System Design II

May 9, 2015

Overview

SDE Interview Overview

Design News Feed

System Design Topics

Design Stats Server

SDE Interview Overview

Skills:

- problem solving (using data structure and algorithm)
- knowledge base (Arch, OS, CC, PL, Net, DB, Web, ML, etc)

Steps:

- resume (your resume can be your friend or enemy)
- phone/screen interview (1-2) (saving engineer's time)
- onsite interview (3-7) (comprehensive)
- offer negotiation (don't be shy)

Questions:

- coding problem
- knowledge probing
- design problem (system design, OO design)
- resume-specific * interviewer-specific
- behavior question

What are interviewers thinking about?

知彼知己，百战不殆。

How interviewers choose questions

- He must know every bit of it (common mistakes, progressive hints).
- Easy to start (even without prior knowledge), lots of followup.
- “deal-breakers”, “hiring-signal-per-second”

How interviewers assess interviewees

- Compare with other interviewees.
- poor/okay/good/great standard.

What do interviewers usually expect from system design questions?

- A working solution with a few lines of **code**. (not just discussion)
  - Don’t need to be compilable. Just express idea through argument/return/functionality.
- Your **logic** and **understanding** of knowledge (not just knowledge)
- Good communication skill. (general background -> specific)

Design News Feed

Design and implement Twitter.

- see the tweets of people you follow.
- follow/unfollow people
- post a tweet.
- comments on a tweet.

Background

Messaging paradigms: (transmission of information)

- P2P
- Broadcast
- Publisher-subscriber (newspapers and readers)

In social network, everyone is content provider as well as content receiver.

What are the design choices here?

- How content is stored/organized.
- How content is distributed.

What are requirement/optimization goal?

- User latency (content retrieving latency.)

Level 1

Database design

- user
- news
- friend
- comment

Database operation

1. get follow list

```
SELECT FollowId FROM friend
WHERE FollowerID = myid;
```

2. get friend's news

```
SELECT * FROM news
WHERE Time > xxx
AND Author IN friendlist
```

UserId	Name	Age
1	Jason	25
2	Michael	26

NewsId	AuthorId	Content	Time
1	2	"hehe"	20150228
2	1	"haha"	20150227

FriendId	FollowerId	FollowId
1	1	2
2	2	1

Problems:

Repeatedly query follow list?

Store each user's follow as a list

Tip: save repeating computations, trade space for latency.

It's costly to check every news' author id if it is in the friend list

Store each user's news id as a list.

Level 2

Querying every follow's news list is still slow.

How about storing my own list of follows' news as well, and get updated when they post?

Tip: we can do some offline computations, for online latency.

This is the choice of Pull and Push.

Latency:

	Refresh (read) (user-facing latency)	Post (write)
Pull	follows' lists	my own list
Push	my own list	followers' lists

Pull: high latency

Push: news delay

Push wins.

Level 3

A problem of Push (and solution), taking heterogenous popularity into account.

- (usually everyone's follow number doesn't vary too much, but followers number varies a lot)
- Celebrities (huge followers)
  - Push will be too slow (update every follower's news list). at the same time, people is less tolerant to celebrities' news delay. => use pull. (just a few celebrities, so latency is okay.)
- => Pull + Push hybrid solution
  - For celebrities' post, don't push.
  - When refresh, query celebrities' news list and merge with my own news list.

Level 4

An improvement for Push, taking heterogeneous user activity into account.

- Some users are more addictive than others.
  - How about push to them with higher priority, and let less frequent users pull. (reduce network traffic costed by useless push => reduce the delay of useful pushes.)

Other things you could mention

1. How to design storage hierarchy?

   Cache frequently accessed news: newer news, celebrities' news, active users' pushed news list.

2. How to partition storage?

   Geographical, graph clustering…

3. How to rank news?

   Time, popularity (number of likes, etc)

Summary: Heterogeneity is a good source of discussion.

   # Followers and user frequency, in Pull/Push tradeoff.

   Memory access frequency, in cache.

