQL
- No special data structure needed

Good for MVP or small systems.

2.Strong Consistency

- ACID properties
- Data always correct

3.Easy Ranking

You can sort by:

ORDER BY score DESC

4.Easy Updates

Insert / delete / update queries easily.

5.Mature Ecosystem

- Indexing
- Replication
- Backup
- Security

Disadvantages of Using SQL DB

1.Very Slow for Large Scale

For millions of queries:

LIKE 'app%'

Full table scan or heavy index scan
Slow per keystroke

2.Not Designed for Prefix Search

SQL indexes are for exact match, not prefix.

Special indexes needed → still slow.

3.High Latency

Autocomplete needs **< 50 ms**.
SQL may take **100–500 ms+**.

4.Poor Scalability

Hard to scale to:

- Billions of queries
- Millions of users typing simultaneously

5.No Built-in Text Ranking

Google-style ranking needs ML + search engines.
SQL is basic.

6.Memory Inefficient

Trie/search index stores compact prefix data.
SQL stores full strings → more storage.

Time Complexity

Operation	Complexity
Prefix Search	O(N) (worst case)
With Index	Still heavy

Compare with Trie: **O(L)** (very fast)

When SQL Approach is Good

Small dataset
Internal tools
Interview brute-force solution
MVP product

When SQL is Bad

Google/Amazon scale
Real-time autocomplete
High QPS systems

Using SQL DB for autocomplete is simple and consistent, but prefix search is slow at scale, has high latency, and does not scale well compared to Trie or search engines.

Approach 2: Using Trie for Autocomplete Query Service

Store all words in a **Prefix Tree (Trie)** so we can find suggestions in **O(L)** time.

What is a Trie?

Trie is a tree where:

- Each node = one character
- Path from root → word
- All words with same prefix share nodes

Example

Words:

apple, app, application

Trie:

```
      a
      |
      p
      |
      p
    /   \
   l     (end)
   |
   e
```

Query Example

User types:

prefix = "app"

Steps:

1. Go to node 'a' → 'p' → 'p'
2. From that node, DFS to find all words
3. Return top K results

Time Complexity

Operation	Complexity
Insert	O(L)
Search Prefix	O(L)
Get Suggestions	O(K * L)

Very fast.

Advantages of Trie

1.Very Fast Prefix Search

No scanning database.
Direct tree traversal.

2.Low Latency

Perfect for autocomplete (<10ms).

3.Shared Prefix Saves Space

"app", "apple", "application" share nodes.

4.Easy to Rank

Store score at nodes.

Problems with Normal Trie (IMPORTANT)

1.Huge Memory Usage

Each node has:

26 pointers (a-z)

For millions of words:

BILLIONS of nodes
Huge RAM consumption

2.Poor Cache Locality

Trie nodes are scattered in memory → cache misses → slower in practice.

3.Hard to Scale in Distributed Systems

Trie is pointer-heavy structure.

Hard to shard across machines.

4.Difficult Updates at Scale

Updating popularity scores in many nodes is complex.

5.Not Good for Disk Storage

Trie needs RAM.
Disk-based Trie is slow.

6.DFS Cost for Suggestions

After prefix match, DFS traversal can be expensive if many words share prefix.

Example:

"app" → 1 million words

7.Not Language Friendly

For Unicode (Hindi, Chinese):

- Alphabet size huge
- Node pointers explode

Trie is memory inefficient and hard to scale for large datasets.

This is the key drawback.

Real Companies Use Optimized Tries

Instead of normal Trie:

- Compressed Trie (Radix Tree)
- Ternary Search Tree
- DAWG
- Finite State Transducers (FST)

Trie provides fast O(L) prefix search but is memory-heavy, hard to scale, and inefficient for large datasets without compression.

Cache-Based Autocomplete

Most users type **same prefixes** like:

a, ap, app, iph, iphone

So we **save these results in cache (Redis)**.

Architecture

User → Cache (Redis) → Trie / DB

How It Works

1.User types: "app"

Step 1: Check Cache

Redis.get("app")

If Found:

Return suggestions instantly (1 ms)

2.If Not in Cache

- Search Trie / Search Engine
- Get top suggestions
- Store in cache

Redis.set("app", ["apple", "app store", "application"])

Why Cache is IMPORTANT

Very Fast

Redis = in-memory → <1 ms latency

Reduces Load

Trie / DB not called every time.

