ReHub. Extending Hub Labels for reverse k-nearest neighbor queries on large-scale networks

ALEXANDROS EFENTAKIS, Research Center “Athena”
DIETER PFOSER, Department of Geography and GeoInformation Science, George Mason University

Quite recently, the algorithmic community has focused on solving multiple shortest-path query problems beyond simple vertex-to-vertex queries, especially in the context of road networks. Unfortunately, those advanced query-processing techniques cannot be applied to large-scale graphs, e.g., social or collaboration networks, or to efficiently answer Reverse k-Nearest Neighbor (RkNN) queries, which are of practical relevance to a wide range of applications. To remedy this, we propose *ReHub*, a novel main-memory algorithm that extends the Hub Labeling technique to efficiently answer RkNN queries on large-scale networks. Our experimentation will show that *ReHub* is the best overall solution for this type of queries, requiring only minimal additional preprocessing and providing very fast query times in all cases.

CCS Concepts:•**Mathematics of computing** → **Graph algorithms;**•**Theory of computation** → **Shortest paths; Nearest neighbor algorithms;**

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Query processing, Graph algorithms, *ReHub* algorithm, RkNN, Reverse k-Nearest Neighbors, RNN, k-Nearest Neighbors, kNN, Hub Labels

ACM Reference Format:
Alexandros Efentakis, and Dieter Pfoser, 2016. ReHub. Extending Hub Labels for reverse k-nearest neighbor queries on large-scale networks *ACM J. Exp. Algor.* V, N, Article XXXX (August 2016), 36 pages.
DOI: http://dx.doi.org/10.1145/2990192

1. INTRODUCTION

During the last two decades, the algorithmic community has produced significant results regarding vertex-to-vertex shortest-path queries, especially in the context of transportation networks (cf. [Bast et al. 2015] for the latest overview). Recently, this focus shifted to additional types of shortest-path (SP) queries, such as *one-to-all* (finding SP distances from a source vertex s to all other graph vertices), *one-to-many* (computing the SP distances between the source vertex s and all vertices of a set of targets T), *range* (find all nodes reachable from s within a given timespan), *many-to-many* (calculate a distance table between two sets of vertices S and T) and k-Nearest Neighbor (kNN) queries. Recent contributions here include [Delling et al. 2011a; 2013a] (one-to-all), [Delling et al. 2011b] (one-to-many, many-to-many), [Efentakis and Pfoser 2014] (one-to-all, range, one-to-many) and [Delling and Werneck 2015; Efentakis et al. 2015b] (kNN queries). Unfortunately, most of these advanced query processing meth-

This work was partially supported by the “Research Programs for Excellence 2014-2016 / CitySense-ATHENA R.I.C.”.
Author’s addresses: A. Efentakis (Corresponding Author), Research Center “Athena’ and D. Pfoser, Department of Geography and GeoInformation Science, George Mason University
Email: efentakis@imis.athena-innovation.gr (A. Efentakis), dpfoser@gmu.edu (D. Pfoser)

ods are tailored for road networks and, thus, they cannot be easily applied to denser, small-diameter graphs, such as social or collaboration networks.

For large-scale networks, the prevailing technique for vertex-to-vertex queries is based on the 2-hop labeling, or, *Hub Labeling* (HL) algorithm [Gavoille et al. 2001; Cohen et al. 2002; Gavoille et al. 2004], in which the preprocessing stage calculates for every vertex v a forward label $forwLab(v)$ and a backward label $backLab(v)$. These labels are then used to very fast answer vertex-to-vertex SP queries. The HL technique has been adapted successfully for vertex-to-vertex queries on road networks [Abraham et al. 2011; 2012b; Delling et al. 2013b; Akiba et al. 2014], undirected, unweighted graphs [Akiba et al. 2013; Delling et al. 2014; Jiang et al. 2014] and public transportation networks [Wang et al. 2015; Delling et al. 2015; Efentakis 2016]. The HL method has also been used for one-to-many, many-to-many and kNN queries on road networks in [Delling et al. 2011b; Delling and Werneck 2015; Abraham et al. 2012a].

Another very important problem variation is the *Reverse k-Nearest Neighbor* (RkNN) query, initially proposed in [Korn and Muthukrishnan 2000]. Given a query point q and a set of targets P, the RkNN query retrieves all the targets in P that have q as one of their k-nearest neighbors according to a distance function $dist()$. RkNN queries may be used in various domains and applications, such as geomarketing, location-based services, resource allocation, profile-based marketing and decision support [Liu and Özsu 2009]. Despite their importance and the fact that there is some scientific literature discussing RkNN queries for road networks [Safar et al. 2009; Cheema et al. 2012; Borutta et al. 2014], to the best of our knowledge, the only RkNN work focusing on other types of graphs is [Yiu et al. 2006]. Unfortunately, all those previous works share some inherent limitations, such as assuming that the graph does not fit in main memory (and therefore is stored on secondary storage), require query times of a few seconds which prohibits their use in real-time applications and most importantly, they do not scale particularly well with respect to the network size, the target density, the distribution of targets and the cardinality of the reverse k-nearest neighbor result.

Putting everything together, the ambition of this work is to provide an efficient and fast main-memory algorithm for answering RkNN queries on large-scale graphs. Our proposed algorithm, termed *ReHub* (*Re*verse kNN + *Hub* labels) extends the Hub Labeling approach to efficiently handle these queries. The main advantage of *ReHub* is that its slower *Offline phase* depends only on the the targets P and has to run only once, whereas its *Online phase* (which depends on the query vertex q) is very fast. Still, even the costlier offline phase hardly needs more than $1s$ (after the creation of the labels), while the online phase requires typically less than $1ms$, making *ReHub* the only RkNN algorithm fast enough for real-time applications and big, large-scale graphs. Moreover, the necessary additional data structures created for *ReHub* may also answer kNN queries and require only a small fraction of the memory required for storing the created hub labels for the typical case of vertex-to-vertex queries. In addition, our experiments will show that by using *ReHub*, we can precompute the RkNN of all graph vertices for large-scale graphs with millions of vertices in just a few minutes, where previous solutions would require several days. Thus, *ReHub* is also the only viable alternative for addressing the related *All-RkNN* problem on large-scale graphs. In this work, we use undirected and unweighted graphs which constitute an important graph class (containing social and collaboration networks) but also pose a significant challenge to Hub Labeling algorithms because of the sheer size of the created labels. However, *ReHub* may be easily adapted for other graph classes where the HL algorithm typically performs well, including road networks.

The outline of this work is as follows. Section 2 presents related work. Section 3 describes the *ReHub* algorithm and provides a theoretical analysis of its performance.

Experiments showcasing *ReHub*'s benefits, in comparison to previous works, are provided in Section 4. Finally, Section 5 gives conclusions and directions for future work.

2. BACKGROUND AND RELATED WORK

Given a query point q and a set of targets P, the ***RkNN*** query (also referred as the monochromatic RkNN query) retrieves all the targets that have q as one of their k-nearest neighbors, according to a distance function $dist()$. Formally RkNN$(q) = \{p \in P : dist(p,q) \leq dist(p,p_k)\}$ where p_k is the k-Nearest Neighbor (kNN) (among the targets P) of p. In Euclidean space, the distance $dist(s,t)$ refers to the Euclidean distance between two points s and t. In graph networks, the query point is a random graph vertex $q \in V$ and $dist(s,t)$ corresponds to the minimum network distance between the two vertices. Throughout this work we use undirected, unweighted graphs $G(V,E)$ (where V represent vertices and E represents arcs), we assume that targets are located on vertices ($P_i \in V$) and we refer to *snapshot* RkNN queries, i.e, targets are not changing. Also, similar to previous works, the term *target density* D refers to the ratio $|P|/|V|$.

2.1. RkNN queries on graphs and applications

RkNN queries on network graphs have a wide range of applications. In road networks, typical applications for RkNN queries include resource allocation, profile-based and location-based marketing and decision support [Liu and Özsu 2009]. For example, RkNN queries may be used to determine the optimal location for opening a new franchise store in a area not covered by existing stores. A new take-away restaurant owner may choose the optimal location of his restaurant according to the locations of other competing restaurants, again by initiating a RkNN query. In this case, the set of targets although not physically moving may change for different kinds of restaurants, e.g., the owner of a Chinese restaurant would consider only locations of competing Chinese restaurants, whereas the targets of a RkNN query for a pizza owner would be restricted to other pizza places. These specific RkNN queries where the set of targets change between individual queries are referred to the bibliography as *ad-hoc* RkNN queries [Yiu et al. 2006].

