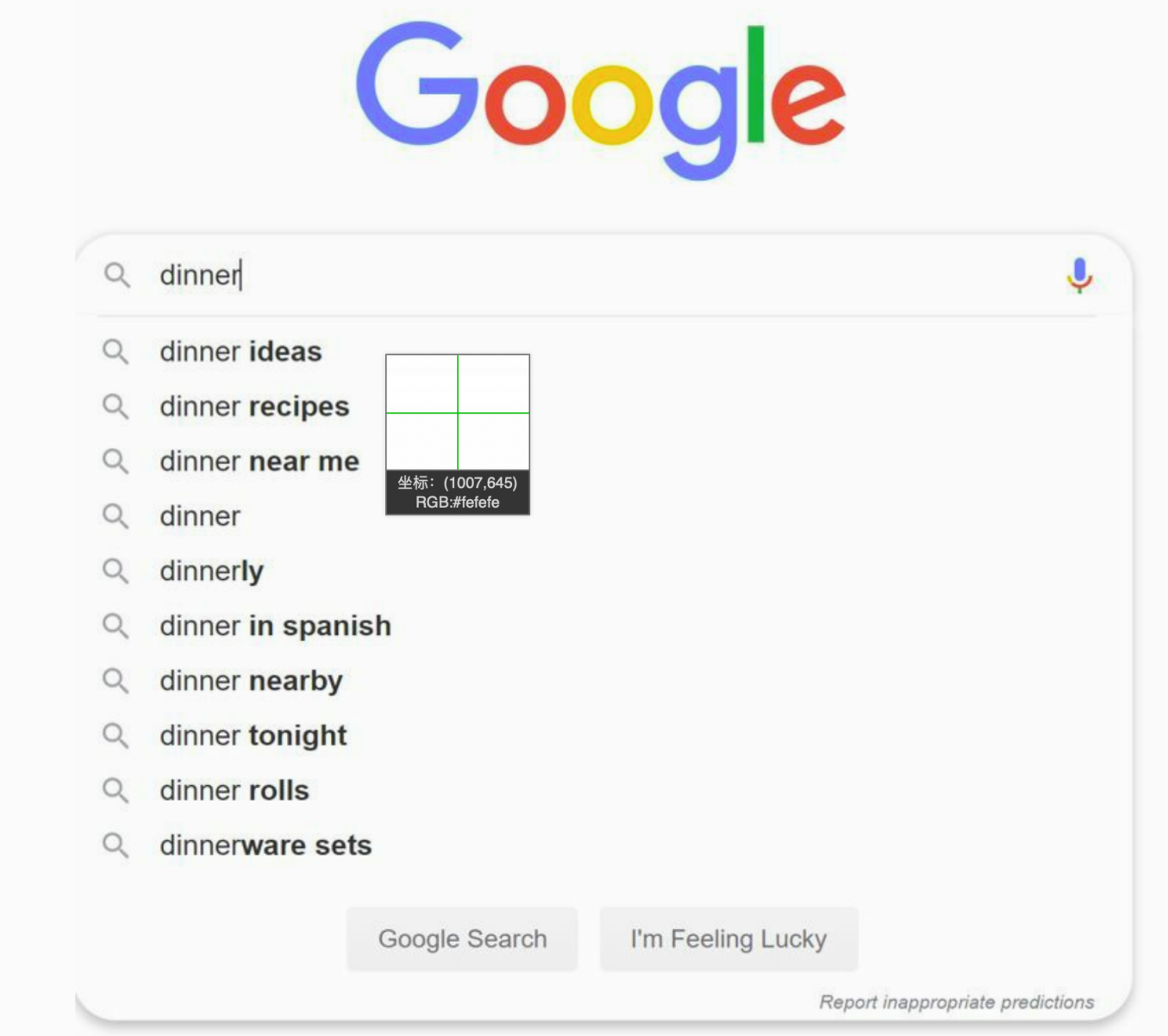
CH13 Design a search autocomplete system



Requirements

- Functional
  - search - query begining - prefix matching
  - 5 suggestions per query
  - ranked by popularity / historical query frequency
  - no spell check
  - English only, extensible to multi-languge

    - only low-case
- Non-functional
    - 10m DAU
    - latency - 100 ms

Back of the envelope estimation

• Assume 10 million daily active users (DAU).

