Community Similarity based on User Profile Joins

Konstantinos Theocharidis
Singapore Management University
Singapore

Hady W. Lauw
Singapore Management University
Singapore

ABSTRACT

Similarity joins on multidimensional data are crucial operators for recommendation purposes. The classic ϵ-join problem finds all pairs of points within ϵ distance to each other among two d-dimensional datasets. In this paper, we consider a *novel and alternative* version of ϵ-join named *community similarity based on user profile joins* (CSJ). The aim of CSJ problem is, given two communities having a set of d-dimensional users, to find *how similar* are the communities by matching every *single* pair of users (a user can be matched with at most one other user) having an *absolute difference* of at most ϵ per dimension. Each dimension in each user vector stores a counter that measures the number of user preferences to a concrete general topic. CSJ uses *low* ϵ and applies the *strict* condition per dimension so as to really find similar *user profiles* among two communities. CSJ applies to a number of cases in which the popular *community detection* and *community search* problems do not suit. This happens since CSJ treats *existing* communities as brands of a *specific* commercial value and *does not* search to form *general* communities as prior works do; these two community types *semantically* differ. Also, CSJ does not rely its execution on social or/and physical links among community users, instead, it only focus on the similarity of *user profiles*. We deploy a suite of 6 methods to solve CSJ; 3 *approximate* and 3 *exact* algorithms. We evaluate our solutions to meaningful *case studies* of real and synthetic datasets having different characteristics. Our experimental results show interesting and diverse conclusions of CSJ applicability to realistic scenarios.

1 INTRODUCTION

Nowadays, social networks and e-commerce (*online systems*) have been established as the main platforms of user interactions. Users in such systems form *communities* (densely connected user subgraphs) via which they exchange opinions about various topics. Much research has been deployed the last years for the *community detection* [55] and *community search* [20] problems. The former finds a number of communities in a network while the latter finds a specific community that contains a given user.

The common point on finding a community is the *structural dependency*, namely the dense social links among community users. In addition to that, some other criteria on forming a community can be further considered, such as the *spatial dependency* [17] (users to be physically close), the *attribute dependency* [16] (each user has specific interests mentioned as topics, keywords, or attributes), and the *topic-aware social influence dependency* [1] (a user influences another with a specific weight for a certain topic).

The main goal of mentioned works is to find a community so as to exploit its members for advertising purposes; e.g., a conference organizer finds suitable members to invite in regards to the topic of the conference. Nevertheless, in this paper, our interest is to exploit the communities as *entities* themselves (namely, as *brands* having a specific *commercial value*). That is inherently feasible since such kind of communities naturally exist in online systems corresponding to *pages* maintained by brand advertisers for social user engagement. In particular, users in online systems opt to subscribe to (*follow*) the online page of a brand in case they want to keep informed about the latest posts (products, ideas, etc.) of brand. The commercial value of a brand is the number of its subscribers, so we claim that every page can be treated as a brand since every page has some subscribers. Prior works completely overlook the fact that such kind of *brand-communities* already exist, and they find a kind of *topic-communities* that are general enough and so cannot be treated as brands. For example, finding a *general* community of people that like *yoga* does not depict the *specific* different brands that provide equipment and services about and around *yoga*. Also, a *less general* search is useless since specific *yoga* communities *already exist* as explained.

1.1 Problem

By contrast, the type of communities we consider in this paper, induce a very basic operation that cannot be addressed neither by *community detection* nor by *community search*. The requested operation is the *community similarity* that we define it as follows: Given two communities (brands) having a set of users (subscribers) where each user is modeled as a d-dimensional vector (each dimension captures an aggregate number of user preferences to a specific category such as *Entertainment*, *Animals*, etc.), find how similar are the communities based on how similar are their users; each user from one community can be matched with *only one* user from the other community (a pair can have the same user). Two users are similar (matched) if their *absolute difference* per dimension is at most ϵ (given threshold); the ϵ used in practice is usually as *minimum* as possible to *really* find similar users. The *strict* condition of absolute difference per dimension enhances further that and justifies our intention of actually seeking for similar *user profiles* among users.

The problem of *community similarity based on user profile joins* (CSJ) is essentially a novel *similarity join* operator we propose in this work. The closest counterpart to the CSJ problem is the ϵ-join [4, 30] that finds all pairs of points within ϵ distance to each other among two d-dimensional datasets. Yet, CSJ has three *distinct* characteristics compared to classic ϵ-join. First, CSJ depends on finding one-to-one user pairs instead of all user pairs among datasets. Second, the ϵ condition is applied per dimension and not over all dimensions in an aggregated way as e.g., Euclidean distance. Third, CSJ uses a meaningful value for ϵ and so avoids the issues of finding a good value for ϵ in regards to the selectivity of the join. Nevertheless, in this paper, we adapt the *state-of-the-art* work [30] of classic ϵ-join problem in order to show how it performs for the CSJ problem.

We emphasize that in CSJ, the *strict* condition of same threshold (ϵ) per dimension, *exclusively* relates with *aggregate counters* per dimension as each dimension represents a *category*. Such category-dimensions are present to a variety of cases in real life. For instance, in social networks, each dimension maps to a category (e.g., *Hobbies*) that has a plethora of pages within it; counter for *Hobbies* is the total number of posts liked by a respective user where each such post published in any page belonging to *Hobbies*. Likewise, category-dimensions can be found in e-commerce (e.g., *Amazon*[1]), movie platforms (e.g., *Netflix*[2]), or song databases (e.g., *Spotify*[3]). In all these cases, a user *constantly* consumes products, movies, or songs (respectively) that relate with specific categories, and so the associated counters to those categories are increased. For example, when a user views a movie that belongs to categories *comedy* and *romance*, the counters in dimensions that map to *comedy* and *romance* increase by one.

1.2 Applicability

The *applicability* of CSJ is evident in several *real-world* scenarios that cannot be adequately treated by existing works (*link-based joins* (LBJ), and *community detection* or *search* (CDS)) since the latter depend on *user connections* (e.g., social, physical links), and that restricts their applicability, as we analytically explain below:
(i) Friend Recommendation. LBJ works [47, 56], which are utilizing SimRank [28] and PageRank [29] similarity measures, join two users based on their social links. Namely, two users are similar (joined and recommended to each other) if they have several common friends. Yet, such a recommendation has two crucial deficiencies. First, it is finite, as there is a point that after that, all the important connections among users will have been explored and so no more friends will be recommended to the respective users. Second, several of the recommended friends may present no interest to the notified users; e.g., the best friend (recommended) of the mother of a user's best friend may have no interest to that user. Instead, nowadays, friend recommendation mostly relies on *similar preferences* among users and not so much on *structural connections* among them. For instance, the social network *LinkedIn*[4] notifies a user x to follow another user y by directly sending to x the message "*people with similar interests (with you) follow user y*". Another similar message to x used in the social network *VK*[5] is "*you have p% similar taste in Music with y*"; which implicitly recommends x to follow y.

Similarity of user preferences can be captured by the *user subscriptions* to communities of the online system; each common subscription found among two users increases further the similarity of such users. However, besides the *same subscriptions*, there is a number of *similar subscriptions* among users that can be found by CSJ and significantly improve friend recommendation.

Also, note that CDS works can find an attributed community for *many* users or a *given* user based on a subset of user's interests (attributes). Still, the recommended friends (community members) to the *relative* user should be connected via a *small number of hops* to that user, and that yields only a *small portion* of total results that can be found in the whole online system. Yet, CSJ finds *holistic* solutions as is not based on user connections.
(ii) Brand Recommendation. As mentioned, each *brand* depicts a specific social network page having a certain number of subscribers. Brand recommendation *exclusively* depends on user preferences and so LBJ works are not applicable to this kind of recommendation since LBJ utilize only structural connections; their usage is restricted to *friend* (user account) *recommendation* as previously discussed. Still, CDS works can apply but they do it having severe limitations.

[1]https://amazon.com/
[2]https://netflix.com/
[3]https://spotify.com/
[4]https://linkedin.com/
[5]https://vk.com/

(ii.a) Business Partner Recommendation. A brand can find promising business partners to cooperate with by solving the CSJ problem. E.g., *Dior* has a contract with *Charlize Theron* for many years; *Kate Winslet* is one of the brand ambassadors of *Longines*. As both celebrities have success on advertising their products, *Dior* and *Longines* could search for similar celebrities to them respectively to form new lucrative collaborations.

To solve that problem, CDS works should first find *all* or a *subset* of communities with a main preference on e.g., *Charlize Theron*. Then, they could select the second most popular preference to each one of formed communities. Last, they could compare the results over all communities, and with a majority-rule way to find the final answer (next business partner). This process is not only prohibitively time-consuming but it is also result-limited due to the inherent constraint of structural connectivity in CDS works. In CSJ, two users can be similar based on their preferences without being connected at all, and that yields unrestricted results. Also, this fact enables the brands (partners) selection for comparison to be much more flexible and the final execution time to be much faster.
(ii.b) Broadcast Recommendation. The online system applies CSJ to a *variety* of community pairs and guided by the results it organizes a prioritized broadcast recommendation process to users. E.g., in case CSJ finds that *Nike* and *Adidas* pages are more similar than *Nike* and *Puma* pages, then the online system recommends to all platform users that follow *Nike* but not *Adidas* and *Puma*, the latter two pages but in different hours; e.g., at the highest peak hour of user engagement in the platform, *Adidas* is recommended, at the second highest hour *Puma* is recommended.

In that scenario, CDS works can *hardly* apply since there is no a specific preference for the attributed communities to be formed on. This means that for *every page* (*preference*) belonging to the *variety* of communities for comparison, communities should be formed and CDS works to follow a process similar to the aforementioned (ii.a)-case of finding partners. Yet, such a solution is not practical due to the much higher computational cost in regards to prior (ii.a)-case.
(ii.c) Content Recommendation. Each community can be treated as a content *feature* of a post published in online system. For example, *content-aware influence maximization* works [27, 32] utilize such kind of features to *tune* the content of a post so as to be viral in the online system. The CSJ results can help a brand in such works to influence a non-subscriber user on different (but similar) features, and *repetitively* doing that, they could gain the subscription of user [48]. Further, CSJ results can help brands to form *diverse* posts (not having the same concept) and *coherent* posts (each feature within a post to naturally coexist with others).