Recently, the emergence of large-scale social networks has provided novel opportunities for uses of RkNN queries. In citation or collaboration networks, the set of targets may represent researchers that belong to a particular discipline (e.g., Computer Science), thematic area (e.g., algorithms) or a hosting university-institution (e.g., Stanford) and a RkNN query may determine which subset of those targets collaborate with a external (query) researcher more closely than colleagues belonging in the same university or thematic area. Similar examples, such as targets corresponding to authors that "*should have exactly two SIGMOD papers*" have been proposed in [Yiu et al. 2006]. For collaboration networks between touring artists where edges connect artists that have toured together, the targets may represent artists belonging to the same subgenre or record company and the RkNN query may be used to determine friendly relationships between bands belonging to a different company or subgenre. In social networks, where targets may represent users known for promoting illegal activities, RkNN queries may identify the cluster of the illegal organization that one suspect-user (the query vertex q) belongs.

In terms of RkNN queries and road networks, the work of [Safar et al. 2009] uses Network Voronoi cells (i.e., the set of vertices and arcs that are closer to the generator object) to answer RkNN queries. This work has only been tested on a relatively small network ($110K$ arcs) and all precomputed information is stored in a database. Despite its costly preprocessing (for calculating the Network Voronoi cells), queries still require $1.5s$ for $D = 0.05$ and $k = 1$. The query times further increase to $32s$ for $k = 20$. Later works focusing on *continuous RkNN queries* on road networks [Cheema et al. 2012] have only been tested with even smaller road networks ($22K$ arcs) and are different in scope from our work, which focuses on snapshot RkNN queries.

To the best of our knowledge, the only work focusing on other graph classes (besides road networks) is [Yiu et al. 2006]. This work proposes the Eager algorithm that traverses the network around query vertex q (in a way similar to Dijkstras algorithm or BFS for unweighted graphs), pruning vertices on the way that may not lead to RkNN results. For static RkNN queries where the target objects do not change and the number k is known in advance, the same authors propose the improved EagerM algorithm that similar to *ReHub*, has an offline and an online phase (that uses the precomputed information obtained from the offline phase) to accelerate RkNN queries. During the offline phase, the EagerM algorithm precomputes the kNN of each graph vertex, by using a combined network expansion from all targets at once. This information is later used to prune the graph traversal around query vertex q during the online phase. Unfortunately this work too, has only been tested on relatively sparse networks, e.g., road networks, grid networks (max degree 10), p2p graphs (avg degree 4) and a very small, sparse co-authorship graph ($4K$ nodes). Furthermore, all experimentation there for values of $k > 1$ (up to $k = 8$) refers to road networks, so the scalability of the proposed algorithms for denser graphs and larger values of k is debatable. Moreover, even for the EagerM algorithm, the online phase still has to perform a pruned Dijkstra-like expansion from the query vertex and thus, cannot be very fast for denser graphs and small values of D. Recently, Borutta et al. [Borutta et al. 2014] extended this work for time-dependent road networks, but the presented results were also not encouraging. The larger road network tested had $50k$ vertices (queries require more than $1s$ for $k = 1$) and for a road network of $10k$ nodes and $k = 8$, RkNN queries take more than $0.3s$ (without even adding the I/O cost). In a nutshell, all existing contributions and methods have not been tested on large-scale graphs, do not scale well for increasing k values and their performance highly depends on the target density D.

2.2. Hub Labels

Our work builds on the 2-hop labeling or Hub Labeling (HL) algorithm of [Gavoille et al. 2001; Cohen et al. 2002; Gavoille et al. 2004] in which, the preprocessing stage stores at every vertex v a forward $forwLab(v)$ and a backward label $backLab(v)$. The forward label $forwLab(v)$ is a sequence of pairs $(u, dist(v, u))$, with $u \in V$. Likewise, the backward label $backLab(v)$ contains pairs $(w, dist(w, v))$. Vertices u and w denote the *hubs of* v. The generated labels conform to *the cover property*, i.e., for any s and t, the set $forwLab(s) \cap backLab(t)$ must contain at least one hub that is on the shortest $s-t$ path. For undirected graphs $backLab(v) = forwLab(v)$. To find the network distance $dist(s, t)$ between two vertices s and t, a HL query must find the hub $v \in forwLab(s) \cap backLab(t)$ that minimizes the sum $dist(s, v) + dist(v, t)$. Since the pairs in each label are sorted by hub, this takes linear time by employing a coordinated sweep over both labels. The HL technique has been successfully used for road networks in [Abraham et al. 2011; 2012b; Akiba et al. 2014; Delling et al. 2013b].

In the case of large-scale graphs, the Pruned Landmark Labeling (PLL) algorithm of [Akiba et al. 2013] orders vertices by degree and then during preprocessing, performs one BFS per graph vertex, starting from the highest-order / degree vertices. At each iteration, each individual BFS is pruned by using the hub labels calculated from the previous searches. With this straightforward strategy, the PLL algorithm produces labels that are *minimal for a specified ordering* [Delling et al. 2014] but also exhibit quite uniform size between the different vertices [Akiba et al. 2013]. The later work of [Delling et al. 2014] improves the previous vertex ordering of [Akiba et al. 2013] and

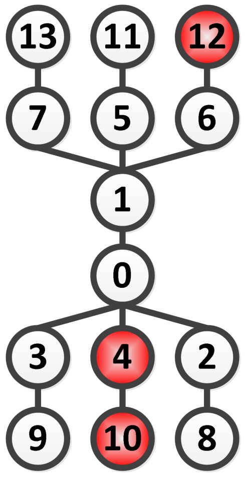


Fig. 1: An example graph G

Vertex	Hub Labels (hub, dist)
0	(0,0)
1	(0,1), (1,0)
2	(0,1), (2,0)
3	(0,1), (3,0)
4	**(0,1), (4,0)**
5	(0,2), (1,1), (5,0)
6	(0,2), (1,1), (6,0)
7	(0,2), (1,1), (7,0)
8	(0,2), (2,1), (8,0)
9	(0,2), (3,1), (9,0)
10	**(0,2), (4,1), (10,0)**
11	(0,3), (1,2), (5,1), (11,0)
12	**(0,3), (1,2), (6,1), (12,0)**
13	(0,3), (1,2), (7,1), (13,0)

Table I: The created hub-labels for the example graph G

the storage scheme of the hub labels for maximum compression. On a similar note, Jiang et al. [Jiang et al. 2014] propose their HopDB algorithm to provide an efficient HL index construction when the given graphs and the corresponding index are too big to fit into main memory. The HL method has also been used for one-to-many, many-to-many [Delling et al. 2011b], kNN queries on road networks in [Delling and Werneck 2015] and in the context of databases in [Abraham et al. 2012a; Efentakis et al. 2015a; Efentakis 2016] respectively. The core contribution of this work is to extend existing HL techniques in the context of RkNN queries on large-scale graphs and the proposed *ReHub* algorithm, presented in the following section.

3. THE *REHUB* ALGORITHM

What follows is the description of the *ReHub* (*Re*verse kNN+ *Hub* labels) algorithm that extends the Hub Labeling approach to efficiently handle RkNN queries on large-scale graphs. *ReHub* consists of two distinct, independent phases: (i) A slower, costlier *Offline* phase that takes place after the creation of the hub labels and depends only on the targets P (regardless of the query vertex q). (ii) An *Online* phase that uses the auxiliary data structures created during the Offline phase to compute the actual RkNN query results. The main benefit of the *ReHub* algorithm is that the costlier offline phase has to run only once and may service all RkNN queries for a specific set of targets, whereas the online phase (that actually depends on the query vertex q) is very fast (typically less than a $1ms$). Hence, *ReHub* may be used within the context of real-time applications, operating on large-scale graphs.

