Step 3 - Design deep dive

In the high-level design, we discussed data gathering service and query service. The high-level design is not optimal, but it serves as a good starting point. In this section, we will dive deep into a few components and explore optimizations as follows:

- Trie data structure
- Data gathering service
- Query service
- Scale the storage
- Trie operations

Trie data structure

Relational databases are used for storage in the high-level design. However, fetching the top 5 search queries from a relational database is inefficient. The data structure trie (prefix tree) is used to overcome the problem. As trie data structure is crucial for the system, we will dedicate significant time to design a customized trie. Please note that some of the ideas are from articles [2] and [3].

Understanding the basic trie data structure is essential for this interview question. However, this is more of a data structure question than a system design question. Besides, many online materials explain this concept. In this chapter, we will only discuss an overview of the trie data structure and focus on how to optimize the basic trie to improve response time.

Trie (pronounced “try”) is a tree-like data structure that can compactly store strings. The name comes from the word re**trie**val, which indicates it is designed for string retrieval operations. 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.

Figure 13-5 shows a trie with search queries “tree”, “try”, “true”, “toy”, “wish”, “win”. Search queries are highlighted with a thicker border.

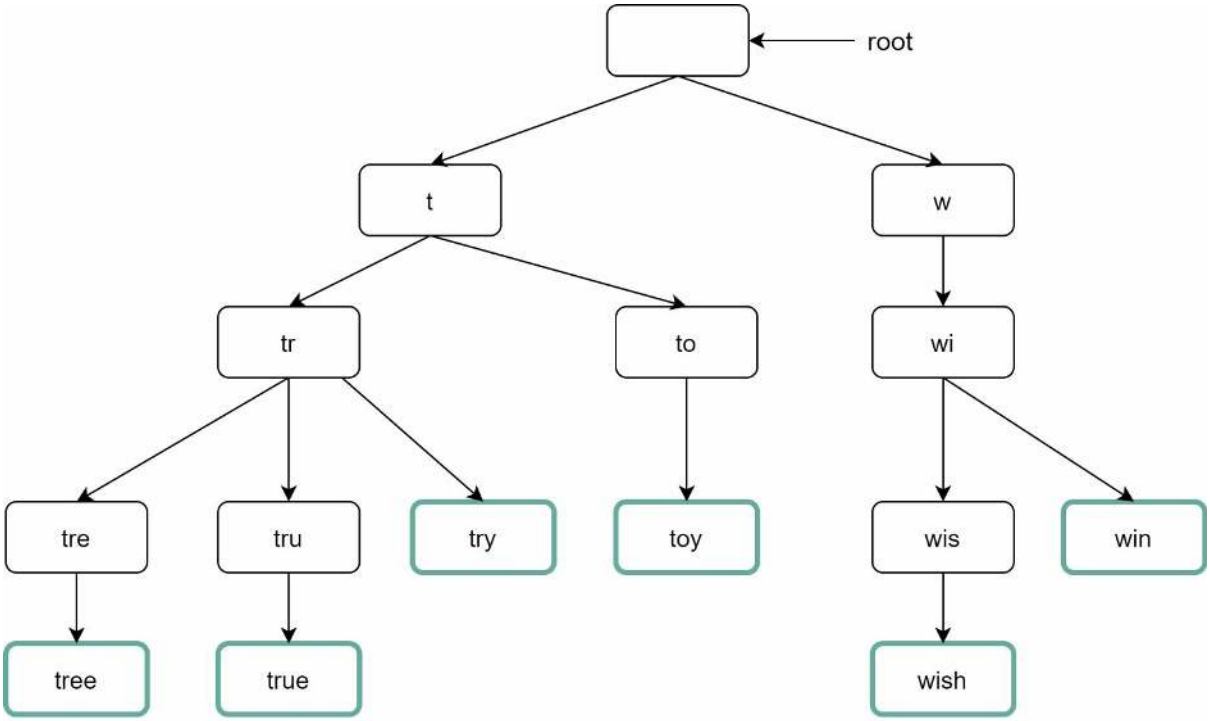

Figure 13-5

Basic trie data structure stores characters in nodes. To support sorting by frequency, frequency info needs to be included in nodes. Assume we have the following frequency table.

