System Design III

May 10, 2015

Overview

Design Web Crawler

Six Degree of Separation

Design Messaging System

Design Web Crawler

Get all web pages from the entire web.

Used in search engine, web data mining (copyright infringement detection), comparison shopping engine

Good use/bad use.

Working Solution

Select a few seed URLs, add them to “frontier URL list”;

Iteratively process URLs in the “frontier URL list”:

- Add the URL in “visited URL list”;
- Fetch the URL and download the web page;
- Extract URLs referred in that web page;
- Add the extracted URLs that are not in the “visited list” to the “frontier list”;

Think about the scale

10 billion web pages. How to distribute the job if we have more machines?

- Each machine works on their own “frontier URL list”
- But all machine needs to query the “visited URL list”.

What’s the data structure for “visited URLs”?

- HashSet.
- Size: 16 billion web pages * 256 characters(Byte) per URL => 4 TB.
  - Only store fixed size checksums of URLs: 16B per URL => 250 GB.
  - We can store them on disk, and cache the popular link (heterogeneity of webpage popularity)
- We have hope to store them all on memory. How to partition?
  - hash(URL): every node need to talk to other nodes frequently, bad.
  - hash(host name): keep the query local! Each node takes care of a set of host names, good.

How about trade accuracy for performance?

- Use a single bit for a URL’s presence.
  - Collision -> false positive. (false negative is catastrophic)
  - Size: 16 G bits = 2GB. Increase size to reduce the probability of collision.
  - Easy to put on the memories of several nodes.
- Bloom filter: multiple hash functions, reduce the probability of collision.

Politeness

Crawler could easily produce burst requests to a certain website.

In turn, website can create “crawler trap” to waster crawler’s resource.

Similar to virus and anti-virus.

- But benign crawlers bring something that website can take advantage of, e.g. more public exposure.

Solution:

- Limit the request rate to a certain website. (Alternate between hostnames, easy to implement based on host name partition.)
- Respect the robot.txt
- In turn, website owners will keep their sensitive information and “crawler traps” out of robot.txt.

BFS or DFS

Key problem here is the order: stack(FILO) or queue(FIFO).

- Note: the parallel solution we just described is not strictly FIFO.

Queue-based design is better:

- More uniform traffic to any certain website. (websites tend to link to themselves more often than others.)
- Discover popular web pages earlier. (Important web pages are often within a few clicks.)

Six Degree of Separation

Given a social network that describe undirected friendship relation, find a shortest path that connects two persons.

Working solution

Model the social network as a (undirected) unweighted graph.

Shortest path => BFS

Think about the scale

Real world graph:

- Huge, but very flatten. (average shortest path length is around 6.)
- Why? Because there are a few people that have huge connections (act as hubs).

This time, “visited list” is not a big problem.

- You have all of them stored already. Just add a visited bit.

In BFS, your “frontier list” will expand quickly. (you will reach hubs pretty soon.)

- Time: $O(b^d)$. d: depth, b: average branching factor. (=O(V+E))

You can expand from start and end bi-directionally.

- Once their “frontier lists” intersect, you find the path.
- Time: $O(b^{d/2})$.
- How to find intersect?

A more memory efficient solution

The “frontier list” consumes $O(b^d)$ memory.

- Note: DFS only consumes O(d) memory. But DFS cannot find shortest path.

Iterative Deepening DFS (IDDFS)

- Iteratively run DFS with max depth of 1, 2, 3, …
- The node discovering order is exactly same as BFS. So we can find shortest path.
- Only consumes O(d) memory.
- But there are repeated visits of upper level nodes.

In our problem, d is very small (around 6).

- At most repeat 6 times.

Design Messaging System

Features:

- Friend list
- Send/receive messages
- View message history

Initial Thought

P2P communication, like SMS.

- Server acts as a router, routing messages to correct receiver.
- Clients receive messages and store them locally.

Advantage:

- Minimum storage requirement on server side.
  - A small buffer for each user is still needed, in case the receiving client is offline.
  - Think about QQ without “cloud history” turned on.

Disadvantages:

- Clients’ storage is limited and volatile (browser cache, mobile app storage)
- No multi-device sync

Working Solution

Server Information storage:

- Users
- Friend relations
  - Friend adjacency list
- Conversations
  - {conversation id, sender id, receiver id, text, time, seen status}

To avoid database query, move everyone's inbox to the server.

- For each user, store conversation with every friend.
  - Text messages are short, no need to store id or excerpt.
- Only cache recent ones.
- Client also can store some cache.

Pull or Push

Remember in news feed

- Pull: read all follows' news lists, high latency
- Push: update all followers' news lists, introduce unnecessary traffic => freshness problem.
  - To reduce unnecessary traffic: don't push for celebrities; only push to active users.

In instant messaging system, Push doesn't really introduce that many traffic.

- only one recipient => updating receiver's inbox is as easy as updating sender's sent box.

"Push notifications" for mobile phone apps, HTTP requests every a few seconds on browser.

- Here "push"/request are client-server communication methods, not the Pull/Push mechanism inside our server.

Followup Questions

1. How to see friends' online status?

   Store online status. Constantly changing, cache.

   How to inform offline (close browser)? Heartbeat request.

   Get friends' online status every a few minutes, unless query.

2. How to support multiple devices?

   Register device information in server. Push notifications on all devices.

   Unread messages: record last read time.

3. How to support group chat?

   Store group information and group chat messages.

System Design Summary

- Single machine performance
  - Interface
  - Data structure, algorithm
- Scaling
  - Storage, QPS estimation
  - Partition jobs and storage, load balancing
  - Storage hierarchy, cache
  - Modularity
  - Replication
- Discussions
  - Heterogeneity -> tradeoff