13장 - 검색어 자동완성 시스템

- 가장 많이 이용된 검색어 k개를 자동완성하여 출력하는 시스템을 설계해본다.

1단계 : 문제 이해 및 설계 범위 확정

지원자: 사용자가 입력하는 단어는 자동완성될 검색어의 첫 부분이어야 하나요?
아니면 중간 부분이 될 수도 있습니까?

면접관: 첫 부분으로 한정하겠습니다.

지원자: 몇 개의 자동완성 검색어가 표시되어야 합니까?

면접관: 5개 입니다.

지원자: 자동완성 검색어 5개를 고르는 기준은 무엇입니까?

면접관: 질의 빈도에 따라 정해지는 검색어 인기 순위를 기준으로 삼겠습니다.

지원자: 맞춤법 검사 기능도 제공해야 합니까?

면접관: 아뇨. 맞춤법 검사나 자동수정은 지원하지 않습니다.

지원자: 질의는 영어입니까?

면접관: 네. 하지만 시간이 허락한다면 다국어 지원을 생각해도 좋습니다.

지원자: 대문자나 특수 문자 처리도 해야 합니까?

면접관: 아뇨. 모든 질의는 영어 소문자로 이루어진다고 가정하겠습니다.

지원자: 얼마나 많은 사용자를 지원해야 합니까?

면접관: DAU 기준으로 천만(10million) 명입니다.

요구사항

- **빠른 응답 속도**
  - 사용자가 검색어를 입력함에 따라 자동완성 검색어도 충분히 빨리 표시되어야 한다.
  - 페이스북 검색어 자동완성 시스템에 관한 문서[1]를 보면 시스템 응답속도는 100밀리초 이내여야 한다.
- **연관성**
  - 자동완성되어 출력되는 검색어는 사용자가 입력한 단어와 연관된 것이어야 한다.
- **정렬**
  - 시스템의 계산 결과는 인기도(popularity) 등의 순위 모델에 의해 정렬되어 있어야 한다.
- **규모 확장성**
  - 시스템은 많은 트래픽을 감당할 수 있도록 확장 가능해야 한다.
- **고가용성**
  - 시스템의 일부에 장애가 발생하거나, 느려지거나,
    예상치 못한 네트워크 문제가 생겨도 시스템은 계속 사용 가능해야 한다.

개략적 규모 추정

- DAU는 천만 명
- 평균 한 사용자당 매일 10건의 검색 수행
- 질의할 때마다 `평균 20바이트의` 데이터 입력
  - 문자 인코딩 방법으로 ASCII사용, (`1문자 = 1바이트`)
  - 질의문은 평균 `4개 단어` , 각 단어는 평균 `5 글자` 로 구성 (`4 * 5 = 20바이트`)
- **검색창에 글자를 입력할 때마다** 클라이언트는 검색어 자동완성 **백엔드에 요청**을 보냄
  - 즉, **평균 1회 검색당 20건의 요청이 백엔드로** 전달
  - ex) 검색창에 dinner라고 하면 다음 같이 요청이 백엔드로 전달 됨

```
search?q=d
search?q=di
search?q=din
search?q=dinn
search?q=dinne
search?q=dinner
```

- 대략 초당 24,000건의 QPS 발생 (= `10,000,000 사용자 * 10질의 / 일 * 20자 / 24시간 / 3600초`)
- 최대 QPS = QPS * 2 = 대략 48,000
- 질의 가운데 20% 정도는 신규 검색어라고 가정
  - 따라서 대략 0.4GB 정도(= 10,000,000 사용자 * 10질의 / 일 * 20자 * 20%)
  - 매일 0.4GB의 신규 데이터가 시스템에 추가 됨

2단계 : 개략적 설계안 제시 및 동의 구하기

- 시스템은 두 부분으로 나뉘며 개략적으로 보면 다음과 같다.
  - **데이터 수집 서비스(data gathering service)**
    - 사용자가 입력한 질의를 실시간으로 수집하는 시스템
    - 데이터가 많은 애플리케이션에 실시간 시스템은 바람직하지 않지만 설계안의 출발점으론 좋음
    - 이 책에서는 상세 설계안을 준비할 때 보다 **현실적인 안으로 교체**한다.
  - **질의 서비스(query service)**
    - 주어진 질의에 다섯 개의 인기 검색어를 정렬해 내놓는 서비스이다.

데이터 수집 서비스

- 질의문과 사용빈도를 저장하는 빈도 테이블(frequency table)이 있다고 가정한다.
- 처음 해당 테이블인 비어 있고,
  사용자가
  `twitch`, `twitter`, `twiter`, `twillo`를 순서대로 검색하면 다음과 같이 바뀌게 된다.