3.1. Offline Phase

The offline phase of the *ReHub* algorithm takes place after the creation of the hub labels. Although the *ReHub* algorithm works with any correct Hub Labeling algorithm, in this work we generate the necessary labels using the PLL algorithm of [Akiba et al. 2013], as provided by its authors in [Akiba et al. 2015]. To highlight the results of the PLL algorithm, the generated labels for the example undirected, unweighted graph G of Figure 1 are shown in Table I. In the remainder of this work we will refer to those labels as the *forward labels*. We also assume that the targets are located at vertices 4,10,12, i.e., $P = \{4, 10, 12\}$. The respective entries are highlighted in Table 1. For each

Hub	Labels (to-many) (id,dist)	kNN Backward Labels (k=2) (id,dist)	kNN Backward Labels (k=1) for ***ReHub*** (idx,dist)
0	(4,1), (10,2), (12,3)	(4,1), (10,2)	(0,1), (1,2)
1	(12,2)	(12,2)	(2,2)
4	(4,0), (10,1)	(4,0),(10,1)	(0,0), (1,1)
6	(12,1)	(12,1)	(2,1)
10	(10,0)	(10,0)	(1,0)
12	(12,0)	(12,0)	(2,0)

Table II: The kNN backward labels creation for *ReHub*, the example graph G, $k = 1$ and $P = \{4, 10, 12\}$

vertex v, the forward label $forwLab(v)$ is a vector of pairs $(u, dist(v, u))$ sorted by hub vertex u. This is the starting point for the offline phase of the *ReHub* algorithm, which in turn is divided in three smaller substages: (i) the *kNN backward labels* construction, (ii) the *batch kNN calculations* from all targets, and (iii) the RkNN *labels* construction. Each of these stages will be described in the following.

3.1.1. The kNN backward labels construction. To efficiently answer one-to-many queries with hub labels, Delling et al. [Delling et al. 2011b] construct an additional data structure (referred hereafter as the *labels-to-many*). The *labels-to-many* are constructed by storing separately the hub labels of the targets $P = \{P_1, \ldots, P_{|p|}\}$ ordered by hub [Delling et al. 2011b]. For each such hub u, those *labels-to-many* is a vector of pairs $(P_i, d(u, P_i))$. Expanding this approach for kNN queries, [Abraham et al. 2012a] showed that if the number k is known in advance (or the maximum k that will be serviced for kNN queries), then for each hub, it suffices to keep the best k pairs with the smallest distances per hub. The corresponding *kNN backward labels* data structure is hence constructed by ordering the labels of targets $P = \{P_1, \ldots, P_{|p|}\}$ by hub and then keeping the best k pairs with the smallest distances per hub. Although these works focused on road networks, their correctness still applies to undirected, unweighted graphs. The corresponding data structures (labels-to-many and kNN backward labels) for our example graph G, $P = \{4, 10, 12\}$ and $k = 2$ are shown in Table II.

```
KNNLAB $(P, |P|, k, forwLab, kNNLab)$
1 // Create a $|V|$-sized vector of empty bounded priority queues of size $k + 1$
2 Initialize$(kNNLab, (|V|, BoundPQue(k + 1)))$
3 for $i = 0$ to $|P|$
4     for $j = 0$ to $forwLab[P[i]].size$
5         $hub = forwLab[P[i]][j].hub$
6         $d = forwLab[P[i]][j].dist$
7         $kNNLab[hub].push(i, d)$
```

To efficiently calculate the kNN backward labels for *ReHub*, we combined elements from previous works, namely the works of [Knopp et al. 2007; Delling et al. 2011b; Delling and Werneck 2015; Geisberger 2011; Abraham et al. 2012a]. Still, we need to do some additional modifications: (i) When answering RkNN queries, we must assume that $k = k + 1$ during the construction of the kNN backward labels. This is necessary, since in our example the NN of target 10 (for $k = 1$) is by definition the same target, but for RkNN queries with $k = 1$, the NN neighbor of 10 is target 4. (ii) Similar to [Knopp et al. 2007], instead of storing the vertex IDs P_i of the targets in the kNN backward labels, we store the array index i of each target, as shown in the last column of Table II. This facilitates faster processing during the remaining substages of the offline and online phase of the *ReHub* algorithm. On the technical side, the kNN backward labels creation is quite fast, since we only have to loop through the forward labels of the targets in P and use a bounded priority queue of size $k+1$ per hub to store the $k+1$ pairs with the smallest distances per hub. This method offers two major advantages. (i) We do not need to build the intermediate labels-to-many data structure (column 2, Table II), which would be much slower, and (ii) when looping through the forward labels of each target, pairs with distances greater than the $k+1$ worst distance previously found for a specific hub may be safely ignored. The resulting pseudocode for the kNN backward labels construction is shown in procedure KNNLAB and throughout this process, for each hub we use a bounded priority queue of size $k+1$ that stores pairs in the form $(idx, dist)$ ordered by distance.

The kNN backward labels for *ReHub*, for the example graph G and $k=1$ are shown in Table II. For small-diameter graphs (like the ones used in this work) we will have many ties (in terms of distance), but keeping at most $k+1$ labels still ensures correctness. Due to the pruning of the PLL algorithm, in our example, *kNN backward labels* do not necessarily have as many as $k+1$ pairs per hub.

Compared to previous works, *ReHub* features some important implementation differences. The first four approaches [Knopp et al. 2007; Delling et al. 2011b; Delling and Werneck 2015; Geisberger 2011] store the entire backward search space from the targets (i.e., the labels-to-many) using a unified vector storing triples $(hub, id, dist)$ that at the end should be sorted according by (hub, id) in [Knopp et al. 2007; Delling et al. 2011b] or $(hub, dist)$ in [Delling and Werneck 2015; Geisberger 2011]. Contrarily, in *ReHub* we only store the $k+1$ pairs per hub ordered by distance, using an adjacency list representation for improving performance. In cases where k is not known in advance, we can store $kmax{+}1$ pairs per hub, where $kmax$ is the maximum value of k we will service for RkNN queries. This optimization originally appeared in [Abraham et al. 2012a; Foti et al. 2012] but with different implementations: On [Abraham et al. 2012a] it was implemented on a relational database and therefore the authors there do not provide any implementation details on how to efficiently do this calculation in main memory. Moreover, that work used the original vertex IDs of the targets (which makes sense in a database), whereas *ReHub* uses the target array indexes to accelerate subsequent computations. Likewise, the [Foti et al. 2012] work is based on Contraction Hierarchies (CH) [Geisberger et al. 2008b; Geisberger et al. 2012] and thus for computing the $k+1$ best pairs per hub, requires $|P|$ (one per target) backward CH searches which will be significantly slower than the main-memory implementation proposed here.

3.1.2. Batch kNN calculations from targets. After creating the kNN backward labels (column 4, Table II), we need to calculate the k-nearest neighbors (kNN) of each target. This is in stark contrast with the work of [Yiu et al. 2006] that needs to calculate the kNN of every graph vertex in the offline phase of the EagerM algorithm. For calculating the kNN of each target, we perform a total of $|P| \times kNN$ calculations, using the created kNN backward labels. Each of those kNN computations uses the method implicitly described in [Abraham et al. 2012a] (but in a database context and thus no main memory implementation details were provided there), with the additional constraint that for each target when traversing the kNN backward labels of one of its hubs, we skip the labels corresponding to this specific target index.

