HLD Case study 2 (Typeahead)

Contents

[System Design Process 2](#_Toc184217474)

[Functional Requirements for Google Autocomplete System 5](#_Toc184217475)

[Non-Functional Requirements for Google Autocomplete System 7](#_Toc184217476)

[Back of the Envelope Estimation 8](#_Toc184217477)

[Read Heavy/Write Heavy 9](#_Toc184217478)

[APIs for Auto Suggest System 12](#_Toc184217479)

[Trie for Autocomplete with Top-K Suggestions 13](#_Toc184217480)

[Updating Search Term Frequency 15](#_Toc184217481)

[Optimizing Read-Write, Sharding 17](#_Toc184217482)

[Recency with Time Decay Factor 20](#_Toc184217483)

## System Design Process

**Overview of System Design Process**

The system design process can be broken into **four key steps**, each with specific objectives and deliverables.

**Step 1: Understanding Requirements**

1. **Requirement Types**:
   * **Functional Requirements**: Define the features and behavior of the system.  
     Examples:
     + Support for group chat or one-on-one messaging.
     + Maximum number of users accessing the system concurrently.
   * **Non-Functional Requirements**: Define performance characteristics.  
     Examples:
     + Consistency vs. availability.
     + Acceptable latency, reliability, and scalability.
2. **Define the Minimum Viable Product (MVP)**:
   * Identify the minimum set of features required to make the product functional for users.
   * Avoid overengineering by focusing on critical features.
3. **Engage with Stakeholders**:
   * Discuss requirements with the interviewer (or stakeholders) to refine scope and prioritize features.

**Step 2: Estimation of Scale**

1. **Key Objectives**:
   * Determine if **sharding** is necessary based on data volume and access patterns.
   * Understand if the system is **read-heavy**, **write-heavy**, or balanced.
   * Perform basic **capacity estimation**:
     + **Machines needed** for the database and application servers.
     + Number of **shards** required.
     + Number of **load balancers** to handle expected traffic.
2. **Practical Applications**:
   * Capacity estimation aids in provisioning resources and budgeting during project planning.

**Step 3: Identifying Key Use Cases and APIs**

1. **Understand External System Interactions**:
   * List the critical use cases that the system must support.
   * Examples: Frequently called APIs, most common operations.
2. **Use Case-Driven Design**:
   * Design choices (e.g., sharding keys, caching mechanisms) should prioritize the most frequent or critical use cases.
3. **Focus on Core Service**:
   * Initially, focus on one major use case or service.
4. **Consider Larger Systems**:
   * In complex systems like Facebook or Flipkart, multiple use cases are supported through **microservices**.
   * Example: Facebook has separate services for news feeds, messaging, pages, etc.

**Step 4: Final Design**

1. **High-Level Architecture**:
   * Define the overall architecture, components, and their interactions.
2. **Specific Features**:
   * Incorporate functional and non-functional requirements into the design.
3. **Schema Design (If needed)**:
   * If required, design schemas by identifying entities and relationships.
4. **Design for Scalability**:
   * Ensure the design supports scaling up for increased usage or data volume.

**Google Search Autocomplete Design**

**Overview**

This section focuses on designing the **autocomplete feature** of Google Search. This is the feature that provides **suggestions** while a user types in the search bar, offering predictions based on partial input.

**Understanding the Autocomplete System**

1. **Definition of Autocomplete**:
   * While typing in a search bar, the system suggests possible queries based on the entered text.
   * Example:
     + Input: **MIC**
     + Suggestions:
       - Michael Jackson
       - Michelle Obama
       - Michael Jordan
       - Microservices
2. **Key Observations**:
   * The suggestions are **strict prefixes** of the typed query.
   * Users can either select a suggestion or continue typing and press "Enter" to perform the final search.
3. **End-to-End Flow**:
   * **Typing Phase**:
     + As the user types, autocomplete provides suggestions in real time.
   * **Search Phase**:
     + If the user selects a suggestion or finishes typing and presses "Enter," the final search query is executed.
   * **Search Results**:
     + Search results display the retrieved pages, with pagination (e.g., page 1, 2, 3).