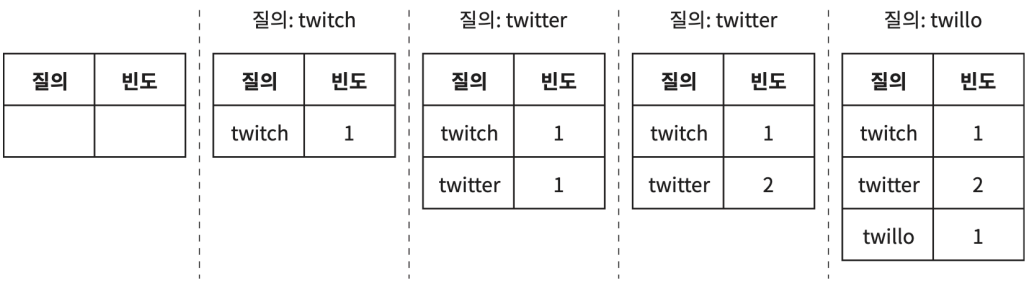

그림 13-2

질의 서비스

- 다음 그림과 같이 빈도 테이블에 두 개의 필드가 존재한다.

query	freuqency
twitter	35
twitch	29
twilight	25
twin peak	21
twitch prime	18
twitter search	14
twillo	10
twin peak sf	8

표 13-1

  - `query` : 질의문을 저장하는 필드
  - `frequency` : 질의문이 사용된 빈도를 저장하는 필드
- 위와 그림과 같은 상태라면 `tw`를 검색창에 입력하면 아래와 같이 top 5 자동완성 검색어가 표시되어야 한다.

그림 13-3

- 이러한 쿼리는 다음과 같은 SQL 문을 사용해 구할 수 있다.

```
SELECT * FROM frequency_table
WHERE query LIKE `prefix%`
ORDER BY frequerncy DESC
LIMIT 5;
```

- 이러한 설계는 데이터 양이 적을 때는 괜찮지만, 데이터가 많아졌을 때 데이터베이스 병목이 될 수 있다.
  - 상세 설계에서 이 문제를 해결한다.

3단계 : 상세 설계

- 컴포넌트를 몇 개 골라 보다 상세시 설계하고 다음 순서로 최적화 방안을 논의한다.
  - **트라이(trie) 자료구조**
  - **데이터 수집 서비스**
  - **질의 서비스**
  - **트라이 연산**
  - **규모 확장이 가능한 저장소**

트라이 자료구조

- 이번 절에서 다루는 트라이의 핵심 아이디어의 상당 부분은 [2]와 [3]에서 차용했다.
- 트라이 자료구조의 핵심 아이디어는 다음과 같다.
  - 트리 형태의 자료구조
  - 이 트리의 루트 노드는 빈 문자열을 나타낸다.
  - 각 노드는 글자(character) 하나를 저장하며, 26개(해당 글자 다음에 등장할 수 있는 모든 글자의 개수)의 자식 노드를 가질 수 있다.
  - 각 트리 노드는 하나의 단어, 또는 접두어 문자열(preifx string)을 나타낸다.
- 다음 그림은 `tree`, `try`, `true`, `toy`, `wish`, `win`이 보관된 트라이이다.

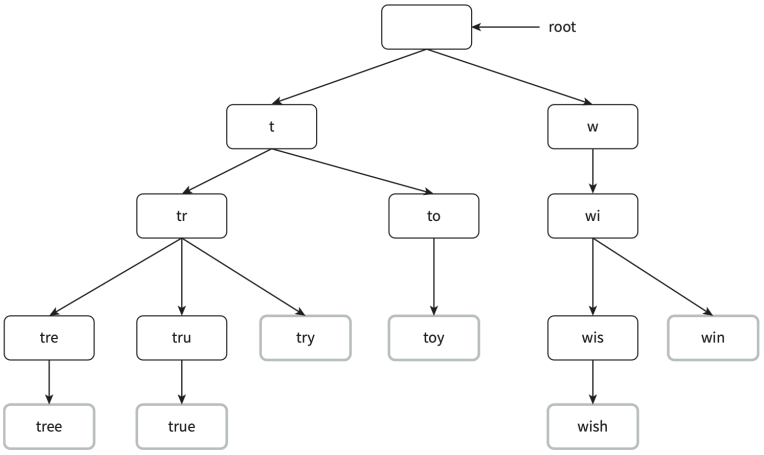

그림 13-5

  - 만약 다음과 같은 빈도 테이블이 있다면 트라이 노드는 오른쪽 그림처럼 된다.

query	frequency
tree	10
try	29
true	35
toy	14
wish	25
win	50

표 13-2

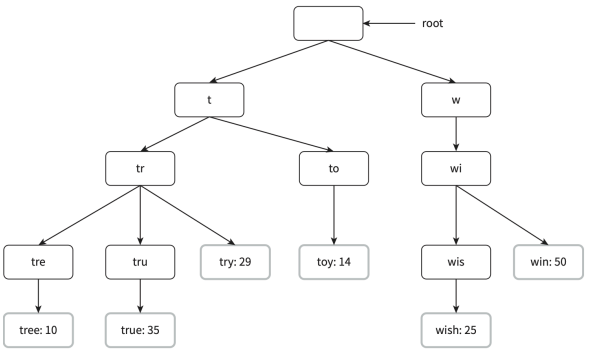

그림 13-6

트라이 알고리즘을 살펴보기 전 용어 정리

- `p` : 접두어(prefix)의 길이
- `n` : 트라이 안에 있는 노드 개수
- `c` : 주어진 노드의 자식 노드 개수

질의어 k는 다음과 같이 찾을 수 있다.