Query	**Frequency**
tree	10
try	29
true	35
toy	14
wish	25
win	50

Table 13-2

After adding frequency info to nodes, updated trie data structure is shown in Figure 13-6.

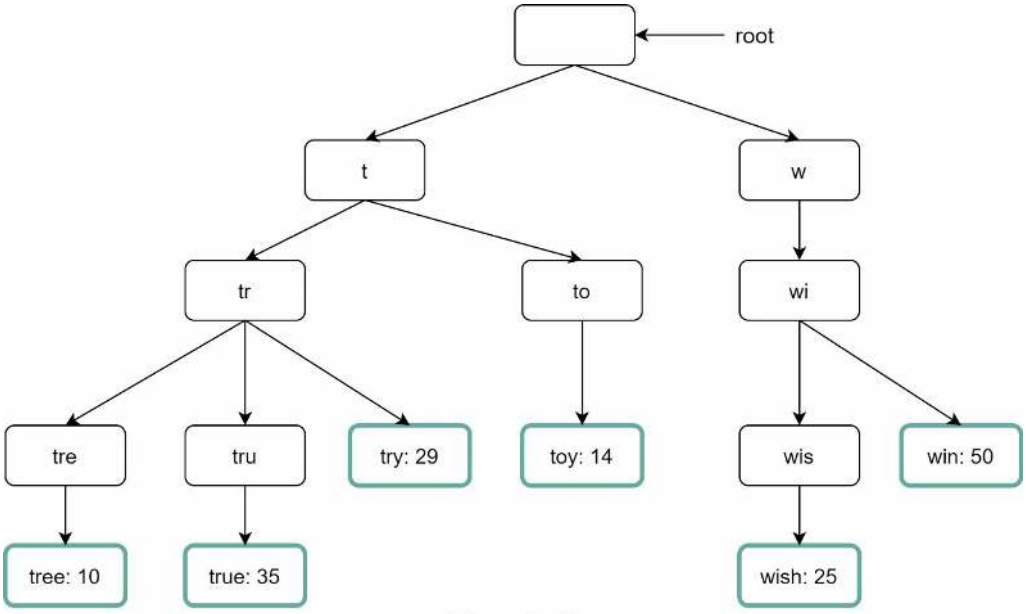

Figure 13-6

How does autocomplete work with trie? Before diving into the algorithm, let us define some terms.

- *p*: length of a prefix
- *n:* total number of nodes in a trie
- c: number of children of a given node

Steps to get top *k* most 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(clogc)*

Let us use an example as shown in Figure 13-7 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 prefix “tr”.

