به نام او، به یاد او

دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

درس ساختمان داده ها و الگوریتم ها
چالش

ارائه شده توسط: دکتر قدسی و دکتر صفرنژاد
طراحان: حامد کلانتری، عرفان والوبیان

شهاب حسینی مقدم 98105716

تیر 1400

آماده سازی: در ابتدای کار برای مشخص کردن اکانت هایی که بیشترین فالور و فالوینگ را دارند یک دور تمام اکانت ها از حافظه خوانده میشوند که تعداد follower ها و following های آنها مشخص شود. برای این کار از hashmap (در داخل پایتون به صورت dict وجود دارد.) استفاده کرده و تمام اکانت هایی را که میخوانیم تعداد following هایش را برای آن منظور میکنیم. این کار از اردر n+e است، چون در hashmap سرچ کردن از اردر 1 است.

همچنین یک کش مخصوص اکانت هایی که تازه درست شده اند به صورت dict داریم.

همچنین هنگام فراخوانی یک بلوک از اکانت ها از حافظه آن ها را در یکی از دو cache_block_0 hashmap و cache_block_1 نگه میداریم که هر کدام دو خصوصیت lock و dirty دارند. این مپ ها هر کدام به طول 100 هستند. هر گاه برای دستوری نیاز به دسترسی به اطلاعات یک اکانت داشتیم، ابتدا به most_accessed_accounts و recently_created_accounts مراجعه کرده و اگر خالی بودند به سراغ cache_block 0 و 1 میرویم. اگر در این بلاک ها اکانت موجود بود از آن میخوانیم و در غیر این صورت طبق قواعد زیر یک بلوک را خالی کرده و یک بلوک 100 خطی حاوی اکانت مورد نظر را از حافظه میخوانیم:

1- اگر lock برای یک بلوک فعال بود یعنی یک رفرنس به یکی از اعضای آن داریم که ممکن است محتویات آن تغییر کند. پس این بلوک کش نباید دست بخورد. مثلا هنگامی که acc1 بخواهد acc2 را فالو کند بلاک حاوی acc1 قفل میشود. چون در غیر این صورت اگر هنگام فراخوانی acc2 بلوک کش از نو نوشته شود، تغییرات acc1 دیگر در دیسک ذخیره نمیشود.
2- اگر dirty فعال بود یعنی در یکی از اعضای این بلوک تغییر ایجاد شده. اگر خواستیم یک بلوک جدید را در این خانه کش بریزیم باید اول این تغییرات را به حافظه برگردانده و سپس کار را انجام دهیم.

برای بلوک های کش میشد به جای دو بلوک 100 تایی از 4 بلوک 50 تایی استفاده کرد. چون در این برنامه این که یک id فراخوانی شود احتمال این که id+1 یا id-1 فراخوانی شود را افزایش نمیدهد. در حالی که اگر 4 بلوک داشتیم، استفاده از دیسک کاهش میافت چون مثلا یک اکانت وقتی آنلاین میشود چند دستور را در بازه کوتاه اجرا میکند. همچنین در این صورت میشد از سیاست هایی همچون LRU استفاده کرد و تعداد دسترسی به هر بلاک کش تعیین میکرد که این بلاک باید override شود یا خیر

ساخت اکانت: هنگامی که یک اکانت جدید میسازیم این اکانت وارد یک hashmap میشود. اگر ظرفیت این مپ از حدی بیشتر شد (مثلا 10 تا اکانت) انگاه وارد حافظه میشود. در نتیجه تعداد دسترسی دیسک کاهش میابد.

بلاک/ آنبلاک: برای هندل کردن این بخش رابطه بلاک به صورت id- در اکانت ثبت میشود. مثلا اگر اکانت 100 اکانت 200 را بلاک کند در روابطش به صورت 200- ثبت میشود. سپس هر گاه که نیاز شد چک کنیم آیا اکانتی اکانت دیگر را بلاک کرده یا خیر، روابط آن اکانت را چک میکنیم. مثلا در هنگام فالو کردن یا در هنگام پیشنهاد دادن اکانت جدید برای فالو. در این بخش از bloom_filter استفاده کردیم. چون احتمال این که دو اکانت همدیگر را بلاک کرده باشند بسیار کم است اما مجبوریم هزینه زیادی برای چک کردن داشته باشیم. همچنین این هزینه هنگام پیدا کردن اکانت جدید برای فالو کردن بشدت زیاد میشود؛ در حالیکه احتمال false-positive بسیار کمی دارد و اگر هنگام پیشنهاد کردن یک اکانت به اکانت دیگر این فیلتر به ما گفت که ممکن است این دو اکانت یکدیگر را بلاک کرده باشند میتوان به آسانی از پیشنهاد صرف نظر کرد (چون پیشنهاد فرد جدید برای فالو کار مهمی نیست که به صورت قطعی بخواهیم بلاک بودن را چک کنیم.) البته ما چون تعداد بلاکی های زیادی نداریم این چک را انجام میدهیم. همچنین برای فالو کردن هم تعداد دفعات مراجعه به دیسک برای این که ببینیم میتوانیم اکانت را فالو کنیم یا بلاک شده ایم کاهش میابد. اما یکی از مشکلات این روش این است که با هر بلاک یا آنبلاک باید دوباره محاسبه را انجام داد و همچنین در صورت بزرگتر شدن تعداد کاربران از عددی مشخص کارایی روش کاهش میابد. اما به طور کلی برای برنامه ما بسیار مفید است.