- 해당 접두어를 표현하는 노드를 찾는다. (시간 복잡도 : `O(p)`)
- 해당 노드부터 시작하는 하위 트리를 탐색하여 모든 유효 노드를 찾는다.
  - 유효한 검색 문자열을 구성하는 노드가 유효 노드 (시간 복잡도 : `O(c)`)
- 유효 노드들을 정렬하여 가장 인기 있는 검색어 k개를 찾는다. (시간 복잡도 : `O(clogc)`)

ex) k = 2 검색창에 `be`를 입력했을 때

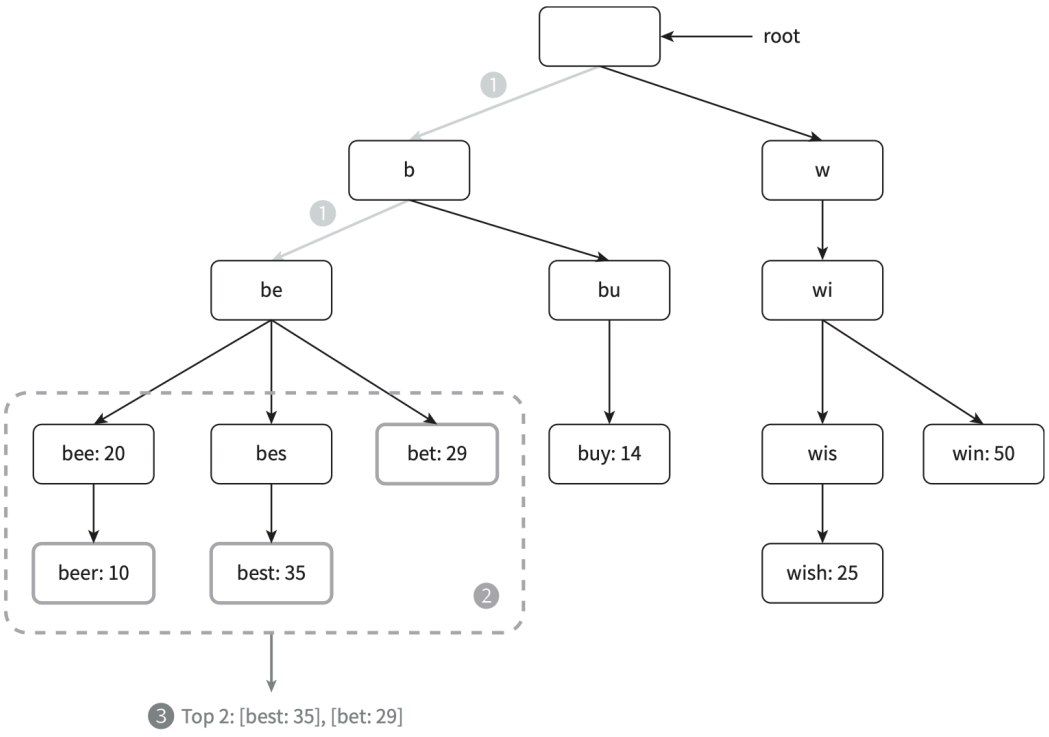

그림 13-7

- 이 알고리즘의 시간 복잡도는 위의 각 단계에서 소요된 시간의 합이다.

- 즉, `O(p)` + `O(c)` + `O(clogc)`
- 알고리즘 자체는 직관적이지만 최악의 경우 k개 결과를 얻으려고 전체 트라이를 다 검색하는 일이 생김

위 알고리즘의 단점을 해결하는 두 가지 방법

▼ 접두어의 최대 길이 제한

- 사용자가 검색창에 긴 검색어를 입력하는 일은 거의 없다.
  - p값을 작은 정숫값(ex. 50)이라고 가정해도 안전
- 최대 길이를 제한할 수 있다면 접두어 노드를 찾는 단계의 시간 복자도는 `O(p)`에서 `O(1)`로 변경

▼ 각 노드에 인기 검색어 캐시

- k개의 인기 검색어를 저장해 두면 전체 트라이를 검색하는 일은 방지할 수 있음
- 5~10개 정도의 자동완성 제안을 표시하면 충분하므로 k는 작은 값
- 이 방식의 단점으로는 **빠른 응답속도를 위해 저장공간을 희생**했다. (캐시 사용으로 인해 저장공간 희생)
- ex) 각 노드에 가장 인기 있는 검색어 다섯 가지를 저장하도록 했음