• An average person performs 10 searches per day.

• 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 x 5 = 20 bytes per query.

• For every character entered into the search box, a client sends a request to the backend for autocomplete suggestions. On average, 20 requests are sent for each search query. For example, the following 6 requests are sent to the backend by the time you finish typing “dinner”.

search?q=d

search?q=di

search?q=din

search?q=dinn

search?q=dinne

search?q=dinner

• ~24,000 query per second (QPS) = 10,000,000 users * 10 queries / day * 20 characters / 24 hours / 3600 seconds.

• Peak QPS = QPS * 2 = ~48,000

• Assume 20% of the daily queries are new. 10 million * 10 queries / day * 20 byte per query * 20% = 0.4 GB. This means 0.4GB of new data is added to storage daily.

High-level Design

- Data gathering service - gather user input queries and aggregates them in real-time
    - table - store `query: freq`
    - Not scalable to large dataset: `select ... order by freq desc limit 5`
    - Optimize
        - Trie data structure
            - root - empty string

- each node has 26 character as children
- candidates store as leafs

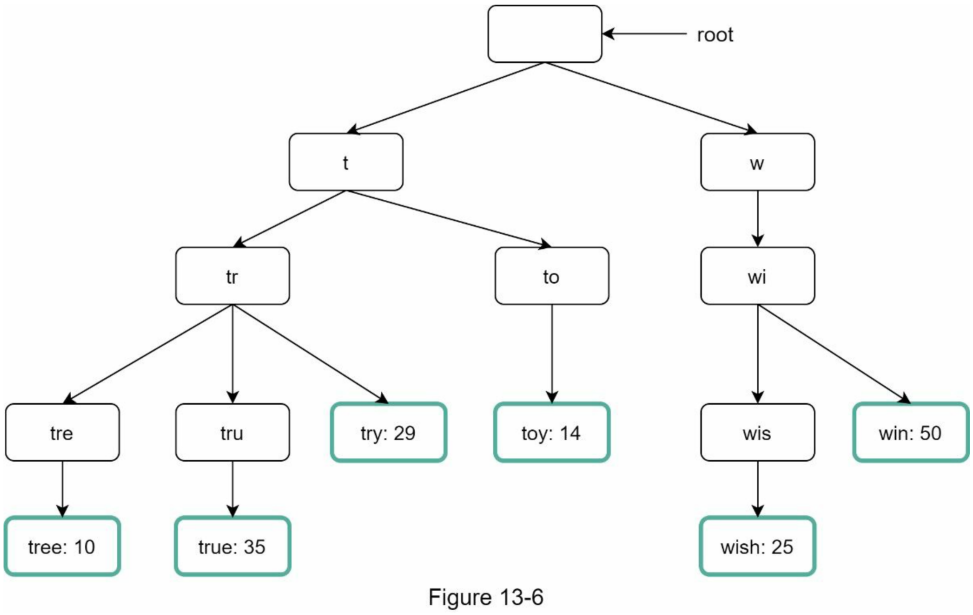


Figure 13-6

- Operations
  - find prefix: O(p)
  - get all valid children: O(c)
  - Sort the children and get topk: O(c log c)

