### **Problem Understanding:**

Query1: Is the matching only supported at the beginning of a search query or in the middle as well? -> Only at the beginning of a search query.

Query2: How many users use the product?

-> 10 million DAU = 10^7

Query3: How many autocomplete suggestions should the system return?

-> 5

Query4: In what order the suggestions need to be shown?

-> This is Determined by popularity, decided by the historical query frequency.

Query5: Does the system support spell check?

-> No, spell check or autocorrect is not supported.

Query6: Does the system support multiple languages or only english?

-> System supports English only but we can discuss multi-language support.

Query7: Do we allow special characters matching?

-> No, we assume all search queries have lower alphabetic characters.

### **Non-functional Requirements:**

1. Fast response time: As a user types a search query, autocomplete suggestions must show up fast enough. Otherwise there would be lag in autocomplete suggestions.
2. Relevant: suggestions should be relevant to the search term.
3. Sorted: results returned by the system must be sorted by popularity or other ranking models.
4. Scalable: The system can handle high traffic volume
5. High availability: The system should remain available and accessible when part of the system is offline, slows down, or experiences unexpected network errors.

### **Back-of-the-envelope estimation:**

1. Assume 10 million = 10^7 DAU
2. An average person performs 10 searches per day.
3. 20 bytes of data per query string:

* Assume we use ASCII character encoding. 1 character = 1 byte
* Assume a query contains 4 words and each word contains 5 characters on average
* That is 4\*5 = 20 bytes / query

1. For each character entered in the search box , a client sends a request to the backend for autocomplete suggestions. On average, 20 requests are sent for a search query.
2. QPS = 10^6 users \*10 queries/day \* 20 characters / (24\*60\*60) = 2\*10^9/(3600\*24)= 2\*10^9/10^4 = 2\*10^5
3. Peak QPS = 2\*QPS = 4\*10^5
4. Assume 20% daily queries are new.

10^6 users\* 10 queries \* 20 characters \* 20%

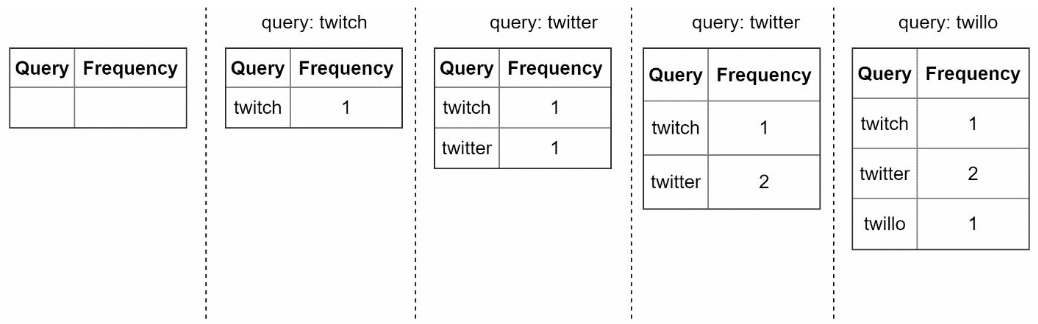
= 2\*10^9 \* 20% = 4\* 10^8 = 0.4GB per day.

**High Level Design and Get Buy-in:**

At the high-level, the system is broken into two parts:

* Data gathering service: Gathers user input queries and aggregates them in real-time. Real-time processing is not practical for such a large data set. Will explore this in a deep dive.
* Query service: Given a search query or prefix, return 5 most frequently searched terms.

**Data Gathering service**

Following shows how query frequency data is updated:  


**Query service**

When a user types tw it should fetch top 5 searched queries based on the updated data. Fo example:

****

To get top 5 searched queries, we can use following SQL query:

Select \* from frequency\_table

Where query like ‘%prefix%’

Order by frequency DESC

LIMIT 5

This is acceptable when data is small. When it is large, accessing the database becomes a bottleneck.