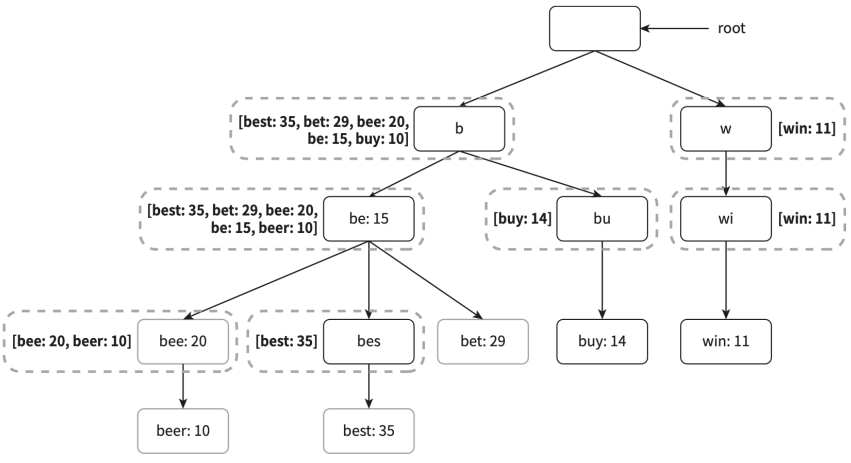

그림 13-8

위 두 가지 최적화 기법을 적용하면 시간 복잡도는 다음과 같이 달라진다.

1. 접두어 노드를 찾는 시간 복잡도는 O(1)

2. 최고 인기 검색어 5개를 찾는 질의의 시간 복잡도도 O(1)
    - 검색 결과가 이미 캐시되어 있기 때문

데이터 수집 서비스

사용자가 검색창에 뭔가 타이핑을 할 때마다 실시간으로 데이터를 수정하면, 다음과 같은 문제로 **실용적이지 않다**.

- 매일 수천만 건의 질의가 일력될 텐데 그때마다 트라이를 갱신하면 질의 서비스는 심각하게 느려질 것이다.
- 트라이가 만들어지고 나면 인기 검색어는 그다지 자주 바뀌지 않을 것이다. (트라이를 자주 갱신할 필요 없음)

규모 확장이 쉬운 데이터 수집 서비스를 만드려면 **데이터가 어디서 오고 어떻게 이용되는지를 살펴야** 한다.

- `트위터` 같은 실시간 애플리케이션이라면 제안되는 검색어를 항상 `신선하게` 유지할 필요가 있지만,

  `구글 검색` 같은 애플리케이션이라면 그렇게 `자주 바꿔줄 이유가 없다`.

용례가 다르더라도 다음 그림처럼 트라이를 만드는 데 쓰이는 데이터는

`분석 서비스나(analytics)`나 `로깅 서비스(logging service)`에서 온다.

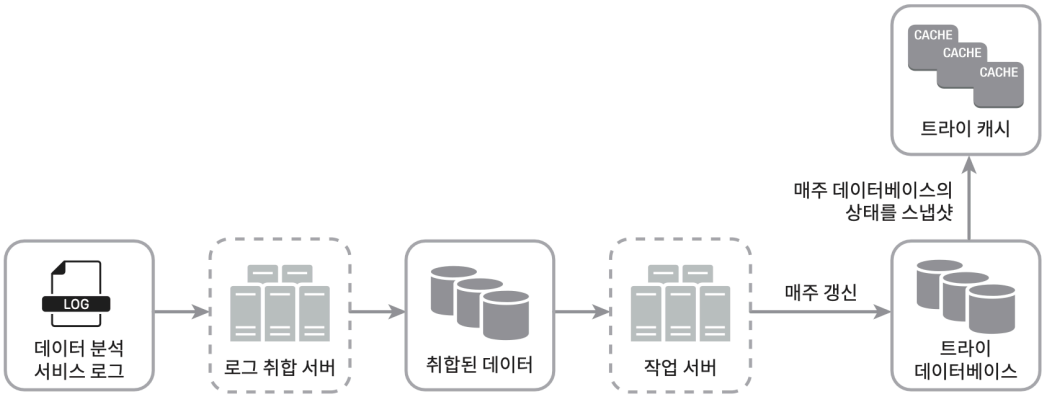

그림 13-9

▼ **데이터 분석 서비스 로그**

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

표 13-3

- 검색창에 입력된 질의에 관한 원본 데이터가 보관
- 새로운 데이터가 추가될 뿐 수정은 되지 않음
- 로그 데이터에는 인덱스를 걸지 않음

▼ **로그 취합 서버**

- 앞 단계인 데이터 분석 서비스로부터 나오는 로그는 양이 많고, 데이터 형식도 제각각이다.
  따라서 이 데이터를 잘 취합하여(aggregation) 우리 시스템이 쉽게 소비할 수 있도록 하면 좋다.
- 데이터 취합 방식은 우리 서비스의 용례에 따라 달라질 수 있음
  - ex)