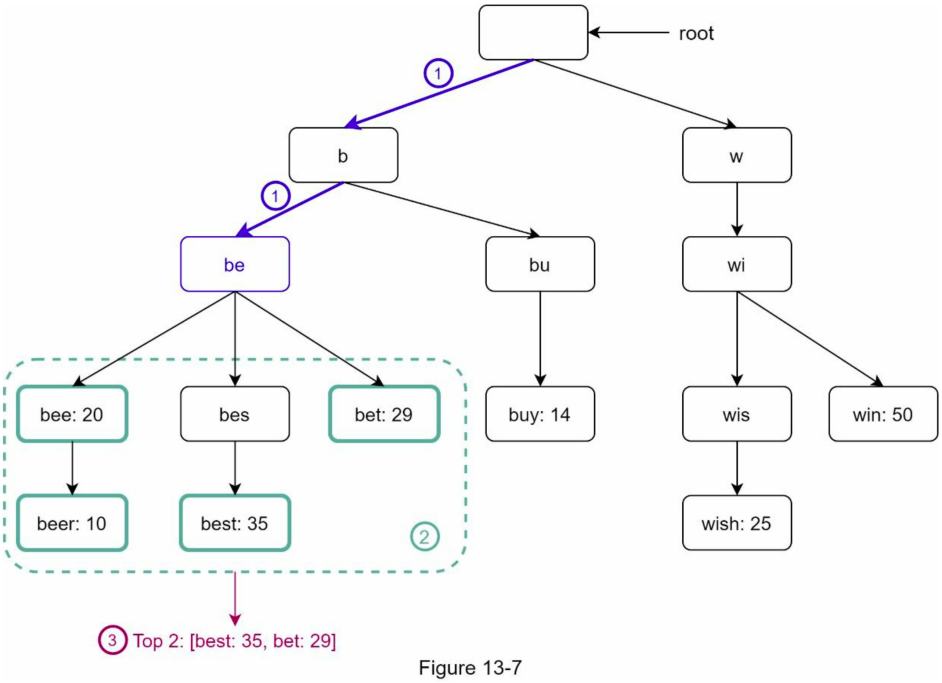


Figure 13-7

  - Slow - due to traverse entire trie to get top k results
  - optimize:
    - limit max length of prefix - reduce O(p) to constant O(1)
    - cache top search queries

- trade space for time

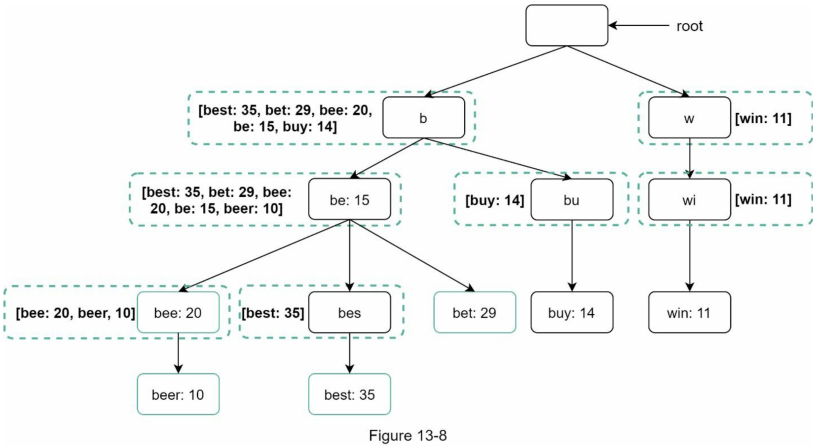


Figure 13-8

- After optimization
  - find prefix: O(1)
  - get candidates: O(1)

- Data gathering service • Query service
- Scale the storage
- Trie operations

- Data gathering service

Figure 13-9

- Aggregator - aggregate data by time interval - by week
- Workers - async jobs to build trie and store in DB
- Trie Cache - snapshot of TrieDB weekly
- TrieDB -
  - document store -mongodb
    - serialize snapshot of trie weekly
  - k, v store
    - every prefix in trie is mapped to a key in hash table
    - data on each trie node as value