The simplified pseudocode for the batch kNN calculations from targets is shown in procedure BATCHKNNCALC. The $kNNResults$ are also stored in a $|P|$-sized vector of bounded priority queues of size k that store pairs in the form $(idx, dist)$ ordered by distance. For each target, when traversing the kNN backward labels of one of its hubs, we skip the pairs corresponding to the index of this specific target (Line 9 in the

Target ID	Forward Labels of targets (hub,dist)	Hub	kNN Backward Labels (k=1) for ***ReHub*** (idx,dist)	kNN Results (k=1) (idx, dist)
4	(0,1), (4,0)	0	(0,1), (1,2)	**(1,1)**
		1	(2,2)	
10	(0,2), (4,1), (10,0)	4	(0,0), (1,1)	**(0,1)**
		6	(2,1)	
12	(0,3), (1,2), (6,1), (12,0)	10	(1,0)	**(0,4)**
		12	(2,0)	

Table III: Batch kNN calculations process for the example graph G, $k = 1$ and $P = \{4, 10, 12\}$

pseudocode). Moreover, every time a new pair is pushed to the corresponding queue (Line 11), our customized push operation checks if the "pushed" target index already exists in the queue with a smaller or equal distance value than the pushed pair. If yes, we can safely ignore this pair. If, on the other hand, this target index exists in the queue with a larger distance value, we update this distance value and resort the queue. If the pushed target index does not already exist in the queue, our custom push operation checks if the queue has less than k items. In that case, the new pair enters the queue and the queue is resorted. If the queue has already k items, our push operation checks if the new pair is better (i.e., corresponds to a smaller distance) than the last (k) element of the queue. If yes, the last element is popped, the new pair enters the queue at the end and the queue is resorted. Since each queue is basically a vector of size k, popping back, pushing back and resorting this (rather small) priority queue are very fast operations.

```
BatchKnnCalc ($P, |P|, k, forwLab, kNNLab, kNNResults$)
1  // Create a $|P|$-sized vector of empty bounded priority queues of size $k$
2  Initialize($kNNResults, (|P|, BoundPQue(k))$)
3  parallel for $i = 0$ to $|P|$
4      for $j = 0$ to $forwLab[P[i]].size$
5          $hub = forwLab[P[i]][j].hub$
6          $d = forwLab[P[i]][j].dist$
7          for $l = 0$ to $kNNLab[hub].size$
8              $idx = kNNLab[hub][l].idx$
9              if $idx \neq i$
10                 $d2 = d + kNNLab[hub][l].dist$
11                 $kNNResults[i].push(idx, d2)$
```

Similar to [Geisberger 2011], every time a new pair $(idx, d2)$ enters the $kNNResults[i]$ queue for a specific target, we check if the queue already has k-items; In that case we store the worst label distance as a separate variable. If the distance d (Line 6) or the distance $d2$ (Line 10) are greater than this worst distance, we can safely skip this particular pair. Especially, in the second case (distance $d2$, Line 10) we can exit the third loop (Line 7) completely, since the kNN backward label of each hub is ordered by distance. This optimization (not shown in the pseudocode for readability) accelerates significantly each individual kNN calculation.

The results of this process are shown on Table III, where the combination of the forward labels of the targets $\{4, 10, 12\}$ with the kNN backward labels shows that the kNN of target 4 is the target with index 1, i.e., target 10, with distance 1. The kNN of target 10 is the target with index 0 (target 4) with the respective distance 1 and

Target ID	kNN Results (k=1) (idx, dist)	Forward Labels of targets (hub,dist)
4	(1,**1**)	(0,1), (4,0)
10	(0,**1**)	~~(0,2)~~, (4,1), (10,0)
12	(0,**4**)	(0,3), (1,2), (6,1), (12,0)

Hub	RkNN Labels (k=1) (idx,dist)
0	(0,1), (2,3)
1	(2,2)
4	(0,0), (1,1)
6	(2,1)
10	(1,0)
12	(2,0)

Table IV: RkNN labels construction for the example graph G, $k = 1$ and $P = \{4, 10, 12\}$

finally, the kNN of target 12 is the target with index 0 (target 4) with the respective distance 4. To facilitate faster computation, each kNN computation may be performed in parallel (Line 3 of procedure BATCHKNNCALC) since there is no interaction between the individual kNN calculations. Considering this is the slower substage of the offline phase (see Section 3.3), employing parallelism significantly drops the total preprocessing time required for *ReHub*'s offline phase. This is also an important advantage of *ReHub* in comparison to EagerM, since the offline phase of EagerM requires a combined network expansion from all targets at once, that cannot be parallelized.

3.1.3. The RkNN labels construction.

After calculating the kNN of each target, for answering RkNN queries it would suffice to run a *one-to-many* HL query from the query vertex q to all targets, by constructing and using the *labels-to-many* of targets P (see column 2, Table II) and then loop through the calculated distances to see if they are smaller or equal to the kNN distances calculated by the previous step. In our experimental section (See Section 4.3), we will refer to this naive approach as the *Naive-ToMany* algorithm. But we can do much better in *ReHub*. We construct an alternative data structure, referred hereafter as the *RkNN labels*, based on the observation that *we need to calculate distances to a specific target, if and only if those distances are equal or smaller than the distance of the kNN of this target*. If the targets are uniformly distributed in the graph, this optimization ensures that only hubs of relatively small distances from each target are added to the RkNN labels. Therefore, during the online phase, if the query vertex q is faraway from some targets, there would be no matching hubs between those targets and the query vertex.

The resulting pseudocode for the RkNN labels construction is shown in procedure RKNNLAB and the entire process is highlighted in Table IV. When we build the RkNN labels for target 10, we skip the pair $(0, 2)$ because the *NN* of target 10 is within distance of 1 and therefore pairs with greater distances than that (for this particular target) may be safely ignored. Again, when building the RkNN labels we use the targets' array indexes, instead of their IDs.

```
RKNNLAB $(P, |P|, k, forwLab, kNNResults, RkNNLab)$
1  Initialize$(RkNNLab, (|V|, vector < (idx, dist) >))$
2  for $i = 0$ to $|P|$
3      for $j = 0$ to $forwLab[P[i]].size$
4          $d = forwLab[P[i]][j].dist$
5          if $d \leq kNNResults[i][k-1]$
6              $hub = forwLab[P[i]][j].hub$
7              $RkNNLab[hub].push\_back(i, d)$
```

Several interesting observations can be made by comparing Tables II and IV. Firstly, as expected, the size of the RkNN labels (column 5, Table IV) is smaller than the labels-to-many (column 2, Table II). Although for our small example graph G this difference is minimal, for larger graphs it becomes significant. Therefore, using the RkNN labels will significantly improve the online phase of the *ReHub* algorithm. This will be clearly showcased in our experimentation presented in Section 4.3 where we compare *ReHub* with the NaiveToMany algorithm. Second, the kNN backward labels (column 4, Table II) are different than the RkNN labels (column 5, Table IV). Note that by using the kNN backward labels we can still answer kNN queries for any query vertex $q \in V$ and by using the RkNN labels we can answer RkNN queries within the same framework.

3.2. Online Phase

The offline phase of the *ReHub* algorithm runs only once for a specific set of targets P. Its final output is (i) The ***kNN Results***, i.e, a matrix of size $|P| \times k$ of (ordered by distance per row) $(idx, dist)$ pairs that contain the kNN of each target and (ii) the RkNN labels. The following online phase of the *ReHub* algorithm is basically a modified one-to-many HL query from the query vertex q that operates on the RkNN labels and is described by the pseudocode of procedure ONLINEPHASE. The output of the online phase is a vector (denoted out in the pseudocode) of size $|P|$ with all values set to infinity, except those that belong to the indexes of the targets of the RkNN set; those values are set to the correct distances from query vertex q to the respective targets. In our running example of the example graph G, for $P = \{4, 10, 12\}$ and $k = 1$, the online phase for a RkNN query from vertex 0 would only have to visit the RkNN labels of hub 0 (see Tables I and IV), the kNN Results for targets 4 and 12 (see Table III) and would finally output the result $out = \{1, \infty, 3\}$, meaning that the targets 4, 12 belong to the RkNN set of vertex 0 with distances 1 and 3 respectively.