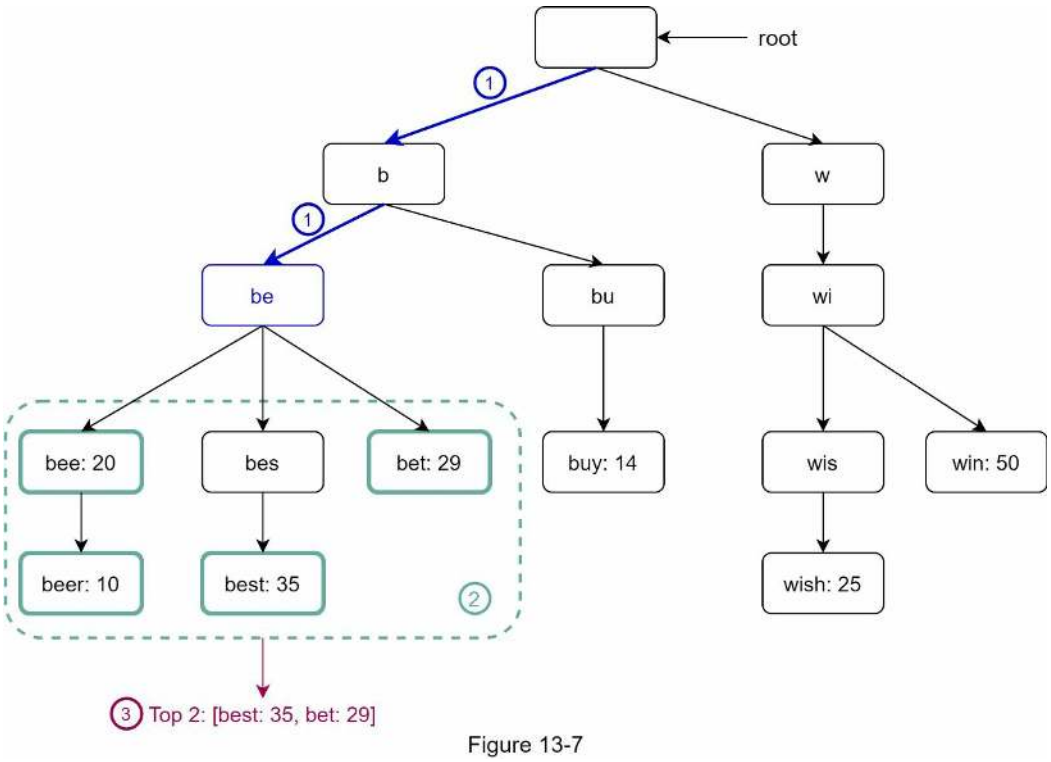

Figure 13-7

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

The above algorithm is straightforward. 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 the max length of a prefix
2. Cache top search queries at each node

Let us look at these optimizations one by one.

Limit the max length of a prefix

Users rarely type a long search query into the search box. Thus, it is safe to say *p* is a small integer number, say 50. If we limit the length of a prefix, the time complexity for "Find the prefix" can be reduced from *O(p)* to *O(small constant)*, aka *O(1)*.

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

Figure 13-8 shows the updated trie data structure. 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].

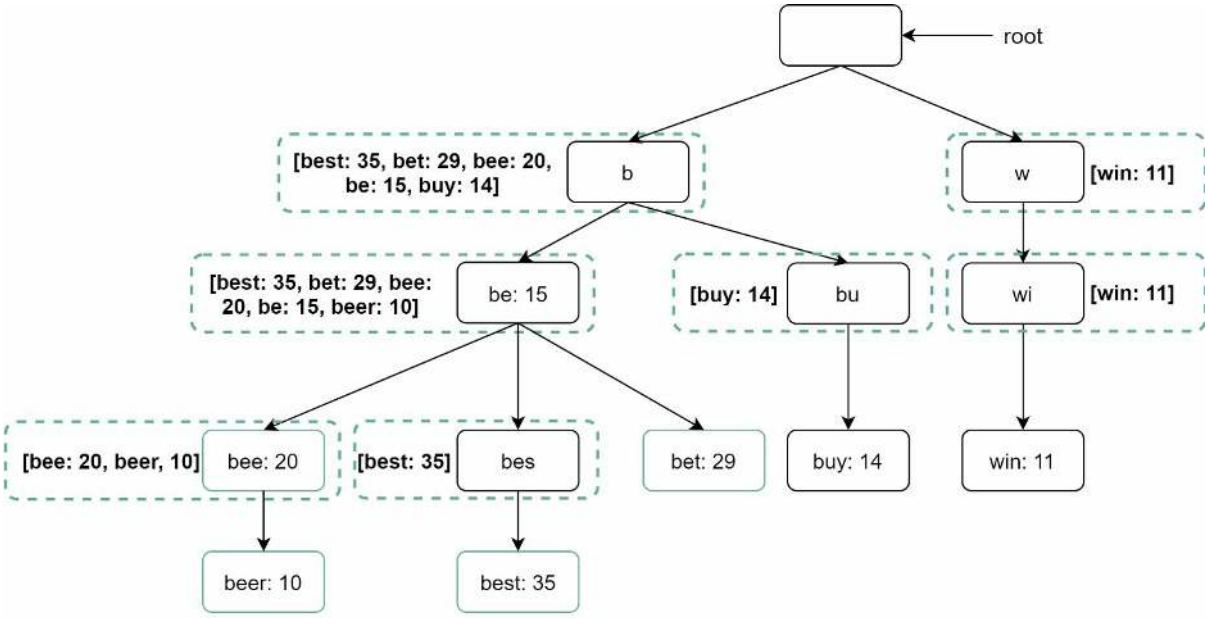

Figure 13-8

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 the 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 unnecessary to update the 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.