## Functional Requirements for Google Autocomplete System

**Overview**

This lecture focuses on identifying the **Minimum Viable Product (MVP)** and **functional requirements** for designing the autocomplete feature of Google Search. The process emphasizes defining clear priorities for the system's first version (V0) and iterating on advanced features in future versions (V1, V2).

**Key Takeaways**

**1. Understanding MVP (Minimum Viable Product)**

* **MVP Definition**: The minimum set of features required to make the system functional for users.
* **Objective**: Deliver essential functionality quickly without unnecessary complexity.

**2. Functional Requirements for Autocomplete MVP**

The following are critical features for V0 (MVP):

1. **Maximum Number of Suggestions**:
   * Limit the number of suggestions displayed to **five** to maintain simplicity and clarity.
2. **Order and Selection of Suggestions**:
   * Suggestions should be ranked based on **popularity**.
   * **Popularity Definition**:
     + **Most Frequent Search Terms**: Select suggestions based on the number of times they have been searched.
     + **No Time Decay for V0**: Ignore recent trends for the MVP and focus solely on absolute search frequency.
   * V1/V2 Enhancement: Incorporate time-based popularity adjustments to prioritize recent trends.
3. **Personalization**:
   * **V0 Decision**: Personalization (e.g., user-specific or location-specific) is **not required**.
     + Examples of personalization:
       - Region-specific popular terms (e.g., IP-based suggestions).
       - User-preference-based terms (e.g., interest in specific topics like cricket).
     + These features will be considered for V1/V2 as they are **"good-to-have"** but not critical for MVP.
4. **Minimum Input Length**:
   * Suggestions will only appear after the user types at least **three characters** (e.g., “MIC” triggers suggestions, but “M” or “MI” does not).
   * Reason: Avoid overwhelming the user with too many irrelevant suggestions at early stages.
5. **Handling Spelling Errors**:
   * **V0 Decision**: Spelling corrections (e.g., mistyped “MCI” instead of “MIC”) are **not required**.
   * This feature can be part of V1/V2 for improved user experience.

|  |  |  |
| --- | --- | --- |
| **Summary of MVP Functional Requirements** | | |
| **Feature** | **Included in V0?** | **Notes** |
| Max Number of Suggestions | Yes | Limit to **5 suggestions**. |
| Popularity-Based Ranking | Yes | Use absolute search frequency (ignore recency for now). |
| Personalization | No | Defer for V1/V2. |
| Min Input Length | Yes | Trigger suggestions after **3 characters**. |
| Spelling Error Handling | No | Defer for V1/V2. |

## Non-Functional Requirements for Google Autocomplete System

**Overview**

This lecture segment outlines the **non-functional requirements (NFRs)** for the autocomplete feature, focusing on system properties like **availability**, **consistency**, and **latency**. These factors determine how the system performs under real-world conditions.

**Key Non-Functional Requirements**

**1. Availability vs. Consistency**

* **Key Decision**: Prioritize **high availability** over strict consistency.
  + **Reason**: Users can tolerate minor inconsistencies (e.g., slightly incorrect suggestion rankings) but expect the system to always provide some suggestions.
  + Inconsistency Example:
    - Actual count for "Michael Jackson" is 5,800 but appears as 5,700 in rankings.
    - This slight inaccuracy might result in showing "Michelle Obama" instead. Such cases are acceptable.
  + **Availability Importance**: Users must see suggestions in real-time without system downtime.

**2. Consistency Definition**

* **Context**: Ensuring that popular terms (based on search frequencies) are shown accurately.
* **Relaxed Consistency**:
  + Counts may be slightly off, and the suggestion order may vary slightly.
  + Exact real-time frequency updates are not critical for user experience.

**3. Latency**

* **Requirement**: Achieve **very low latency** (in the millisecond range).
  + **Reason**: The system must keep up with users' typing speed, providing suggestions as they type.
  + **Goal**: Deliver suggestions within milliseconds to avoid interrupting the typing flow.

|  |  |  |
| --- | --- | --- |
| **Summary of Non-Functional Requirements** | | |
| **Requirement** | **Priority** | **Details** |
| **Availability** | **High** | Ensure system is always responsive, showing suggestions at any given time. |
| **Consistency** | **Low** | Accept slight inaccuracies in suggestion rankings or counts. |
| **Latency** | **Very High** | Maintain millisecond response times to match user typing speed. |

## Back of the Envelope Estimation

**Goals of Estimation**

1. Determine whether **sharding** is needed.
2. Assess if the system is **write-heavy** or **read-heavy**.
3. Estimate required **capacity** (number of machines, storage, etc.).