We have also experimented with a hash map implementation of results (instead of using a $|P|$-sized vector) but our experiments showed that the proposed vector implementation was consistently faster for all tested datasets. This is attributed to several facts (i) At line 8 we have to check if the distance calculated for object P_i is better than previously calculated distance for the same object, which is faster using a vector (ii) Since the $|P|$-sized vector stores distances which are unsigned $8 - bit$ integers the corresponding size of the vector is quite small (at least for our tested datasets) and especially for small values of target density D. (iii) Modern compilers optimize the initialization of vectors (Line 1) using SIMD instructions for fill operations. Thus, initializing the vector is also a very fast operation.

```
ONLINEPHASE $(q, P, |P|, k, forwLab, kNNResults, RkNNLab, out)$
1  Initialize$(out, (|P|, \infty))$
2  for $i = 0$ to $forwLab[q].size$
3      $hub = forwLab[q][i].hub$
4      $d = forwLab[q][i].dist$
5      for $j = 0$ to $RkNNLab[hub].size$
6          $idx = RkNNLab[hub][j].idx$
7          $d2 = d + RkNNLab[hub][j].dist$
8          if $d2 < out[idx]$ &
           $d2 \leq kNNResults[idx][k-1].dist$
9              $out[idx] = d2$
```

THEOREM 3.1. *The* ReHub *algorithm is correct.*

PROOF. Building the kNN backward labels and then performing the batch kNN calculations to calculate the kNN of each target is correct, because it is based on the methodology of Abraham et al. [Abraham et al. 2012a] who proved its correctness. Building the RkNN labels is also correct, since we just reorder all labels of the targets according to hub, except those that correspond to distances greater than the kNN of each target. This ensures that we can calculate correct distances to any of those targets from any query vertex, except when this query vertex is farther than the kNN of a specific target. The online phase is also correct, since it operates on the RkNN labels and updates the result vector out for a specified target, only when the calculated distance is smaller or equal than the distance of the kNN of this target (Line 8, procedure ONLINEPHASE). Therefore the *ReHub* algorithm is also correct. □

The main advantage of the *ReHub* algorithm, in comparison to previous works, is the separation between the costlier offline phase, which runs only once for a specific set of targets and its very fast online phase. Although the EagerM algorithm of [Yiu et al. 2006] was based on the same principle, its corresponding online phase still needs to perform a slow BFS-like graph traversal from the query vertex q, which cannot be fast enough for real-time applications. Contrarily, *ReHub*'s online phase is orders of magnitude faster that its offline phase and thus rarely takes more than $1ms$. An additional benefit of *ReHub* compared to the works of [Yiu et al. 2006; Borutta et al. 2014] is that not only *ReHub* calculates the RkNN set of the query vertex but it also *calculates the correct network distances from the query vertex to any of the targets belonging in the RkNN set*. Regarding the online phase, operating on the RkNN labels is significantly faster, since for large graphs the size of the RkNN labels is significantly smaller than the *labels-to-many*. This will be clearly showcased in our experiments (see Section 4.3) where the online phase of *ReHub* will be significantly faster than the *NaiveToMany* implementation. Also the usage of target array indexes instead of the target IDs accelerates the whole process, since the final results vector out is of size $|P|$ instead of $|V|$ which makes its initialization faster (Line 1, procedure ONLINEPHASE), especially for smaller values of D. Also, accessing the kNN results of each target (Line 9) and the previous best value of results table (Line 8) are very cheap operations, since they operate on smaller vectors of size $|P|$. Moreover, the memory required for storing these intermediate data structures is also significantly smaller. This will be further quantified in the next section, where we analyse the complexity and memory requirements of the *ReHub* algorithm.

3.3. Complexity Analysis and Memory Requirements

If D is the target density, defined as $D = \frac{|P|}{|V|}$, then the number of targets is $D \cdot |V|$. The forward label of each vertex has an average of $\frac{|HL|}{|V|}$ hubs, where $|HL|$ is the total number of labels created by the hub labeling algorithm (PLL in our case). For this specific algorithm, Akiba et al. [Akiba et al. 2013] have shown that the "*size of the created labels does not differ much for different vertices and few vertices have much larger labels than the average*". Since we have $D \cdot |V|$ targets and $\frac{|HL|}{|V|}$ hubs per target, then the labels-to-many will have on average $D \cdot |HL|$ pairs. Regarding the offline phase, the kNN backward labels construction needs to access all those $D \cdot |HL|$ pairs (same as the labels-to-many) to construct the kNN backward labels that have a maximum of $k + 1$ pairs per hub. In the batch kNN calculations, we have a total of $D \cdot |V|$ kNN queries that each needs to access on average $(k + 1) \cdot \frac{|HL|}{|V|}$ pairs to create the kNN results of size of $k \cdot D \cdot |V|$. Therefore, the complexity of the batch kNN calculations will be $(k+1) \cdot D \cdot |HL|$. Finally, for the RkNN labels construction we need to access all $D \cdot |HL|$

Stage	Complexity	Memory for storing result (B)
kNN backward labels construction	$D \cdot \|HL\|$	$5 \cdot (k+1) \cdot \|V\|$
Batch kNN calculations	$(k+1) \cdot D \cdot \|HL\|$	$5 \cdot k \cdot D \cdot \|V\|$
RkNN labels construction	$D \cdot \|HL\|$	$5 \cdot \varepsilon \cdot D \cdot \|HL\|$
Online Phase	$\varepsilon \cdot D \cdot (\frac{\|HL\|}{\|V\|})^2$ (lower bound) $\varepsilon \cdot D \cdot \|HL\|$ (upper bound)	$D \cdot \|V\|$

Table V: *ReHub* complexity and memory requirements

pairs (same as the labels-to-many) and the $k \cdot D \cdot |V|$ results (to retrieve the worst k label per target). Conclusively, both the kNN and RkNN backward labels construction have a complexity of $D \cdot |HL|$ each (since $|HL| \gg |V|$), where the most costly batch kNN calculations stage has a complexity of $(k+1) \cdot D \cdot |HL|$. In all previous calculations, we assume that using and maintaining the bounded priority queues of size $k+1$ (kNN backward labels construction) or size k (batch kNN calculations) has no impact on the corresponding complexity due to the relatively small value of k.

Regarding the online phase, for large values of k, at the worst case, the online phase of *ReHub* will degrade to a one-to-many query between the query vertex q and the set of targets P. Therefore, we will first analyze the complexity of a one-to-many HL query. As showed earlier, the labels-to-many will have on average $D \cdot |HL|$ pairs. On the best case, those pairs will be equally distributed per hub and each hub on the labels-to-many will have an average of $D \cdot \frac{|HL|}{|V|}$ pairs. Since the forward label of the query vertex q will have on average of $\frac{|HL|}{|V|}$ hubs, on the best case a one-to-many query from the query vertex will access on average $D \cdot (\frac{|HL|}{|V|})^2$ pairs. At the worst case, the corresponding one-to-many query will have to access all $D \cdot |HL|$ pairs of the labels-to-many. Hence, the complexity of a one-to-many query will range between $D \cdot (\frac{|HL|}{|V|})^2$ (best case) and $D \cdot |HL|$ (worst case). Likewise, the online phase of *ReHub* will access between $\varepsilon \cdot D \cdot (\frac{|HL|}{|V|})^2$ (best case) and $\varepsilon \cdot D \cdot |HL|$ (worst case) pairs, where $\varepsilon < 1$ (since the size of the RkNN labels is smaller than the labels-to-many) and $\varepsilon = f(k, D, |B|)$, i.e., the value of ε for a specific graph depends on the target density D, the cardinality k of the RkNN result and the distribution $|B|$ of targets. In fact, our experimentation has shown that ε becomes smaller for larger values of D and smaller values of k and $|B|$. Our experimental results of Section 4.3 will also show that for the largest datasets and for small values of k, *ReHub*'s online phase complexity is close to the lower bound $\varepsilon \cdot D \cdot (\frac{|HL|}{|V|})^2$, whereas the *NaiveToMany* algorithm's complexity converges to the upper bound $D \cdot |HL|$. The aforementioned theoretical results are summarized in Table V where we also report the memory required for storing the results of each stage, considering that each pair requires 5 bytes for storage (4 bytes for target index + 1 byte for distance due to the small-world nature of large-scale graphs) and the output of the online phase is a $D \cdot |V|$ sized vector of distances.

3.4. Extension to Directed and Weighted Graphs

Throughout this work and the experimentation described in Section 4, we use undirected and unweighted graphs. However, the *ReHub* algorithm may be easily extended to directed graphs with the following changes: (i) In the offline phase the kNN backward labels must be constructed from the backward labels (ii) In the online phase we must use the backward labels of query vertex q. As before, the RkNN labels will still be constructed from the forward labels, even for directed networks. Note that *most* *previous methods like [Yiu et al. 2006; Borutta et al. 2014] have only been applied on undirected networks*. For weighted graphs *ReHub* will work out of the box, without requiring any further modifications.