Figure 13-9 shows the redesigned data gathering service. Each component is examined one by one.

Figure 13-9

Analytics Logs. It stores raw data about search queries. Logs are append-only and are not indexed. Table 13-3 shows an example of the log file.

query	time
tree	2019-10-01 22:01:01
try	2019-10-01 22:01:05
tree	2019-10-01 22:01:30
toy	2019-10-01 22:02:22
tree	2019-10-02 22:02:42
try	2019-10-03 22:03:03

Table 13-3

Aggregators. The size of analytics logs is usually very large, and data is not in the right format. We need to aggregate data so it can be easily processed by our system.

Depending on the use case, we may aggregate data differently. For real-time applications such as Twitter, we aggregate data in a shorter time interval as real-time results are important. On the other hand, aggregating data less frequently, say once per week, might be good enough for many use cases. During an interview session, verify whether real-time results are important. We assume trie is rebuilt weekly.

Aggregated Data.

Table 13-4 shows an example of aggregated weekly data. “time” field represents the start time of a week. “frequency” field is the sum of the occurrences for the corresponding query in that week.

query	Time	frequency
tree	2019-10-01	12000
tree	2019-10-08	15000
tree	2019-10-15	9000
toy	2019-10-01	8500
toy	2019-10-08	6256
toy	2019-10-15	8866

Table 13-4

Workers. Workers are a set of servers that perform asynchronous jobs at regular intervals. They build the trie data structure and store it in Trie DB.

Trie Cache. Trie Cache is a distributed cache system that keeps trie in memory for fast read. It takes a weekly snapshot of the DB.

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

1. Document store: Since a new trie is built weekly, we can periodically take a snapshot of it, serialize it, and store the serialized data in the database. Document stores like MongoDB [4] are good fits for serialized data.

2. Key-value store: A trie can be represented in a hash table form [4] by applying the 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.

Figure 13-10 shows the mapping between the trie and hash table.