That scenario is a *middle case* among (ii.a) and (ii.b) for CDS works. Namely, the kind of features in current post along with recent past posts may request ad-hoc feature comparisons ((ii.a)-case) or several feature comparisons ((ii.b)-case). The CDS deficiencies are the same and worsen as moving from (ii.a) to (ii.b).

1.3 Contributions

The main *contributions* of this paper are the following:
Problem. We propose and study the CSJ problem that constitutes a *new and alternative* variant of classic ϵ-join operator. Given two communities (brands), CSJ finds their similarity by applying a *strict ϵ absolute difference* condition per dimension over *single* pairs of users; a user can be matched with at most one other user.
Methods. We present a suite of 6 methods to solve CSJ along with an insightful discussion over them. Specifically, we deploy

three *approximate* and three *exact* solutions that are both useful under different settings. The best CSJ methods employ a novel *encoding scheme* that has a general applicability besides CSJ.

Experiments. We present analytic experiments for different *case studies* on real and synthetic datasets. Results show the importance of CSJ and its interesting technical challenges.

2 RELATED WORK

Similarity Joins. *Similarity joins* are important operators finding applicability in fields such as data mining, data cleaning, recommendation, clustering, outlier detection, association rule mining, etc. [6, 9, 14, 23, 31, 33, 34, 42]. They were first utilized as spatial joins [39, 43] but through the years they undergo several variations; they have been modified to operate as set joins [2, 3, 45, 50, 51, 54], string joins [22], spatio-textual joins [5, 49], link-based joins [47, 56], and ϵ-joins [4, 30, 37]. Other works relative to similarity joins such as similarity search and top-k retrieval can be found in [7, 10, 13, 36].

Spatial joins find pairs of points (entities), one from each dataset, such that they satisfy a specific spatial constraint (e.g., distance, intersection). A special category of them are the ϵ-joins where all pairs of points should be within ϵ distance to each other. Usually, ϵ-joins apply on d-dimensional data [4, 30] where d mostly takes values ranging from 2 to 32. Set joins receive a collection of set-valued data and find pairs of sets in that collection such that their similarity to be no less than a given threshold. Set joins can also be utilized for string matching (find how similar are two strings) that is the problem studied by string joins. Spatio-textual joins are a combination of spatial and set (or string) joins. Namely, given a collection of spatio-textual objects, such a join identifies pairs of objects in that collection that are both spatially close and textually similar. Last, link-based joins take two sets of nodes in a graph and find k pairs of nodes which are ranked the highest depending on linked-based similarity scores like SimRank [28] or PageRank [29].

The CSJ problem we study in this paper is related with the ϵ-joins literature. In particular, the popular EGO-join approach is initially proposed in [4]. According to that, the two datasets of points being joined are first sorted based on the EGO (Epsilon Grid Order). The EGO-join procedure pertains to a recursive process and a strategy is utilized for pruning dataset segments that is guaranteed to be non-joinable. Afterwards, several other works tried to improve the performance of EGO-join and the best of them is the work in [30] that also constitutes the *state-of-the-art* for the classic ϵ-join problem. To enable a comparison with that work [30], we adapt its ϵ-join distance condition to *correctly* apply for CSJ purposes.

Community Detection. The problem of *community detection* discovers all the communities in a graph such as social networks or bibliographic networks. Communities found in graphs can be attributed [38, 40, 44, 52, 53, 57] or non-attributed [21, 41], spatial [8, 15, 24] or non-spatial [21, 41]. Yet, community detection is a NP-hard problem [55] and so not suitable for *online* purposes.

Community Search. The problem of *community search* obtains communities based on a *specific* query request; a *given* user who is the member that searched community should include. Due to the *single* community found, community search is much faster than community detection and so more appropriate for online executions. Still, there are works on attributed [16, 19] and non-attributed [11, 12, 25, 26, 35, 46], as also spatial [17] and non-spatial [18] communities. Some works also consider social influences among users based on certain topics [1]. Community search is a more popular problem than community detection due to its practical and online applicability; [20] shows a survey.

3 PROBLEM STATEMENT

The problem of *community similarity based on user profile joins* (CSJ) computes the similarity of two communities by examining the similarity of their subscribers (each user represented as a d-dimensional vector) where one user from one community can be matched with *only one* user from the other community (including the same user in pair); the matched users share similar *user profile*. A *user profile* is similar to another when the *absolute difference* per dimension is at most ϵ; ϵ is a given threshold and is as *minimum* as possible to abide by the *strict* condition per dimension. Formally:

CSJ Definition. Given two communities B and A having $|B|$ and $|A|$ number of subscribers respectively, and $\left\lceil \frac{|A|}{2} \right\rceil \leq |B| \leq |A|$, the CSJ join operator finds the matched d-dimensional user pairs $\prec b, a \succ$ with $b \in B$ and $a \in A$ having $|b_i - a_i| \leq \epsilon$ for each $i \in [1, d]$, where each b can be matched with at most one a and vice versa; b_i and a_i depict the *aggregate counters* of b and a in dimension i. The *exact* method of CSJ finds *all* the matched user pairs $\prec b, a \succ$ among B and A (depicted as *matched_user_pairs*(B, A)), while the *approximate* method of CSJ finds a percentage p of total matched user pairs $\prec b, a \succ$, $p \cdot |$*matched_user_pairs*(B, A)$|$ where $p \in (0, 1]$. In both cases, CSJ computes the *similarity* among B and A as follows:

$$similarity(B, A) = p \cdot \frac{|matched\_user\_pairs(B, A)|}{|B|} \quad (1)$$

, where $p = 1$ for *exact* and $p \in (0, 1]$ for *approximate*

Throughout the paper, we depict the less-followed community by B and the more-followed community by A, while to keep things concise, we refer to *similarity*(B, A) just by using the word *similarity*. We stress that *similarity* is meaningful to be computed only when the size of B is at least the half of the size of A (as dictated in aforementioned CSJ definition), since otherwise, chances are that B will be a significant subset of A and that does not capture the CSJ similarity semantics. Moreover, the logic behind *one-to-one* matches among B and A is that CSJ treats the subscribers of compared communities as a different audience in which it aims to find the maximum number of common users. So, if a user in B matched with a user in A (and vice versa), CSJ interprets the matched users as being the *same person* belonging to a different kind of audience.

To clarify how *exact* and *approximate* methods solve the CSJ problem, we provide the following indicative example:

Example. Assume $\epsilon = 1$ and three categories ($d = 3$) named *Music*, *Sport*, and *Education*. Suppose also two users in B depicted as b_1 = {Music: 3, Sport: 4, Education: 2} and b_2 = {Music: 2, Sport: 2, Education: 3}, and three users in A depicted as a_1 = {Music: 2, Sport: 3, Education: 5}, a_2 = {Music: 2, Sport: 3, Education: 1}, and a_3 = {Music: 3, Sport: 3, Education: 3}. Numbers in previous sets denote the number of posts that users liked where such posts published in pages belonging to respective categories. Since $|B| = 2$ is at least the ceiling half of $|A| = 3$, we proceed to the computation of *similarity*. We see that b_1 can be matched with a_2 and a_3 while b_2 can be matched only with a_3. An *exact* method finds the matched user pairs $\prec b_1, a_2 \succ$ and $\prec b_2, a_3 \succ$ and so it yields a $similarity = 100\%$. On the contrary, an *approximate* method may assign b_1 with a_3, so b_2 is left unmatched, yielding a less accurate $similarity = 50\%$.

This example shows that *approximate* method can be easily less accurate than *exact* method. However, this does not affect its value and importance as a method. The usage of *approximate* method is to *fast* find a group of *similar-enough* community pairs for impending precise *similarity* computation. When such a group is found, the *exact* method applies to improve the *similarity* of community pairs belonging to that group. The *online system* executes the respective recommendation case *exclusively* based on the precise results derived from the *exact* method. Namely, the **time-consuming** *exact* method uses the results of **fast** *approximate* method as input to alleviate its total execution overhead. So, both methods are useful in CSJ for different reasons.

4 MINMAX METHOD

In this section we present the `MinMax` method including its *approximate* and *exact* versions for the CSJ problem. The competitor CSJ methods to `MinMax` are discussed in the next section. We also highlight that `MinMax` uses a novel *encoding scheme* (outlined in Figure 1) that can find a broader applicability besides CSJ.

In Figure 1, a *user vector* of d-size is segmented to 4 *parts* marked by *red*, *green*, *blue*, and *purple* colors for illustration clarity. The selection of a 4-*parts*-segmentation achieves the *best tradeoff* since a lower number of *parts* is more time-costly (due to less effective pruning) and a higher number of *parts* is more space-consuming (due to more memory required over all *parts*). The sum of values over d dimensions gives the *encoded_ID* of user. Also, since $\epsilon = 1$, the *ranges* [2,11], [8,20], [5,16], and [13,26] capture all the possible values[6] per *part* that may satisfy an absolute difference of at most ϵ per dimension. So, the sum of *minimum* and *maximum* values in these *ranges* yields the *encoded_Min* and *encoded_Max* of user. Note that such a user with *encoded_ID* = 46 may be matched for CSJ purposes **only** with users where their *encoded_ID* falls within the range [*encoded_Min* = 28, *encoded_Max* = 73] associated with *encoded_ID* = 46.

`MinMax` algorithms (*approximate* and *exact*) presented in the following, take as input the communities B and A, the number of dimensions d, and the absolute-difference-per-dimension threshold ϵ, and they compute as output the *similarity* among B and A. Further, in the discussion of algorithms, the word *parts* mentions to the sums of each part, e.g., numbers 5, 13, 9, and 19 in Figure 1, while the word *ranges* mentions to the final ranges associated with those parts, e.g., [2,11], [8,20], [5,16], and [13,26] in Figure 1.

Moreover, during the pairing execution process of currently examined users $b \in B$ and $a \in A$, both `MinMax` algorithms yield 5 kind of events, i.e., MIN PRUNE, MAX PRUNE, NO OVERLAP, NO MATCH, and MATCH. An event MIN PRUNE means that current user $b \in B$ cannot be further matched with *any* user $a' \in A$ where $a'.encoded\_Min \geq a.encoded\_Min$. Instead, an event MAX PRUNE denotes that current user $a \in A$ cannot be further matched with *any* user $b' \in B$ where $b'.encoded\_ID \geq b.encoded\_ID$. An event NO OVERLAP means that there is no *complete* overlap among *parts* of current user $b \in B$ and *ranges* of current user $a \in A$, and so there is no need to proceed to an ϵ-comparison on their d-dimensional vectors, since it is sure that they do not match. An event NO MATCH denotes that an ϵ-comparison on prior d-dimensional vectors is executed but no match is found among current users (i.e., there is at least one dimension $i \in [1, d]$ in which the absolute difference of user counters is higher than ϵ). Finally, an event MATCH means that prior d-dimensional comparison matches the examined users.