Scales Easily

Millions of users → cache handles traffic.

Cache Problems

- Memory expensive
- Need eviction (LRU)
- Data can be stale

We use Redis cache for hot prefixes to reduce latency and backend load, falling back to Trie or search index on cache miss.

Without cache:
Every user hits Trie (slow)

With cache:
Most users get instant results from RAM

Cache only:

- **Top 1% most popular prefixes**
- **Not all prefixes**

What is Data Gathering Service?

It collects:

- User searches
- Clicks
- Popular queries
- Trending topics

Then it updates autocomplete data.

Two Ways to Update Data

1.Real-Time System (Instant Update)

Update cache **immediately when user searches**.

Flow

User search → Kafka → Stream Processor → Redis / Trie Update

Example:

User types:

"iphone 16"

Immediately:

- Increase score in Redis
- Add to autocomplete cache

Advantages

- Fresh trending suggestions
- Perfect for breaking news / trending products

Disadvantages

- Complex system
- High cost
- Concurrency issues
- Hard to scale

2.Almost Real-Time System (Batch Updates)

Update autocomplete **every few minutes or hours**.

Flow

Logs → S3 / HDFS → Spark Batch Job → Build Trie → Deploy

Example:

- Every 5 min / 1 hour update popular queries

Advantages

- Simple
- Cheap
- Stable
- No race conditions

Disadvantages

- Slight delay in trends
- New query appears after some time

How to Cache Data Gathering Results

Cache Layer (Redis)

Store:

"app" → ["apple", "app store", "application"]

Strategy 1: Write-Through Cache

When data pipeline updates index → also update Redis.

Strategy 2: Cache Refresh Job

Cron job rebuilds cache every 5 min.

Strategy 3: Hot Prefix Cache

Only cache:

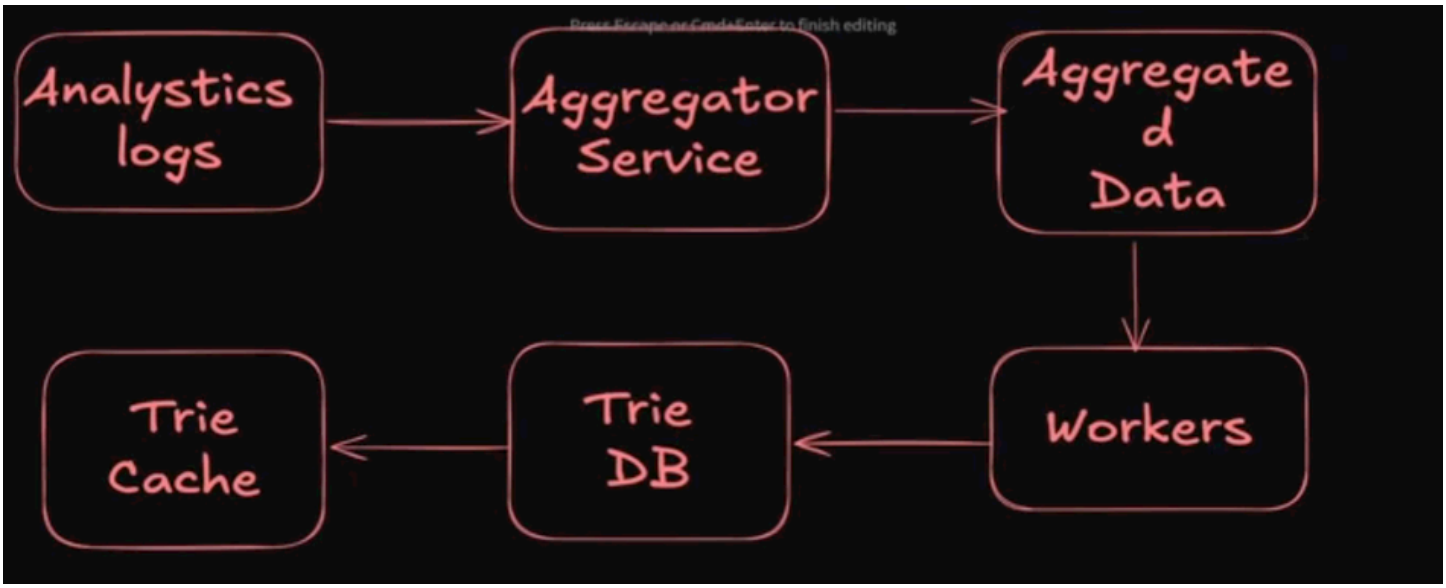
- Most popular prefixes
- Trending queries