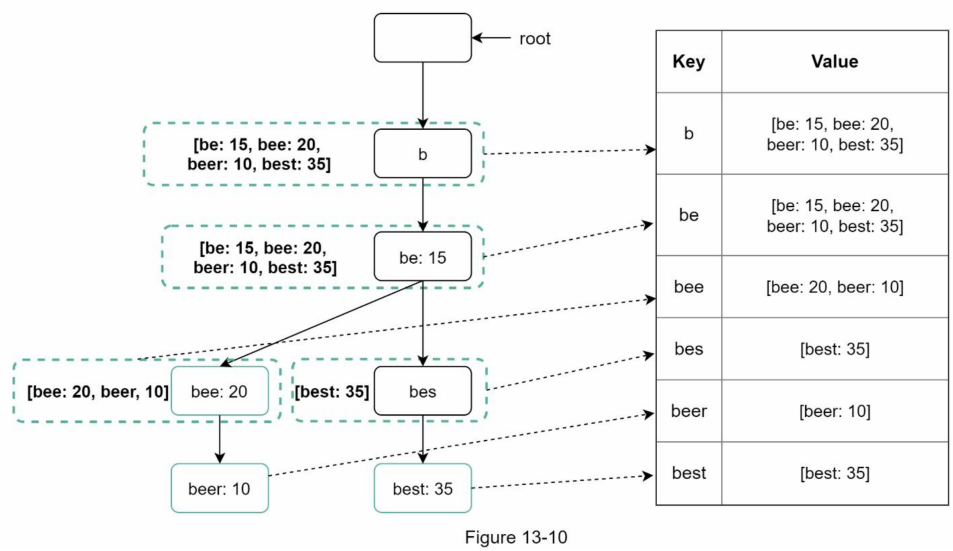
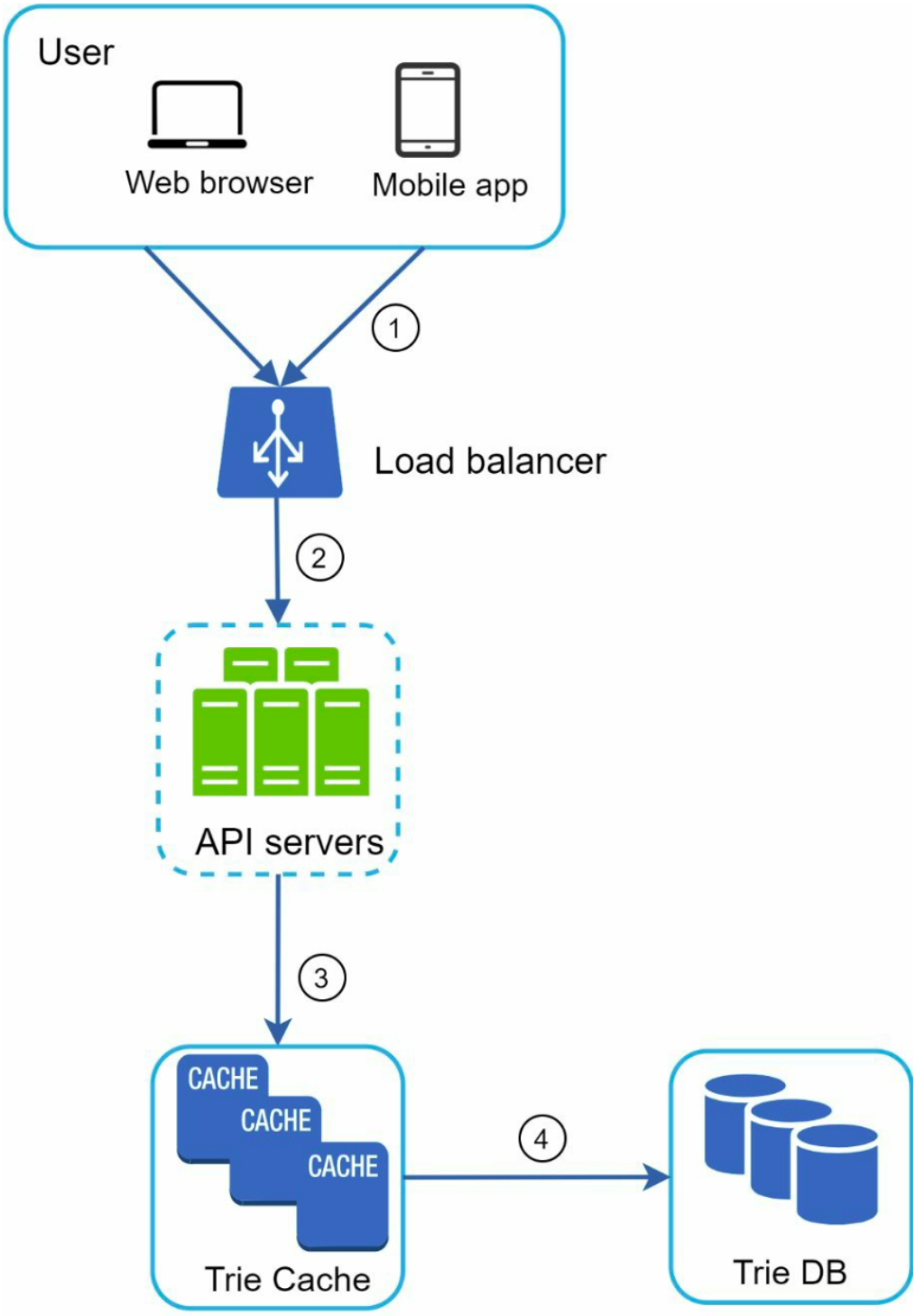


