Real-Time Gaining Subscribers by Messaging

Konstantinos Theocharidis
Singapore Management University
Singapore

Hady W. Lauw
Singapore Management University
Singapore

Abstract

The research on how *brands* can *gain subscribers using content* in their *online pages* (e.g., social networks, e-commerce) is a recent and interesting problem. In particular, in each round k features and m users are searched so as the m users to be notified *once* by brand with *messages* containing k features to subscribe to brand page. However, (k, m)-queries have some noticeable *semantic* and *technical* limitations. In this paper, we propose k-queries to gain subscribers in order to *overcome* the limitations of (k, m)-queries. Specifically, we study the realistic problem of *Real-Time Gaining Subscribers by Messaging* (RGSM). RGSM executes in many rounds. In each round, advertiser searches to message the m users who are most interested to *predefined* k features, as in many real-world scenarios, advertiser *already* knows some k features in the system that are similar to their brand content. The focus on RGSM problem is each k-query to be answered in *real-time* over *many* users having *dynamic* preferences across rounds. We deploy the algorithms Scan, Groupy, and Buffer to solve RGSM on *different* and *realistic* query types. Results show that Groupy is faster than Scan and that Buffer is emphatically superior to others; Buffer solves RGSM in *real-time*.

CCS Concepts

• **Information systems** → **Information retrieval query processing**; **Social networking sites**; **Content match advertising**.

Keywords

real-time query processing, subscription gain, messaging

ACM Reference Format:
Konstantinos Theocharidis and Hady W. Lauw. 2018. Real-Time Gaining Subscribers by Messaging. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Finding the proper *keywords* to stimulate the interest of users is a well-known problem [19, 20, 26, 27, 29]. Yet, in social network and e-commerce platforms (hereafter, *online systems*) such *keywords* are *given* since they correspond to specific *online pages* that inherently represent the *preferences* of users. This happens because users *follow* various pages in online systems to get notified of relative posts as they have an interest in such pages [6, 10]; e.g., users that follow *Nike* page have *Nike* in their preference set. So, in this paper we consider *given* keywords to form *messages* that motivate users to positively react to them. In particular, we aim to gain the *subscription* of notified users by sending messages containing *limited* keywords that they like most; those keywords are a subset of a feature universe (each *feature* corresponds to a specific *online page*) and the *followers* of an online page are the *subscribers* of that page.

Every *service* that helps *brands* to *gain subscribers* is *important* as it provides a concrete way for a brand *advertiser* to increase the subscribers of brand page in *online systems* and *offline systems*. In this work, we focus on *online systems* as most brands nowadays maintain social network and e-commerce pages in such systems. Yet, note that even classic websites of brands (we consider them as *offline systems*) can also acquire a portion of the new subscribers gained in online systems since brands can advertise their classic websites in their online system pages. The *benefit* of advertisers possessing subscribers in an offline system is the direct engagement with them via e-mail (e.g., *newsletters*) *without having any restriction* in the number of e-mails they can send. However, the number of messages is *limited per day* in online systems; e.g., an advertiser having an account in the social network VK[1] can only send 20 messages per 12 hours to any non-friend VK user (one message per user). In both cases, *subscribers* are very significant for a brand to *grow* its popularity and revenues since they not only *start* the influence propagation process of brand (in online systems) but they are also the *sole* users that receive its e-mails (in offline systems).

Gaining subscribers using content, introduced in [26], is a recent and interesting *service* for brand advertisers to increase the subscribers of their pages in online systems. Specifically, the work [26] studies the problem of *Gaining Subscribers by Messaging* (GSM) that pertains to (k, m)-queries, one (k, m)-query per round. Authors in that work search in each round both for k features (online pages; keywords) and m non-subscriber users to send messages containing k features to m users aiming to gain the subscription of m users.

In this paper, we study the same *service* (*gaining subscribers using content*) but we address it in a much more *practical* and *realistic* way. Particularly, we propose the problem of *Real-Time Gaining Subscribers by Messaging* (RGSM) that pertains to k-queries; a k-query finds in each round the m non-subscriber users having the highest preference to *given* k features (online pages; keywords). So, advertiser messages the m users on *predefined* k features. We claim that k-queries overcome the *practical limitations* (*semantic* and *technical*) of (k, m)-queries as we analytically explain as follows.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

[1] https://vk.com/

[2] Description Codes are: *woman-looking-at-window-6aqjJOZfe5o* for picture **1**, *person-on-body-of-water-AIfbUyzhKz0* for picture **2**, *woman-wearing-pink-blouse-5fiibEXMHRI* for picture **3**, *woman-wearing-brown-hat--Hr_uHRfImo* for picture **4**, *selective-focus-photo-of-woman-in-gray-sleeveless-top-DJNoNHpQK_I* for picture **5**, *a-mans-hand-holding-onto-a-watch-on-a-dock-w_kQm6F2h54* for picture **6**, *person-in-blue-long-sleeve-shirt-showing-left-hand-DfteCt2Wyvc* for picture **7**, *person-wearing-silver-round-analog-watch-with-black-leather-strap-ap5lEDv4Z60* for picture **8**, and *round-silver-colored-watch-on-rack-during-sunset-Tb38UzCvKCY* for picture **9**.

Figure 1: An example depicting 2 *cases* **that mention to the difficulties that advertisers face when the** k **features are** *not given*. **Pictures** 1-9 **mention to** *first case* **while** 10**th picture mentions to** *second case*. **The description of** 2 *cases* **lies in the Example 1. All pictures taken from** *Unsplash* **by typing in a browser** `https://unsplash.com/photos/` **followed by the** ***description code***[2] **for each picture. Each picture (apart from** 10**) is described by a set of keywords that generally capture the watch-concept of each picture.**

1.1 Semantic Contributions of k-queries over (k, m)-queries

In many realistic scenarios, the k features are *already* defined by the brand advertiser as we show in the next cases: (a) *John* is a *watch designer* of a new watch brand and wants to get subscribers to his brand online page. He *already* knows that his watch brand has similar properties with k other known watch brands of online system; (b) *Maria* is a *landscape photographer* and she *already* has a portfolio that is possible to be liked by users that have a high preference on k popular pages of online system relative to photography, architecture, or design; (c) *Giorgio* has *already* found a k-size good content to publish to the *news magazine* page of the company in which he works as an advertiser. However, a replacement of prior k-queries with (k, m)-queries of work in [26] can create significant *inconvenience* and *delay*, as we explain in the following example:

Example 1. *Consider the previous* (a)*-scenario relative to John with the difference that now he uses a* (k, m)*-query instead of a* k*-query to advertise his watch brand. So, John solves the problem in [26] and assume that the* k *features returned to his* (k, m)*-query for current round are the* k *pages (suppose* $k = 3$*) "travel is a pleasure", "we love sea", and "nature in our planet". Then, John sees that he must find pictures depicting watches having a concept relative to "travel", "sea", and "nature". Suppose that there are* 2 cases *for John:* (i) *his watches have* specific *characteristics,* (ii) *his watches have a* variety *of characteristics. In both cases, assume that John opts to search in Unsplash platform (that has a plethora of copyright-free pictures) to find suitable watch pictures to a "travel-sea-nature" concept. Based on current content in Unsplash (**realistic facts**), after* 100 *mouse-scrolls in his watch-keyword search, John finds* only *the* 9 *pictures depicted in Figure 1 (pictures* 1–9*); note that the general character of "travel" keyword allows a flexible utilization of pictures that can be relevant to travel. In the scenario of* (i)*-case and assuming that all the watches designed by John have a round-big-shape, he decides to reject the pictures* 3, 4, *and* 7 *while being uncertain of the picture* 8 *due to its watch blur depiction. So, he creates a collage-post including the pictures* 1, 2, 5, 6, (8), *and* 9. *In the scenario of* (ii)*-case, he creates a collage-post like the one depicted in picture* 10. *In both cases, the created collage-post constitutes the post of current round. However, the* big problem *highlighted in both cases, is that John needs to spend significant time (*100 *mouse-scrolls in Unsplash) to find proper watch pictures to create a post under the specific* k*-concept defined by his* (k, m)*-query. To stress further that, imagine that in any of next rounds, some of the* k *pages also relate with a "travel-sea-nature" concept, and as it does not make sense the same picture to be used twice, John may spend the double time (e.g.,* 200 *mouse-scrolls) in regards to current round till finding a proper watch picture to include in his collage-post. So, overall, and as rounds evolve, it can be cumbersome for advertisers to efficiently utilize the results of* (k, m)*-queries where the* k *features are* ***arbitrary****. Instead,* k*-queries are* ***time-saving*** *as they depend on existing known content (see Example 3 for more details on* k*-queries).*

1.2 Technical Contributions of k-queries over (k, m)-queries

The technical difference among k-queries and (k, m)-queries is that the latter searches for 2 parameters (k features and m users) compared to only one parameter searched by the former (m users). This fact challenges the *real-time* applicability of (k, m)-queries, especially when it is tested on realistic cases (*big* user datasets with users having *dynamic* preferences). Nevertheless, authors in [26] do not study *any of the mentioned criteria* relative to their proposed (k, m)-queries, and that restricts the applicability of those queries and as a result their usefulness towards brand advertisers. By contrast, the k-queries we propose in this work are studied in *real-time* over *many* users with *dynamic* preferences. Based on the arguments we provide in previous section (semantic contributions of k-queries over (k, m)-queries), we claim that k-queries have clearly higher chances to be used more in real world than (k, m)-queries. This induces that *several* advertisers may *concurrently* apply k-queries to online system, and so their *real-time* response is *actually* necessary. We stress that *need* of *real-time* applicability in the next example:

Example 2. *Consider the popular online system Instagram*[3]*. Instagram may face k-query cases such as:* (A) different *advertisers representing different pages apply k-queries in similar time periods;* (B) *a* group *of advertisers being responsible for the content of the same page, apply simultaneously k-queries to form their content publishing schedule;* (C) *a* single *advertiser of a page applies a k-query but selects only a subset of the returned m users, then repeats another k-query, and so on, till forming a set of m users;* (D) *any* combination *among* (A)–(C)*. Assuming that Instagram devotes a single machine with* serial execution *for the new k-query service, and assuming also a logical number of* nearby-time *k-query requests, Instagram wants to avoid scenarios where several advertisers in queue wait long enough times leading to an impractical k-query service. For instance, if the query reply time is* 2 *seconds and have* 30 *advertisers concurrently apply k-queries, the* 30*th advertiser will wait almost* 1 *minute to get their m users, which is time-prohibitive for Instagram.*

1.3 Work Contributions

Therefore, based on above arguments we discussed in previous two sections, we contend that the RGSM problem (k-queries) we propose in this paper is not only more *practical* and *realistic* than GSM problem [26] ((k, m)-queries) but is also more *technically challenging* due to the actual necessity of real-time applicability.