3.5. The All-RkNN problem

Another alternative for answering RkNN queries for a static set of targets and a fixed value of k is to precompute the Reverse k-Nearest Neighbors for every graph vertex $q \in V$. We will refer hereafter to this variant of the RkNN query as the *All-RkNN problem*. With *ReHub* or EagerM (the best RkNN alternatives for a static set of objects) that would require to run the offline phase once and then perform the online phase for every graph vertex, for a total of $|V|$ iterations. As our experimentation will show (see Section 4.4), *ReHub* is the only viable solution for such an effort, since it will require less than $25min$ at the worst case, while *EagerM would require as much as 123 days*.

However, although precomputing the Reverse k-Nearest Neighbors for every graph vertex is now feasible with *ReHub*, is not always advisable. During the offline phase of the All-RkNN computation, we will need to have access to all *ReHub*'s data structures (kNN backward labels, kNN results and the RkNN labels), including the forward labels. Storing the complete RkNN results for all graph vertices would require an additional vector (denoted hereafter as the *RkNN results vector*) of memory size $5 \cdot |V| \cdot |RkNN|$ bytes (since each RkNN of a vertex requires 5 bytes for storage), where the number of the RkNN per vertex (contrary to kNN results where the respective size is at most k) is hard to predict. As expected, our experimentation has shown that the number of RkNN per vertex typically increases for larger values of k, D or $|B|$.

When we compared the necessary data structures for strictly answering RkNN queries for *ReHub* (kNN results and the RkNN labels) and the All-RkNN variation (the *RkNN results vector*), results showed that $ReHub$ requires as little as $700\times$ less memory (see Section 4.4). Moreover, in case of a live-online system that answers RkNN queries where objects might change at infrequent intervals, *ReHub* will require less than $1s$ (offline phase) to accommodate updates when targets change, whereas in the All-RkNN variation the system will have to stay offline for several minutes, rendering the corresponding solution totally impractical. Note that although the All-RkNN variation will still have faster query times (as the RkNN query will just require an O(1) access to the *RkNN results vector*), *ReHub* would require typically less than $1ms$ for the same query. For a typical web service this difference is minimal, considering that for such short query times the true bottleneck of the service would be to construct and return the (JSON or XML) response (i.e, the RkNN result) to the end user and not the actual RkNN query times. Thus, *ReHub* will still be the most pragmatic solution for RkNN queries on large-scale graphs, even compared to the All-RkNN variation.

4. EXPERIMENTS

To evaluate the performance of *ReHub* on various large-scale graphs, we conducted experiments on a workstation with a 4-core Intel i7-4771 processor clocked at 3.5GHz and 32 GB of RAM, running Ubuntu 14.04. Our code was written in C++, with GCC 4.8 and optimization level 3. We used OpenMP for parallelization. For benchmarking *ReHub*, we also implemented optimized, main-memory versions of the state-of-the-art *Eager* and *EagerM* algorithms [Yiu et al. 2006] for unweighted, undirected graphs (replacing Dijkstra with faster BFS expansions) and using adjacency arrays [Mehlhorn and Sanders 2008] (instead of adjacency lists) for the main-memory graph representation to facilitate faster performance. Note that our versions of the Eager and EagerM algorithms are significantly faster than the original paper, even after considering the fact that we are using a superior workstation for testing.

Graph	| V |	| E |	AVG degree	| HL | / | V |	PLL Preproc. Time (s)	Graph Size (Mb)	HL Size (Mb)
Amazon	334,863	925,872	3	689	230	9	1,101
Citeseer	268,495	1,156,647	4	408	110	10	523
Citeseer2	434,102	16,036,720	37	4,457	5,946	124	9,229
DBLP	540,486	15,245,729	28	3,628	5,720	118	9,352
Facebook	4,039	88,234	22	26	0.03	1	1
Gowalla	196,591	950,327	5	100	13	8	95
NotreDame	325,729	1,090,108	3	55	6	10	87
Slashdot	77,360	469,180	6	204	11	4	76
Slashdot2	82,168	504,230	6	216	13	4	85
Youtube	1,134,890	2,987,624	3	167	123	27	906

Table VI: Networks graphs statistics

The network graphs used in our experiments are taken from the Stanford Large Network Dataset Collection [Leskovec and Krevl 2014] and the 10th Dimacs Implementation Challenge [Bader et al. 2014]. All graphs are undirected, unweighted and connected. We used collaboration graphs (DBLP, Citeseer, Citeseer2) [Geisberger et al. 2008a], social networks (Facebook [McAuley and Leskovec 2012], Slashdot and Slashdot2 [Leskovec et al. 2009]), networks with ground-truth communities (Amazon, Youtube) [Yang and Leskovec 2012], web graphs (Notre Dame) [Albert et al. 1999] and location-based social networks (Gowalla) [Cho et al. 2011]. The graphs' average degree is between 3 and 37 and the PLL algorithm creates $26 - 4,457$ hub/distance pairs per vertex, requiring $0.03 - 5,950s$ for the hub labels' construction (see Table VI). We also report the memory size occupied for storing the original graphs (forward star representation) and for storing the labels, assuming that each $(id, dist)$ pair requires 5 bytes for storing, since distance is an unsigned $8 - bit$ integer (an optimization also used in the original PLL code), due to the small-world nature of the datasets. In fact, for our test datasets the graph diameter was less than 100. For each individual RkNN experiment we generate randomly 20 sets of targets of size $D \cdot |V|$ and then we generate 50 random query vertices per set (for a total of 1000 test cases), making sure that each query vertex q does not belong to the corresponding target set. For all experiments, we measure the running times of the offline and online phases of *ReHub* and EagerM separately. For those algorithms, the reported total time is the sum of the average running times of the online and offline phases.

4.1. Overall Performance

In this section, we evaluate the performance of *ReHub* in comparison with the *Eager* and *EagerM* algorithms of [Yiu et al. 2006] for ad-hoc RkNN queries. For *ReHub* and EagerM we report the total time required for both the offline and online phases. For $ReHub$'s offline phase, we only parallelized the batch kNN computations from targets. Contrarily, the EagerM offline phase cannot be parallelized, since it uses a single combined network expansion from all targets at the same time. Online phase is always sequential for all algorithms. Note that the PLL algorithm preprocessing time should not be added to *ReHub*'s offline phase, because it will take place only once for any set of targets P for the same graph. This makes sense, especially for ad-hoc queries or applications in which users may want to perform multiple RkNN queries over different sets of targets for the same graph. Moreover, with the PLL preprocessing we can still answer vertex-to-vertex queries, which is not possible with either Eager or EagerM. In addition, *ReHub* will work with any correct HL algorithm, and thus, for any forthcoming, faster HL algorithm, *ReHub* will still work without requiring any modifications.

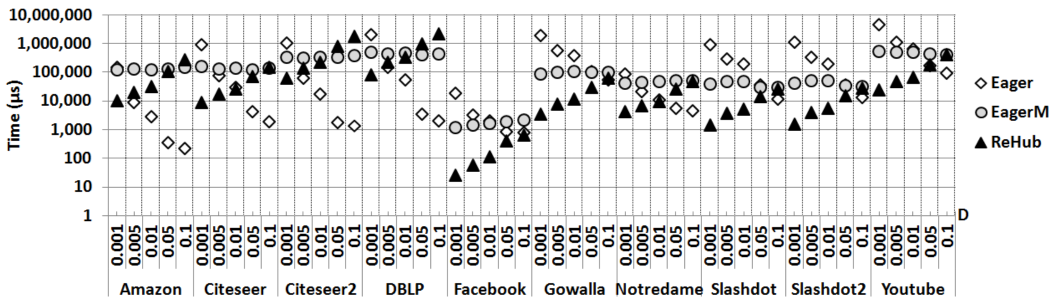


Fig. 2: *ReHub*, Eager and EagerM performance for $k = 1$ and varying values of D

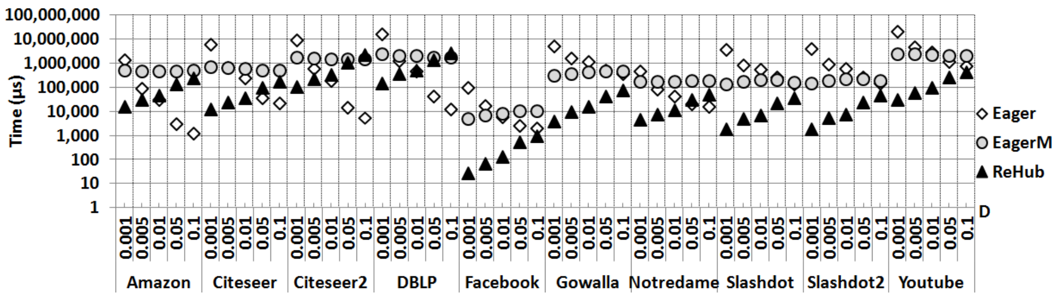


Fig. 3: *ReHub*, Eager and EagerM performance for $k = 4$ and varying values of D

4.1.1. Impact of target density D. In our first round of experiments we evaluate the performance of *ReHub* in comparison to Eager and EagerM, according to the target density $D = |P|/|V|$, i.e., for $k = 1$ and $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$ similar to the methodology followed in [Yiu et al. 2006]. Results are shown in Figure 2.