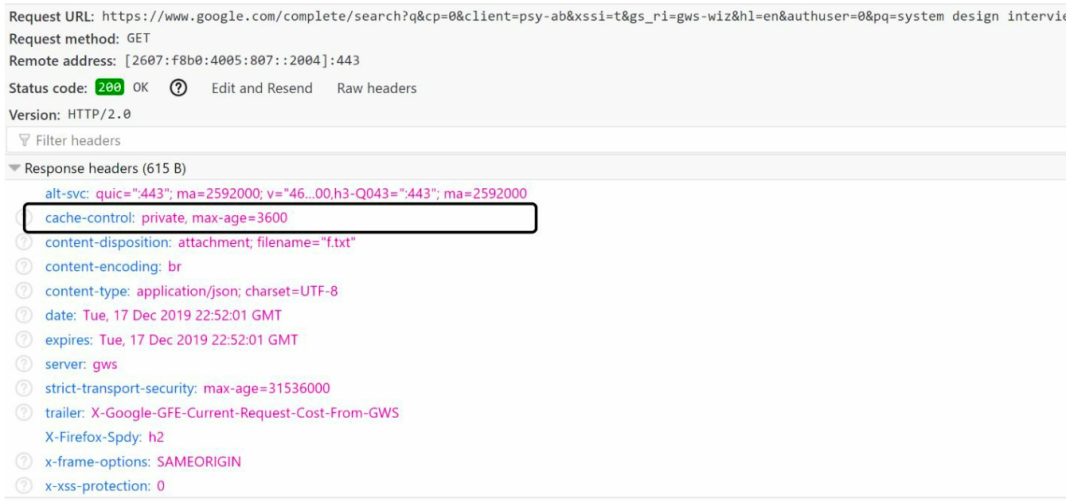
Figure 13-10

- ? how to get top k by frequency with prefix matching using hash
  - redis support sorted set: hash table + skip list

- Query service - Given a search query or prefix, return 5 most frequently searched terms.



-
- Optimizations
  - AJAX request without reload page
  - Browser caching



- Trie Operations
  - Create
  - Update
    - Option 1: weekly update whole trie
    - Option 2: update individual trie node directly
      - try to avoid as it's slow, only do if trie size is small, else need to update all ancestors
  - Delete
    - filter layer
      - remove asynchronically and physically in db
- Storage scale-up
  - Shard by alphabetics
  - Merge low frequent nodes

Extend

- multi-linguistics - unicode
- trending query for candidates
  - reduce working data set by sharding
  - assign more weight to recent search query - exponentially
  - do not need to access all data at once, use streaming data
    - (Steaming, realtime) : Analytics Log -> Producer/Broker -> Kafka Topic / Queue -> Flink / Worker / Consumer -> DB
    - Batch (non realtime) : DB -> Spark / Cron job-> DB