The *main contributions* of this paper are the following:

Problem. We propose and study the RGSM problem, the *first* one that relates with k-queries for gaining subscribers using content in *real-time* over *many* users with *dynamic* preferences. RGSM is an important problem as it provides to brands a realistic and efficient way to increase their subscribers.

Algorithms. We develop three RGSM algorithms, named Scan, Groupy, and Buffer. We consider Scan as *baseline*, Groupy as *advanced baseline*, and Buffer as *state-of-the-art* for the RGSM problem.

Experiments. We conduct a thorough experimental evaluation on six different and realistic query types (*case studies*) by providing scalable and sensitivity analysis results of RGSM algorithms.

[3]https://instagram.com/

2 Problem Statement

We define the **RGSM problem** as follows:

Problem 1. *Given an online system depicting a social network or e-commerce platform with $|V|$ users, a feature universe L, a weighted feature set F_v of size $|L|$ capturing the preferences of each user v, a number of m notification messages, each one, containing k features corresponding to an entry of a queries set $\mathcal{Q}$, and a number of rounds n, find in real-time in each round t which m non-subscriber users to message so as to maximize the total subscription gain SG over n rounds. A user that receives a message cannot be notified again. The SG of a user v in round t ($v.SG$) is computed as the sum of values (counters) present in round t at the k positions of F_v. The preferences of users are dynamic; the counter of any feature in each F_v may increase over rounds. So, the goal of RGSM is the following:*

$$SG = \underset{m}{\text{argmax}} \sum_{t=1}^{n} SG_t(k, m) \quad s.t.\ k \text{ fixed, } m \text{ sought} \tag{1}$$

The typical scenario in RGSM is that different advertisers *simultaneously* apply k-queries to the online system for finding their m users; so, the response to those queries should be in *real-time*. We consider that each k-query belongs to a *fixed* set $\mathcal{Q}$ since the k-feature(keyword)-combinations that interest the advertiser are *concrete*. The reason is that each advertiser uploads content for an online page that has a specific purpose (e.g., relates with *sport fashion* or *business economics* or *diet products*, etc.). For clarity, we provide an example about how $\mathcal{Q}$ is computed for k-queries, below:

Example 3. *Assume an advertiser that runs a brand page about "sport fashion" in an online system. Advertiser observes that the product characteristics of their brand look similar to the ones that are present in posts of* Nike, Adidas, Puma, *and* Reebok *pages of the same online system. Yet, advertiser knows that only $k = 3$ features are available to take place in a message. So, in this case, $\mathcal{Q}$ includes* 3*-out-of-*4 *feature combinations. Namely, $\mathcal{Q} = \{$(*Nike, Adidas, Puma*), (*Nike, Adidas, Reebok*), (*Nike, Puma, Reebok*), (*Adidas, Puma, Reebok*)$\}$. Depending on current brand content, advertiser selects the proper k-query $\in \mathcal{Q}$ to request online system for finding the proper m users.*

3 RGSM Algorithms

Before proceeding to the description of RGSM algorithms, we present a summary of them that outlines their connection at a high level. Scan relies its execution on a priority queue without storing any information about updated SG of users over rounds. Groupy incorporates this information that Scan misses by storing all non-notified users in a distributed structure and uses a candidate pruning technique based on the maximum *likes* of users per day. Buffer converts the distributed structure of Groupy to a linear one that stores a limited number of users with highest SG at that time, exploiting also the candidate pruning technique of Groupy.

3.1 Scan Algorithm

Algorithm Scan presents the Scan method to solve RGSM.

The logic of Scan is to maintain a priority queue to find the m users with highest SG in each round and marked them as notified for next rounds. In specific, in each round t, advertiser first sets a k-query to the online system (Line 3). Then, for each non-notified

user v, Scan computes their SG and may update the priority queue Q_1 (Lines 4–10). Note that Q_1 is ascending-sorted on $v.SG$ meaning that its *top* entry denotes the entry with the lowest SG. When all non-notified users of round t are examined, Q_1 contains the users to notify in round t and SG increases accordingly (Lines 11–12).

Algorithm Scan

```
   Input  : V, F_v, Q, k, m, n
   Output : SG // total subscription gain over n rounds
1  SG = 0; notified = ∅; // notified users over n rounds
2  for t = 1, ..., n do
3      Advertiser applies a k-query ∈ Q; // each F_v may have now
       higher values relative to k features
4      Q_1 = ∅; // pr. queue with entries asc-sorted on v.SG
5      for each user v ∈ V \ notified do
6          Compute v.SG based on k positions of F_v;
7          if |Q_1| < m then Q_1.push(⟨v, v.SG⟩);
8          else if |Q_1| = m and v.SG > top.SG then
9              Q_1.pop(⟨top, top.SG⟩);
10             Q_1.push(⟨v, v.SG⟩);
11     for each user v ∈ Q_1 do
12         SG = SG + v.SG; notified.insert(v);
13 return SG;
```

3.2 Groupy Algorithm

Algorithm Groupy presents the Groupy method for RGSM.

Groupy utilizes a *group structure* (GS) in its operation. GS stores for each *query* a *fixed* number ($groupsN$) of *equal-size* ($groupS$) group triples (*gp-triples*). Each *gp-triple* is a map of a *groupId* to triples of the form $\langle v.userId, v.SG, v.lastUpdate\rangle$ sorted in descending order on $v.SG$ that is the SG of user v to the k keywords of *query*; $v.lastUpdate$ denotes the last round t in which $v.SG$ was computed. Initially, $groupsN * groupS = |V|$, but over rounds less than $|V|$ users are stored in GS due to skipping of $notified$ users. Yet, a user v has the same *groupId* for each *query* throughout the whole execution of algorithm. Namely, distribution of users in groups is done once by serially scanning of V and is the same for each *query*.

Along with GS, Groupy uses a cache (Qcache) that comprises the queries stored in GS. However, in real world, Qcache cannot contain all the queries of advertiser, so we also implement a cache-replacement policy that replaces the least-frequent *query* present in cache with the new received one; in case of tie, the least-recent *query* is replaced. Qcache plays an important role in the operation of Groupy since in case of replacement, we load all the non-notified users to $GS[query]$, but in opposite case, we do not modify at all the content of $GS[query]$; in both cases, we save time.

Further, Groupy leverages a *beneficiary value* ($benVal$) in its operation that depicts the maximum number of *likes* (clicking the *like* icon) that any user can yield to the posts published by any feature within the period of a whole day.

The logic of Groupy is to avoid updating GS as much as possible and in case of updates they should be efficient and meaningful. As meaningful considered the updates relative to users that could be included in the next m-users (if not taking place in current m-ones) and also can lead to an efficient pruning of total candidate users in GS. Namely, Groupy addresses a k-query of current round t by maximum-exploit the GS content of previous rounds and efficiently-and-minimum-updating GS for current round.

Algorithm Groupy

```
   Input  : V, F_v, Q, k, m, n
   Output : SG // total subscription gain over n rounds
   Param. : groupsN, groupS, cacheS, benVal
1  SG = 0; notified = ∅; // notified users over n rounds
2  Q.cache = ∅; GS = ∅; // entry: query to gp-triples
3  for t = 1, ..., n do
4      Advertiser applies a k-query ∈ Q; // each F_v may have now
       higher values relative to k features
5      Update Q.cache and (may) GS for query based on groupsN,
       groupS, cacheS, and notified;
6      Q_1 = ∅; // priority queue with entries asc-sorted first on
       v.lastUpdate and second on v.SG
7      Select the first m triples (one per group) from GS[query]
       whose users ∉ notified to fill Q_1;
8      if GS[query] already updated for round t then
9          Sequentially examine each group in GS[query] to update
           Q_1 until top.SG ≥ v.SG for current user v in group;
           // examination skips notified users
10         for each user v ∈ Q_1 do
11             SG = SG + v.SG; notified.insert(v);
12     else if GS[query] not updated for round t then
13         minQ_2 = ∅; maxQ_2 = ∅; // priority queues with entries
           asc-sorted and desc-sorted (respectively) first on v.SG and
           second on v.lastUpdate
14         updates = ∅; // entry: user to pair of updates
15         Fill minQ_2, maxQ_2, and updates from Q_1;
16         Sequentially benVal-examine each group in maxQ_2 to
           may update minQ_2 and updates; // examination skips
           notified users
17         Repeat the benVal-examine process of Line 16 for each
           group ∈ GS[query] \ maxQ_2;
18         for each user v ∈ updates \ minQ_2 do
19             Update GS[query] for pair updates of v;
20         for each user v ∈ minQ_2 do
21             SG = SG + v.SG; notified.insert(v);
22 return SG;
```

In more detail, Groupy handles a k-query in each round t (Line 4) by first updating Qcache and GS (Line 5). Qcache is always updated while GS gets updated only when *query* is not found in GS (*query* either first time applied or previously replaced by other ones). In case of updating GS, all non-notified users along with their SG and $lastUpdate$ values relative to round t are inserted to $GS[query]$.