### **Design Deep Dive:**

### **Trie Data Structure:**

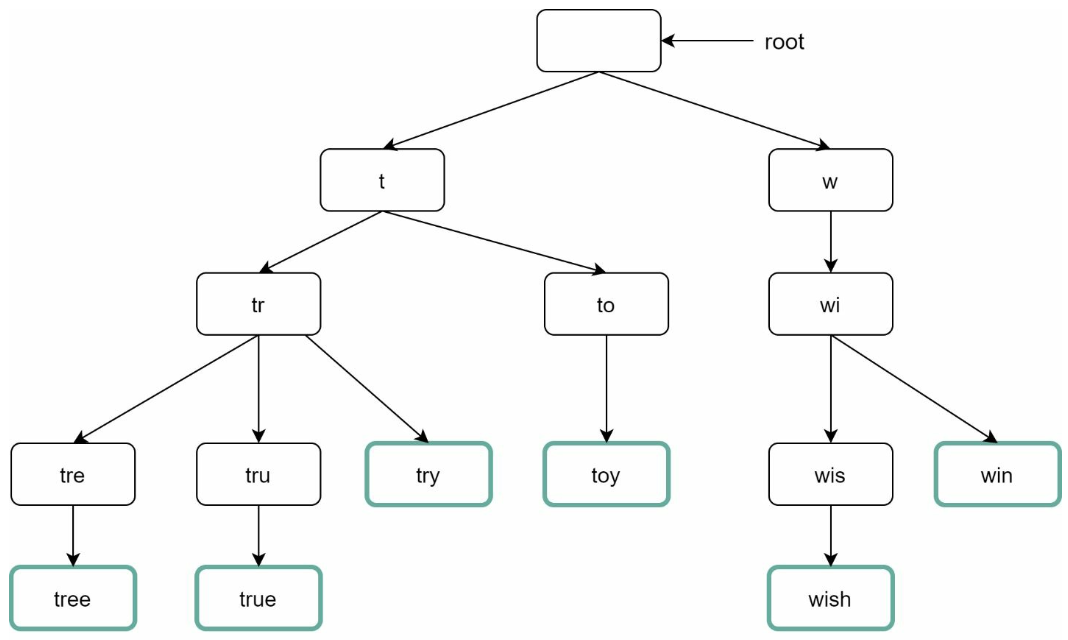
RDBMS would be used for storage. However, fetching top 5 search queries from RDBMS is inefficient. Trie data structure is used to overcome this problem.

The main idea of trie consists of the following:

• A trie is a tree-like data structure.  
• The root represents an empty string.

• Each node stores a character and has 26 children, one for each possible character. To save space, we do not draw empty links.

• Each tree node represents a single word or a prefix string.



To support sorting by frequency, frequency info needs to be included in nodes.

Before diving into the algorithm, let us define some terms:

* P: length of prefix
* N: total number of nodes in a trie
* C: number of children of a given node

Steps to get top k searched queries are listed below:

1. Find the prefix. Time complexity: O(p)
2. Traverse the subtree from the prefix node to get all valid children. A child is valid if it can form a valid query string. Time complexity: o(c)
3. Sort the children and get top k. Time complexity: O(c log c)

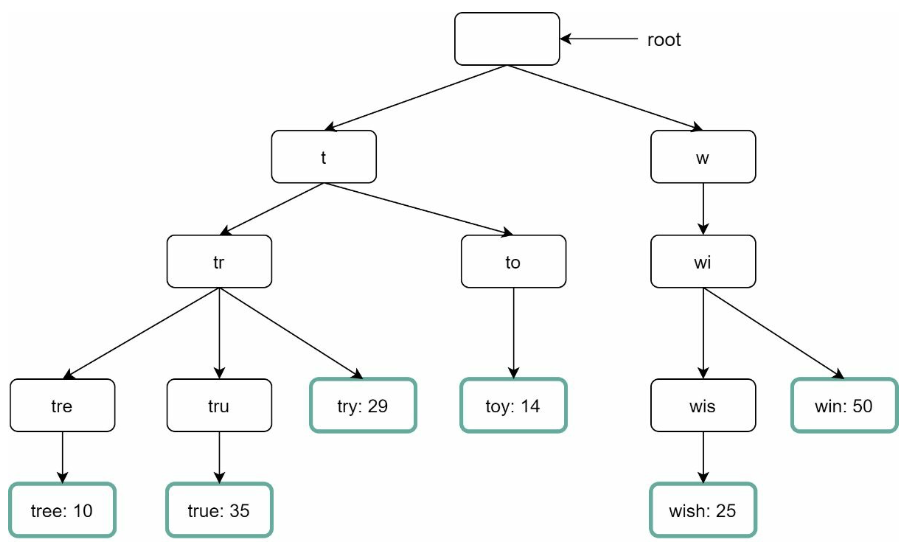
Let us use the below example to explain the algorithm. Assume k equals to 2 and a user types “tr” in the search box.