Similar to EagerM, *ReHub* is mainly optimized for static sets of targets, i.e., running the offline phase once and run multiple iterations of the online phase for different query vertices $q \in V$. However, the main advantage of *ReHub* is that its online phase is orders of magnitude faster than its offline phase (see Section 4.2), whereas in EagerM the running times of the offline and online phases are comparable to each other. But even for ad-hoc queries (where the targets change), results show that *ReHub* is faster for all values of D except $D = 0.1$ (i.e., for very dense targets) than both Eager and EagerM for the smallest datasets (Facebook, Gowalla, Slashdot, Slashdot2). For the remaining datasets (Amazon, CiteSeer, CiteSeer2, DBLP, Notredame, Youtube) results are evenly mixed: *ReHub* is typically faster for sparser targets ($D = 0.001, D = 0.005$) and Eager is faster for $D = 0.1$ and $D = 0.05$. However, on the majority of cases, *ReHub* still surpasses EagerM's performance. Note that although *ReHub*'s performance degrades for larger values of D, it is more stable than Eager and thus only exceeds $1s$ only for the worst performing graphs (DBLP $1.9s$, Citeseer2 $2.34s$) and dense targets ($D = 0.1$). Contrarily, for sparse targets ($D = 0.001$) Eager's performance is much worse, requiring more than $1s$ for Citeseer2 ($1.09s$), DBLP ($2.1s$), Gowalla ($2.0s$), Slashdot2 ($1.2s$) and YouTube ($4.9s$).

Repeating the previous experiments for $k = 4$ (see Fig. 3) further highlights the performance advantages of *ReHub*. Now *ReHub* is faster than both Eager and EagerM on Facebook, Gowalla, Slashdot, Slashdot2, YouTube for all values of D, faster on Citeseer, Notredame for $D = 0.001, 0.005, 0.01$ and faster on Amazon, CiteSeer2, DBLP for $D = 0.001, 0.005$. Note that for the YouTube dataset and $D = 0.001$ (the worst perform-

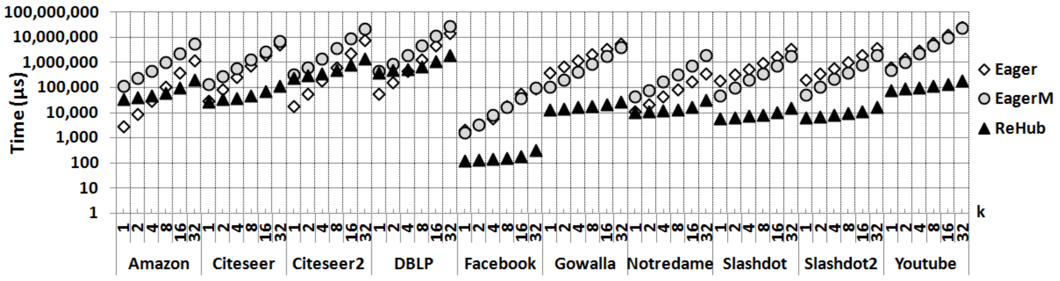


Fig. 4: *ReHub*, Eager and EagerM performance for $D = 0.01$ and varying values of k

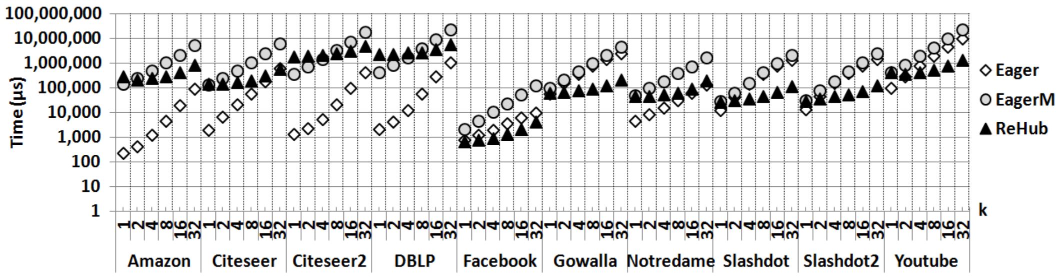


Fig. 5: *ReHub*, Eager and EagerM performance for $D = 0.1$ and varying values of k

ing case for the family of Eager algorithms), Eager requires more than $25s$, EagerM requires $2.5s$, while *ReHub* always requires less than $2.8s$ on all datasets.

4.1.2. Impact of cardinality k. In our second round of experiments, we assess the performance of the *ReHub*, *Eager* and *EagerM* algorithms in comparison to k. Again, for *ReHub* and EagerM we report the total time required for both the offline and online phases and for $ReHub$'s offline phase, we parallelized only the batch kNN computations from targets. Similar to the methodology of [Yiu et al. 2006], Figure 4 reports the corresponding results for $D = 0.01$ and $k = \{1, 2, 4, 8, 16, 32\}$.

Results show that *ReHub* exhibits excellent and stable performance, regardless of the value of k, contrarily to Eager and EagerM's performance which degrades polynomially with increasing values of k. As a result, *ReHub* is faster than both Eager and EagerM for $k \geq 8$ on all datasets and faster than EagerM for all tested values of k. Especially for large values of k ($k = 16, k = 32$) *ReHub* is *2-3 orders of magnitude faster than both Eager and EagerM* on all tested datasets, except Amazon, Citeseer2 and DBLP (there *ReHub* is still $3 - 8\times$ faster than Eager). Moreover, for the datasets of Citeseer, Facebook, Gowalla, NotreDame, Slashdot, Slashdot2 and YouTube, *ReHub* is the fastest algorithm for all values of k. Furthermore, *ReHub* requires more than $1s$ only for the Citeseer2 and DBLP datasets and $k = 32$, whereas Eager requires more than $1s$ for $k = 32$ on all datasets except Facebook and Notredame. In addition, Eager requires $4 - 25s$ for $k = 32$ on the datasets Citeseer2, DBLP, Gowalla, Slashdot2 and YouTube, with EagerM's performance being slightly better. Overall, *ReHub* exhibits excellent and stable performance for all values of k, contrary to Eager and EagerM who do not scale well for increasing values of k.

We repeat the previous experiments for $D = 0.1$ (see Fig. 5) which is the most favorable value of D for the Eager and EagerM algorithms (as shown previously in Fig. 2 and 3). However, *ReHub* still outperforms EagerM for $k \geq 8$ on all datasets and Eager

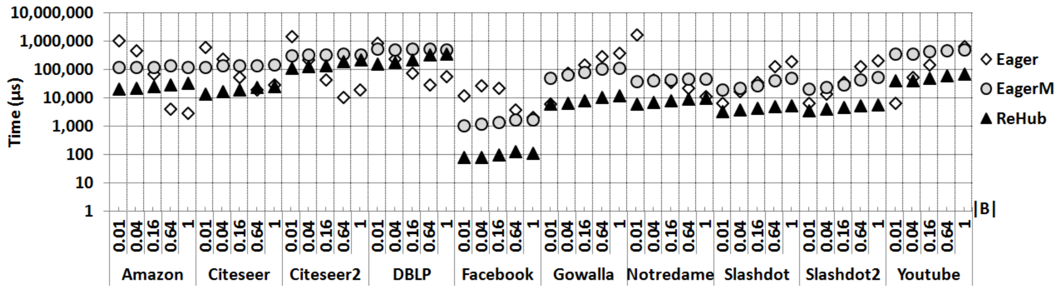


Fig. 6: *ReHub*, Eager and EagerM performance for $D = 0.01$, $k = 1$ and varying values of $|B|$

on Facebook, Gowalla, Slashdot, Slashdot2, YouTube for $k \geq 2$. Contrarily, Eager is better than *ReHub* on the datasets with the largest $|HL|/|V|$ ratio (Amazon, Citeseer, Citeseer2, DBLP).