Trie Implementation

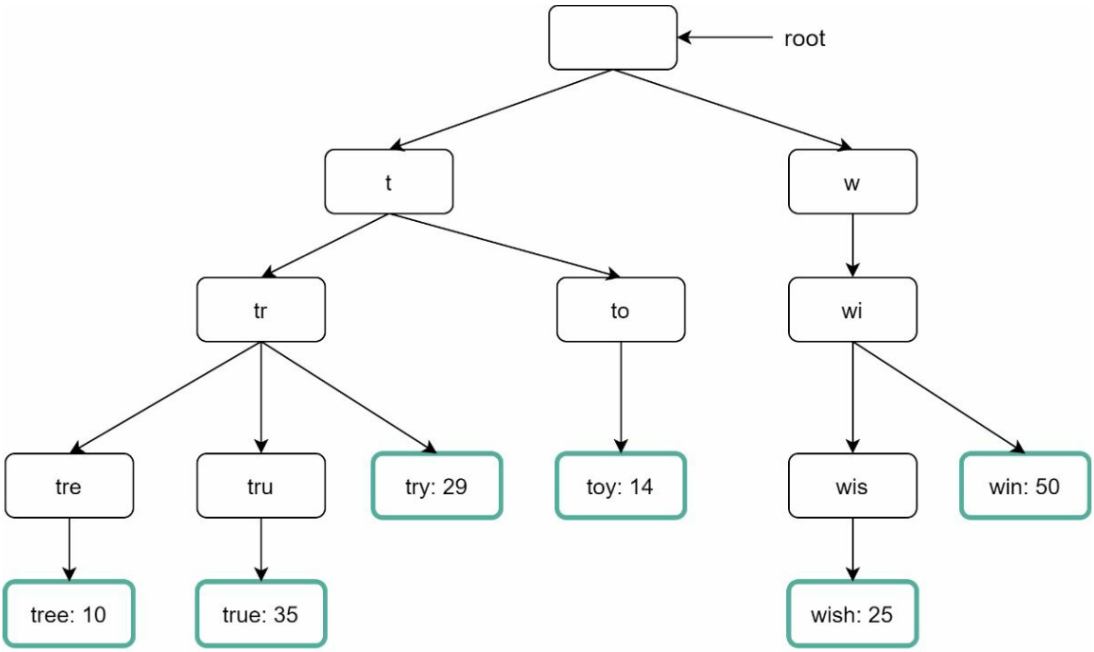


Figure 13-6

- tree
- root: empty string ""
- each node has 26 children

""
| \
"a" "b"
|
"aa"

key: prefix
value: `[(leaf_child, freq)...]`

- sorted by freq

to limit memory usage, we can limit length of list, e.g.: len = 5

hash table: easy to update
key: prefix
value: `[(leaf_child, freq)...]`

better?

- redis
  - sorted set: hash table + skip list

Linked List:
head -> node1 -> node2 -> null
O(n) to search

Skip-list: O(logn)

aa ------------ > ad ------------> ah
| |
aa -------ac ----->ad 7 ----------|
| |
1 ----2---3

ba ----

```
layers = [
  [aa,          ad,           ah],
    |            |             |
  [aa, ab, ac, ad, ...]
]
```

```
class TrieNode:
    def __init__(self):
        self.children = {}  # Dictionary to store child nodes
        self.is_end_of_word = False  # Indicates if this node is the end of
a word


class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word: str) -> bool:
        node = self.root
        for char in word:
```

```
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

    def starts_with(self, prefix: str) -> bool:
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True

    def _autocomplete_helper(self, node, prefix, suggestions):
        if node.is_end_of_word:
            suggestions.append(prefix)
        for char, child_node in node.children.items():
            self._autocomplete_helper(child_node, prefix + char,
suggestions)

    def autocomplete(self, prefix: str) -> list:
        node = self.root
        suggestions = []

        # Traverse the Trie to the end of the prefix
        for char in prefix:
            if char not in node.children:
                return []  # If prefix is not in Trie, return an empty list
            node = node.children[char]

        # Gather all autocomplete suggestions
        self._autocomplete_helper(node, prefix, suggestions)
        return suggestions
```

usage:

```
trie = Trie() # Insert words into the Trie words = ["apple", "app",
"application", "apt", "banana", "band", "bandana"] for word in words:
trie.insert(word) # Search for a word print(trie.search("app")) # Output:
True print(trie.search("apple")) # Output: True print(trie.search("banana"))
# Output: True print(trie.search("bandana")) # Output: True
print(trie.search("bandage")) # Output: False # Check if any words start
with a prefix print(trie.starts_with("ban")) # Output: True
print(trie.starts_with("cat")) # Output: False # Autocomplete suggestions
```

```
print(trie.autocomplete("app")) # Output: ['apple', 'app', 'application']
print(trie.autocomplete("ban")) # Output: ['banana', 'band', 'bandana']
print(trie.autocomplete("cat")) # Output: []
```

Auto-completion with typo

Demo

- https://github.com/nullpointer0xffff/autocomplete_demo