[6]For instance, the value 2 of range [2,11] relative to 1st-part of user in Figure 1 (let call that user x) can be related to a user y with 1st-part arrangement 0|0|0|0|1|1 and to another user z with 1st-part arrangement 0|2|0|0|0|0. Among y and z, only y has an absolute difference $\leq \epsilon$ with x in regards to their 1st-part. Yet, `MinMax` algorithms should consider both users for examination based on the *encoding scheme* to avoid possible *false misses*.

4.1 Approximate `MinMax`

Algorithm `Ap-MinMax` presents the *approximate* execution of `MinMax` method. It first forms two buffers (structures) named *Encd_B* and *Encd_A* (Lines 1–4). *Encd_B* contains a *triple-entry* for each user $b \in B$, i.e., the encoded ID of user, the 4 parts of that ID, and the real ID of user. *Encd_A* includes a *quadruple-entry* for every user $a \in A$, i.e., the encoded Min and Max of user, the ranges of user encoded ID, and the user real ID. *Encd_B* and *Encd_A* are sorted in ascending order of encoded ID and encoded Min so as to enable the occurrence of MIN PRUNE and MAX PRUNE events. The terms parts, ranges, encoded ID, Min, and Max have been explained in the beginning of this section and are also illustrated in Figure 1.

Algorithm `Ap-MinMax` (Approximate MinMax)

```
Input  : $B, A, d, \epsilon$
Output : $similarity$ // $similarity(B, A)$ depicted in Eq. (1)
1  Compute the $encd\_ID$ and its $parts$ for each user $b \in B$;
2  Store all results ($encd\_ID$, $parts$, $real\_ID$) to a buffer $Encd\_B$
   that is ASC-sorted on $encd\_ID$;
3  Compute the $encd\_Min$, $encd\_Max$, and the $ranges$ of
   $encd\_ID$ for each user $a \in A$;
4  Store all results ($encd\_Min$, $encd\_Max$, $ranges$, $real\_ID$) to a
   buffer $Encd\_A$ that is ASC-sorted on $encd\_Min$;
5  $offset = 0$;
6  for each $eB \in Encd\_B$ do
7      $skip = 1$;
8      for each $eA$ using offset $\in Encd\_A$ do
9          if $eB.encd\_ID < eA.encd\_Min$ then break; // MIN
           PRUNE (go to next $eB$)
10         else if $eB.encd\_ID \geq eA.encd\_Min$ and
           $eB.encd\_ID \leq eA.encd\_Max$ then
11             if $eB.parts$ do not overlap with $eA.ranges$ then
               continue; // NO OVERLAP (go to next $eA$)
12             else Compare $eB$ with $eA$ based on $\epsilon$ using their
               $real\_ID$ to access their $d$-dimensional vectors;
               // MATCH (go to next $eB$) or NO MATCH
13         else $offset$++ when $skip = 1$; // MAX PRUNE
14 $similarity = |matched\_user\_pairs(B, A)| \div |B|$;
15 return $similarity$;
```

The pairing process unfolds in a double-loop fashion over an entry *eB* of *Encd_B* and an entry *eA* of *Encd_A*. There are three main cases. First, in case that the encoded ID of *eB* is smaller than the encoded Min of *eA* then *eB* is *min pruned* as it is sure that there is no a match for it (Line 9). Second, in case that the encoded ID of *eB* falls within the encoded Min and Max range of *eA* then there is a chance of possible matching (Line 10). If *any* of the *parts* of *eB* does not fall within the respective *range* of *eA* then there is no *complete* overlap among them, and so it is sure that *eB* and *eA* do not match (Line 11). Yet, if there is a *complete* overlap among respective *parts* and *ranges*, the d-dimensional comparison takes place (Line 12). In case of *match*, the execution continues with the processing of next *eB* and that justifies its *approximate* nature. Third, if the previous two main cases are not met, it means that the encoded ID of *eB* is bigger than the encoded Max of *eA* (Line 13), and so *eA* can be *max pruned in some settings*.

```
user vector = 1|0|0|0|2|2|0|0|2|1|1|5|4|0|3|0|0|1|4|1|0|3|5|4|1|2|4          ε = 1 , d = 27

1st-Part:  1|0|0|0|2|2     = 5        1st-Part:  1|0|0|0|2|2     => [0,2]|[0,1]|[0,1]|[0,1]|[1,3]|[1,3]        => [2,11]
2nd-Part:  0|0|2|1|1|5|4   = 13       2nd-Part:  0|0|2|1|1|5|4   => [0,1]|[0,1]|[1,3]|[0,2]|[0,2]|[4,6]|[3,5]  => [8,20]
3rd-Part:  0|3|0|0|1|4|1   = 9        3rd-Part:  0|3|0|0|1|4|1   => [0,1]|[2,4]|[0,1]|[0,1]|[0,2]|[3,5]|[0,2]  => [5,16]
4th-Part:  0|3|5|4|1|2|4   = 19       4th-Part:  0|3|5|4|1|2|4   => [0,1]|[2,4]|[4,6]|[3,5]|[0,2]|[1,3]|[3,5]  => [13,26]

encoded_ID = 5 + 13 + 9 + 19          encoded_Min  = 2 + 8 + 5 + 13      =>  encoded_Min = 28
encoded_ID = 46                       encoded_Max = 11 + 20 + 16 + 26 => encoded_Max = 73
```

Figure 1: An example of the *encoding scheme* used in CSJ.

These settings are related with the variables *skip* and *offset* that we also use in other parts of code but we omitted their overall reference for simplicity. In short, *skip* is a flag activated per *eB* and denotes consecutive entries to *eA* that can be safely skipped for next *eB*, operated by the respective movement of *offset*. The deactivation of *skip* is done when a comparison takes place, even a *part-range* comparison. Finally, *similarity* is computed based on the total number of matched pairs (Line 14). Details about gradually forming and using a buffer of matches (depicted as *matched_user_pairs*(B, A); see Eq. (1)) are also omitted.

To further illustrate the execution of Ap-MinMax, we provide a running example in Figure 2 consisting of 8 sequential instances. Numbers in parenthesis are the *ecnd_Min* and *ecnd_Max* of each user $a \in A$ while the number associated with each user $b \in B$ is their *ecnd_ID*. Actually, in each instance, the left column of A users maps to a more compact version of *Encd_A* while the right column of B users maps to a less verbose version of *Encd_B*. All numbers are intentionally selected to be different for clarity. Ap-MinMax finds two matched pairs, $< b_2, a_3 >$ and $< b_5, a_5 >$, so *similarity* = 40%. During execution we observe all the possible events (MIN PRUNE, MAX PRUNE, NO OVERLAP, NO MATCH, and MATCH) that can take place in Ap-MinMax. For example, in instance $\ll$ **1** $\gg$, b_1 is eventually *min pruned* by a_3 after two *no overlap* comparisons with a_1 and a_2. However, in instances $\ll$ **3** $\gg$ and $\ll$ **4** $\gg$, a_1 and a_2 are *max pruned* by b_3, respectively. Further, note that in instance $\ll$ **6** $\gg$, b_4 starts comparing with a_4 by using the *offset* changed by b_3 in instances $\ll$ **3** $\gg$ and $\ll$ **4** $\gg$ where *skip* was activated.

4.2 Exact MinMax

Algorithm Ex-MinMax shows the *exact* execution of MinMax method that computes the maximum possible *similarity* among communities B and A. To avoid *false misses* of Ap-MinMax, Ex-MinMax considers all the possible matches among currently examined user $b \in B$ and its relative matches in A. This consideration is done during the pairing process of each b, and derives from a MIN PRUNE on b along with a MAX PRUNE over all the matches of b to A. To apply such a MAX PRUNE to a set of users in A, Ex-MinMax uses a variable *maxV* (Line 6) to store the *maximum* encoded Max value over users (matches of b) in that set of A.

Ex-MinMax forms *Encd_B* and *Encd_A* in Line 5 as Ap-MinMax does it but also uses 4 new structures denoted in Lines 1–4. E.g., *matched_B* is a *map* that stores for each user b all its matches in A and *sortedM_B* is a *map* that is ascending sorted on the *cardinality* of those matches to all the users in B who have such a *cardinality*. Further, besides *skip* and *offset*, Ex-MinMax also uses the variable *maxV* to enable, as earlier discussed, a MAX PRUNE event over all the matches in A of currently examined user b. The pairing process among *eB* and *eA* entries is similar to the one of Ap-MinMax and the *new code* added is related with the management of *maxV*.

In particular, when a new *match* arises (Line 20), *maxV* may update its value with the encoded Max value of current matched user a (Lines 22–23). In case of a MIN PRUNE event (Line 10), if the encoded ID of next-to-current-user b is greater than *maxV*, this safely (no *false misses*) enables a MAX PRUNE application to all the matches of b in A. So, due to MIN PRUNE, current user b cannot be further matched with A, and due to MAX PRUNE, all the matched users of b in A cannot be further matched with B.

```
Algorithm Ex-MinMax ( Exact MinMax )
   Input   : B, A, d, ϵ
   Output  : similarity // similarity(B, A) depicted in Eq. (1)
 1 matched_B = ∅; // map : a user b TO matches in A
 2 matched_A = ∅; // map : a user a TO matches in B
 3 sortedM_B = ∅; // map : |matches in A| TO relative users b
 4 sortedM_A = ∅; // map : |matches in B| TO relative users a
 5 Form Encd_B and Encd_A as Ap-MinMax in its Lines 1–4;
 6 offset = 0; maxV = 0; // the maximum encoded Max value
   over matches in A found for every examined user in B
 7 for each eB ∈ Encd_B do
 8     skip = 1;
 9     for each eA using offset ∈ Encd_A do
10         if eB.encd_ID < eA.encd_Min then // MIN PRUNE
11             Let the next entry of eB in Encd_B be next_eB;
12             if next_eB.encd_ID > maxV then
                   // MAX PRUNE applies to a set of users in A
13                 Use matched_B and matched_A to fill
                   sortedM_B and sortedM_A, respectively;
14                 Update matched_user_pairs(B, A) by calling
                   the function CSF(matched_B, matched_A,
                   sortedM_B, sortedM_A);
15                 Set maxV = 0 and Empty matched_B,
                   matched_A, sortedM_B, and sortedM_A;
16             break; // go to next eB
17         else if eB.encd_ID ≥ eA.encd_Min and
           eB.encd_ID ≤ eA.encd_Max then
18             if eB.parts do not overlap with eA.ranges then
               continue; // NO OVERLAP (go to next eA)
19             else Compare eB with eA based on ϵ using their
               real_ID to access their d-dimensional vectors;
               // MATCH or NO MATCH
20             if MATCH then
21                 matched_B[eB.real_ID].insert(eA.real_ID);
                   matched_A[eA.real_ID].insert(eB.real_ID);
22                 if eA.encd_Max > maxV then
23                     maxV = eA.encd_Max;
24         else offset++ when skip = 1; // MAX PRUNE
25 similarity = |matched_user_pairs(B, A)| ÷ |B|;
26 return similarity;
```