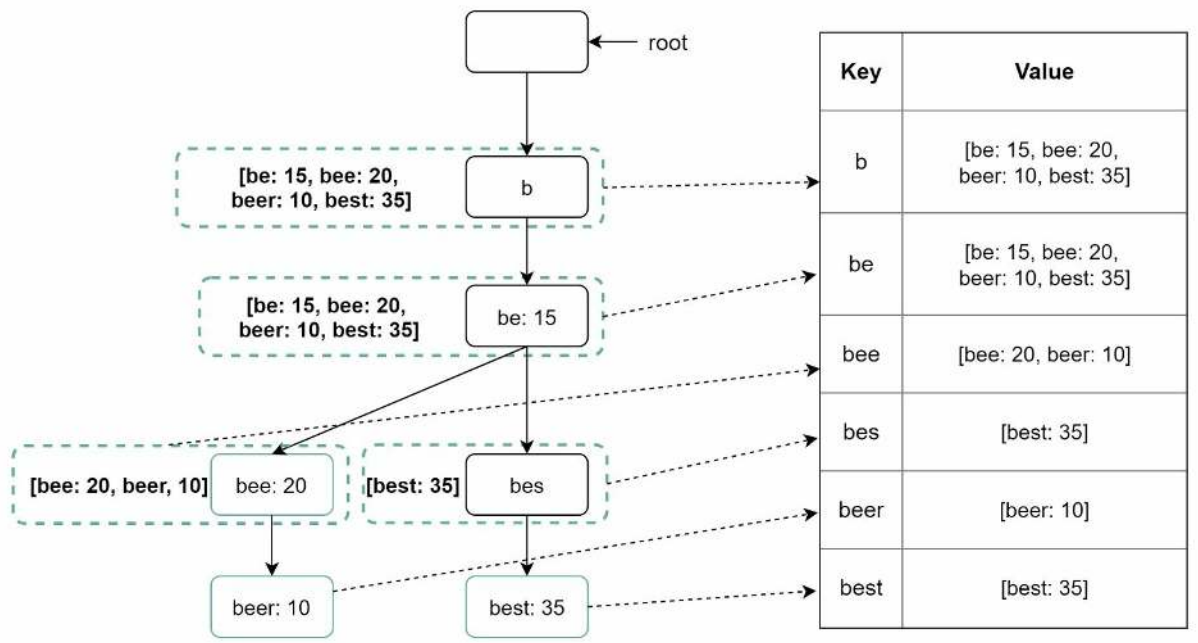

Figure 13-10

In Figure 13-10, each trie node on the left is mapped to the <*key*, *value*> pair on the right. If you are unclear how key-value stores work, refer to Chapter 6: Design a key-value store.

Query service

In the high-level design, query service calls the database directly to fetch the top 5 results. Figure 13-11 shows the improved design as previous design is inefficient.

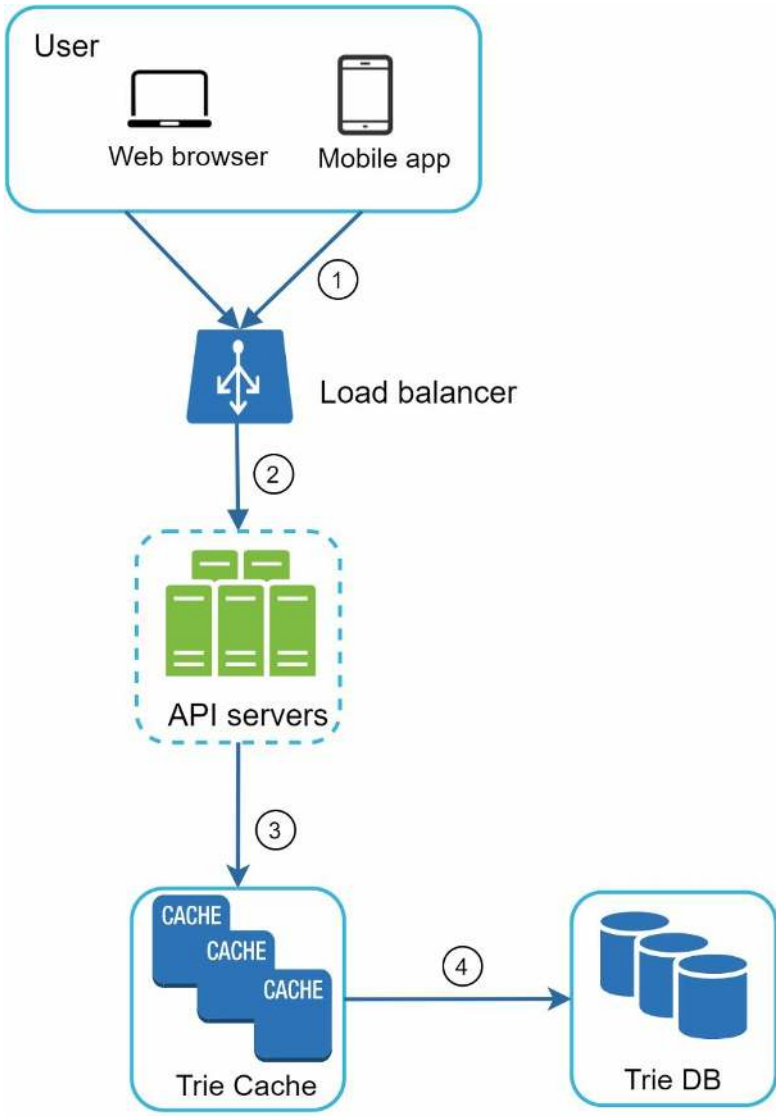

Figure 13-11

1. A search query is sent to the load balancer.
2. The load balancer routes the request to API servers.
3. API servers get trie data from Trie Cache and construct autocomplete suggestions for the client.
4. In case the data is not in Trie Cache, we replenish data back to the cache. This way, all subsequent requests for the same prefix are returned from the cache. A cache miss can happen when a cache server is out of memory or offline.

Query service requires lightning-fast speed. We propose the 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. Figure 13-12 shows the response header when you type “system design interview” on the Google search engine. As you can see, Google

caches the results in the browser for 1 hour. 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.