After updating Qcache and GS, Groupy uses a priority queue Q_1 to store m non-notified users (along with their SG and $lastUpdate$ values), each one belonging to a different group in $GS[query]$, who updated their SG more recently than others; in case of tie, users with higher SG are selected (Lines 6–7). The reason of such selection is to quickly find a strong *m-candidate-set* since distributed-and-recently-updated users can have a good pruning effect to other candidates in GS. In case $GS[query]$ updated, Q_1 has updated values relative to round t (Line 8), else, it has non-updated values relative to previous rounds of t (Line 12).

In the former case (Line 8), Groupy simply examines each user-triple in a group of GS until a user v is found that cannot modify the content of Q_1. Then, Groupy examines the next group in GS, and so on (Line 9). When modifications in Q_1 are over, notified users of round t are found and increase the total SG (Lines 10–11).

In the latter case (Line 12), Groupy moves the content of Q_1 (after updating $v.SG$ for all triples in Q_1) to the priority queues $minQ_2$ and $maxQ_2$ (inverse to each other) as also to the $updates$ map (Lines 13–15). $maxQ_2$ is used for examination (having the user with highest SG as its top entry) while $minQ_2$ is used for modifications as it keeps the final m-users (top entry corresponds to the user with lowest SG). The map $updates$ stores, for each user that incurs a modification to $minQ_2$, a pair entry of user previous $\langle SG, lastUpdate\rangle$ values (present in $GS[query]$ from earlier rounds) and user current $\langle SG, lastUpdate\rangle$ values. If a user v inserted to $updates$ it means that v is a *strong candidate* (managed to modify $minQ_2$) to take place in final set of m-users of current round or close next ones.

After initializing $minQ_2$, $maxQ_2$, and $updates$, Groupy proceeds to an examination process based on $benVal$ to may update the content of $minQ_2$ and $updates$, by first examining the groups of $maxQ_2$ (Line 16) and then the other groups of $GS[query]$ (Line 17). The reason for such examination ordering is that it is expected that the users with high SG (present in $maxQ_2$) can reduce the total number of processed users needed over all groups in $GS[query]$ in order to find the final m users for messaging in round t.

Function `BenValExamine` presents the $benVal$-examine process that is utilized by Groupy and Buffer algorithms.

Function BenValExamine

Input : top, v, t, bV, MS (GS or BS), $query$
Output : $minQ_2$, $updates$ // modified **or** unchanged

1 $dif_1 = t - v.lastUpdate$;
2 $val_1 = v.SG + \lceil dif_1 * bV - ((dif_1 - 1)/2.0) * bV \rceil$;
3 **if** $top.SG > val_1$ **then** // top of $minQ_2$
4 $dif_2 = t -$ last t' where $query$ inserted to MS;
5 $val_2 = v.SG + \lceil dif_2 * bV - ((dif_2 - 1)/2.0) * bV \rceil$;
6 **if** $top.SG > val_2$ **then** // top of $minQ_2$
7 **break**; // process next group of $GS[query]$ **or** complete the examination of $buffer$ in $BS[query]$
8 **else**
9 **continue**; // process next user in same group of $GS[query]$ **or** next triple of $buffer$ in $BS[query]$
10 Check whether v modifies $minQ_2$ and $updates$;
11 **return** $minQ_2$, $updates$;

Its input relates with top (top entry of $minQ_2$), v (currently examined user in GS for Groupy or BS for Buffer algorithm we explain later), t (current round), bV (depicts $benVal$ for brevity), MS (main structure; GS for Groupy or BS for Buffer), and $query$ for the query of round t. Its output relates with any modifications to $minQ_2$ and $updates$. In short, `BenValExamine` computes a *first* beneficiary value val_1 depicting a *maximum* possible increased $v.SG$ considering the *delta* among current round t and last round where $v.SG$ updated (Line 2). In case that val_1 is at least equal to $top.SG$ (Line 3) then the *actual* $v.SG$ may incur modifications to $minQ_2$ and $updates$ (Line 10). Otherwise, `BenValExamine` computes a *second* beneficiary value val_2 depicting a *maximum* possible increased $v.SG$ considering the *delta* among current round t and smallest round where $v'.SG$ updated over any $v' \in MS[query]$ (Line 5); that smallest round equals to the last round where $query$ inserted to MS (either as first time applied or after replacement). In case that val_2 is less than top entry in $minQ_2$ then Groupy proceeds with the examination of next group in GS (Lines 6–7), as its SG-order verifies that no user in same group with v can incur any changes to the final m-users, and so no reason to compute their updated values for round t. Else, Groupy proceeds with the examination of next user in same group of GS (Lines 8–9).

When $benVal$ examination finishes, Groupy updates $GS[query]$ only for the users stored in $updates$ but not in $minQ_2$ (Lines 18–19). The effective operation of `BenValExamine` entails that *few users* are found in $updates$ and that saves update time to $GS[query]$. Distribution of users in groups also helps updates to apply to small segments (groups) and that decreases further the update time; that is why GS formatted in a distributed way. Finally, the users kept in $minQ_2$ are the final m-users to notify in round t and they respectively increase the total SG (Lines 20–21).

3.3 Buffer Algorithm

Algorithm Buffer presents the Buffer method for RGSM.

Algorithm Buffer

Input : V, F_v, Q, k, m, n
Output : SG // total subscription gain over n rounds
Param. : $bufS$, $benVal$

1 $SG = 0$; $notified = \emptyset$; // notified users over n rounds
2 $BS = \emptyset$; // entry: $query$ **to** $triples$
3 **for** $t = 1, \ldots, n$ **do**
4 Advertiser applies a k-$query \in Q$; // each F_v may have now **higher** values relative to k features
5 $minQ_2 = \emptyset$; // priority queue with entries asc-sorted first on $v.SG$ **and** second on $v.lastUpdate$
6 $updates = \emptyset$; // entry: user **to** pair of updates
7 **if** $query \in BS$ **then** // $buffer$ already loaded
8 Fill $minQ_2$ with the first m triples of $BS[query]$ whose users $\notin notified$;
9 Insert to $updates$ the m triples of Line 8;
10 Sequentially $benVal$-**examine** each triple (after m-th triple) of $BS[query]$ to may update $minQ_2$ and $updates$; // examination skips $notified$ users
11 **if** $benVal$-*examination of Line 10 completed* **then**
12 Repeat Lines 18–21 of Groupy ; // BS for GS
13 **else if** $query \notin BS$ ***or*** $benVal$-*examine not over* **then**
14 Find the $bufS + m$ users with the highest SG as Scan does it for m users in its Lines 4–10;
15 Clear $BS[query]$ and move the $bufS$ users with lowest SG to $BS[query]$; // load $buffer$
16 **for** *each user* v *of rest* m *users* **do** // highest SG
17 $SG = SG + v.SG$; $notified$.insert(v);
18 **return** SG;

We explained the execution of Groupy in an elaborate way since Buffer can be considered as a *parameterized* version of Groupy but it works emphatically faster than Groupy. In fact, Buffer was created by addressing the bottlenecks of Groupy and it constitutes a much simpler implementation despite its impressive performance.

After analysis, we found that the overhead of Groupy is attributed to two main things. First and foremost, the distributed format of GS does not usually find quickly an effective enough *m-candidate-set* and this entails several users to be processed till examination of all groups in GS to complete (Lines 13–17). A secondary overhead related to the first is the time required to fill the priority queue Q_1 (Lines 6–7). The reason is that as rounds evolve, the use of Q_1 to form an initial *m-candidate-set* seems not crucial in algorithm effectiveness due to the user distribution in GS.

The workaround to both prior mentioned bottlenecks of Groupy is, instead of using many groups, Buffer uses just a single group (*buffer*) to store and modify the content of each *query*. Now, Buffer (i) uses a *buffer* (noted by BS) as main structure and cache concurrently, (ii) avoids the problems relative to user distribution, (iii) does not need an initial *m-candidate-set* to start its operation, and (iv) retains the small segment that is time-saving to updates application.

In more detail, Buffer uses a *buffer* of size $bufS$ to store a *query* content to $BS[query]$ either when *query* is first time applied ($query \notin BS$) or when $benVal$-examine process did not finish (did not meet a *break* condition; Line 7 in `BenValExamine`) as shown in Line 13. The *query* content of $BS[query]$ comprises triples of the form $\langle v.userId, v.SG, v.lastUpdate\rangle$ sorted in descending order on $v.SG$ and carries the same semantics we mentioned for Groupy. The *loading* (caching) of *buffer* is done as follows: First, a set of $bufS+m$ users are selected having the highest SG at the moment (round t) that advertiser applies the k-query (Line 14). Then, the $bufS$ users of them with the lowest SG form the *buffer* (along with their SG and $lastUpdate$ values) that inits or replaces a prior *buffer* in $BS[query]$ (Line 15).