    트위터와 같은 **실시간 서비스**, (결과를 빨리 보여주는 것이 중요. **데이터 취합 주기를 짧게**)

    구글 검색과 같은 대부분의 서비스는 **일주일에 한 번** 정도로 **데이터(로그)를 취합**해도 충분

▼ **취합된 데이터**

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

표 13-4

- `time` : 해당 주가 시작한 날짜
- `frequency` : 해당 질의가 해당 주에 사용된 횟수의 합

▼ 작업(worker) 서버

- 주기적으로 비동기적 작업을 실행하는 서버 집합
- 트라이 자료구조를 만들고 트라이 데이터베이스에 저장하는 역할 담당

▼ 트라이 캐시

- 분산 캐시 시스템으로 트라이 데이터를 메모리에 유지하여 읽기 연산 성능을 높힌다.
- 매주 트라이 데이터베이스의 스냅샷을 떠서 갱신

▼ 트라이 데이터베이스

- 지속성 저장소로 트라이 데이터베이스를 위해 사용할 수 있는 선택지는 다음의 두 가지가 있음

1. 문서 저장소(document store)
    - 새 트라이를 매주 만들 것이므로, 주기적으로 트라이를 직렬화하여 데이터베이스에 저장 가능
    - MongoDB 같은 문서 저장소를 활용하면 이런 데이터를 편리하게 저장할 수 있음
2. 키-값 저장소
    - 트라이는 아래 로직을 적용하면 해시 테이블 형태로 변환 가능
    - 트라이에 보관된 모든 접두어를 해시 테이블 키로 변환