**Key Factors Influencing Sharding**

1. **Size of data**: Primary determinant for the need for sharding.
2. **Nature of stored data**: In this context, storing:
   * Search terms (not just keywords, as they can be multi-word phrases, e.g., "Roger Federer").
   * Frequency of search terms (as a number).
3. **Scope of personalization**: Assuming no user-level personalization; data is stored at a global level.

**Steps for Estimation**

**1. Total Queries Per Day**

* Estimated traffic: **8 billion search queries per day** (approximation for Google-like scale).
* Assumption: Not all queries are unique; many queries repeat.

**2. Proportion of New Search Terms**

* About **10% of daily queries** result in **new search terms** (based on historical data shared by Google, where 10-30% of queries are unique).
* New terms per day: .

**3. Estimating New Terms Over 10 Years**

* Days in 10 years: .
* Total new terms in 10 years: .

**4. Storage Size Estimation**

* Average size of a search term: .
* Total storage for 10 years: .

## Read Heavy/Write Heavy

**1. Understanding Reads and Writes**

* **Read Operations (Reads):**
  + Triggered by each keystroke after the third character in a search bar.
  + Example: Typing "Sachin Tendulkar":
    - Typing "SAC" → 1 read.
    - Typing "H" → 1 read.
    - Typing "I" → 1 read.
    - Total reads increase with each character until the suggestion is selected.
* **Write Operations (Writes):**
  + Occurs when the user selects a suggestion or presses enter.
  + Updates the frequency of the selected search term.
  + Example: Searching "Sachin Tendulkar" → frequency is updated by +1.
  + Writes do not increase for partial terms in a good system (e.g., "Sachin" or "Tendulkar" individually).

**2. Characteristics of Reads and Writes**

* **Good Auto-Suggest System:**
  + Minimal reads: High-quality suggestions shown early reduce unnecessary keystrokes.
  + Balanced read-write ratio: For each read, there’s typically one corresponding write.
* **Bad Auto-Suggest System:**
  + Multiple reads before finding a relevant suggestion.
  + Imbalanced read-write ratio.
* **General Nature of the System:**
  + The system is both read-heavy and write-heavy at high traffic volumes.
  + Requires optimization to handle both efficiently.

**3. Optimization Strategies**

* **Absorb Reads:**
  + Use caching to minimize reads to the database.
  + Cache frequently searched terms and their suggestions.
* **Absorb Writes:**
  + Batch writes or use systems that tolerate eventual consistency.
  + Aggregate frequency updates for the same term before writing.

**4. Capacity Estimation**

* **Steps to Estimate Capacity:**
  + Calculate **Queries Per Second (QPS)**:
    - Total queries per day ÷ Total seconds in a day.
    - Example: 24 billion operations per day = ~300,000 QPS.

|  |
| --- |
| The total number of queries per day (24 billion) is derived based on the assumption that:   1. **Searches Per Day (Writes):**    * The system handles **8 billion searches per day** globally. 2. **Reads Per Search:**   For every search, **2 read operations** are assumed on average:   * + One or more read operations are triggered by the auto-suggest feature while the user types.   + A good system minimizes reads but assumes at least 2 reads per write (search).  1. **Total Queries (Reads + Writes):**   Total queries = **Reads + Writes** = 8 billion writes + (8 billion×2 reads per write)  =24 billion queries/day.  **Calculating Queries Per Second (QPS)**   * Total seconds in a day = . * ​. Substituting values:   **Why Assume 2 Reads per Write?**   * **User Interaction with Auto-Suggest:**   + In a good system, relevant suggestions are shown within a few keystrokes.   + On average, it’s assumed that 2 read requests are enough to fetch and refine the suggestions before the user selects or presses enter. |

* + Determine machine bottlenecks:
    - **CPU** for compute-intensive tasks.
    - **Memory (RAM)** for caching or in-memory operations.
    - **Network IO** for large or frequent data transfers.
  + Decide the number of machines based on QPS and the identified bottleneck.
* **Bottleneck Analysis:**
  + **Example Use Cases:**
    - Movie uploads → Network-heavy.
    - Code execution → CPU-heavy.
    - Frequent hash table lookups → Memory-heavy.

