HLD-9
Monday, 24 May, 2021 8:34 PM

Start at 9:05 AM

Last class learning

1. Data storage in NoSQL
2. Computation
3. LSM trees of sorted sets

Agenda

1. HLD Case study (Search typeahead)
2. Process of HLD breakdown

Search typeahead System

MIC — Michael Jackson
← Michelle Obama
← Microsoft

1. Processing of HLD problem

1) MVP (Minimum viable product) / Requirement Gathering
functional requirements
→ Minimum requirements
eg. WhatsApp: Sending & receiving messages (FR)
- High throughput (NFR)
- Low latency
Non functional requirements

2) Estimation of Scale
- Latency estimation
- QPS (RPM → request per minute)
→ Queries per second
- Traffic estimation

3) Design Goal

4) API design
eg. post/get/put/delete, minimal
search (search)

5) Detailed Implementation

Search typeahead System

MVP

1. Show suggestions which has input text as "Starts" prefix
   - Not accounting for spelling mistake
   - Strict prefix matching
2. Limit input to first 5 suggestions
3. Top 5 suggestions based on frequency of search term

Functional Requirement →

Non functional requirements:

1. Highly Available
2. High throughput
3. Horizontally scalable
4. Low latency

Minimal requirements

1. Personalized suggestions
2. Spell Check / Autocorrect

Scale estimation

1. Storage estimation
2. QPS
3. Traffic → read heavy / write heavy or mixed workload

100 RPM
10 RPM X 10 = 100 (10x2) : 2 minutes

What users want to about?

Query Heavy → Frequently

New / old

Storage estimation

→ 8 Billion searches per day
20% New, 70% older
→ 2.4 Billion searches per day

→ One record = 100 bytes
= 2.4 Billion X 100 bytes
= 24 GB x 100
= 240 GB / day

Bytes
1000 B = 1 KB
1000 KB = 1000 X 1000 = 10^6 bytes
1000 MB = 1GB = 1000 x 1000 x 1000 = 10^9 B
= 1 Billion B
1000 GB = 1 TB
1000 TB = 1 PB

Not 10 years :-
240 x 10 x 365 = 876 TB
↳ So we may need data sharding

QPS (Queries per second)
(8 Billion)/(12 X 60 X 60) = 93K QPS

Traffic
Mixed Workload (Read + Write Heavy)

Design Goal

P | A | C | E | C

→ Low latency
→ High availability

API design

1. get/put/suggestions (prefix) — (search)
2. increase Frequency (search term) — (search)

Implementation details

1. Key Value Store
   (Search term → counter) key
   Check key's (optional to design)
2. Key-Value Store
   prefix → (Top 5 suggestions)

Read

WRITE

MISSPELL → MSC → C
MISH → C
MISCH → C

1. Periodic Update
   + not lower the number of writes
   − Trends
2. P_1 P_2 P_3 P_4 P_5 ... (Update on multiplication)
   → trending topics

3. SAMPLING → If your queries are truly random, then for frequency not break we need not look at all queries but a random sample
   S_1 / S_2 / S_3 / S_4 ... / S_{100} / S_{200} / S_{300} ..... S_{n^2}

Model

Periodic Update

void recomputeFrequency (search_term) {
  int x = rand() % 100
  if (x == 0) {
    update_freq(search_term, freq)
  }
}
recomputeFrequency();

Work Heavy System

Read Heavy System

gzip CDN → Cache
Zoho
CDN → cache

Probability of a search term $f(x)$ = (occurrence, freq)

Time decay factor
decay factor → Time

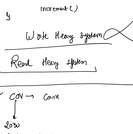
Mishael : D1 : 10
D2 : 8
D3 : 6
D4 : 5

decay factor (d) > 1

d = 2

day-1 Mishael = 100
day-2 = 50 + (50) = 100
day-3 = 50 + 100 = 150
day-4 ⇒ 0

OSI Model (Network)

P / D / N / T / S / P / A

L4 LB → (1)
L7 LB →