The algorithm works as follows:

• Step 1: Find the prefix node “tr”.

• Step 2: Traverse the subtree to get all valid children. In this case, nodes [tree: 10], [true: 35], [try: 29] are valid.

• Step 3: Sort the children and get top 2. [true: 35] and [try: 29] are the top 2 queries with the prefix “tr”.

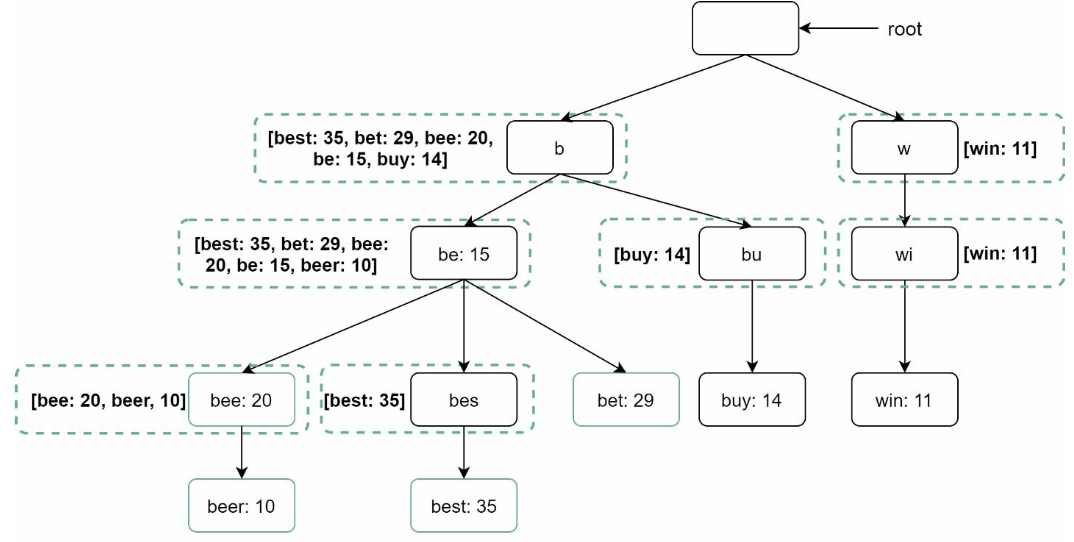


The time complexity of this algorithm is the sum of time spent on each step mentioned above: O(p) + O(c) + O(clogc)

However it is too slow because we need to traverse the entire trie to get top k results in the worst case scenario. Below are two optimizations:

1. **Limit max length of prefix:** Users rarely type a large query(more than 50 characters). If we limit the length of the prefix, the complexity for “find prefix” can be reduced from O(p) to O(contant) ~ O(1).
2. **Cache top search queries at each node:** To avoid traversing the whole trie, we store top k most frequently used queries at each node. Since 5 to 10 autocomplete suggestions are enough for users, k is a relatively small number. In our specific case, only the top 5 search queries are cached.

By caching top search queries at every node, we significantly reduce time complexity to retrieve top 5 queries. However, this design requires a lot of space to store top queries at every node. Trading for space for time is well worth it as fast response time is very important.