Real-Time vs Almost Real-Time

Feature	Real-Time	Almost Real-Time
Update Delay	ms–sec	min–hours
Complexity	High	Low
Cost	High	Low
Use Case	Twitter trends	Google autocomplete

We can update autocomplete data in real-time using streaming pipelines like Kafka, but most systems use almost real-time batch jobs to reduce cost and complexity while keeping data reasonably fresh.

- **Google: almost real-time (minutes delay)**
- **Amazon: near real-time for products**
- **Twitter: real-time trending**



What This Diagram Shows

Autocomplete Data Gathering + Index Building Pipeline

This is **offline / backend pipeline** that prepares data for Trie & Cache.

Analytics Logs → Aggregator → Aggregated Data → Workers → Trie DB → Trie Cache

1.Analytics Logs

What is it?

Raw user activity logs:

- What users searched
- What they clicked
- What they typed
- Time, location, device

Example Logs:

"iphone 15"

"iphone case"

"app store"

"amazon prime"

This data is HUGE and dirty.

2.Aggregator Service

What is it?

A service that **cleans and summarizes logs**.

What it does:

1.Count frequency

"iphone" searched 1M times

"iphone case" 200K times

2.Remove noise

- Typos
- Spam queries
- Rare useless queries

3.Compute Scores

score = frequency + recency + CTR

3.Aggregated Data

What is it?

Final clean dataset with popularity score.

Example Output:

Query	Score
iphone	1000000
iphone case	200000
app store	900000

This is input for Trie building.

4.Workers

What are Workers?

Background processes that:

- Build Trie
- Update DB
- Push to cache

What they do:

1.Build Trie Index

Insert queries into Trie with score.

2.Shard Trie

Split data:

a-m → Worker 1

n-z → Worker 2

3.Batch Update

Run every:

- 5 min
- 1 hour
- 1 day

5.Trie DB

What is Trie DB?

Persistent storage of Trie index.

Why DB?

- Trie too big for RAM
- Need backup
- Need versioning

Example:

- RocksDB

- Cassandra
- Custom key-value store
- Disk-based FST

6.Trie Cache

What is Trie Cache?

Fast in-memory cache for popular prefixes.

Example:

Redis:

"app" → ["apple", "app store", "application"]

"iph" → ["iphone", "iphone 15"]

Query service reads from here first.

How Query Works with This Pipeline

User types: "app"

1. Check Trie Cache (Redis)
2. If miss → Trie DB lookup
3. Return suggestions
4. Cache result

Real-Time vs Batch in This Diagram

This pipeline is **Almost Real-Time**

- Runs every few minutes/hours
- Not per user search

Cheaper and scalable.

User logs are collected in analytics, aggregated to compute popularity scores, workers build and update the Trie index in a database, and hot prefixes are pushed to an in-memory cache for low-latency autocomplete queries.

- **Analytics logs = Raw customer feedback**
- **Aggregator = Accountant counting popular items**
- **Workers = Factory building dictionary**
- **Trie DB = Warehouse**

- **Cache = Shop counter with best items**

Important

- **Batch jobs (Spark/Flink)**
- **Kafka for logs**
- **Redis for cache**
- **Trie/FST for index**
- **Sharding workers**
- **Blue-green deployment for Trie updates**

Normal HTTP Request

- The browser reloads the whole page.
- Server sends full HTML page again.
- It is **slow** and uses more data.

Example: You search something → page refreshes.

AJAX Request

- The page **does NOT reload**.
- Browser asks server only for **small data (JSON)**.
- Page updates in the background.
- It is **fast** and smooth.

Example: Google search suggestions appear without page refresh.

- **HTTP request** = full page reload
- **AJAX request** = only data, no reload

We use AJAX instead of normal HTTP requests to avoid full page reload and fetch only required data, which improves speed and user experience.