Hence, the set of *matched_user_pairs*(B, A) get *safely* updated via the function CSF (discussed later) in Line 14. CSF finds the

```
<< 1 >>
a1:(30, 55)    b1:40
a2:(33, 60)    b2:48
a3:(42, 72)    b3:67
a4:(45, 73)    b4:71
a5:(50, 80)    b5:74
=====
* b1:40 IN a1:(30, 55) ⇒ NO OVERLAP
* b1:40 IN a2:(33, 60) ⇒ NO OVERLAP
* b1:40 <  a3:(42, 72) ⇒ MIN PRUNE

<< 2 >>
a1:(30, 55)    b2:48
a2:(33, 60)    b3:67
a3:(42, 72)    b4:71
a4:(45, 73)    b5:74
a5:(50, 80)
=====
* b2:48 IN a1:(30, 55) ⇒ NO MATCH
* b2:48 IN a2:(33, 60) ⇒ NO MATCH
* b2:48 IN a3:(42, 72) ⇒ MATCH

<< 3 >>
a1:(30, 55)    b3:67
a2:(33, 60)    b4:71
a4:(45, 73)    b5:74
a5:(50, 80)
=====
* b3:67 > a1:(30, 55) ⇒ MAX PRUNE

<< 4 >>
a2:(33, 60)    b3:67
a4:(45, 73)    b4:71
a5:(50, 80)    b5:74
=====
* b3:67 > a2:(33, 60) ⇒ MAX PRUNE

<< 5 >>
a4:(45, 73)    b3:67
a5:(50, 80)    b4:71
               b5:74
=====
* b3:67 IN a4:(45, 73) ⇒ NO MATCH
* b3:67 IN a5:(50, 80) ⇒ NO OVERLAP

<< 6 >>
a4:(45, 73)    b4:71
a5:(50, 80)    b5:74
=====
* b4:71 IN a4:(45, 73) ⇒ NO OVERLAP
* b4:71 IN a5:(50, 80) ⇒ NO MATCH

<< 7 >>
a4:(45, 73)    b5:74
a5:(50, 80)
=====
* b5:74 > a4:(45, 73) ⇒ MAX PRUNE

<< 8 >>
a5:(50, 80)    b5:74
=====
* b5:74 IN a5:(50, 80) ⇒ MATCH

MATCHES = {<b2, a3>, <b5, a5>}
```

Figure 2: An example showing the execution of Approximate MinMax.

```
<< 1 >>
a1:(30, 55)    b1:40
a2:(33, 60)    b2:58
a3:(38, 57)    b3:67
a4:(45, 73)    b4:74
a5:(50, 80)    b5:81
=====
* maxV = 0
* b1:40 IN a1:(30, 55) ⇒ MATCH (maxV = 55)
* b1:40 IN a2:(33, 60) ⇒ NO OVERLAP
* b1:40 IN a3:(38, 57) ⇒ MATCH (maxV = 57)
* b1:40 <  a4:(45, 73) ⇒ MIN PRUNE (b2 > maxV)
                       ⇒ CSF(<b1, a1>, <b1, a3>)

<< 2 >>
a2:(33, 60)    b2:58
a4:(45, 73)    b3:67
a5:(50, 80)    b4:74
               b5:81
=====
* maxV = 0
* b2:58 IN a2:(33, 60) ⇒ MATCH (maxV = 60)
* b2:58 IN a4:(45, 73) ⇒ MATCH (maxV = 73)
* b2:58 IN a5:(50, 80) ⇒ NO MATCH (b3 < maxV)

<< 3 >>
a2:(33, 60)    b3:67
a4:(45, 73)    b4:74
a5:(50, 80)    b5:81
=====
* maxV = 73
* b3:67 > a2:(33, 60) ⇒ MAX PRUNE

<< 4 >>
a4:(45, 73)    b3:67
a5:(50, 80)    b4:74
               b5:81
=====
* maxV = 73
* b3:67 IN a4:(45, 73) ⇒ MATCH (maxV = 73)
* b3:67 IN a5:(50, 80) ⇒ NO MATCH (b4 > maxV)
                       ⇒ CSF(<b2, a2>, <b2, a4>, <b3, a4>)

<< 5 >>
a5:(50, 80)    b4:74
               b5:81
=====
* maxV = 0
* b4:74 IN a5:(50, 80) ⇒ NO OVERLAP

<< 6 >>
a5:(50, 80)    b5:81
=====
* maxV = 0
* b5:81 > a5:(50, 80) ⇒ MAX PRUNE

MATCHES = {Results of CSF}
```

Figure 3: An example showing the execution of Exact MinMax.

maximum number of *one-to-one* matches in given set of matched user pairs $< b, a >$. Note that *generally* that set may also include users b' who examined after the examination of current b but we omitted such *edge cases* from Ex-MinMax *code* to keep things simple. Yet, the running example we present later for Ex-MinMax demonstrates such *edge cases*. After CSF call, $maxV$ is set to 0 and the 4 new structures get empty to initiate the process of next segment of B matching with A (Line 15).

Figure 3 presents a running example pertaining to the execution of Ex-MinMax that comprises 6 consecutive instances. For clarity, all numbers are selected to be different and the presentation setting relative to *Encd_A* and *Encd_B* is the same as in Figure 2. The variation in execution depends on the usage of $maxV$. In instance $\ll \mathbf{1} \gg$, b_1 updates two times $maxV$ and $\{a_1, a_3\}$ are the matches of b_1 in A. The MIN PRUNE event on b_1 by a_4 along with MAX PRUNE event on $\{a_1, a_3\}$ due to $b_2 > maxV$, denote that b_1 is *min pruned* (no other match exists for b_1 in A) and $\{a_1, a_3\}$ are *max pruned* (no other match exists for them in B), so CSF can be safely called without losing any other matches relative to b_1, a_1, or a_3. In this case, CSF updates $matched\_user\_pairs(B, A)$ with either $< b_1, a_1 >$ or $< b_1, a_3 >$. Execution flows similar to Ap-MinMax in other instances except for instance $\ll \mathbf{4} \gg$ where CSF is called again. Note that while prior instance $\ll \mathbf{1} \gg$ depicts a *normal case* for the given input to CSF, instance $\ll \mathbf{4} \gg$ captures an *edge case* as the input to CSF contains more than one examined users in B (b_2 and b_3). Instances $\ll \mathbf{2} \gg$–$\ll \mathbf{4} \gg$ show that *edge cases* also use $maxV$ till reaching a *safe point* where all matches are found between current segments B and A before calling CSF. The set $matched\_user\_pairs(B, A)$ has the final matches of Ex-MinMax gradually formed by CSF.

Last, we discuss the run of function CSF (CoverSmallestFirst) that finds the *maximum* number of matches to the current set of matched user pairs $< b, a >$. CSF assigns a match to the *smallest* users (having the *smallest* number of matches) so as to find a pair for them (*cover* them). Covering smaller users, and so excluding them from the pairing process, leaves a bigger portion of available pairs in order more matches overall to be found.

Specifically, CSF utilizes $sortedM\_B$ *and* $sortedM\_A$ in a *loop* (Line 1) to first find the smallest user b (Line 4) or the smallest user a (Line 7). Then, it pairs prior user b with the smallest user a (Line

5) or pairing prior user a with the smallest user b (Line 8). In case of a tie (Line 9), the previous two steps are combined as mentioned (Line 10). When a pair $\prec b, a \succ$ having *minimum* connections in B and A is found, it updates $matched\_user\_pairs(B, A)$ by its insertion and the input of CSF by its deletion (Lines 11–12). The *loop* terminates when all the input of CSF is processed (Line 13).

```
Function CSF : Cover Smallest First
   Input  : $matched\_B$, $matched\_A$, $sortedM\_B$, $sortedM\_A$
   Output : $matched\_user\_pairs(B, A)$
1  while true do
2      $sB = sortedM\_B$.first(); $sA = sortedM\_A$.first(); // get the first entries of respective sorted maps
3      if $sB.|matches\ in\ A| < sA.|matches\ in\ B|$ then
4          for each user $b \in sB$ do
5              Find the user $a \in matched\_B[b]$ having the smallest matches to $B$; // break if single match
6      else if $sB.|matches\ in\ A| > sA.|matches\ in\ B|$ then
7          for each user $a \in sA$ do
8              Find the user $b \in matched\_A[a]$ having the smallest matches to $A$; // break if single match
9      else
10         Repeat Lines 4–5; Repeat Lines 7–8 in case no a user $a \in matched\_B[b]$ found having a single match to $B$;
11     Insert to $matched\_user\_pairs(B, A)$ the found pair $\prec b, a \succ$ having minimum connections in $B$ and $A$;
12     Update $matched\_B$, $matched\_A$, $sortedM\_B$, and $sortedM\_A$ by removing $\prec b, a \succ$ related information;
13     Exit from loop if $sortedM\_B$ or $sortedM\_A$ gets empty;
14 return $matched\_user\_pairs(B, A)$;
```

5 COMPETITOR METHODS

In this section we present two competitor methods to Ap-MinMax and Ex-MinMax. The first is a Baseline method while the second is an adaptation of SuperEGO method used in [30].

5.1 Baseline Method

Approximate Baseline. Approximate Baseline (Ap-Baseline) uses a nested loop; outer for $b \in B$ and inner for $a \in A$. When a match is found among current users b and a, execution proceeds with the next user b. Yet, *skip* and *offset* are used similarly to Ap-MinMax for the faster processing of nested loop join.