Top 5 queries are stored on each node. For example, the node with prefix “be” stores the following: [best: 35, bet: 29, bee: 20, be: 15, beer: 10]. 

Let us revisit the time complexity of the algorithm after applying those two optimizations:

1. Find the prefix node. Time complexity: O(1)  
2. Return top k. Since top k queries are cached, the time complexity for this step is O(1).

As the time complexity for each of the steps is reduced to O(1), our algorithm takes only O(1) to fetch top k queries.

### **Data Gathering Service**

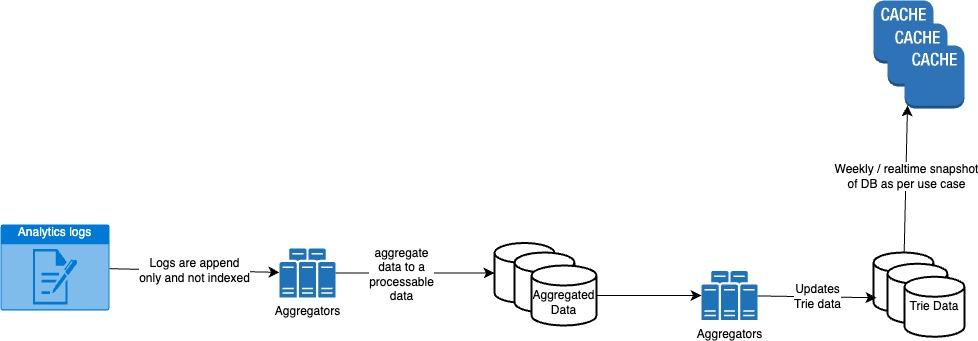
In our previous design, whenever a user types a search query, data is updated in real-time.

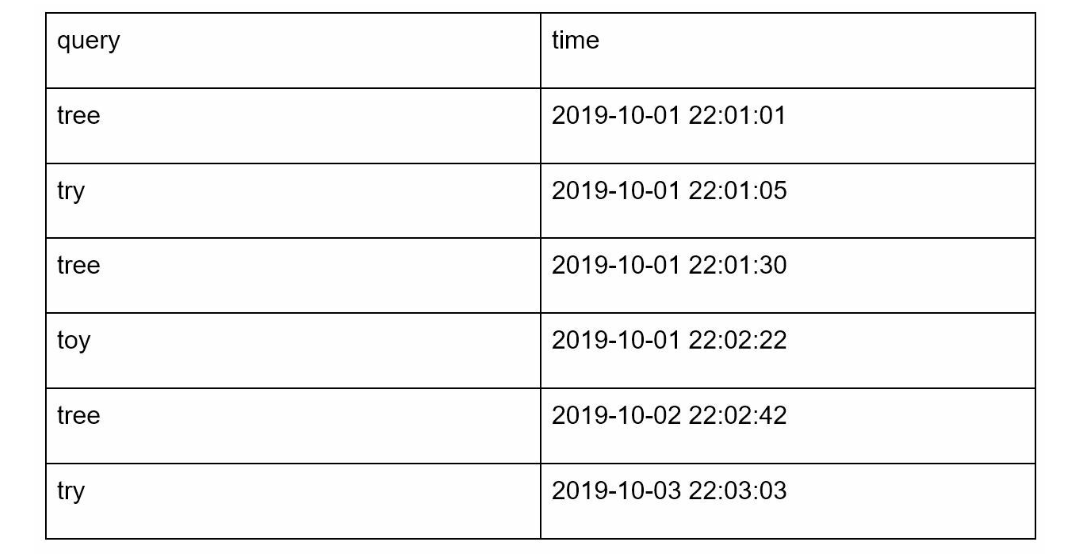
This approach is not practical for following two reasons:

* Users may enter billions of queries per day. Updating the trie on every query significantly slows down the query service.
* Top suggestions may not change much once the trie is built. Thus, it is necessary to update trie frequently.