فالو/ آنفالو: برای این بخش ابتدا چک میکنیم که دو اکانت یکدیگر را بلاک نکرده باشند، (پس آن ها را لود میکنیم، اما بلاک اکانت اول را قفل میکنیم چون قرار است در آینده در داده هایش تغییر داشته باشیم.) سپس تغییر در داده را ایجاد میکنیم.

```
can't follow: you have blocked this user
can't follow: you have been blocked by this user
```

یافتن دوستان آنلاین: از آنجا که محدودیت در حافظه داریم، جا برای کش کردن آنلاین ها و آفلاین ها نبود، در نتیجه مجبوریم اکانت ها را از حافظه لود کنیم.

```
online friends of account 120 are : ['2143', '5802', '2435', '6694', '4065', '7566', '7538', '2939', '3406',
offline friends of account 120 are : ['3349', '9217', '239', '7271', '4400', '8596', '6981', '3914', '3456',
online friends of account 10001 are : ['2133']
offline friends of account 10001 are : []
online friends of account 1 are : ['6561', '877', '7145', '3463', '7755', '1385', '5118', '433', '7560', '981
offline friends of account 1 are : ['6149', '7340', '6033', '6076', '665', '7915', '6860', '5375', '8593', '
```

(برای اطمینان از درست کار کردن تابع، اکانت های آفلاین هم خروجی داده میشوند)

پیشنهاد دوستان جدید: این بخش پرهزینه ترین بخش است. ابتدا اکانت در حافظه لود میشود. سپس تاریخچه پیشنهاد ها برای اکانت نیز لود میشود. برای این که فرآیند خیلی طول نکشد یک حد بالا برای افراد معرفی میکنیم. که مثلا اگر الگوریتم 200 نفر شخص متمایز پیدا کرد، دیگر ادامه ندهد. الگوریتم به این صورت است که برای هر یک از دوستان دوستان فرد، اگر فالو یا بلاک نشده باشند، یک امتیاز نسبت میدهد. در نهایت حداکثر 20 نفر با توجه به امتیازی (نحوه امتیاز دهی توضیح داده میشود) که دارند پیشنهاد میشوند:

```python
def calculate_adj_score(self, acc_1: account, acc_2: account, acc_3: account) -> int:
    score = 1
    if acc_3.connections.__contains__(str(acc_1.id)):
        score += 3
    if acc_3.connections.__contains__(str(acc_2.id)):
        score += 2
    if acc_2.connections.__contains__(str(acc_1.id)):
        score += 5
    return score
```

(اکانت 1 همان شخص اصلی است. اکانت 2 دوستش و اکانت 3 کسی است که قرار است پیشنهاد شود. هر بار که در طی مسیر به این شخص برخوردیم، این امتیاز را دوباره حساب کرده و به مجموع امتیازش اضافه میکنیم.)

در ابتدا برای این که سرچ زیادی داریم اکانت ها را در یک hashmap نگهداری کرده و در نهایت هنگامی که میخواهیم k اکانت با بالاترین امتیاز را پیشنهاد دهیم از priorityQueue استفاده میکنیم.

```
recommended accounts  account:priority
9112:6
4034:8
6589:6
4985:6
5317:6
5447:6
2553:6
5928:6
4115:6
241:6
6886:6
2169:6
4195:6
9071:6
1738:8
6031:6
```

البته در شرایط واقعی تر و با محدودیت کمتر، انتظار میرود امتیاز ها بیشتر از این باشند.

در نهایت هم برای نگهداری تاریخچه پیشنهاد ها یک دیسک میسازیم و در هنگام پیشنهاد دادن این تاریخچه را از دیسک میخوانیم. هر پیشنهادی را که داده ایم را با فرمت id:time ذخیره میکنیم. سپس در هنگامی که بین اکانت ها میگردیم چک میکنیم که اگر در تاریخچه به صورت id:-1 ذخیره شده بود دیگر آن را پیشنهاد نمیدهیم. در غیر این صورت اگر خواستیم این اکانت را پیشنهاد بدهیم آن را هم به تاریخچه اضافه میکنیم. id:time در صورتی به id:-1 تبدیل میشود که 10 روز از پیشنهاد گذشته باشد یا شخص فالو نشده باشد.