- 다음 그림처럼 각 트라이 노드에 보관된 모든 데이터를 해시 테이블 값으로 변환

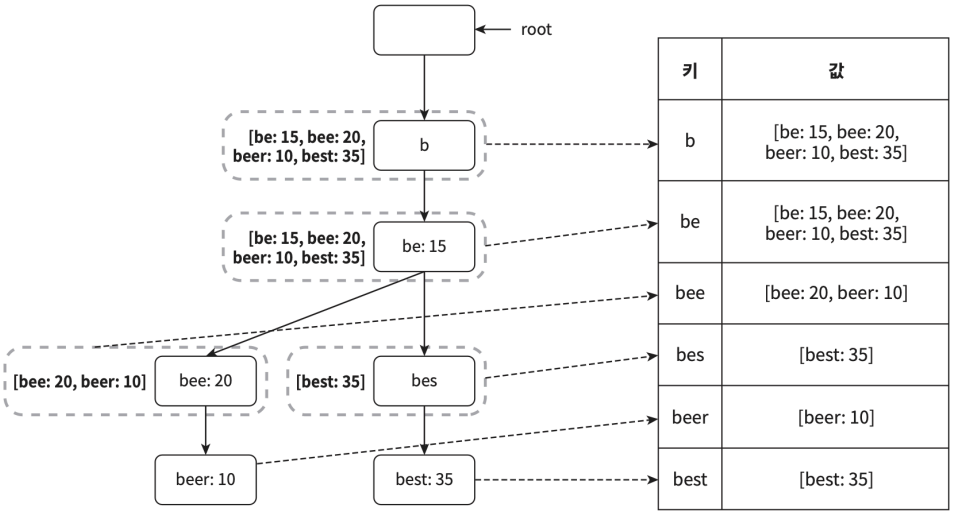

그림 13-10

- 위 그림처럼 각 트라이 노드는 하나의 <`키, 값`> 쌍으로 변환

질의 서비스

- 처음 설계에서 비효율성을 개선된 `새 설계안`의 그림

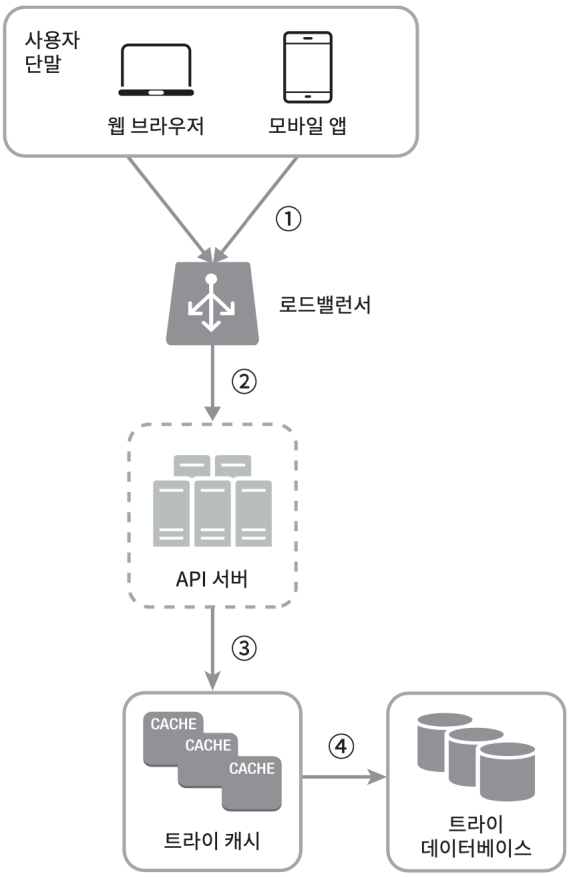

그림 13-11

1. 검색 질의가 로드밸런서로 전송
2. 로드밸런서는 해당 질의를 API 서버로 전송
3. API 서버는 트라이 캐시에서 데이터를 가져와 해당 요청에 대한 자동완성 검색어 제안 응답 구성
4. 데이터가 트라이 캐시에 없는 경우 데이터베이스에서 가져와 캐시에 채움

> **질의 서비스는 빨라야 한다**. 이를 위한 **최적화 방안**은 다음과 같다.

- **AJAX 요청**
  - 웹 애플리케이션의 경우 브라우저는 보통 AJAX 요청을 보내어 자동완성된 검색어 목록을 가져옴
  - 이 방법을 사용하면 요청을 보내고 받기 위한 페이지를 새로고침 할 필요가 없음
- **브라우저 캐싱(browser caching)**

- 대부분 자동완성 검색어 결과 목록이 짧은 시간 안에 바뀌지 않는다.
  따라서, 제안된 검색어들을 브라우저 캐시에 넣어두면 후속 질의의 결과는 훨씬 빨라지게 된다.
- **구글 검색 엔진이 이런 캐시 매커니즘을 사용**한다.

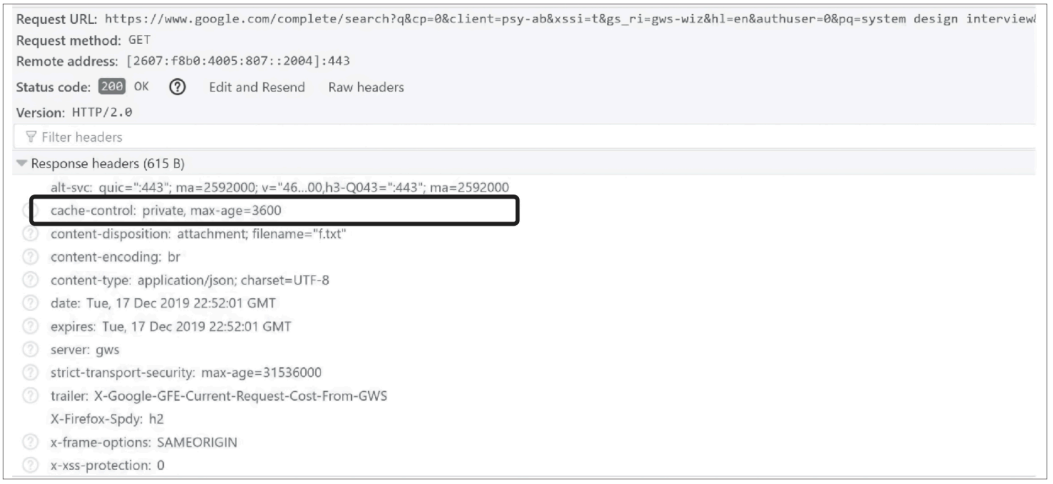

그림 13-12

system design interview라고 입력했을 때 응답 헤더 (제안된 검색어를 한 시간 동안 캐시 함)

- `cache-control: private` :
  해당 응답이 요청을 보낸 사용자의 캐시에만 보관될 수 있으며, 공용 캐시에 저장되면 안 된다는 뜻
- `max-age = 3600` : 캐시 항목으로 3600초 동안 유요하다는 뜻이다. (한 시간)

- **데이터 샘플링(data sampling)**
  - N개 요청 가운데 1개만 로깅하는 방식 (모든 질의 결과를 로깅하지 않음)

트라이 연산

- 검색어 자동완성 시스템의 **핵심 컴포넌트**
- 트라이 관련 연산들이 어떻게 동작되는지 살펴보자.