Exact Baseline. Exact Baseline (Ex-Baseline) uses a nested loop to first find all matches among B and A. Then, it forms the 4 structures $matched\_B$, $matched\_A$, $sortedM\_B$, and $sortedM\_A$ (same as in Ex-MinMax) and calls *once* the CSF function.

5.2 SuperEGO Method

The SuperEGO method [30] is the *state-of-the-art* algorithm for the classic similarity ϵ-join problem that finds all pairs of points within ϵ distance to each other among two d-dimensional datasets. Algorithm SuperEGO presents the general framework of SuperEGO.

The logic of SuperEGO is to apply a *divide-and-conquer* **recursive** approach on the input community pair $\prec B, A \succ$. Specifically, SuperEGO gradually reduces B and A to increase the chances to prune user pairs $\prec b, a \succ$ in them that it is sure that cannot be matched. This pruning is done by the EGO-Strategy function of SuperEGO (Line 1) that is its core component for efficiency. As B and A get smaller but not pruned by EGO-Strategy then SuperEGO meets one of the four cases mentioned (Lines 2–3, Lines 4–6, Lines 7–9, Lines 10–12). In all such cases we have a split of segments having a size no less than t (predefined parameter of SuperEGO) and then the recursive process continues. The only exception is the *base case of recursion* in Lines 2–3 where denotes the worst-case scenario of SuperEGO. It means that pruning was not possible in previous recursive steps and it is also not meaningful the segments B and A to be further divided; as a result, the classic *nested loop join* applies on respective segments B and A.

```
Algorithm SuperEGO
   Input  : $B, A, d, \epsilon$
   Output : $matched\_user\_pairs(B, A)$
   Param. : $t$
1  if EGO-Strategy($B, A, d, \epsilon$) = 1 then return $\emptyset$;
2  if $|B| < t$ and $|A| < t$ then
3      return NestedLoopJoin($B, A, d, \epsilon$);
4  if $|B| < t$ and $|A| \geq t$ then
5      $\{A_1, A_2\}$ = Split($A$);
6      return SuperEGO($B, A_1, d, \epsilon$) $\bigcup$ SuperEGO($B, A_2, d, \epsilon$);
7  if $|B| \geq t$ and $|A| < t$ then
8      $\{B_1, B_2\}$ = Split($B$);
9      return SuperEGO($B_1, A, d, \epsilon$) $\bigcup$ SuperEGO($B_2, A, d, \epsilon$);
10 if $|B| \geq t$ and $|A| \geq t$ then
11     $\{B_1, B_2\}$ = Split($B$); $\{A_1, A_2\}$ = Split($A$);
12     return SuperEGO($B_1, A_1, d, \epsilon$) $\bigcup$ SuperEGO($B_1, A_2, d, \epsilon$) $\bigcup$
       SuperEGO($B_2, A_1, d, \epsilon$) $\bigcup$ SuperEGO($B_2, A_2, d, \epsilon$);
```

SuperEGO can solve CSJ by *properly* adapting the ϵ parameter. E.g., if $\epsilon = 1$ and $d = 27$ then the relative ϵ for SuperEGO should be 27 (marking the *minimum* distance derived from an absolute difference of at most 1 per dimension) since SuperEGO applies an *aggregate* distance over d dimensions in its operation.

Approximate SuperEGO. The Approximate SuperEGO method (Ap-SuperEGO) executes SuperEGO with only difference the replacement of NestedLoopJoin with the one used in Ap-Baseline.

Exact SuperEGO. Exact SuperEGO (Ex-SuperEGO) replaces the NestedLoopJoin of SuperEGO with that of Ex-Baseline to find all matches among current B and A. When the recursion of SuperEGO ends, it forms $matched\_B$, $matched\_A$, $sortedM\_B$, and $sortedM\_A$ (same as in Ex-MinMax) and calls *once* the CSF.

6 EXPERIMENTAL EVALUATION

Our code is in C++ and ran experiments on an Intel CPU i7-11700 machine at 2.5 GHz with 24GB RAM and Windows 10 64-bit.

6.1 Setup

Datasets. We used one *real* and one *synthetic* dataset for the CSJ problem, mentioned as VK and Synthetic in the paper. In particular, we sampled 7.8M users from the *VK*[7] social network and for their *preferences* we selected the 20 most popular pages from the 27 categories of VK; so, 540 pages in total. Each *user vector* is formed based on the **real likes** of that user to all VK posts published in any of the 540 pages over the 2010-2019 years of VK. Actually, each *user vector* has 27 dimensions ($d = 27$) mapping to the 27 categories of VK, and the value in each dimension depicts the *aggregate* number of likes of user over all the posts

[7]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 purposes.

associated with the respective 20 pages of category. Regarding the Synthetic dataset, we used the same number of users with VK (7.8M) but each *user vector* (also $d = 27$) is filled with values derived from a **uniform generator** we created. Table 1 shows the *distribution* (notably different among datasets) over the number of *total likes* per dimension (category).

The *maximum* number of likes per dimension over all users in VK is 152532 while in Synthetic is 500000. We created Synthetic to be *semi-realistic.* On one hand, platforms have many features and so the aggregation per category can be high enough, that is why we selected the value of 500000. On the other hand, the user reactions are often not so uniform since users tend to like some things much more than others (as our real VK dataset depicts). Yet, Synthetic plays the role of a not so common but simultaneously possible case of real-world. We contend that by following such an approach for forming VK and Synthetic datasets, we provide broad and general results for our CSJ methods.

Methods. We evaluate the CSJ performance of 3 *approximate* and 3 *exact* methods. The former are Ap-Baseline, Ap-MinMax, and Ap-SuperEGO, while the latter are Ex-Baseline, Ex-MinMax, and Ex-SuperEGO. For SuperEGO methods all data are *normalized* to fit in $[0, 1]^d$ domain since else the algorithm does not work.

Parameters. In all experiments, $d = 27$ and $\epsilon = 1$ for VK while $\epsilon = 15000$ for Synthetic. For SuperEGO methods, ϵ is 27(1/152532) and 27(15000/500000) for VK and Synthetic, respectively. We remind that we adapted the ϵ of SuperEGO for CSJ to denote a *right* aggregate absolute distance over all d dimensions. Also, as SuperEGO can run in parallel, we used 1 thread for serial execution and fair comparison. For the *encoding scheme* used in MinMax methods we selected 4 *parts* for reasons explained in Section 4.

Case Studies. We evaluate our CSJ solutions to **two types** of *case studies.* The former mentions to *different categories* where each $< B, A >$ community pair (of totally 10 pairs) has a *similarity* $\geq$ 15%, whereas the latter mentions to *same categories* where each $< B, A >$ community pair (of totally 10 pairs) has a *similarity* $\geq$ 30%. Each community pair is denoted by a unique *cID* (*couple ID*) in all results. Detailed information about the community pairs we compare is shown in Table 2. The same 20 couples are compared in both VK and Synthetic. Yet, we stress that the selection of those 20 couples done in an *exploration* way under the **realistic** settings of VK. Namely, we tested various couples belonging to *different* or *same* categories till finding 10 couples in total that **really** have a *similarity* at least 15% or 30% in each case.

Tables. Tables 1 and 2 already explained. Tables 3–10 show the *evaluation results* of all CSJ methods. In each such table, size_B is $|B|$, size_A is $|A|$, all *percentage* values mention to respective computed *similarity* (Eq. (1)), and each number in *parenthesis* is the respective *execution time* in *seconds.* Finally, Table 11 shows the *scalability results* relative only to Ex-MinMax method that *generally* is the most accurate and practical method to solve the CSJ problem; *execution time* is also in *seconds.*

6.2 Results

Experiments on VK Dataset. Table 3 shows the CSJ results of *approximate* methods on VK dataset for *different* categories. Compared to Ap-Baseline, in most cases Ap-MinMax achieves a slightly higher *similarity* (accuracy) and a much higher efficiency (less execution time). Compared to Ap-SuperEGO, Ap-MinMax achieves significantly higher *similarity* due to accuracy loss derived from *normalized* data conversion in Ap-SuperEGO, yet Ap-SuperEGO is notably more efficient than Ap-MinMax due to its divide-and-conquer recursive approach along with its efficient EGO-Strategy. Yet, we stress that for the CSJ purposes, accuracy is of higher priority than efficiency, so Ap-MinMax has an edge over the competitor methods.

Table 4 presents the CSJ results of *exact* methods on VK dataset for *different* categories. Ex-Baseline and Ex-MinMax yield the same *similarity*, while Ex-SuperEGO is crucially less accurate for *normalized* data conversion reasons but it remains faster than others. Yet, Ex-MinMax is now emphatically faster than Ex-Baseline and that makes it a clear winner over compared methods as it has the best accuracy-efficiency performance over others.

Tables 5 and 6 present the CSJ results of *approximate* and *exact* methods on VK dataset for *same* categories. The general trend is similar to the one discussed before for Tables 3 and 4 respectively, although the *execution time* of all methods is now generally higher due to the double *similarity* effect (15% to 30%).

Experiments on Synthetic Dataset. Table 7[8] presents the CSJ results of *approximate* methods on Synthetic dataset for *different* categories. Now, Ap-Baseline has similar accuracy and running times as Ap-MinMax. Further, Ap-SuperEGO is now eminently more competitive on accuracy to others but it no longer has a commanding lead on running time. The improvement in accuracy for Ap-SuperEGO relates with the much higher ϵ that is used for Synthetic as also with Synthetic *uniform* nature; these factors restrict accuracy loss during *normalized* data conversion. Yet, the higher value of ϵ limits the efficient pruning of EGO-Strategy leading to more recursive calls that cost.

Table 8 shows the CSJ results of *exact* methods on Synthetic dataset for *different* categories. Here, Ex-SuperEGO does not have any accuracy loss and so all methods achieve the same *similarity.* Still, Ex-MinMax is obviously faster than Ex-Baseline and competitive on running time to Ex-SuperEGO. The *exact* operation of CSF may address the risky conversion needed to SuperEGO.

Tables 9 and 10 show the CSJ results of *approximate* and *exact* methods on Synthetic for *same* categories. Besides the general increase in execution time due to the higher *similarity*, results are similar to the ones in Tables 7 and 8, correspondingly. Some worthy differences are that Ap-SuperEGO is notably less competitive on accuracy in Table 9 than in Table 7 and Ex-MinMax is less competitive on running time in Table 10 than in Table 8.