**5. System Scale Assumptions**

* Equally distributed requests: Assumes uniform traffic throughout the day.
* Burst traffic: Systems like ticket booking might concentrate traffic in short intervals, requiring recalibration of QPS calculations.

## APIs for Auto Suggest System

* **Core APIs:**
  + **Get Suggestions:** Fetch top suggestions for a given prefix.
    - Example: Input "SAC" → Output: ["Sachin Tendulkar", "Sacramento", etc.].
  + **Register Search:** Log the occurrence of a search term when selected.
    - Example: User selects "Sachin Tendulkar" → Frequency is updated.
* **Async Operations:**
  + Search frequency updates are best effort and asynchronous to reduce latency.

**Separation of Concerns**

* **Auto Suggest Service:** Provides prefix-based suggestions.
* **Search Service:** Fetches search results for a query after enter is pressed.
  + Example: "Michael Jackson" → Returns relevant web pages.

**Key Design Considerations**

* **Improving Suggestions:** Show popular terms early to reduce unnecessary reads.
* **Handling Skewness:** Systems like IRCTC or ticket booking require different handling for burst traffic.
* **Database Selection:** Choose databases optimized for either reads or writes, then optimize for the other aspect through caching or batching.

## Trie for Autocomplete with Top-K Suggestions

**Problem Context**

* **Task**: Implement a system on a single machine to:
  1. **Find the top-K suggestions** (e.g., top 5) given a prefix.
  2. **Update search frequency** of a term when searched.
* **Challenges**:
  1. Regular **Trie** works well for insertions and searches but is slow for finding top-K suggestions (requires traversing all child nodes to calculate top suggestions).
  2. **Read** (top-K) needs to be extremely fast, even if **write** (frequency update) is slower.

**Optimized Trie Solution**

**Data Structure Design**

Each node in the trie stores:

1. **Frequency**: Number of searches for the word ending at that node.
   * Example: If node corresponds to "Michael", frequency = number of times "Michael" was searched.
2. **Top-K Suggestions**: List of the top-K terms (e.g., top 5) starting with the prefix at this node.
   * Stored as an array or heap.

**API Design**

1. **Read API**: Retrieve the top-K suggestions for a given prefix.
   * **Time complexity**: O(length of prefix). The node already stores top-K suggestions, so retrieval is direct.
2. **Write API**: Update the frequency of a term when searched.
   * **Steps**:
     1. Locate the node corresponding to the term (e.g., "Michael").
     2. Increment the frequency at that node.
     3. Propagate changes upwards in the trie:
        + Update the top-K suggestions of ancestor nodes if the frequency change makes the term eligible for top-K.

**Write Implementation Details**

1. **Frequency Update**:
   * Increment the frequency at the node where the term ends.
2. **Top-K Update**:
   * For each ancestor node of the updated term:
     1. Check if the term's new frequency makes it eligible for top-K (compare with the least frequency in top-K).
     2. If yes, replace the least frequent term in top-K with the updated term.
     3. Continue until the root or no further updates are needed.

**Properties of the Solution**

* **Read**: Fast because the top-K suggestions are precomputed and stored in each node.
* **Write**: Slightly slower as updates propagate to ancestor nodes.
* **Space Overhead**: Increased due to storing top-K suggestions at every node.

**Illustrative Example**

**Trie Structure Example**

Prefix = "MIC". Words starting with MIC:

* "Michael Jackson" (frequency: 500)
* "Michelle Obama" (frequency: 498)
* "Michael Jordan" (frequency: 300).

**Node for "MIC"**:

* **Frequency**: 0 (if "MIC" is not a complete word).
* **Top-K**: ["Michael Jackson (500)", "Michelle Obama (498)", "Michael Jordan (300)"].