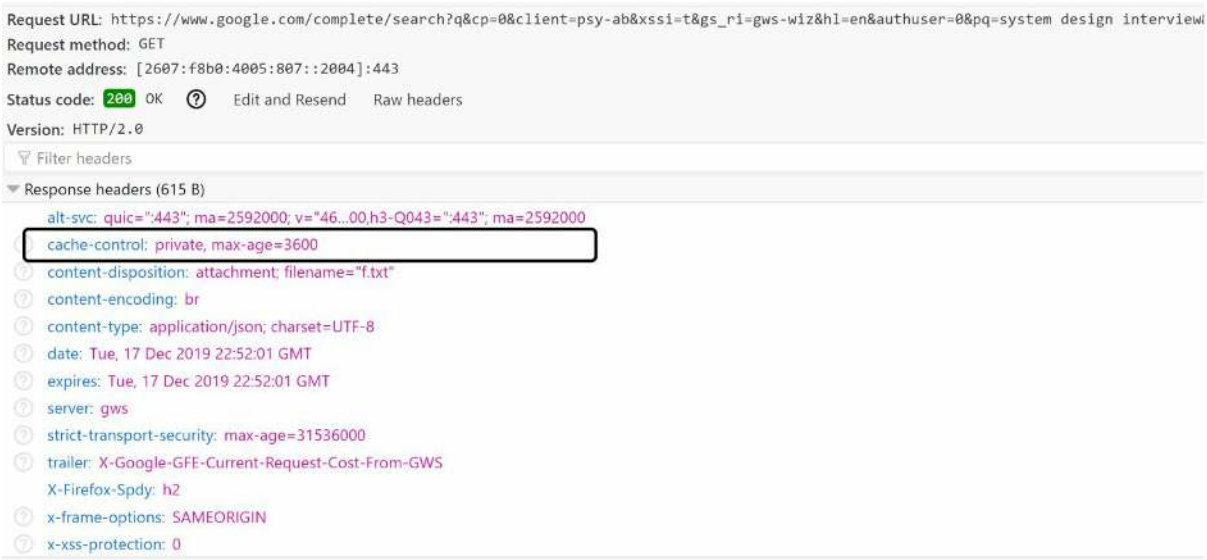

Figure 13-12

• Data sampling: For a large-scale 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* requests 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.