Experimental Conclusion. Overall, the MinMax methods are the best choice to solve the CSJ problem with a few exceptions taking place in Tables 8 and 10 where Ex-SuperEGO has an edge. Yet, we stress that these exceptions occur in the Synthetic dataset that as earlier mentioned it is not so common (although possible) in real-world due to its *uniform* formation. In more realistic scenarios, such as the ones captured by VK dataset, the *non-uniform* preference distribution of users will incur inaccurate *normalized* data conversions that will crucially limit (as experimentally shown) the accuracy of SuperEGO methods. Since accuracy is the most important metric for CSJ problem, the MinMax methods represent the most practical way to solve it, without having too significant efficiency penalty compared to SuperEGO methods.

Moreover, we stress that even if there was a way SuperEGO to work for numeric (*non-normalized*) data, a combined algorithm MinMax-SuperEGO would be faster than SuperEGO itself. Specifically, as we observed, both SuperEGO methods essentially replace the NestedLoopJoin part of original SuperEGO framework with that used in Baseline to solve the CSJ problem. Yet, that replaced NestedLoopJoin part is notably slower than the

[8]The *similarity* results for $cID = 10$ constitute an *edge case* in Tables 7 and 8.

Table 1: The *ranking* per category based on *total_likes* in descending order for *VK* and *Synthetic* datasets.

rank	**Dataset**	Category	total_likes	**Dataset**	Category	total_likes
1	VK	Entertainment	2,111,519,450	Synthetic	Hobbies	4,030,521,210
2	VK	Hobbies	602,445,614	Synthetic	Social_public	3,899,674,411
3	VK	Relationship_family	384,993,747	Synthetic	Job_search	3,894,770,484
4	VK	Beauty_health	318,695,199	Synthetic	Medicine	3,879,329,978
5	VK	Media	296,466,970	Synthetic	Home_renovation	3,840,633,803
6	VK	Social_public	255,007,945	Synthetic	Celebrity	3,784,173,891
7	VK	Sport	245,830,867	Synthetic	Education	3,783,409,580
8	VK	Internet	206,085,821	Synthetic	Entertainment	3,763,167,129
9	VK	Education	197,289,902	Synthetic	Sport	3,718,424,135
10	VK	Celebrity	167,468,242	Synthetic	Tourism_leisure	3,702,498,557
11	VK	Animals	159,569,729	Synthetic	Transportation_Services	3,685,969,155
12	VK	Music	153,686,427	Synthetic	Finance_insurance	3,680,184,922
13	VK	Culture_art	141,107,189	Synthetic	Culture_art	3,680,041,975
14	VK	Food_recipes	140,212,548	Synthetic	Consumer_Services	3,668,738,029
15	VK	Tourism_leisure	140,054,637	Synthetic	Professional_Services	3,623,780,227
16	VK	Auto_motor	136,991,765	Synthetic	Products_stores	3,565,053,769
17	VK	Products_stores	131,752,523	Synthetic	Relationship_family	3,560,196,074
18	VK	Home_renovation	120,091,854	Synthetic	Cities_countries	3,552,381,297
19	VK	Cities_countries	74,006,530	Synthetic	Food_recipes	3,550,668,794
20	VK	Professional_Services	33,024,545	Synthetic	Internet	3,521,866,267
21	VK	Medicine	32,135,820	Synthetic	Animals	3,517,540,727
22	VK	Finance_insurance	30,961,892	Synthetic	Media	3,514,872,848
23	VK	Restaurants	6,473,240	Synthetic	Auto_motor	3,469,592,249
24	VK	Job_search	1,853,720	Synthetic	Communication_Services	3,446,086,841
25	VK	Transportation_Services	1,385,538	Synthetic	Restaurants	3,415,910,481
26	VK	Consumer_Services	810,889	Synthetic	Music	3,297,277,125
27	VK	Communication_Services	474,492	Synthetic	Beauty_health	3,292,929,613

Table 2: The *names* and *VK-ids* of compared community pairs. Each id (id_B, id_A) maps to a specific VK social network page that is accessible just by typing in a browser the id of page (after *public*) as follows: *https://vk.com/publicID*. In case of visit, we recommend using the *Google Chrome* browser for translation since most pages are written in Russian language.

cID	name_B	id_B	name_A	id_A
1	Quick Recipes	165062392	Salads | Best Recipes	94216909
2	Happiness	23337480	Sportshacker	128350290
3	Moment of history	143826157	This is a fact | Science and Facts	45688121
4	Health secrets. What is said by doctors?	55122354	Fashionable girl	36085261
5	First channel	25380626	Nice line ...	26669118
6	About women's...	33382046	Successful girl	24036559
7	The best of Saint Petersburg	31516466	Vandrouki | Travel almost free	63731512
8	Housing problem	42541008	Business quote book	28556858
9	Jah Khalib	26211015	My audios	105999460
10	Job in Moscow	31154183	VK Pay	166850908
11	Cooking: delicious recipes	42092461	Cooking at home: delicious and easy	40020627
12	Simple recipes	83935640	Best Chef's Recipes	18464856
13	FC Barcelona	22746750	Football Europe	23693281
14	World Russian Premier League	51812607	Football Europe	23693281
15	World of beauty	34981365	Fashionable girl	36085261
16	Beauty | Fashion | Show Business	32922940	Fashionable girl	36085261
17	More than just lines	32651025	Just love	28293246
18	Modern mom	55074079	MAMA	20249656
19	Business quote book	28556858	Business Strategy | Success in life	30559917
20	Smart Money | Business Magazine	34483558	Business Strategy | Success in life	30559917

encoded nested loop join used in `MinMax` methods since it is shown that `Ap-MinMax` and `Ex-MinMax` are emphatically faster than `Ap-Baseline` and `Ex-Baseline`. So, we claim that `MinMax` contributes to better CSJ results even in that *theoretic* case of *non-normalized* data for `SuperEGO`.

Scalability Study on VK Dataset. To highlight the practical applicability of `Ex-MinMax`, we present *scalability* results on `VK` dataset in Table 11. Each row maps to a different category and has four points of reference related to the *average size* of four different and realistic couples within category. The difference in sizes depends on the difference in popularity (number of subscribers) of the communities of each category. This enables us to have various sizes to compare. Generally, as size increases, the associated execution time increases in a logical pace. The highest running time is noted for the *size*_4 of *Entertainment* category, where for a couple with *average size* equals to 1110846 subscribers, `Ex-MinMax` needs almost 18 hours to solve the CSJ problem. However, this is a *rare case* since most communities in VK have a number of subscribers ranging around 150000 - 200000. That is why we solved CSJ for such kind of communities.

Table 3: ***Approximate*** **methods on** VK **dataset for** $\epsilon = 1$ **and** ***different*** **categories where** $similarity \geq 15\%$**.**

cID	Categories (B | A) / Methods	Ap-Baseline	Ap-MinMax	Ap-SuperEGO	size_B | size_A
1	Restaurants | Food_recipes	20.56% (442 s)	20.58% (116 s)	19.68% (18 s)	109,176 | 116,016
2	Hobbies | Sport	15.40% (1826 s)	15.42% (590 s)	15.16% (19 s)	156,213 | 230,017
3	Culture_art | Education	24.82% (761 s)	24.82% (177 s)	24.26% (19 s)	134,961 | 138,199
4	Medicine | Beauty_health	16.30% (1011 s)	16.26% (232 s)	16.06% (15 s)	120,783 | 185,393
5	Media | Entertainment	17.32% (3640 s)	17.34% (1501 s)	16.70% (60 s)	197,415 | 330,944
6	Social_public | Relationship_family	24.31% (600 s)	24.31% (154 s)	24.10% (8 s)	118,993 | 131,297
7	Cities_countries | Tourism_leisure	22.18% (1733 s)	22.19% (838 s)	21.83% (35 s)	140,114 | 257,419
8	Home_renovation | Products_stores	15.45% (1457 s)	15.46% (359 s)	15.15% (33 s)	167,585 | 182,815
9	Celebrity | Music	17.36% (1183 s)	17.36% (272 s)	16.86% (16 s)	125,248 | 189,937
10	Job_search | Finance_insurance	20.95% (219 s)	20.72% (51 s)	19.40% (12 s)	55,918 | 109,622

Table 4: ***Exact*** **methods on** VK **dataset for** $\epsilon = 1$ **and** ***different*** **categories where** $similarity \geq 15\%$**.**

cID	Categories (B | A) / Methods	Ex-Baseline	Ex-MinMax	Ex-SuperEGO	size_B | size_A
1	Restaurants | Food_recipes	20.81% (1198 s)	20.81% (133 s)	20.15% (27 s)	109,176 | 116,016
2	Hobbies | Sport	15.46% (4254 s)	15.46% (597 s)	15.22% (30 s)	156,213 | 230,017
3	Culture_art | Education	24.95% (1985 s)	24.95% (226 s)	24.58% (51 s)	134,961 | 138,199
4	Medicine | Beauty_health	16.42% (2466 s)	16.42% (239 s)	16.20% (21 s)	120,783 | 185,393
5	Media | Entertainment	17.52% (8220 s)	17.52% (1552 s)	16.92% (75 s)	197,415 | 330,944
6	Social_public | Relationship_family	24.38% (1603 s)	24.38% (186 s)	24.20% (37 s)	118,993 | 131,297
7	Cities_countries | Tourism_leisure	22.22% (4192 s)	22.22% (863 s)	21.91% (57 s)	140,114 | 257,419
8	Home_renovation | Products_stores	15.53% (3539 s)	15.53% (392 s)	15.29% (41 s)	167,585 | 182,815
9	Celebrity | Music	17.52% (2790 s)	17.52% (288 s)	17.06% (32 s)	125,248 | 189,937
10	Job_search | Finance_insurance	21.57% (679 s)	21.56% (147 s)	20.09% (114 s)	55,918 | 109,622