The structure, meaning, and purpose of using $minQ_2$ and $updates$ is identical to Groupy, yet note that now the *m-candidate-set* is found much faster and that contributes to much less users (notification candidates for round t) to be processed. The reason is that instead of distributed users, users in *buffer* are now the really strong ones (having the highest SG over all non-notified users) and that helps the $benVal$-examine process to complete much more quickly. However, as rounds evolve, $benVal$-examine process can be incomplete (not all possible candidates are considered) over the fixed users present in *buffer*, and so *buffer* needs to be reloaded with another portion of users. Yet, in case of reloading, the m users of round are already found and modify total SG and $notified$ (Lines 16–17) as Buffer exploits the sorting needed to form a new *buffer* for $BS[query]$.

Last, the examination for finding the m-users to notify in round t is much faster due to the absence of Q_1 and $maxQ_2$ (Lines 7–10).

4 Experimental Evaluation

4.1 Experimental Setup

Implementation. We wrote code in C++ and we did experiments on an AMD Ryzen 5 4600U CPU @2.1 GHz machine with 16GB RAM running Linux Ubuntu 20.04.3 LTS 64-bit.

Data. We sampled data from the social network VK[4] that expresses the Russian version of Facebook. In particular, we sampled 2M users having preferences (F_v) over 540 features of VK that is the sum of the 20 most popular features (pages) from each category of VK; VK has in total 27 categories (see Table 1). Initially (round 0), $F_v(f)$ contains the *aggregate* number of *likes* that user v yield to any post published by f (in the page of f) over the years 2010-2017 of VK. We then use a period of 500 days, where the first day (round 1) maps to 1st January 2018, to consider any post published in VK by any f in that period and increases the counters in each $F_v(f)$.

Table 1: The 27 categories of VK in alphabetical order.

VK Categories
Animals | Auto_motor | Beauty_health | Celebrity | Cities_countries | Communication_Services | Consumer_Services | Culture_art | Education | Entertainment | Finance_and_Insurance | Food_recipes | Hobbies | Home_renovation | Internet | Job_search | Media | Medicine | Music | Products_stores | Professional_Services | Relationship_family | Restaurants | Social_and_public_organizations | Sport | Tourism_and_Leisure | Transportation_Services

Table 2: The 6 query types based on 12 VK categories.

Type	Name	Categories
A	Beautician	Beauty_health | Medicine
B	Traveler	Cities_countries | Tourism_and_Leisure
C	Educator	Culture_art | Education
D	Economist	Finance_and_Insurance | Job_search
E	Chef	Food_recipes | Restaurants
F	Interior Designer	Home_renovation | Products_stores

Query Types. To capture the *realistic* and *dynamic* change of $F_v(f)$ over rounds, we opted to form 6 different *query types* (as shown in Table 2) to evaluate the RGSM performance. Specifically, Q includes 20 (3-out-of-6) k-query feature combinations to take place for each query type, where $k = 3$ and the number 6 maps to a set comprising the 3 (this value is irrelevant to the value of k) most popular features from each pair of relative categories. E.g., for the query type A (Beautician), the mentioned set contains the three most popular features in category "Beauty_health" and the three most popular features in category "Medicine"; 6 elements in total (6 corresponds to the value 4 in Example 3). This respectively holds for the rest query types in Table 2. For more details on Q, we remind readers to see the Example 3.

We stress that we consider VK as a big social network and not as a single dataset. Instead, we consider each query type as a different dataset; each query type represents a different *case study* since $F_v(f)$ changes *do not correlate* at all among different query types. In other words, each query type enables a *distinct* RGSM evaluation besides the fact that all evaluations are done in VK. We believe that the experiments we provide on 6 *separate* and *unrelated* query types (case studies; datasets) are sufficient to prove our contributions.

Cache. The cache of Groupy is denoted by Qcache while for Buffer the *buffer* (stored per $BS[query]$) operates as cache itself. To be realistic enough to the presence of massive data in online systems, we decide a 25% *cache decrease policy* for both algorithms. For Groupy, this means that Qcache can store only the 75% of the queries stored in GS. For Buffer, the effect is that the size of *buffer* is 7.5K

[4]VK (https://vk.com/) represents the Russian version of Facebook in terms of scale, functionalities, variety of topics, user accounts, brand pages, etc. It has a much more flexible and unrestricted API (https://dev.vk.com/en/reference) than rest social networks. Further, according to *Wikipedia*, VK had been the 16th most visited website in the world and at the moment it has more than 800M users. All these factors make it very suitable as a social network data source for research-oriented purposes.

that equals to the group size of Groupy decreased by 25%; we remind that *buffer* operates as a single group of Groupy, so the maximum size that *buffer* can have equals to *groupS* (group size of Groupy).

In more detail, the 25% *cache decrease policy* applies differently in Groupy and Buffer due to their different implementation. Since *GS* stores much more data than *BS* per query, we have 25% queries decrease in $|Q|$ (set of all k-queries) contained in Qcache that benefit *GS*; $|Q\text{cache}| = 75\%|Q|$ (we denote $|Q\text{cache}|$ by *cacheS*). On the contrary, *BS* does not store much data per query, so $|Q|$ does not affect Buffer. For that, we apply 25% decrease for the size of *buffer* relative to each query stored in *BS* (*BS*[*query*]). As *buffer* continuously updated, it plays the role of cache for Buffer; $bufS = 75\%groupS$ (we denote the size of *buffer* by $bufS$).

Parameters. The default parameters used in our experiments are as follows: $|V| = \{1M, 1.5M, 2M\}$, $|L| = 540$, $m = 20$, $k = 3$, $n = 500$, $|Q| = 20$, $benVal = 10$. For Groupy, $groupsN = \{100, 150, 200\}$ respectively for $|V|$, $groupS$ = 10K, $cacheS$ = 15. For Buffer, $bufS$ = 7.5K. Also, the sequence of k-queries $\in Q$ are selected *uniformly-at-random* (with a different initialization seed) in each experiment; that enhances further the independence among experiments we present in this work (additionally to the notion of aforementioned *case studies*).

Algorithms. Besides the Scan, Groupy, and Buffer algorithms shown in Section 3, there is no other work that can be used to solve RGSM. We analytically justify this argument in Sections 5 and 6. In the former, we explain why our prior GSM solutions [26] are not meaningful for the RGSM problem. In the latter, we additionally discuss other related works that are also not close to RGSM purposes. We remind that Scan is the *baseline*, Groupy is the *advanced baseline*, and Buffer is the *state-of-the-art* method to solve the RGSM problem.

SG Values. Our RGSM algorithms do not actually differ in their achieved *SG* values[5]. In particular, the optimal *SG* values per round are derived from Scan, and the *SG* values per round of Groupy and Buffer are almost identical to the ones of Scan, as we show in some of our experiments later. Any differences in *SG* values are either minor (due to the $benVal$ parameter used in Groupy and Buffer) or derived from *SG*-ties. Based on that, we stress that the RGSM algorithms in this work are exclusively evaluated on their *efficiency* (running time) as their *efficacy* (*SG* values) can be considered identical.

4.2 Experimental Approach

We present our experimental approach in the next two sections; the current section operates as preamble to familiarize the readers for the context that will follow.

In the first section (4.3), we evaluate the performance of RGSM algorithms in *scalable datasets* only on *running time* since all methods produce almost identical *SG*; any differences among found m users over rounds are either minor or derived from *SG*-ties. Each experiment relates with 500 rounds, so we opted to include only the necessary information (*running time*) for ease of readability.

In the second section (4.4), we present the *actual SG* values of algorithms along with their *running time* per round. This section illustrates the *sensitivity analysis* of algorithms' performance based on different selection values for k and m parameters. Although *SG* values do not differ among algorithms (as in previous section), we show them for completeness and verification reasons; the 100 rounds per experiment here enable better the *SG* values depiction.

[5]The *SG* value per round is computed as the sum of separate *SG* values for each one of the m users selected to be notified in that round.

4.3 Performance: Scalability Results

Figure 2 presents the *running time* (depicting how fast a k-query replied including the update time in the algorithm structures) over 500 rounds of Scan, Groupy, and Buffer algorithms in 1M, 1.5M, and 2M users for query type A. Figure 3 does it for query type B; Figure 4 does it for query type C; Figure 5 does it for query type D; Figure 6 does it for query type E; and Figure 7 does it for query type F.

We see that in all cases, Buffer is significantly faster than Scan and Groupy. On average, for the query types A, B, C, and F, Buffer is roughly at least 2 and 2.5 orders of magnitude faster than Groupy and Scan, respectively. Even for the query types D and E (in which F_v is not updated as often as in other query types and so $benVal$-examine process is not similarly effective), Buffer has several actual real-time responses. When Buffer performs similarly to Scan in some rounds, this means that a new *buffer* loaded at such points.

Regarding the performance of Groupy, we observe that in almost all cases is clearly faster than Scan (on average) till the middle of rounds (round 250). After that point, it remains better than Scan (on average) but its superiority declines as rounds evolve. At rounds where Groupy is less efficient than Scan, it happens due to cache replacements. The general behavior of Groupy is attributed to the not so effective *m-candidate-set* due to the distributed format of *GS*. In first rounds, that set works adequately as $benVal$-examine process can easily find the strong users (with high *SG*) that separate over others. Yet, as rounds evolve, the deficiencies of a random distribution slow down the responses of Groupy.