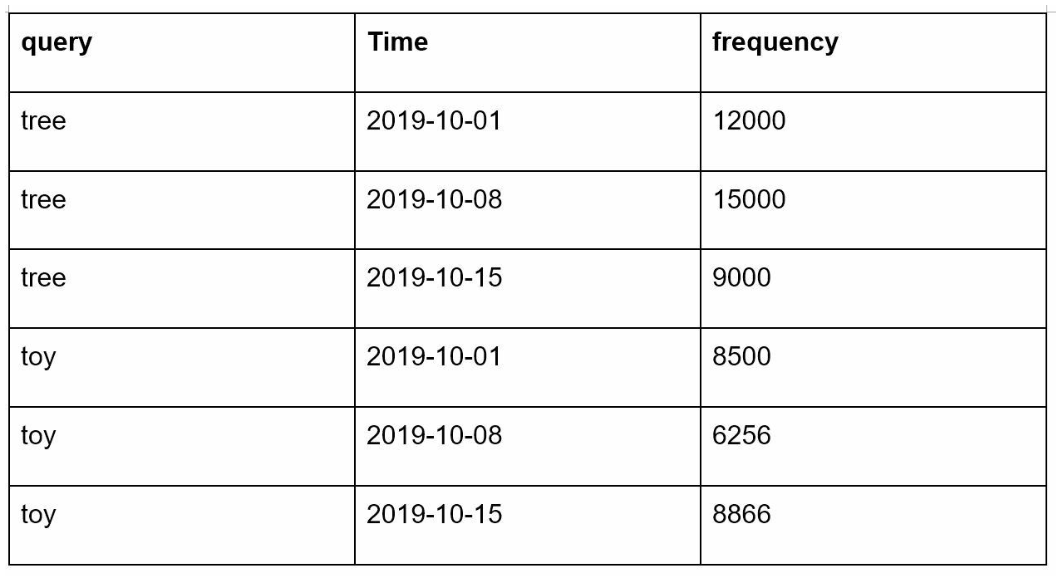
To design a scalable data gathering service, we examine where data comes from and how data is used. Real time applications like Twitter require up to date autocomplete suggestions. However, autocomplete suggestions for many Google keywords might not change much on a daily basis.

Despite the differences in use cases, the underlying foundation for data gathering service remains the same because data used to build the trie is usually from analytics or logging services.



Analytics logs file example: 

Aggregated data example:

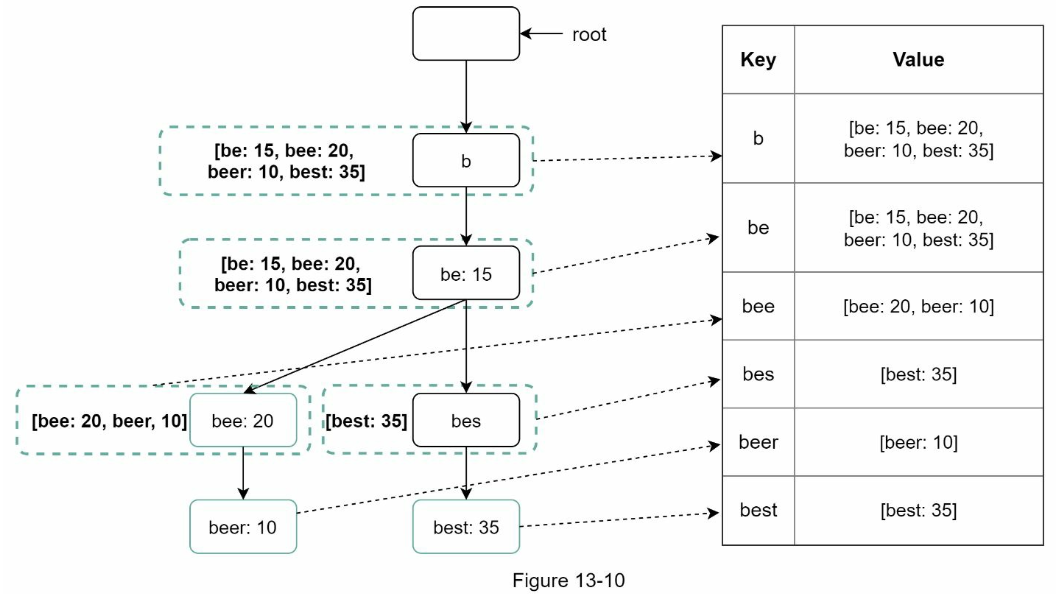


Trie Db is the persistent storage. Two options are available to store data:

1. Document store: Since a new trie is built weekly, we can periodically take a snapshot of it, serialize it and store the deserialized data in the database. Document stores like MongoDB are good fit for serialized data.
2. Key-value store: A trie can be represented in a hash table form by applying following logic:

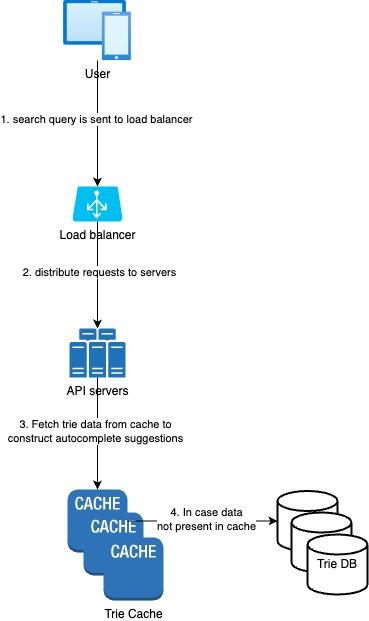
• Every prefix in the trie is mapped to a key in a hash table.

• Data on each trie node is mapped to a value in a hash table.



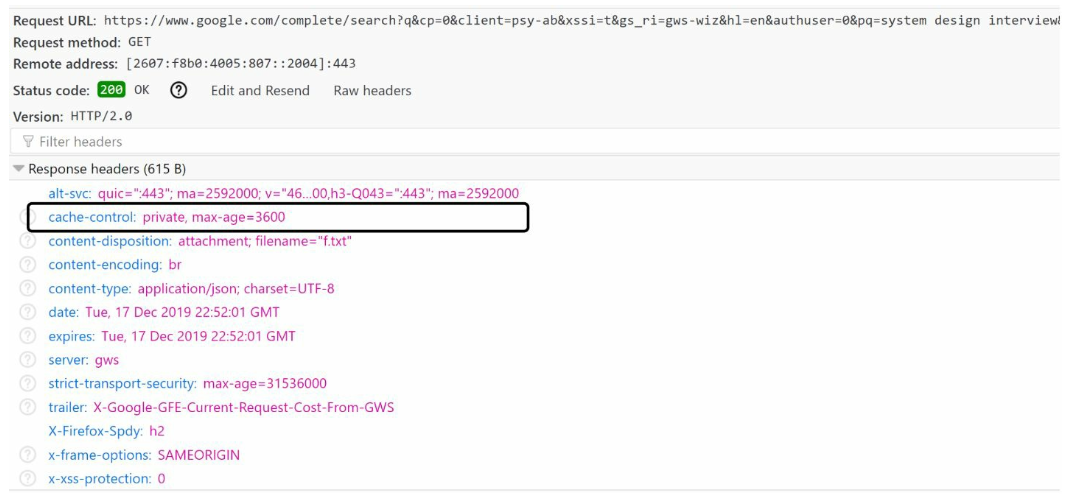
Each trie node is mapped to <key, value> pair.

### **Query Service**



Query service requires lightning fast speed. We propose following optimizations:

* AJAX request: For web applications, browsers usually send AJAX requests to fetch autocomplete results. The main benefit of AJAX is that sending/receiving a request/response does not refresh the whole web page.
* Browser caching: For many applications, autocomplete search suggestions may not change much within a short time. Thus, autocomplete suggestions can be saved in browser cache to allow subsequent requests to get results from the cache directly. Google search engine uses the same cache mechanism. Please note: “private” in cache-control means results are intended for a single user and must not be cached by a shared cache. “max- age=3600” means the cache is valid for 3600 seconds, aka, an hour.