**Update Scenario**:  
If "Michael" frequency increases from 100 to 301:

1. Update frequency at "Michael".
2. Check top-K at "MIC":
   * If "Michael (301)" is greater than the least frequent term (e.g., "Michael Jordan (300)"), update top-K by replacing "Michael Jordan" with "Michael".

**Performance Analysis**

1. **Read Complexity**:
   * O(length of prefix) since top-K is precomputed.
2. **Write Complexity**:
   * O(length of word × K) for propagating changes to top-K in ancestor nodes.
   * Efficient because updates are localized to the ancestor chain of the updated term.
3. **Space Complexity**:
   * Extra space for storing top-K at each node.
   * Trade-off: Improves read speed at the cost of storage.

## Updating Search Term Frequency

**Overview**

The lecture discusses techniques to manage and update search term frequencies efficiently for an auto-complete system. The focus is on optimizing writes to the database (DB) and maintaining trends while reducing latency and computational costs. Two key approaches are explained:

1. **Buffering**
2. **Sampling**

**1. Buffering Approach**

**Key Idea**

Buffer frequent updates in a temporary storage (e.g., a cache or local HashMap) and batch-process only popular search terms to the database.

**Steps**

1. **Temporary Storage**:
   * Maintain a cache or local HashMap with keys as search terms and values as counts.
   * All search queries update this local storage instead of directly interacting with the database.
2. **Threshold-Based Updates**:
   * Define a **threshold count** (e.g., 100).
   * Once a term’s count in the cache reaches the threshold:
     + Update the database with a batch increment (e.g., +100).
     + Reset the local count for that term to 0.
3. **Advantages**:
   * Reduces the number of writes to the database (e.g., 1 write instead of 100 incremental writes).
   * Saves on database operations like tree traversal and top-N updates.
   * Reduces overall network calls.
4. **Eviction Policy for Cache**:
   * Use **Least Frequently Used (LFU)**:
     + Evict terms with the least search frequency.
   * If frequencies are tied, evict the oldest term.

**2. Sampling Approach**

**Key Idea**

Process only a **random sample** of search queries to estimate trends while ignoring the exact counts.

**Concept:**

* Trends in a **random sample** statistically resemble the trends in the full dataset.
* Similar to exit polls in elections, where a small sample accurately reflects overall voting behaviour.

**Steps:**

1. **Random Sampling**:
   * For every search query, generate a random number R in the range [0, 99] (e.g., using rand() % 100).
   * Only process the query if R == 0 (1% probability).
2. **Processing Sampled Queries**:
   * Update the database or frequency structures (e.g., trie or HashMap) for the sampled queries.
   * Ignore 99% of the queries.
3. **Advantages**:
   * Significantly reduces write volume to approximately **1% of the original**.
   * Maintains popularity trends despite not processing all data.
4. **Limitations**:
   * Sampling is effective only when trends are more important than exact counts.
   * Not suitable for systems where precise frequency counts are critical.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Buffering** | **Sampling** |
| **Purpose** | Reduce write frequency to the DB | Reduce the volume of processed queries |
| **Storage Requirement** | Cache or HashMap for temporary counts | No additional storage required |
| **Write Efficiency** | Batches updates based on popularity | Processes only a small subset (1%) |
| **Trend Accuracy** | Very high | High (but not exact counts) |
| **Complexity** | Moderate (manages counts and thresholds) | Simple (random number generation) |

## Optimizing Read-Write, Sharding

**1. Read-Write Optimization**

**1.1 Buffering Approach:**

* **Concept**:
  + Accumulate counts in a cache or local hash map for popular terms.
  + Only push data to the database when a threshold is reached (e.g., count ≥ 100).
* **Advantages**:
  + Reduces the frequency of writes to the database.
  + Avoids frequent traversal of the Trie or HashMap for every search term.
  + Network and DB load are reduced significantly.
* **Implementation**:
  + Cache maintains key-value pairs (term -> count).
  + When the count for a term reaches the threshold, add it in bulk to the DB and reset the cache count to 0.
  + Eviction in cache: Use **Least Frequently Used (LFU)** eviction strategy with tie-breaking based on age.