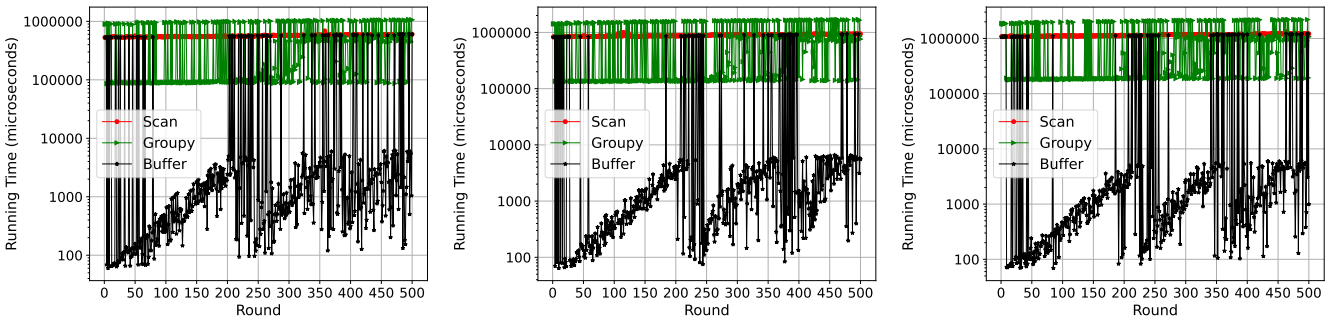
Last, a specific remark on scalable user sizes, is that we see that Buffer increases even more its speedup over competitors as dataset grows, while Groupy performs arbitrarily. The reason is that we initialize the random query process with a different seed each time; Groupy is affected by that due to its randomized distribution effects.

4.4 Performance: Sensitivity Analysis Results

Figure 8 presents the *running time* and *SG* results of Scan, Groupy, and Buffer algorithms in 1M users for query type A and (a) $k = 2$, $m = 10$, (b) $k = 2$, $m = 20$, (c) $k = 3$, $m = 10$, (d) $k = 3$, $m = 20$. Figure 9 does it for query type B; Figure 10 does it for query type C; Figure 11 does it for query type D; Figure 12 does it for query type E; and Figure 13 does it for query type F. In all figures, the colors *red*, *green*, and *black* (of solid lines) depict the *running time* of Scan, Groupy, and Buffer (as in Figures 2–7), while the colors *blue*, *cyan*, and *purple* (of dashed lines) depict their *SG* values, respectively. A zoom of **400% scale** is advised to see the *SG* results analytically.

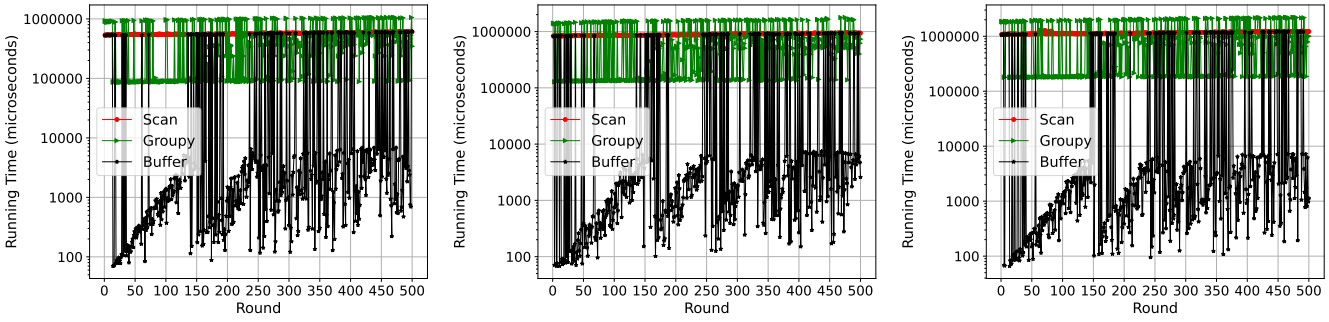
We remind that when $k = 3$ then $|Q| = 20$ and $cacheS = 15$. Yet, when $k = 2$ then $|Q| = 15$ and $cacheS = 11$ that maps to 25% loss of $|Q|$. The cache size of Buffer ($bufS$) remains the same in all cases since as stated the cache of Buffer is irrelevant to $|Q|$ changes.

All results make sense under various aspects. First, the *SG* values of all algorithms are almost identical as expected (that is why we did not show such results in Figures 2–7). Second, as m grows over a fixed k, both *time* and *SG* increase since more users involved that require more processing but concurrently yield higher efficacy (*SG*). Third, the *SG* values get more compact as k grows due to



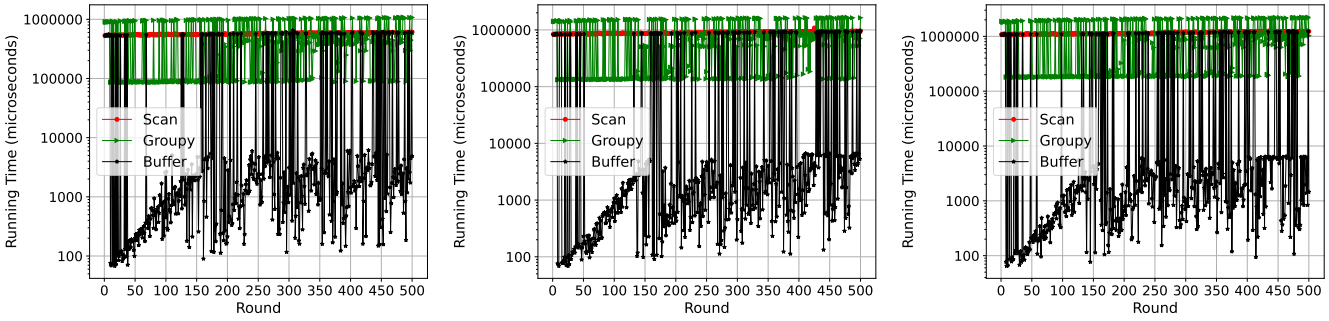
(a) 1M (b) 1.5M (c) 2M

Figure 2: Running time of Scan, Groupy, and Buffer for *Beautician* (type A) over 500 rounds in 1M, 1.5M, and 2M users.



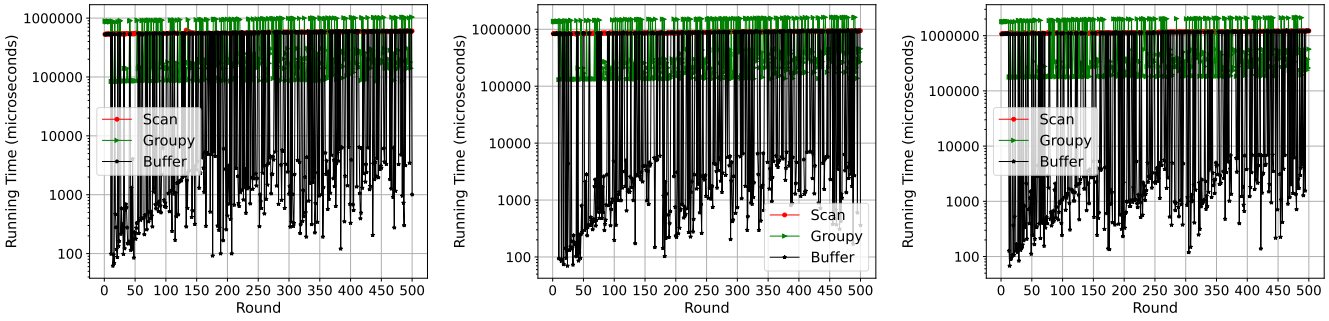
(a) 1M (b) 1.5M (c) 2M

Figure 3: Running time of Scan, Groupy, and Buffer for *Traveler* (type B) over 500 rounds in 1M, 1.5M, and 2M users.



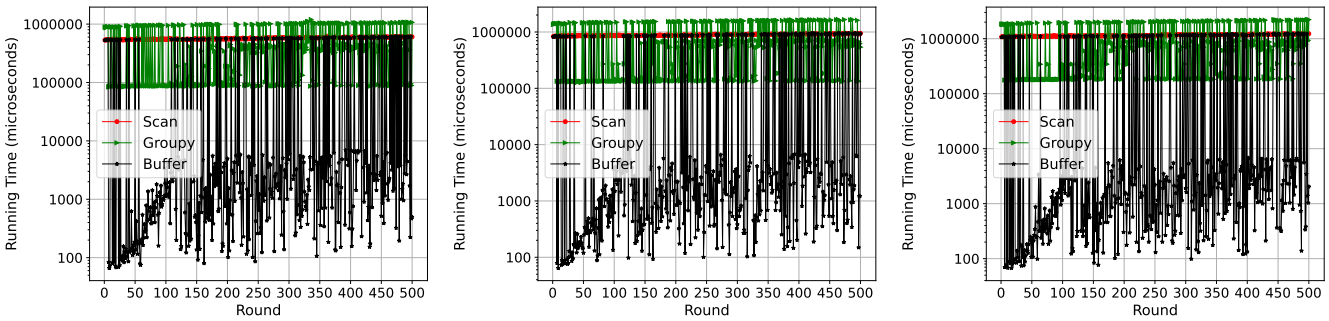
(a) 1M (b) 1.5M (c) 2M

Figure 4: Running time of Scan, Groupy, and Buffer for *Educator* (type C) over 500 rounds in 1M, 1.5M, and 2M users.