* Data Sampling: For a large-system, logging every search query requires a lot of processing power and storage. Data sampling is important. For instance, only 1 out of every N request is logged by the system.

**Trie operations:**

Trie is a core component of the autocomplete system. Let us look at how trie operations (create, update, and delete) work.

**Create:**

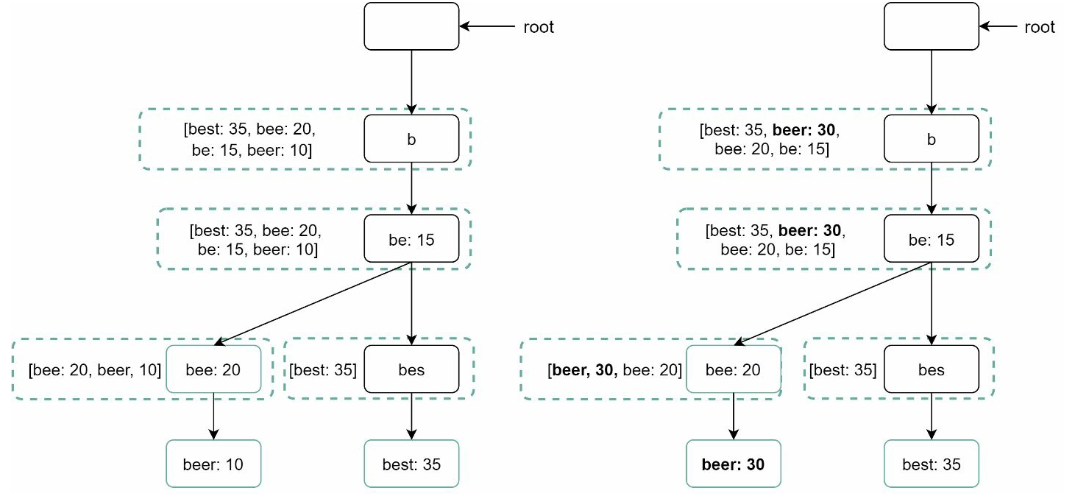
Trie is created by workers using aggregated data. The source of data is from Analytics Log/DB.

**Update:**

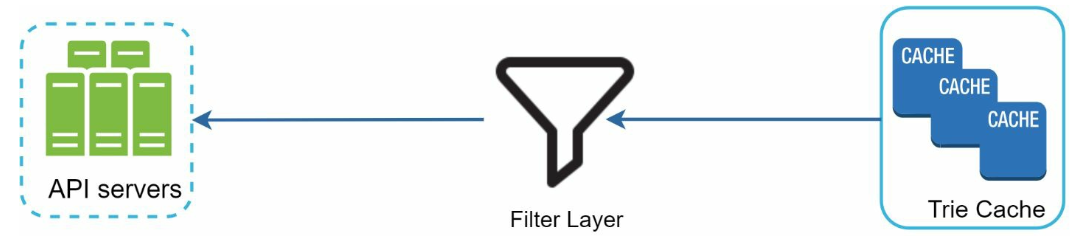
There are two ways to update the trie:

Option 1: Update the trie weekly. Once a new trie is created, the new trie replaces the old one.

Option 2: Update individual trie node directly. We try to avoid this operation because it is slow. However, if the size of the trie is small, it is an acceptable solution. When we update a trie node, its ancestors all the way up to the root must be updated because ancestors store top queries of children. Figure 13-13 shows an example of how the update operation works. On the left side, the search query “beer” has the original value 10. On the right side, it is updated to 30. As you can see, the node and its ancestors have the “beer” value updated to 30.



**Delete**

We have to remove hateful, violent, sexually explicit, or dangerous autocomplete suggestions. We add a filter layer (Figure 13-14) in front of the Trie Cache to filter out unwanted suggestions. Having a filter layer gives us the flexibility of removing results based on different filter rules. Unwanted suggestions are removed physically from the database asynchronously so the correct data set will be used to build trie in the next update cycle. 