트라이 생성

- 작업 서버가 담당하며, 데이터 분석 서비스의 로그나 데이터베이스로부터 취합된 데이터 이용

트라이 갱신의 두 가지 방법

1. **매주 한 번** 갱신하는 방법 : 새로운 트라이를 만든 다음에 기존 트라이 대체
2. 트라이의 **각 노드를 개별적으로 갱신**하는 방법
   - 이 책에서는 사용하지 않은 방식 (**성능이 좋지 않음**, 트라이가 작을 때는 고려해도 괜찮음)
   - 트라이 노드를 갱신할 때 상위 노드에도 인기 검색어 질의 결과가 포함되기 때문에 갱신에 필요한 모든 상위 노드(ancestor)도 갱신해야 한다. 다음 그림은 이 갱신 연산이 어떻게 동작하는지 보여준다.

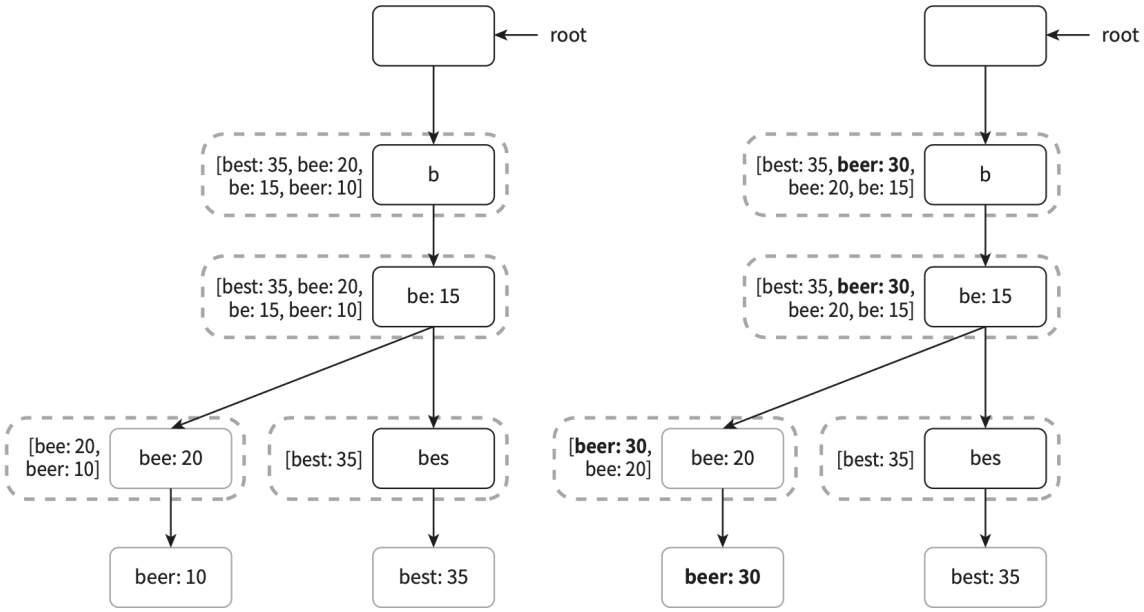

그림 13-13

beer 의 검색 빈도를 10 → 30 갱신하는 상황

검색어 삭제

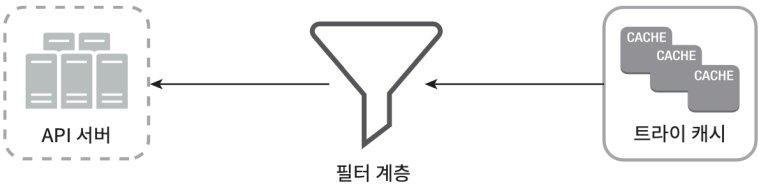

그림 13-14

- 트라이 캐시 앞에 필터 계층을 두고 부적절한 질의어가 반환되지 않도록 할 수 있다.
  - 필터 계층을 두면 필터 규칙에 따라 검색 결과를 자유롭게 변경할 수 있다는 장점이 있음
- 데이터베이스에서 해당 검색어를 물리적으로 삭제하는 것은 다음번 업데이트 사이클에 비동기적으로 진행

규모 확장이 가능한 저장소

- 일단 **영어만 지원**되기 때문에 간단하게 `첫 글자를 기준으로 샤딩`하는 방법을 생각할 수 있음
  - 검색어를 보관하기 위해 두 대 서버가 필요하다면 `a` 부터 `m`까지 글자로 시작하는 검색어는 첫 번째 서버에 저장하고, 나머지는 두 번째 서버에 저장한다.
  - 세 대 서버가 필요하다면 `a`부터 `i`까지는 첫 번째 서버에, `j`부터 `r`까지는 두 번째 서버, 나머지는 세 번째 서버에 저장한다.
  - 이 방법을 쓰는 경우 가능한 서버는 최대 26대로 제한된다. (영어 알파뱃의 최대 글자는 26)
    - 따라서 **이 이상 서버 대수를 늘리려면 샤딩을 계층적**으로 해야 됨
  - 이 방식이 괜찮아 보이지만, `c`로 시작하는 단어가 `x`로 시작하는 단어보다 많다. 즉, 데이터를 각 서버에 균등하게 배분하기 어렵다.
  - 따라서 이 책에서는 다음 그림처럼 과거 질의 **데이터의 패턴을 분석하여 샤딩하는 방식**을 **제안**한다.
- 과거 질의 **데이터의 패턴을 분석하여 샤딩하는 방식**