**1.2 Sampling Approach:**

* **Concept**:
  + Only process 1% of the search terms (or a chosen sampling rate) and discard the rest.
  + Ensure trends in the sample reflect trends in the larger data set.
* **Advantages**:
  + Drastically reduces the number of writes (e.g., by 99% for a 1% sampling rate).
  + Lightweight and scalable.
* **Implementation**:
  + Use a random number generator (Rand mod 100).
  + If the number equals 0 (1% chance), process the search term; otherwise, discard.

**2. Storing Search Data Using Trie or HashMap**

* **HashMap**:
  + Simple key-value structure (term -> frequency or term -> [top suggestions]).
  + Efficient for fast reads and writes.
* **Trie**:
  + Hierarchical structure optimized for prefix-based queries.
  + Each node contains:
    - Links to children.
    - Top 5 search suggestions and their frequencies.

**3. Sharding Strategies**

**3.1 Sharding HashMap-Based Systems:**

* **Key Idea**: Sharding based on keys is straightforward since each key is independent.
* **Implementation**:
  + Use consistent hashing on the key to determine the shard.
  + Example: MIC -> Shard 1, MI -> Shard 2.

**3.2 Sharding Trie-Based Systems:**

* **Challenges**:
  + Trie traversal must remain efficient; breaking nodes across machines adds latency.
  + Subtrees must be allocated to the same shard to maintain continuity.
* **Solution**:
  + Only split at level 3 (after three characters are typed).
  + Each subtree rooted at the third character combination (e.g., "MIC", "SHA") is treated as a unit.
  + Use consistent hashing to assign these subtrees to shards.
  + Example:
    - Total subtrees: .
    - Shards: 100.
    - Distribute 17,576 subtrees across 100 shards randomly using consistent hashing.
* **Advantages**:
  + Ensures popular and less popular prefixes are evenly distributed.
  + Each shard handles multiple subtrees but maintains contiguous data.

## Recency with Time Decay Factor

**Concept Overview**

* **Challenge**: Counting search terms based only on absolute frequency can fail to account for recency.
  + **Example**: A term frequently searched years ago may have a high count but is no longer relevant, whereas a recently trending term might have a lower count but is more pertinent for suggestions.
  + Goal: Incorporate recency into search counts to balance absolute frequency and recent popularity.

**Solution: Time Decay Factor (TDF)**

Time Decay Factor is used to diminish the weight of older counts progressively over time, ensuring recent activity has greater influence.

**Algorithm**

1. **Core Idea**
   * At the end of each day, all counts are reduced by dividing them by a decay factor (TDF), which is greater than 1.
   * New counts for the day are added after decaying the older counts.
2. **Calculation**
   * For any term on day n:
3. **Impact of Decay Factor (TDF)**
   * **Higher TDF**: Faster decay; older counts diminish more rapidly.
   * **Lower TDF**: Slower decay; older counts retain more relevance.

**Illustrative Examples**

**Scenario 1: Decay with TDF = 2 (Fast Decay)**

* **Day 1**: Count = 10,000
* **No further searches over 10 days**

**Scenario 2: Decay with TDF = 1.1 (Slower Decay)**

* **Day 1**: Count = 10,000
* **No further searches over 10 days**

**Scenario 3: Minimal Decay with TDF = 1.01**

* **Day 1**: Count = 10,000
* **No further searches over 10 days**

**Implementation Steps**

1. **Daily Decay Job**
   * At the end of each day, reduce all term counts using the formula:
   * Add counts for the current day.
2. **Older Count Cleanup**
   * Periodically, delete terms whose counts fall below a threshold (e.g., counts approaching 0).
3. **Choosing TDF**
   * **Typical values**:
     + Fast decay: TDF = 2
     + Moderate decay: TDF = 1.1
     + Slow decay: TDF = 1.01
   * Adjust based on system requirements (e.g., trending vs. long-term relevance).

**Advantages of Time Decay Factor**

1. **Balances Recency and Popularity**: Prioritizes recent activity while considering older counts.
2. **Scalable**: Lightweight computation; applicable for large datasets.
3. **Flexibility**: TDF can be adjusted to fit different application needs.
4. **Automatic Trend Adjustment**: Ensures trending terms rise in rankings naturally.