Table 5: ***Approximate*** **methods on** VK **dataset for** $\epsilon = 1$ **and** ***same*** **categories where** $similarity \geq 30\%$**.**

cID	Categories (B | A) / Methods	Ap-Baseline	Ap-MinMax	Ap-SuperEGO	size_B | size_A
11	Food_recipes | Food_recipes	31.42% (1610 s)	31.44% (472 s)	30.94% (29 s)	180,158 | 196,135
12	Food_recipes | Food_recipes	32.01% (2329 s)	32.05% (1049 s)	31.30% (45 s)	180,351 | 272,320
13	Sport | Sport	39.24% (2070 s)	39.33% (763 s)	37.53% (45 s)	179,412 | 234,508
14	Sport | Sport	36.66% (2234 s)	36.48% (745 s)	34.85% (54 s)	184,663 | 234,508
15	Beauty_health | Beauty_health	36.83% (1330 s)	36.85% (393 s)	36.47% (14 s)	163,176 | 185,393
16	Beauty_health | Beauty_health	30.46% (1534 s)	30.45% (404 s)	30.11% (15 s)	178,138 | 185,393
17	Relationship_family | Relationship_family	35.25% (1427 s)	35.26% (369 s)	34.97% (14 s)	165,509 | 190,027
18	Relationship_family | Relationship_family	32.21% (1125 s)	32.23% (326 s)	31.76% (20 s)	147,140 | 175,929
19	Products_stores | Products_stores	31.79% (1700 s)	31.82% (479 s)	31.36% (37 s)	182,815 | 201,038
20	Products_stores | Products_stores	33.40% (1475 s)	33.42% (466 s)	33.07% (30 s)	161,991 | 201,038

Table 6: ***Exact*** **methods on** VK **dataset for** $\epsilon = 1$ **and** ***same*** **categories where** $similarity \geq 30\%$**.**

cID	Categories (B | A) / Methods	Ex-Baseline	Ex-MinMax	Ex-SuperEGO	size_B | size_A
11	Food_recipes | Food_recipes	31.52% (4168 s)	31.52% (600 s)	31.20% (143 s)	180,158 | 196,135
12	Food_recipes | Food_recipes	32.10% (5945 s)	32.10% (1194 s)	31.63% (150 s)	180,351 | 272,320
13	Sport | Sport	39.54% (5314 s)	39.54% (997 s)	38.62% (227 s)	179,412 | 234,508
14	Sport | Sport	37.10% (5527 s)	37.10% (1037 s)	35.81% (419 s)	184,663 | 234,508
15	Beauty_health | Beauty_health	36.93% (3765 s)	36.93% (508 s)	36.67% (159 s)	163,176 | 185,393
16	Beauty_health | Beauty_health	30.57% (3952 s)	30.58% (515 s)	30.28% (133 s)	178,138 | 185,393
17	Relationship_family | Relationship_family	35.35% (3835 s)	35.35% (520 s)	35.11% (154 s)	165,509 | 190,027
18	Relationship_family | Relationship_family	32.26% (3063 s)	32.26% (413 s)	31.93% (103 s)	147,140 | 175,929
19	Products_stores | Products_stores	31.88% (4389 s)	31.88% (600 s)	31.59% (159 s)	182,815 | 201,038
20	Products_stores | Products_stores	33.50% (3932 s)	33.50% (545 s)	33.23% (135 s)	161,991 | 201,038

Table 7: ***Approximate*** **methods on** $Synthetic$ **dataset for** $\epsilon = 15000$ **and** ***different*** **categories where** $similarity \geq 15\%$**.**

cID	Categories (B | A) / Methods	Ap-Baseline	Ap-MinMax	Ap-SuperEGO	size_B | size_A
1	Restaurants | Food_recipes	17.57% (389 s)	17.56% (307 s)	17.53% (285 s)	109,176 | 116,016
2	Hobbies | Sport	15.87% (1494 s)	15.86% (1610 s)	15.79% (766 s)	156,213 | 230,017
3	Culture_art | Education	24.00% (603 s)	23.96% (516 s)	23.88% (390 s)	134,961 | 138,199
4	Medicine | Beauty_health	16.46% (872 s)	16.46% (816 s)	16.40% (459 s)	120,783 | 185,393
5	Media | Entertainment	15.37% (3035 s)	15.36% (3240 s)	15.29% (1384 s)	197,415 | 330,944
6	Social_public | Relationship_family	24.42% (499 s)	24.39% (417 s)	24.30% (330 s)	118,993 | 131,297
7	Cities_countries | Tourism_leisure	22.04% (1501 s)	22.02% (1602 s)	21.97% (734 s)	140,114 | 257,419
8	Home_renovation | Products_stores	15.38% (1203 s)	15.36% (1090 s)	15.31% (632 s)	167,585 | 182,815
9	Celebrity | Music	15.79% (931 s)	15.77% (883 s)	15.73% (500 s)	125,248 | 189,937
10	Job_search | Finance_insurance	7.76% (171 s)	7.76% (134 s)	7.73% (130 s)	55,918 | 109,622

Table 8: *Exact* methods on *Synthetic* dataset for $\epsilon = 15000$ and *different* categories where $similarity \geq 15\%$.

cID	Categories (B | A) / Methods	Ex-Baseline	Ex-MinMax	Ex-SuperEGO	size_B | size_A
1	Restaurants | Food_recipes	17.74% (1151 s)	17.74% (252 s)	17.74% (206 s)	109,176 | 116,016
2	Hobbies | Sport	16.00% (3880 s)	16.00% (1382 s)	16.00% (549 s)	156,213 | 230,017
3	Culture_art | Education	24.15% (1806 s)	24.15% (460 s)	24.15% (314 s)	134,961 | 138,199
4	Medicine | Beauty_health	16.57% (2396 s)	16.57% (713 s)	16.57% (337 s)	120,783 | 185,393
5	Media | Entertainment	15.49% (7308 s)	15.49% (3093 s)	15.49% (974 s)	197,415 | 330,944
6	Social_public | Relationship_family	24.56% (1556 s)	24.56% (364 s)	24.56% (264 s)	118,993 | 131,297
7	Cities_countries | Tourism_leisure	22.13% (3950 s)	22.13% (1516 s)	22.13% (554 s)	140,114 | 257,419
8	Home_renovation | Products_stores	15.57% (3279 s)	15.57% (982 s)	15.57% (457 s)	167,585 | 182,815
9	Celebrity | Music	15.90% (2550 s)	15.90% (783 s)	15.90% (359 s)	125,248 | 189,937
10	Job_search | Finance_insurance	7.85% (544 s)	7.85% (113 s)	7.85% (91 s)	55,918 | 109,622

Table 9: *Approximate* methods on *Synthetic* dataset for $\epsilon = 15000$ and *same* categories where $similarity \geq 30\%$.

cID	Categories (B | A) / Methods	Ap-Baseline	Ap-MinMax	Ap-SuperEGO	size_B | size_A
11	Food_recipes | Food_recipes	30.46% (1339 s)	30.42% (1311 s)	30.30% (717 s)	180,158 | 196,135
12	Food_recipes | Food_recipes	30.44% (2017 s)	30.43% (2211 s)	30.34% (952 s)	180,351 | 272,320
13	Sport | Sport	33.58% (1642 s)	33.56% (1763 s)	33.43% (829 s)	179,412 | 234,508
14	Sport | Sport	30.70% (1722 s)	30.68% (1812 s)	30.56% (860 s)	184,663 | 234,508
15	Beauty_health | Beauty_health	36.48% (1094 s)	36.46% (1066 s)	36.30% (586 s)	163,176 | 185,393
16	Beauty_health | Beauty_health	30.21% (1244 s)	30.19% (1180 s)	30.09% (650 s)	178,138 | 185,393
17	Relationship_family | Relationship_family	35.16% (1157 s)	35.14% (1133 s)	34.97% (610 s)	165,509 | 190,027
18	Relationship_family | Relationship_family	31.58% (940 s)	31.55% (869 s)	31.42% (509 s)	147,140 | 175,929
19	Products_stores | Products_stores	31.31% (1404 s)	31.28% (1385 s)	31.14% (737 s)	182,815 | 201,038
20	Products_stores | Products_stores	33.11% (1226 s)	33.10% (1225 s)	32.97% (638 s)	161,991 | 201,038

Table 10: *Exact* methods on *Synthetic* dataset for $\epsilon = 15000$ and *same* categories where $similarity \geq 30\%$.

cID	Categories (B | A) / Methods	Ex-Baseline	Ex-MinMax	Ex-SuperEGO	size_B | size_A
11	Food_recipes | Food_recipes	30.63% (3914 s)	30.63% (1301 s)	30.63% (636 s)	180,158 | 196,135
12	Food_recipes | Food_recipes	30.57% (5471 s)	30.57% (2207 s)	30.57% (827 s)	180,351 | 272,320
13	Sport | Sport	33.73% (4701 s)	33.73% (1780 s)	33.73% (757 s)	179,412 | 234,508
14	Sport | Sport	30.85% (4827 s)	30.85% (1806 s)	30.85% (756 s)	184,663 | 234,508
15	Beauty_health | Beauty_health	36.64% (3372 s)	36.64% (1107 s)	36.64% (577 s)	163,176 | 185,393
16	Beauty_health | Beauty_health	30.41% (3636 s)	30.41% (1167 s)	30.41% (583 s)	178,138 | 185,393
17	Relationship_family | Relationship_family	35.31% (3562 s)	35.31% (1157 s)	35.31% (591 s)	165,509 | 190,027
18	Relationship_family | Relationship_family	31.72% (2823 s)	31.72% (861 s)	31.72% (453 s)	147,140 | 175,929
19	Products_stores | Products_stores	31.48% (4052 s)	31.48% (1384 s)	31.48% (667 s)	182,815 | 201,038
20	Products_stores | Products_stores	33.27% (3594 s)	33.27% (1226 s)	33.27% (589 s)	161,991 | 201,038

Table 11: *Scalability* results for `Exact MinMax` on *VK*. Size is the *average size* of a *different* couple for each category.