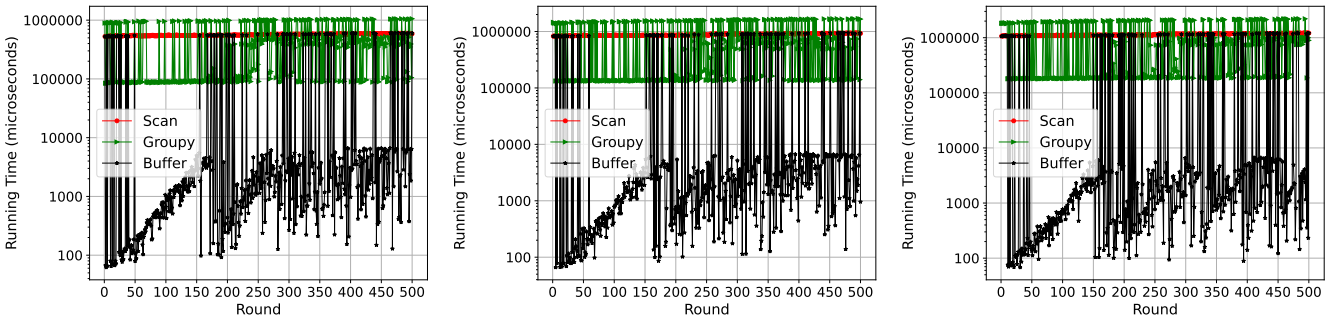
(a) 1M (b) 1.5M (c) 2M

Figure 5: Running time of Scan, Groupy, and Buffer for *Economist* (type D) over 500 rounds in 1M, 1.5M, and 2M users.



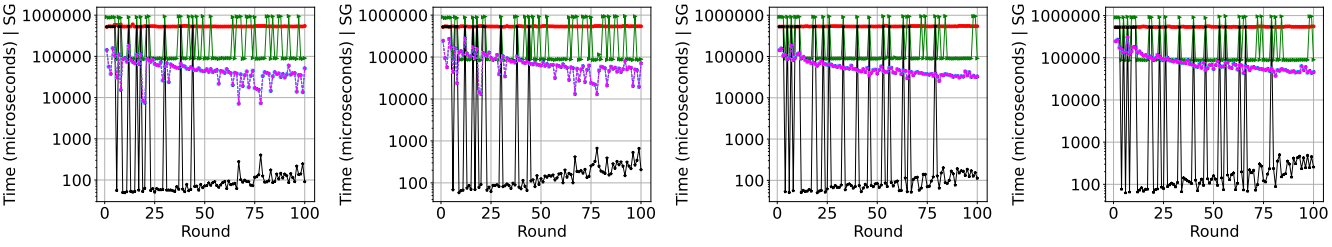
(a) 1M (b) 1.5M (c) 2M

Figure 6: Running time of Scan, Groupy, and Buffer for *Chef* (type E) over 500 rounds in 1M, 1.5M, and 2M users.



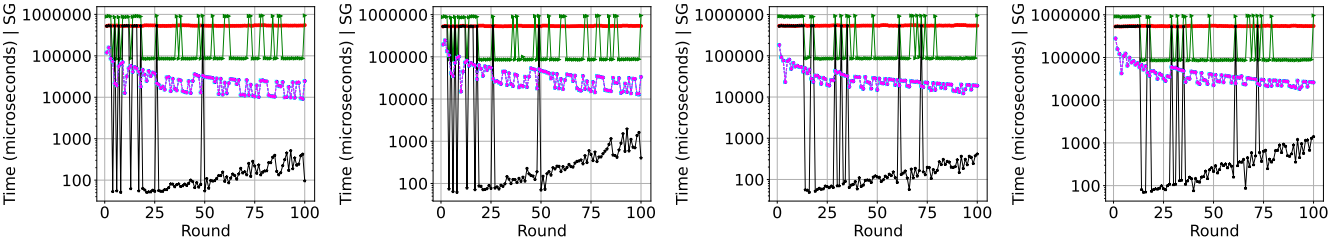
(a) 1M (b) 1.5M (c) 2M

Figure 7: Running time of Scan, Groupy, and Buffer for *Interior Designer* (type F) over 500 rounds in 1M, 1.5M, and 2M users.



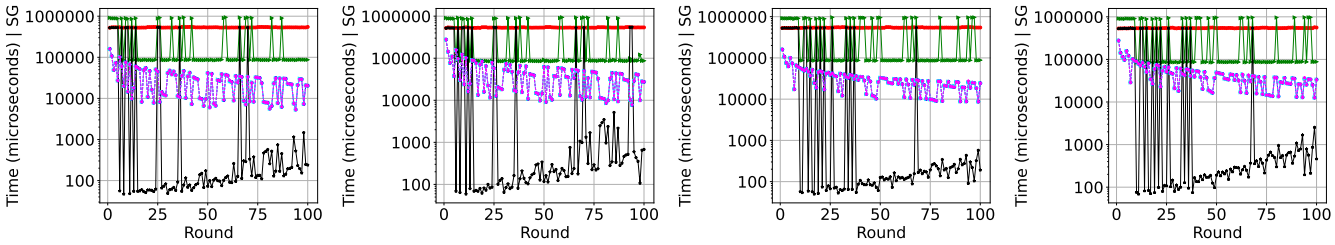
(a) $k = 2$, $m = 10$ (b) $k = 2$, $m = 20$ (c) $k = 3$, $m = 10$ (d) $k = 3$, $m = 20$

Figure 8: Running time and SG results of Scan, Groupy, and Buffer for *Beautician* (type A) over 100 rounds in 1M users.



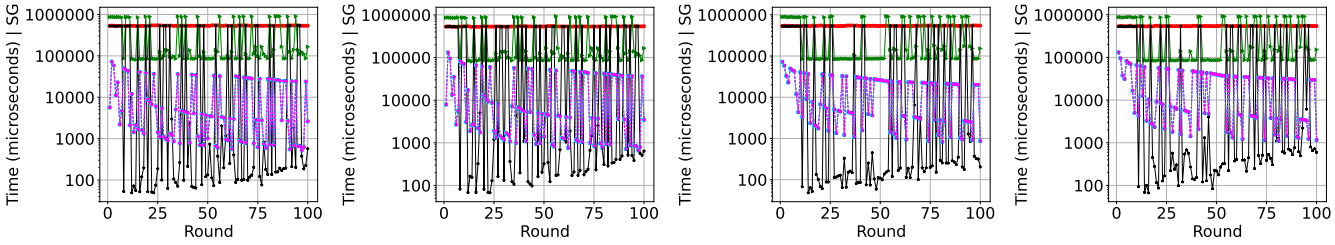
(a) $k = 2$, $m = 10$ (b) $k = 2$, $m = 20$ (c) $k = 3$, $m = 10$ (d) $k = 3$, $m = 20$

Figure 9: Running time and SG results of Scan, Groupy, and Buffer for *Traveler* (type B) over 100 rounds in 1M users.



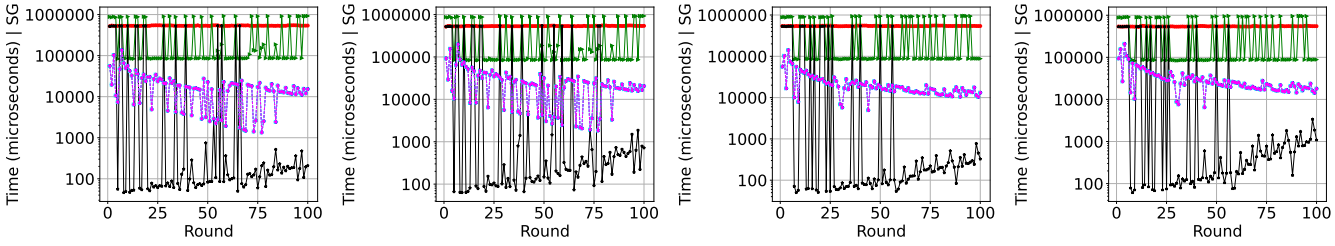
(a) $k = 2$, $m = 10$ (b) $k = 2$, $m = 20$ (c) $k = 3$, $m = 10$ (d) $k = 3$, $m = 20$

Figure 10: Running time and SG results of Scan, Groupy, and Buffer for *Educator* (type C) over 100 rounds in 1M users.



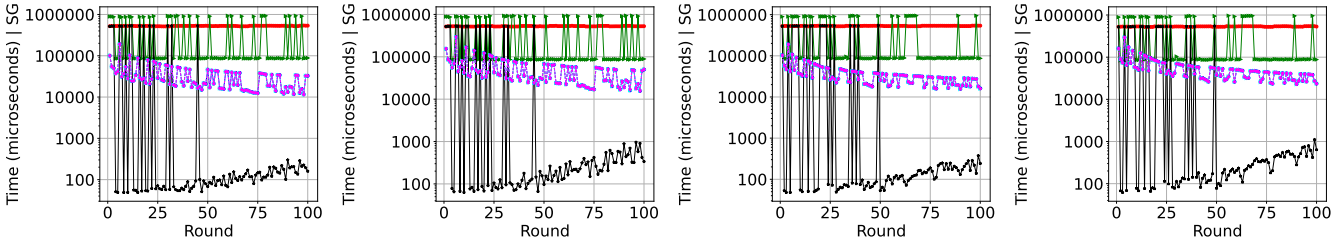
(a) $k = 2$, $m = 10$ (b) $k = 2$, $m = 20$ (c) $k = 3$, $m = 10$ (d) $k = 3$, $m = 20$

Figure 11: Running time and SG results of Scan, Groupy, and Buffer for *Economist* (type D) over 100 rounds in 1M users.