### **Scale the storage**

Now that we have developed a system to bring autocomplete queries to users, it is time to solve the scalability issue when the trie grows too large to fit in one server.

Since English is the only supported language, a naive way to shard is based on the first character. Here are some examples.

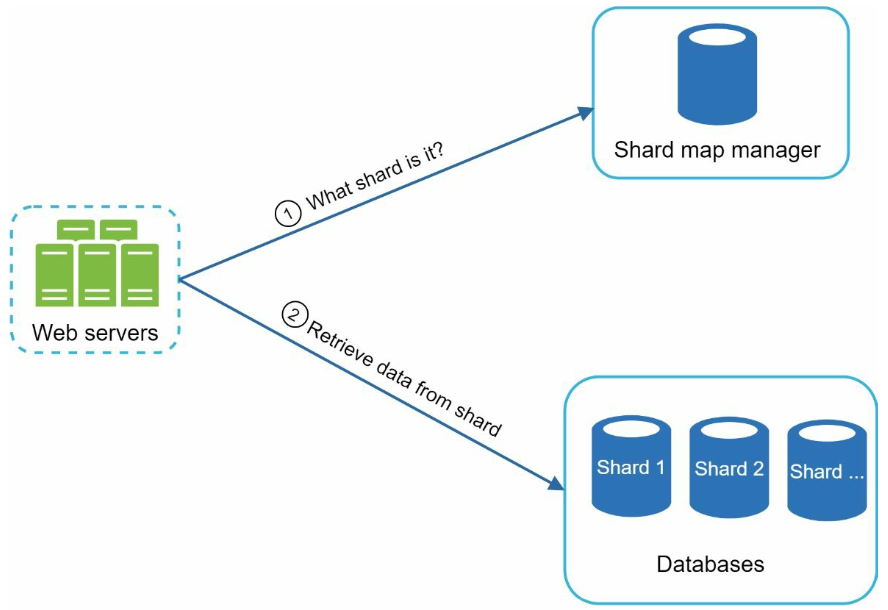
• If we need two servers for storage, we can store queries starting with ‘a’ to ‘m’ on the first server, and ‘n’ to ‘z’ on the second server.

• If we need three servers, we can split queries into ‘a’ to ‘i’, ‘j’ to ‘r’ and ‘s’ to ‘z’.

Following this logic, we can split queries up to 26 servers because there are 26 alphabetic characters in English. Let us define sharding based on the first character as first level sharding. To store data beyond 26 servers, we can shard on the second or even at the third level. For example, data queries that start with ‘a’ can be split into 4 servers: ‘aa-ag’, ‘ah- an’, ‘ao-au’, and ‘av-az’.

At the first glance this approach seems reasonable, until you realize that there are a lot more words that start with the letter ‘c’ than ‘x’. This creates uneven distribution.

To mitigate the data imbalance problem, we analyze historical data distribution pattern and apply smarter sharding logic as shown in Figure 13-15. The shard map manager maintains a lookup database for identifying where rows should be stored. For example, if there are a similar number of historical queries for ‘s’ and for ‘u’, ‘v’, ‘w’, ‘x’, ‘y’ and ‘z’ combined, we can maintain two shards: one for ‘s’ and one for ‘u’ to ‘z’.

****

**Interviewer: How do you extend your design to support multiple languages?**

To support other non-English queries, we store Unicode characters in trie nodes. If you are not familiar with Unicode, here is the definition: “an encoding standard covers all the characters for all the writing systems of the world, modern and ancient” [5].

**Interviewer: What if top search queries in one country are different from others?**

In this case, we might build different tries for different countries. To improve the response time, we can store tries in CDNs.

**Interviewer: How can we support the trending (real-time) search queries?**

Assuming a news event breaks out, a search query suddenly becomes popular. Our original design will not work because:

• Offline workers are not scheduled to update the trie yet because this is scheduled to run on weekly basis.

• Even if it is scheduled, it takes too long to build the trie.