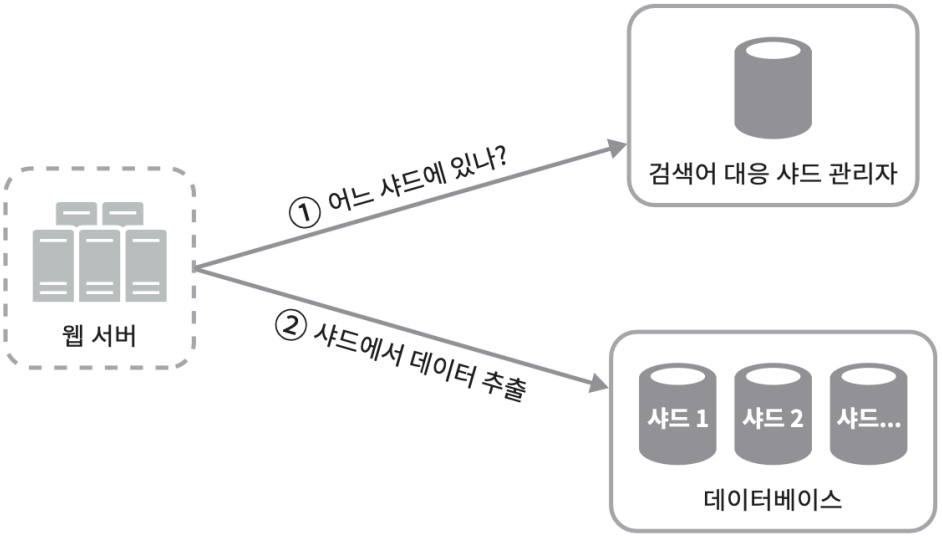

그림 13-15

- 위 그림에서 `검색어 대응 샤드 관리자(shard map manager)`는
  어떤 검색어가 어느 저장소 서버에 저장되는지에 대한 정보를 관리한다.
- ex) `s`로 시작하는 검색어의 양이 `u`, `v`, `w`, `x`, `y`, `z`로 시작하는 검색어를 전부 합친 것과 비슷하다면 `s`**에 대한 샤드 하나**와 `u`**부터** `z`**까지**의 검색어를 위한 **샤드 하나**를 두어도 충분하다.

4단계 : 마무리

추가적으로 생각

- `다국어 지원`이 가능하도록 시스템을 확장하려면?
  - 트라이에 `유니코드` 데이터를 저정해야 한다.
    - **유니코드** : 세상에 존재하는 모든 문자 체계를 지원한느 표준 인코딩 시스템
- `국가별`로 인기 검색어 **순위**가 다르다면?
  - 국가별로 다른 트라이를 사용하면 됨
  - 트라이를 CDN에 저장하여 응답속도를 높이는 방법도 있음
- `실시간`으로 변하는 검색어의 추이를 반영하려면?
  - 다음과 같은 이유로 **현재 설계는 실시간을 다루기 적합하지 않다**.
    - 작업 서버가 매주 한 번씩만 돌도록 되어 있어 적절하게 트라이를 갱신하지 못함

        - 설사 때맞춰 서버가 실행되더라도 트라이를 구성하는 데 너무 많은 시간이 걸림
- 실시간에 대해서는 실제 설계는 못 하고 다음과 같은 **아이디어를 제시**
    - 샤딩을 통하여 작업 대상 데이터의 양을 줄임
    - 순위 모델(ranking model)를 바꾸어 최근 검색어에 보다 높은 가중치를 주도록
    - 데이터가 스트림 형태로 올 수 있다. (한번에 모든 데이터를 동시에 사용할 수 없을 가능성이 있음)
        - 데이터가 스트리밍 된다는 것은 데이터가 지속적으로 생성된다는 뜻
        - 스트림 프로세싱에는 다음과 같은 특별한 종류의 시스템이 필요
            - 아파치 하둡 맵리듀스(Apache Hadoop MapReduce)
            - 아파치 스파크 스트리밍(Apache Spark Streaming)
            - 아파치 스톰(Apache Storm)
            - 아파치 카프카(Apache Kafka)

Chapter 13: DESIGN A SEARCH AUTOCOMPLETE SYSTEM

1. The Life of a Typeahead Query
2. How We Built Prefixy: A Scalable Prefix Search Service for Powering Autocomplete
3. Prefix Hash Tree An Indexing Data Structure over Distributed Hash Tables
4. MongoDB Wikipedia
5. Unicode FAQs
6. Apache Hadoop
7. Spark Streaming
8. Apache Storm
9. Apache Kafka