(a) $k = 2$, $m = 10$ (b) $k = 2$, $m = 20$ (c) $k = 3$, $m = 10$ (d) $k = 3$, $m = 20$

Figure 12: Running time and SG results of Scan, Groupy, and Buffer for *Chef* (type E) over 100 rounds in 1M users.



(a) $k = 2$, $m = 10$ (b) $k = 2$, $m = 20$ (c) $k = 3$, $m = 10$ (d) $k = 3$, $m = 20$

Figure 13: Running time and SG results of Scan, Groupy, and Buffer for *Interior Designer* (type F) over 100 rounds in 1M users.

the more competitive *SG* value of each notified user. Fourth, more different queries (larger $|Q|$ derived from higher k) usually induce more $buffer$ initializations for Buffer but less $buffer$ replacements, while usually incur better cache utilization for Groupy; any performance fluctuations depend on the random seed of each query type as we also mentioned for Figures 2–7. Last, all *running time* results are in line with the ones discussed in previous section.

5 GSM Work

Here, we elaborate on basic aspects of our prior work that proposed the mentioned (k, m)-queries to address the *Gaining Subscribers by Messaging* (GSM) problem in [26]. We do that to clarify the difference between the RGSM problem (k-queries) we study in this work and the GSM problem ((k, m)-queries) we studied before [26].

We remind the definition of **GSM problem** as follows:

Problem 2. *Given a social network $G = (V, E)$ with $|V|$ users and $|E|$ edges, a feature universe L, a weighted feature set F_v of size $|L|$ capturing the preferences of each user v, a budget k, a limited number of m notification messages, a similarity threshold d, and a number of rounds n, find in each round t what k content features to publish and which m users to notify (via messages containing the k features) so as to maximize the cumulative subscription gain SG over n rounds:*

$$SG = \underset{k,m}{\text{argmax}} \sum_{t=1}^{n} SG_t(k, m) \quad s.t.\ k \text{ and } m \text{ sought} \tag{2}$$

Same to RGSM (Section 2), $SG_t(k, m)$ denotes the *subscription gain* of m users over k features for round t; $SG_t(k, m)$ is defined as a weighted sum depicting the aggregate preference of m users to k features in round t. Note also that although F_v in GSM can express *dynamic preferences* as in RGSM, in our GSM work [26] we considered only *static preferences* (fixed F_v; once tuned never change). Still, as in RGSM, a user can be notified *only once* in GSM.

We continue by describing the execution of Substitute algorithm that solves the GSM problem (Substitute is the best and optimized version of all other algorithms studied in [26]). Algorithm Substitute presents its execution for GSM.

Algorithm Substitute

```
   Input  : V, E, L, F_v, k, m, d, n
   Output : SG, R // total subscription gain and reach (out-degree) of
                  m users over n rounds
1  SG, R = 0;  notified = ∅; // notified users over n rounds
2  for each c_f from k-feature-combinations-of-L do
3      fC[c_f] = ∅; // the m users with highest SG for c_f
4      selV[c_f].insert((v, v.SG)); // desc-sorted on v.SG
5  for t = 1, ..., n do
6      roundBest = ∅; // entry: ⟨users, features, SG, R⟩
7      for each c_f from k-feature-combinations-of-L do
8          if users in notified not contained in fC[c_f] then
9              Check whether fC[c_f] updates roundBest using d
               for fC[c_f].SG comparisons and fC[c_f].R as second
               filter when needed;
10             continue; // proceed with the next c_f
11         uC = ∅; // the m-size user combinations for c_f
12         Enrich uC the minimum the possible by using
           SG-comparison-optimizations based on d over a gradual
           processing of users in selV[c_f];
13         Compute fC[c_f] by considering all entries in uC;
14         Repeat Line 9 to possibly update roundBest;
15     SG = SG + roundBest.SG;  R = R + roundBest.R;
16     for each v ∈ roundBest.users do notified.insert(v);
17     Delete from each selV[c_f] all users in notified;
18 return SG, R;
```

The main logic of Substitute is in each round t to compute *roundBest* (includes the (k, m) solution) by constantly updating it via the processing of each feature combination c_f. In particular, Substitute first initializes $fC[c_f]$ and computes $selV[c_f]$ for each c_f (Lines 2–4); the former intended to store the m-size user combination that yields the highest SG for cf, while the latter stores a portion of users in V (based on n value) in a descending order on their personal SG values ($v.SG$) for c_f. Then, it starts the processing of each round for the computation of *roundBest* (Line 5).

There are two ways for each c_f to update *roundBest*. The first and time-saving way is the *notified* users of previous rounds to not included in $fC[c_f]$, and so $fC[c_f]$ should not be computed again; in that case, $fC[c_f]$ uses its existing value to may update *roundBest* (Lines 8–9). The second and time-costly way is Substitute to generate all the necessary m-size user combinations for comparison so as to find the m users with highest SG for c_f to be stored in $fC[c_f]$, and use that $fC[c_f]$ to may update *roundBest* (Lines 11–14).

For $fC[c_f]$ to update *roundBest*, it entails that either $fC[c_f]$ yields clearly higher SG than *roundBest* based on the similarity threshold d, or $fC[c_f].SG$ and *roundBest*.SG are similar (based on d) but the reach of $fC[c_f]$ (total out-degree in G of m users in $fC[c_f]$) is higher than of *roundBest* ($fC[c_f].R > roundBest.R$).

Regarding the second case about updating *roundBest* (Lines 11–14) that is the main costly part of the algorithm, the SG-comparison-optimizations relate with the processing of $selV[c_f]$. Specifically, Substitute gradually examines which lower-ordered users in $selV[c_f]$ can *substitute* higher-ordered users in $selV[c_f]$ until no more substitutions are possible. The meaning of a *substitution* is that a lower-ordered user can generate (after substitution) an m-size user combination to include in uC (Line 11) that is necessary to be included to avoid false misses during the total comparison over uC for finding the m users with highest SG for $fC[c_f]$; uC operates as a *knowledge base* that expands with each new substitution. However, to improve the efficiency of Substitute, in [26] we applied additional optimizations (related with the notion of *invisibility* in [26]) that enable some false misses to take place, and so the optimal SG value per round (and overall) to be affected. In short, a user is marked *invisible* in $selV[c_f]$ when it is estimated that the user (besides the fact that the user can substitute others) cannot yield a strong enough m-size user combination to include in uC; so, that user does not expand uC and it is ignored in the rest processing of $selV[c_f]$.

Having described the Substitute algorithm, we emphasize that Substitute cannot meaningfully apply to solve the RGSM problem (we study in this paper), and in case of adaptation, it would be similar to Scan algorithm we presented for RGSM in Section 3. Specifically, in RGSM the k is fixed, so Substitute will have a single c_f to consider. Also, the RGSM problem does not consider the similarity threshold parameter d, so all the optimizations of Substitute based on d are not necessary for RGSM. Further, RGSM does not use a second filter (like the *reach* used in GSM). Last, RGSM considers dynamic preferences (F_v change over rounds) instead of static preferences used in GSM. After considering all such details, for Substitute to solve RGSM it needs to just recompute $selV[c_f]$ (where c_f is fixed) in each round (due to dynamic change of F_v) and take the m highest-ordered users in $selV[c_f]$ to compute the SG of current round. That functionality of $selV[c_f]$ is modeled by the priority queue Q_1 used in Scan (Section 3).

Finally, it is important for readers to understand that the technical differences among GSM and RGSM problems derive from their different applicability. In more detail, the (k, m)-queries used in GSM, help an advertiser to *explore* what content would be promising when the advertiser does not have concrete ideas about what content to publish in their brand page. By the time the advertiser

finds what kind of content works well for their page (after running several (k, m)-queries) then the advertiser may stop issuing (k, m)-queries and may start issuing k-queries used in RGSM by focusing on specific aspects of content (fixed k). We claim that the real-time applicability is not so critical for the *exploration* nature of (k, m)-queries, and so longer response times to such queries are permissible for GSM purposes. In contrast, k-queries are much more *frequent* in real world (as explained in Section 1), and that does requests for their real-time response that it is *feasible* to be achieved due to the search of a single parameter (m) in RGSM instead of two parameters (k, m) that are sought in GSM. To enhance further the chances for k-queries to apply under real-time settings, we do not use in RGSM the discussed parameters d and R that are used in GSM. To conclude, the more complex modeling around (k, m)-queries leaves their real-time response an *open research challenge* whereas the more frequent but simplified k-queries allow their real-time study *in a feasible way*, and that is the focus of this paper.

6 Related Work

Gaining subscribers is valuable to brands for *Influence Maximization* (IM) purposes; we describe why in the next paragraph. IM asks for the k users who can maximize the influence of a given post in a social network, and is a very popular problem that has been intensely studied the last two decades. The work in [11] was the one that first formally studied the IM problem and subsequent works tried to improve its efficacy and efficiency [1–5, 7, 15, 23]. The work in [16] presents a detailed survey on IM and discusses several of its variants. One of such variants is the *Content-Aware Influence Maximization* (CAIM) problem that we introduced and studied in [10]. CAIM can be considered as the inverse variant of IM, since instead of seeking for k users to promote a given post, in CAIM we search for k features to tune an influential post that starts its propagation from the social network page subscribers of major-advertised brand in that post. CAIM paved the way for next works [12, 24, 26] to achieve content-based influence in a network via looking for the proper features to form the relative content.