   Some functions are more frequently called than others. Can you do more in less-frequently-called functions to reduce the work in hot functions? (refresh is more than post.)

System Design Topics

Computer System Architecture
    Performance
Operating System
    Concurrency
Compiler Construction
Programming Language
    OO design
Database
Networking

Performance

What do you consider aside from Big-O time complexity?

- constant factor (e.g. selection sort/insertion sort)
- how large is N (e.g. quick sort + insertion sort)
- space's implications on performance (memory hierarchy)
- memory access pattern (e.g. top-k problem, linked-list/array)
- bottleneck/hotspot profiling
- trade accuracy for performance (e.g. sliding window average)
- compiler options
- * pipeline, super-scalar, branch prediction
- parallelism (e.g. matrix multiplication)

Latency numbers everyone should know

```
L1 cache reference ..........................  0.5 ns
Branch mispredict ............................    5 ns
L2 cache reference ...........................    7 ns
Mutex lock/unlock ............................   25 ns
Main memory reference ......................  100 ns
Compress 1K bytes with Zippy ............. 3,000 ns  =   3 µs
Send 2K bytes over 1 Gbps network ....... 20,000 ns  =  20 µs
SSD random read ........................ 150,000 ns  = 150 µs
Read 1 MB sequentially from memory ..... 250,000 ns  = 250 µs
Round trip within same datacenter ...... 500,000 ns  = 0.5 ms
Read 1 MB sequentially from SSD* ..... 1,000,000 ns  =   1 ms
Disk seek ........................... 10,000,000 ns  =  10 ms
Read 1 MB sequentially from disk .... 20,000,000 ns  =  20 ms
Send packet CA->Netherlands->CA .... 150,000,000 ns  = 150 ms
```

Design Stats Server

Count the number of events (requests, error, cache hit, etc) in last second, last minute, last hour, last day.

Interface

```
Class SMHDCounter{
  void inc();
  int getLastSecCount();
  int getLastMinCount();
  int getLastHourCount();
  int getLastDayCount();
}
```

Tip: When asked for implementing a server, just implement a function. If necessary, explain a bit: we can export the function as an RPC available to be called by clients via HTTP request.

Initial Thoughts

Create four counters, and increment them on each inc(), separate threads to set per-sec counter to 0 every sec, per-min counter to 0 every min, etc.

Wrong, because we want a sliding window count, not current calendar sec, min, hour count.

Level 1

Keep a list of all timestamps. Count on every query.

Storage: O(query-per-day)

Time: O(query-per-second), O(query-per-hour), etc.

This is accurate, but not scalable if qps is large.

Level 2

If QPS is high, we must sacrifice accuracy for performance.

Count events in per-sec buckets. (granularity: second)

86400 buckets (circular buffer).

- Inc: counter[now_by_second()%86400]++.
- GetLastSecCount: counter[(now_by_second()-1)%86400].
  - not exactly “last” sec.
- GetLastHourCount: **Sum**(counter, (now_by_second()-3600)%86400, (now_by_second()-1)%86400).

Storage: O(second-per-day)

Time: O(second-per-second), O(second-per-hour), etc.

Did you see errors?

You are using circular buffers, full of history data. How would you zero them?

If Inc() is in frequent, e.g. there is no inc() in the 42-th second. your 42-th bucket still has the value of last day's value.

Level 3

What if GetXXXCount() is frequent?

Record accumulated count in buckets.

Keep a total_count.

- Inc: increment the total_count, and write total_count to counter[now_by_second()%86400].
- GetLastHourCount: counter[now_by_second() - 1) % 86400 - (now_by_second() - 3600) % 86400).

Time: O(1).

Level 4

What if memory is constraint?

We can let the bucket size vary.

only 3600 per-sec buckets, but 60*24 per-min buckets.

per-day count's granularity becomes 1 min. (acceptable.)

Level 5

You can apply the idea and design multiple stages of buckets. The earlier, the larger granularity.

Client servers that sent us inc() requests can also do some aggregation, to minimize network traffic.