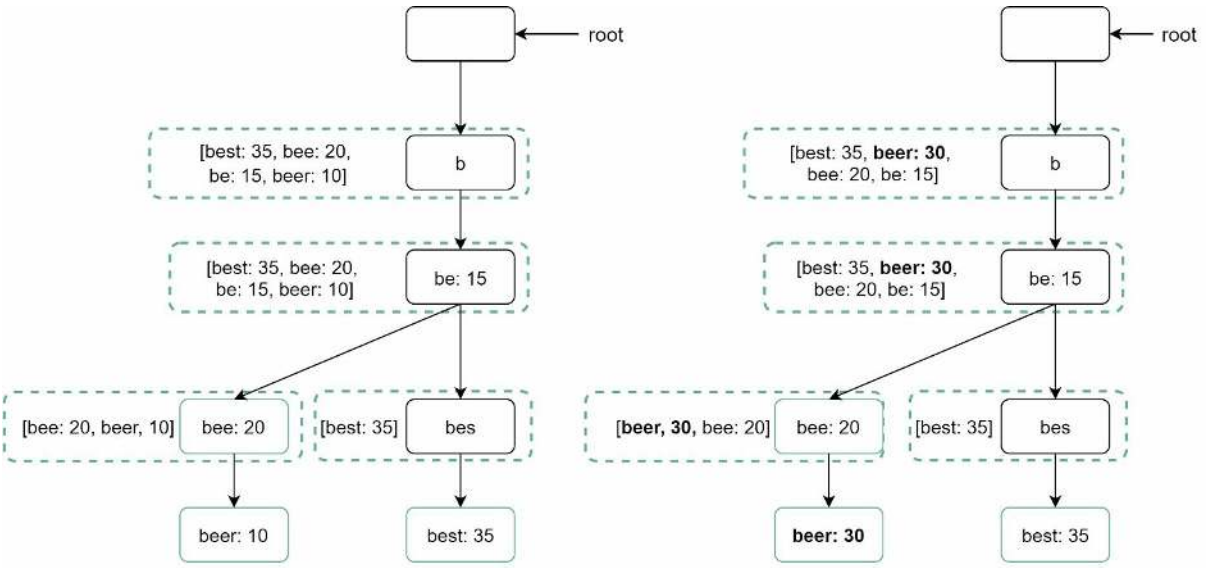

Figure 13-13

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 asynchronically so the correct data set will be used to build trie in the next update cycle.

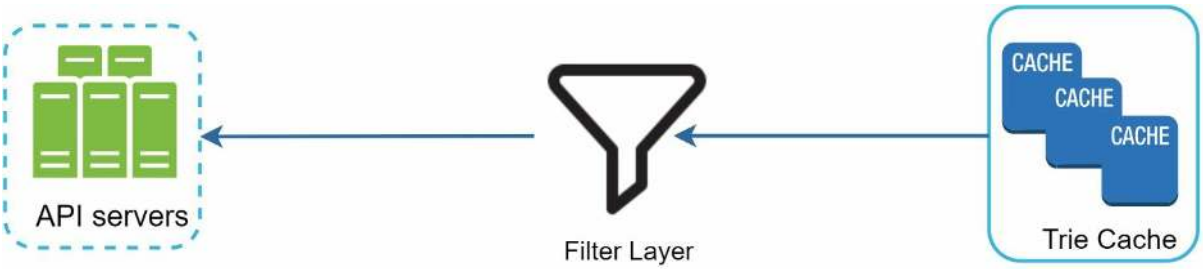

Figure 13-14

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'.

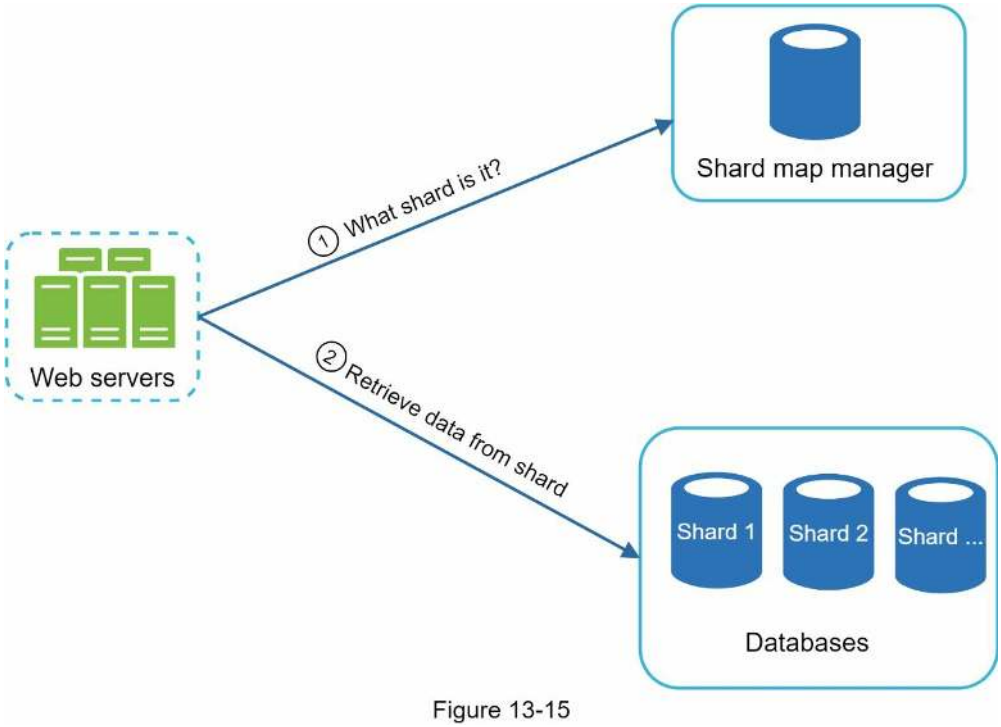

Figure 13-15