Category	size_1	Ex-MinMax	size_2	Ex-MinMax	size_3	Ex-MinMax	size_4	Ex-MinMax
Food_recipes	124,453	165 s	200,966	670 s	332,977	3,676 s	417,492	7,020 s
Restaurants	27,733	5 s	50,802	26 s	71,114	34 s	111,713	93 s
Hobbies	212,071	807 s	326,951	3,387 s	432,853	7,900 s	538,492	12,979 s
Sport	107,770	140 s	156,762	278 s	199,233	590 s	248,901	1,381 s
Education	128,905	173 s	200,466	517 s	317,041	2,663 s	414,692	6,891 s
Culture_art	54,381	25 s	106,885	125 s	157,236	360 s	228,763	997 s
Beauty_health	149,171	204 s	211,701	710 s	256,387	1,660 s	318,470	3,218 s
Medicine	21,290	4 s	41,438	16 s	62,333	38 s	84,311	66 s
Entertainment	445,364	8,371 s	651,230	22,328 s	841,407	35,648 s	1,110,846	63,873 s
Media	117,231	130 s	220,804	1,057 s	335,845	2,920 s	406,973	7,444 s
Relationship_family	121,910	167 s	169,862	324 s	212,582	840 s	283,532	2,304 s
Social_public	80,552	65 s	135,060	194 s	182,865	426 s	269,604	1,797 s
Tourism_leisure	104,403	105 s	147,984	245 s	204,376	605 s	248,205	1,510 s
Cities_countries	53,271	30 s	94,130	86 s	133,765	214 s	163,201	292 s
Products_stores	112,425	127 s	157,593	335 s	219,171	735 s	265,760	2,181 s
Home_renovation	101,381	107 s	149,484	275 s	188,986	527 s	274,326	1,889 s
Celebrity	105,339	112 s	160,277	340 s	206,374	907 s	255,239	1,096 s
Music	110,695	119 s	158,516	264 s	201,757	714 s	251,919	1,118 s
Finance_insurance	24,620	5 s	49,505	10 s	70,196	48 s	108,028	162 s
Job_search	16,728	1 s	30,787	6 s	45,597	14 s	62,418	28 s

7 CONCLUSION

In this paper we proposed and studied the *community similarity based on user profile joins* (CSJ) problem. We deployed 3 *approximate* and 3 *exact* methods to solve it. The best CSJ method utilizes a novel *encoding scheme* we developed that can find a general applicability besides CSJ. We thoroughly evaluated our solutions on real and synthetic datasets having distinct characteristics. To do that, we formed a number of *case studies* pertaining to different and same categories. Experimental results verify the applicability of CSJ to real-world scenarios and the challenges that it sets.

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] Ahmed Al-Baghdadi and Xiang Lian. 2020. Topic-Based Community Search over Spatial-Social Networks. *PVLDB* 13, 12 (2020), 2104–2117.

[2] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-Similarity Joins. In *VLDB*. 918–929.

[3] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up All Pairs Similarity Search. In *WWW*. 131–140.

[4] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. 2001. Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. *SIGMOD Rec.* 30, 2 (2001), 379–388.

[5] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. 2012. Spatio-Textual Similarity Joins. *PVLDB* 6, 1 (2012), 1–12.

[6] S. Chaudhuri, V. Ganti, and R. Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE*. 5–5.

[7] Muhammad Aamir Cheema, Xuemin Lin, Haixun Wang, Jianmin Wang, and Wenjie Zhang. 2011. A unified approach for computing top-k pairs in multidimensional space. In *ICDE*. 1031–1042.

[8] Yu Chen, Jun Xu, and Minzheng Xu. 2015. Finding community structure in spatially constrained complex networks. *IJGIS* 29, 6 (2015), 889–911.

[9] Zhaoqi Chen, Dmitri V. Kalashnikov, and Sharad Mehrotra. 2009. Exploiting Context Analysis for Combining Multiple Entity Resolution Systems. In *SIGMOD*. 207–218.

[10] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. 2000. Closest Pair Queries in Spatial Databases. *SIGMOD Rec.* 29, 2 (2000), 189–200.

[11] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online Search of Overlapping Communities. In *SIGMOD*. 277–288.

[12] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local Search of Communities in Large Graphs. In *SIGMOD*. 991–1002.

[13] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2021. New Trends in High-D Vector Similarity Search: Al-Driven, Progressive, and Distributed. *PVLDB* 14, 12 (2021), 3198–3201.

[14] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. 2007. Duplicate Record Detection: A Survey. *TKDE* 19, 1 (2007), 1–16.

[15] Paul Expert, Tim S Evans, Vincent D Blondel, and Renaud Lambiotte. 2011. Uncovering space-independent communities in spatial networks. *Proceedings of the National Academy of Sciences* 108, 19 (2011), 7663–7668.

[16] Yixiang Fang, Reynold Cheng, Yankai Chen, Siqiang Luo, and Jiafeng Hu. 2017. Effective and efficient attributed community search. *VLDBJ* 26, 6 (2017), 803–828.

[17] Yixiang Fang, Reynold Cheng, Xiaodong Li, Siqiang Luo, and Jiafeng Hu. 2017. Effective Community Search over Large Spatial Graphs. *PVLDB* 10, 6 (2017), 709–720.

[18] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective Community Search for Large Attributed Graphs. *PVLDB* 9, 12 (2016), 1233–1244.

[19] Yixiang Fang, Reynold Cheng, Siqiang Luo, Jiafeng Hu, and Kai Huang. 2017. C-Explorer: Browsing Communities in Large Graphs. *PVLDB* 10, 12 (2017), 1885–1888.

[20] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *VLDBJ* 29, 1 (2020), 353–392.

[21] Santo Fortunato. 2010. Community detection in graphs. *Physics Reports* 486, 3 (2010), 75–174.

[22] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate String Joins in a Database (Almost) for Free. In *VLDB*. 491–500.

[23] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. 1998. CURE: An Efficient Clustering Algorithm for Large Databases. *SIGMOD Rec.* 27, 2 (1998), 73–84.

[24] D. Guo. 2008. Regionalization with dynamically constrained agglomerative clustering and partitioning (REDCAP). *IJGIS* 22, 7 (2008), 801–823.

[25] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying K-Truss Community in Large and Dynamic Graphs. In *SIGMOD*. 1311–1322.

[26] Xin Huang, Laks V. S. Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate Closest Community Search in Networks. *PVLDB* 9, 4 (2015), 276–287.

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

[28] Glen Jeh and Jennifer Widom. 2002. SimRank: A Measure of Structural-Context Similarity. In *KDD*. 538–543.

[29] Glen Jeh and Jennifer Widom. 2003. Scaling Personalized Web Search. In *WWW*. 271–279.

[30] Dmitri V. Kalashnikov. 2013. Super-EGO: fast multi-dimensional similarity join. *VLDBJ* 22, 4 (2013), 561–585.

[31] Dmitri V. Kalashnikov and Sharad Mehrotra. 2006. Domain-Independent Data Cleaning via Analysis of Entity-Relationship Graph. *TODS* 31, 2 (2006), 716–767.

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

[33] Edwin M. Knorr and Raymond T. Ng. 1998. Algorithms for Mining Distance-Based Outliers in Large Datasets. In *VLDB*. 392–403.

[34] Krzysztof Koperski and Jiawei Han. 1995. Discovery of spatial association rules in geographic information databases. In *Advances in Spatial Databases*. 47–66.

[35] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2015. Influential Community Search in Large Networks. *PVLDB* 8, 5 (2015), 509–520.

[36] Yifan Li, Xiaohui Yu, and Nick Koudas. 2021. LES3: Learning-Based Exact Set Similarity Search. *PVLDB* 14, 11 (2021), 2073–2086.

[37] Michael D. Lieberman, Jagan Sankaranarayanan, and Hanan Samet. 2008. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *ICDE*. 1111–1120.

[38] Yan Liu, Alexandru Niculescu-Mizil, and Wojciech Gryc. 2009. Topic-Link LDA: Joint Models of Topic and Author Community. In *ICML*. 665–672.

[39] Ming-Ling Lo and Chinya V. Ravishankar. 1996. Spatial Hash-Joins. *SIGMOD Rec.* 25, 2 (1996), 247–258.

[40] Ramesh M. Nallapati, Amr Ahmed, Eric P. Xing, and William W. Cohen. 2008. Joint Latent Topic Models for Text and Citations. In *KDD*. 542–550.

[41] Mark EJ Newman and Michelle Girvan. 2004. Finding and evaluating community structure in networks. *Phys. Rev. E* 69, 2 (2004), 026113.

[42] Rabia Nuray-Turan, Dmitri V. Kalashnikov, Sharad Mehrotra, and Yaming Yu. 2012. Attribute and Object Selection Queries on Objects with Probabilistic Attributes. *TODS* 37, 1 (2012), 1–41.

[43] Jignesh M. Patel and David J. DeWitt. 1996. Partition Based Spatial-Merge Join. *SIGMOD Rec.* 25, 2 (1996), 259–270.

[44] Yiye Ruan, David Fuhry, and Srinivasan Parthasarathy. 2013. Efficient Community Detection in Large Networks Using Content and Links. In *WWW*. 1089–1098.

[45] Sunita Sarawagi and Alok Kirpal. 2004. Efficient Set Joins on Similarity Predicates. In *SIGMOD*. 743–754.

[46] Mauro Sozio and Aristides Gionis. 2010. The Community-Search Problem and How to Plan a Successful Cocktail Party. In *KDD*. 939–948.

[47] Liwen Sun, Reynold Cheng, Xiang Li, David W. Cheung, and Jiawei Han. 2011. On Link-Based Similarity Join. *PVLDB* 4, 11 (2011), 714–725.

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

[49] George Tsatsanifos and Akrivi Vlachou. 2015. On Processing Top-k Spatio-Textual Preference Queries. In *EDBT*. 433–444.

[50] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. 2009. Top-k Set Similarity Joins. In *ICDE*. 916–927.

[51] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient Similarity Joins for Near-Duplicate Detection. *TODS* 36, 3 (2011), 1–41.

[52] Zhiqiang Xu, Yiping Ke, Yi Wang, Hong Cheng, and James Cheng. 2012. A Model-Based Approach to Attributed Graph Clustering. In *SIGMOD*. 505–516.

[53] Jaewon Yang, Julian McAuley, and Jure Leskovec. 2013. Community Detection in Networks with Node Attributes. In *ICDM*. 1151–1156.

[54] Alexandros Zeakis, Dimitrios Skoutas, Dimitris Sacharidis, Odysseas Papapetrou, and Manolis Koubarakis. 2022. TokenJoin: Efficient Filtering for Set Similarity Join with Maximum Weighted Bipartite Matching. *PVLDB* 16, 4 (2022), 790–802.

[55] Fan Zhang, Xuemin Lin, Ying Zhang, Lu Qin, and Wenjie Zhang. 2019. Efficient community discovery with user engagement and similarity. *VLDBJ* 28, 6 (2019), 987–1012.

[56] Weiguo Zheng, Lei Zou, Yansong Feng, Lei Chen, and Dongyan Zhao. 2013. Efficient Simrank-Based Similarity Join over Large Graphs. *PVLDB* 6, 7 (2013), 493–504.

[57] Yang Zhou, Hong Cheng, and Jeffrey Xu Yu. 2009. Graph Clustering Based on Structural/Attribute Similarities. *PVLDB* 2, 1 (2009), 718–729.