Gaining subscribers is crucial both for IM and CAIM. In the former, the k sought influential users to initiate the propagation of a given post, are more feasibly and beneficially found in the page subscribers set of interested-to-get-advertised brand. Having more subscribers entails that even more influential and loyal initial adopters can be found in the subscribers set. In the latter, the post propagation takes place only in case that there are some subscribers in the brand page. All such subscribers are utilized in the propagation process as all of them operate as initial adopters; so, the more the page subscribers the higher the propagation of post in a network.

In addition, *gaining subscribers* is vital when the *network topology* of online systems is *partially* agnostic [9, 13, 22]; e.g., it is observed that many social networks provide limited access to their data. In such cases, the subscribers of respective brand are the *sole* users for analysis and advertising since any or all connections beyond their level may not be available; so, all the influence techniques are exclusively deployed and depending on those subscribers which are always accessible even if that access requests a manual retrieval.

The first endeavor to *gain subscribers using content* is our prior work in [26] that we analytically discussed in Section 5 in order to clarify the difference among that work and RGSM. Moreover, other works relative to RGSM problem are the studies [19, 20, 27, 29]; [29] implements a system that learns how to extract *keywords* from web pages for advertisement targeting; [27] shows that extracted *keywords* from different types of meta-data are better suited for content-based recommendation than the manually assigned ones; [20] analyzes the performance of search engine queries comprising *keywords* relative to gender and branded terms; [19] improves the performance of e-commerce session-based recommendation by generating *keywords* entirely from the click sequence in the current session. Yet, all previous works stimulate the interest of users by *searching* the proper keywords to do that whereas RGSM relies on *given* keywords. So, their methods are not suitable for RGSM since they are scheduled based on a different objective function.

Our *gaining subscribers* problems (GSM in [26] and RGSM here) can also naturally and meaningfully utilize the components deployed in our recently published works [24, 25]. Specifically, the work [24] adaptively finds (based on social network feedback) influential features to tune the content of a post in each round, aiming to maximize the cumulative influence of posts in a social network over all rounds. Features are randomly selected in each round from a set of *non-eliminated* features. Instead of random selection, features could be selected by the advertiser of brand page for GSM or RGSM purposes without affecting the functionality of learners developed in [24]. In addition, the work [25] finds how similar are two features; we remind that each feature corresponds to a social network page in all our relevant works [10, 24–26], and current one. So, highly similar features derived from [25] could enrich the search feature space for GSM and the capacity of Q for RGSM.

Finally, it still makes sense to refer some *complementary studies* that can stimulate further the interest of m non-subscriber users in GSM and RGSM problems; further stimulation relates with any actions that additionally boost the intention of notified users (found m users in GSM and RGSM) to subscribe to the brand page of respective advertiser. Such complementary studies pertain to *personalized influence maximization* works [8, 14, 17, 18, 21] that maximize the influence on *targeted* users based on their preferences, and to *active friending* works in social networks [28, 30]. Active friending is a recommendation strategy that guides the interested user to systematically approach their *specific* friending targets; referred works maximize the probability of friending targets to accept the invitation of interested user (become social friends with user).

7 Conclusion

We proposed how brands can utilize the *existing content* in their pages to online systems (social networks, e-commerce) to increase their *subscribers* in *real-time*. Specifically, we studied the novel RGSM problem and deployed the algorithms Scan, Groupy, and Buffer to solve it over many and scalable users with dynamic preferences. Results showed that, overall, Buffer *actually* solves RGSM in *real-time*, and this fact aligns with the targets of this work.

Acknowledgments

This research/project is supported by the National Research Foundation, Singapore under its AI Singapore Programme (AISG Award No: AISG2-RP-2021-020).

References

[1] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. 2014. Maximizing Social Influence in Nearly Optimal Time. In *SODA*. 946–957.

[2] Wei Chen, Chi Wang, and Yajun Wang. 2010. Scalable Influence Maximization for Prevalent Viral Marketing in Large-Scale Social Networks. In *KDD*. 1029–1038.

[3] Wei Chen, Yifei Yuan, and Li Zhang. 2010. Scalable Influence Maximization in Social Networks under the Linear Threshold Model. In *ICDM*. 88–97.

[4] Yi-Cheng Chen, Wen-Chih Peng, and Suh-Yin Lee. 2012. Efficient algorithms for influence maximization in social networks. *KAIS* 33, 3 (2012), 577–601.

[5] Suqi Cheng, Huawei Shen, Junming Huang, Wei Chen, and Xueqi Cheng. 2014. IMRank: Influence Maximization via Finding Self-Consistent Ranking. In *SIGIR*. 475–484.

[6] Kevin Curran, Sarah Graham, and Christopher Temple. 2011. Advertising on Facebook. *International Journal of E-Business Development* 1, 1 (2011), 26–33.

[7] Amit Goyal, Francesco Bonchi, and Laks V. S. Lakshmanan. 2011. A Data-Based Approach to Social Influence Maximization. *PVLDB* 5, 1 (2011), 73–84.

[8] Jing Guo, Peng Zhang, Chuan Zhou, Yanan Cao, and Li Guo. 2013. Personalized Influence Maximization on Social Networks. In *CIKM*. 199–208.

[9] Thibaut Horel and Yaron Singer. 2015. Scalable Methods for Adaptively Seeding a Social Network. In *WWW*. 441–451.

[10] Sergei Ivanov, Konstantinos Theocharidis, Manolis Terrovitis, and Panagiotis Karras. 2017. Content Recommendation for Viral Social Influence. In *SIGIR*. 565–574.

[11] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the Spread of Influence through a Social Network. In *KDD*. 137–146.

[12] Ansh Khurana, Alvis Logins, and Panagiotis Karras. 2020. Selecting Influential Features by a Learnable Content-Aware Linear Threshold Model. In *CIKM*. 635–644.

[13] Paul Lagrée, Olivier Cappé, Bogdan Cautis, and Silviu Maniu. 2019. Algorithms for Online Influencer Marketing. *TKDD* 13, 1 (2019), 1–31.

[14] Jong-Ryul Lee and Chin-Wan Chung. 2014. A query approach for influence maximization on specific users in social networks. *TKDE* 27, 2 (2014), 340–353.

[15] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne VanBriesen, and Natalie Glance. 2007. Cost-Effective Outbreak Detection in Networks. In *KDD*. 420–429.

[16] Yuchen Li, Ju Fan, Yanhao Wang, and Kian-Lee Tan. 2018. Influence Maximization on Social Graphs: A Survey. *TKDE* 30, 10 (2018), 1852–1872.

[17] Yuchen Li, Dongxiang Zhang, and Kian-Lee Tan. 2015. Real-Time Targeted Influence Maximization for Online Advertisements. *PVLDB* 8, 10 (2015), 1070–1081.

[18] Qi Liu, Zheng Dong, Chuanren Liu, Xing Xie, Enhong Chen, and Hui Xiong. 2014. Social marketing meets targeted customers: A typical user selection and coverage perspective. In *ICDM*. 350–359.

[19] Yuanxing Liu, Zhaochun Ren, Wei-Nan Zhang, Wanxiang Che, Ting Liu, and Dawei Yin. 2020. Keywords Generation Improves E-Commerce Session-Based Recommendation. In *WWW*. 1604–1614.

[20] Partha Mukherjee and Bernard J. Jansen. 2014. Performance analysis of keyword advertising campaign using gender-brand effect of search queries. *Electronic Commerce Research and Applications* 13, 2 (2014), 139–149.

[21] Hung T. Nguyen, Thang N. Dinh, and My T. Thai±. 2016. Cost-aware Targeted Viral Marketing in billion-scale networks. In *INFOCOM*. 1–9.

[22] Lior Seeman and Yaron Singer. 2013. Adaptive Seeding in Social Networks. In *FOCS*. 459–468.

[23] Youze Tang, Xiaokui Xiao, and Yanchen Shi. 2014. Influence Maximization: Near-Optimal Time Complexity Meets Practical Efficiency. In *SIGMOD*. 75–86.

[24] Konstantinos Theocharidis, Panagiotis Karras, Manolis Terrovitis, Spiros Skiadopoulos, and Hady W. Lauw. 2024. Adaptive Content-Aware Influence Maximization via Online Learning to Rank. *TKDD* 18, 6, Article 146 (2024).

[25] Konstantinos Theocharidis and Hady W. Lauw. 2024. Community Similarity based on User Profile Joins. In *EDBT*. 572–583.

[26] Konstantinos Theocharidis, Manolis Terrovitis, Spiros Skiadopoulos, and Panagiotis Karras. 2022. A Content Recommendation Policy for Gaining Subscribers. In *SIGIR*. 2501–2506.

[27] Christian Wartena, Wout Slakhorst, and Martin Wibbels. 2010. Selecting Keywords for Content Based Recommendation. In *CIKM*. 1533–1536.

[28] De-Nian Yang, Hui-Ju Hung, Wang-Chien Lee, and Wei Chen. 2013. Maximizing acceptance probability for active friending in online social networks. In *KDD*. 713–721.

[29] Wen-tau Yih, Joshua Goodman, and Vitor R. Carvalho. 2006. Finding Advertising Keywords on Web Pages. In *WWW*. 213–222.

[30] Yapu Zhang, Jianxiong Guo, Wenguo Yang, and Weili Wu. 2020. Targeted activation probability maximization problem in online social networks. *IEEE Transactions on Network Science and Engineering* 8, 1 (2020), 294–304.