4.1.3. Impact of target distribution. In our third experiment, we evaluate the impact of targets distribution to the performance of *ReHub*, Eager and EagerM. To that purpose, we adapt a methodology similar to [Delling et al. 2011b]. We pick a vertex at random and run BFS from it until reaching a predetermined number of vertices $|B|$. If B is the set of vertices visited during this search, we pick our targets P as a random subset of B, i.e., we select our targets from a ball of fixed size $|B|$. Hence, smaller values of $|B|$ correspond to targets that are closely together, whereas $|B| = 1$ represents random selected targets scattered uniformly in the graph network. We keep the density of targets steady at $D = 0.01$ and for $k = 1$ we experiment with different values of $|B|$ represented as percent of the total graph vertices. Figure 6 reports the corresponding results for $D = 0.01$, $k = 1$ and $|B| = \{0.01, 0.04, 0.16, 0.64, 1\}$.

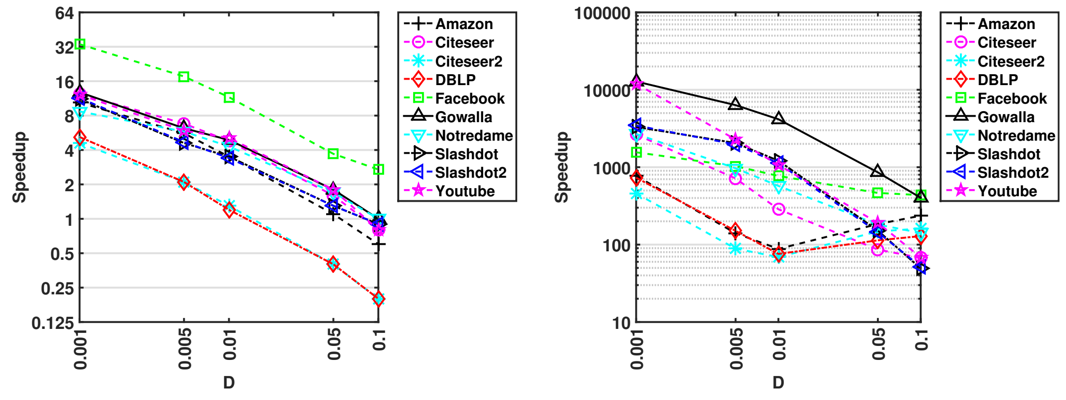
Results show that again, *ReHub* is the most stable algorithm, requiring less than $0.4s$ on all cases and always outperforming EagerM (which also provides stable performance), whereas Eager's performance fluctuates with the value of $|B|$, requiring more than $1s$ on multiple occasions. In addition, *ReHub* (and EagerM to a lesser degree) seems to benefit from denser distribution of targets (i.e., $|B| = 0.01$), since the more closely together are the targets, the more closer are the kNN of each target and thus the smaller is the size of the RkNN labels (see Section 3.1.3). Contrarily, Eager performs significantly worse on the largest datasets for those cases, since it has to visit largest portion of the graph if the query vertex q is faraway from the targets. Overall, on the majority of cases, *ReHub* is the fastest algorithm, exhibiting excellent performance for all values of $|B|$. Conclusively, *ReHub* provides the most stable performance regardless of the values of D, k or $|B|$ and outperforms Eager and EagerM on the majority of cases, especially for larger values of k where *ReHub* is $2 - 3$ orders of magnitude faster than its competitors.

4.2. Online and offline phase performance and memory requirements

In the previous section, we have demonstrated that *ReHub* outperforms the Eager and EagerM algorithms on the majority of cases for ad-hoc queries and varying values of D, k or $|B|$. However the main advantage of *ReHub* is the separation between the costlier offline phase which takes place only once for a fixed set of targets P and the very fast online phase with depends on the query vertex q. Accordingly in this section, we will compare the offline and online phases of *ReHub* and EagerM (Eager's perfor-

	Time (μs)									
	Offline					**Online**				
Dataset	**0.001**	**0.005**	**0.01**	**0.05**	**0.1**	**0.001**	**0.005**	**0.01**	**0.05**	**0.1**
Amazon	11,124	22,661	33,945	113,102	307,270	12.4	18.4	21.9	41.3	73.9
Citeseer	9,492	18,161	27,388	75,191	168,775	11.1	16.4	26.4	51.2	100.9
Citeseer2	65,733	156,484	248,521	847,910	**1,926,590**	94.3	132.2	171.2	302.6	628.8
DBLP	80,396	239,673	372,245	**1,072,020**	**2,479,130**	119.2	205.1	232.9	345.4	572.7
Facebook	30	71	125	432	705	0.1	0.2	0.3	0.7	0.9
Gowalla	3,935	8,567	12,463	31,550	65,445	2.9	8.0	12.9	51.5	89.5
Notredame	4,468	7,200	9,829	27,461	49,058	1.6	3.7	6.1	20.9	33.2
Slashdot	1,654	4,096	5,659	15,117	26,705	6.2	14.5	22.9	79.2	133.9
Slashdot2	1,728	4,462	6,194	16,579	31,006	6.2	15.6	26.5	83.9	155.5
YouTube	28,019	49,553	71,925	194,560	459,870	19.1	87.9	164.0	492.4	**1,179.3**

Table VII: *ReHub* offline and online phase performance for $k = 1$ and $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$



(a) Speedup of *ReHub* in comparison to EagerM for the offline phase

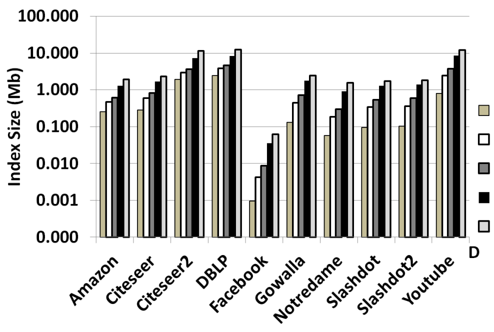
(b) Speedup of *ReHub* in comparison to EagerM for the online phase.

Fig. 7: Offline and online phases of *ReHub* and EagerM for $k = 1$ and varying values of D

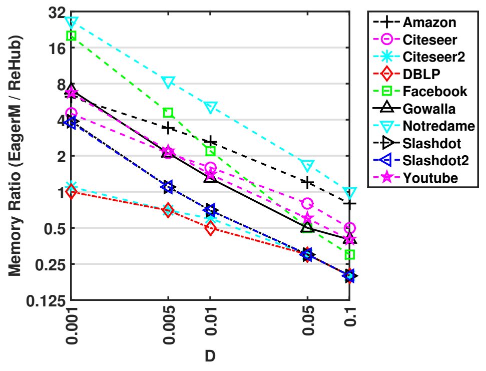
mance will be exactly the same, since it does not use any preprocessing) for a static set of targets and varying values of D, k or $|B|$.

4.2.1. Impact of target density D. In our first set of experiments we evaluate the performance of the offline and online phases of *ReHub* and EagerM, in comparison to the target density $D = |P|/|V|$. Figure 7 reports the speedup of *ReHub* in comparison to EagerM for the offline and online phases for $k = 1$ and $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$. Table VII also reports the corresponding absolute times (μs) for *ReHub* and we highlight in bold (i) the offline times when they surpass $1s$ and (ii) the online times when they surpass $1ms$ for easier reference.

Regarding the offline phase, *ReHub* is faster on all datasets and values of D, except Amazon, Citeseer2 and DBLP for $D = 0.05$ and $D = 0.1$. Moreover, for sparse targets ($D = 0.001$) *ReHub*'s offline phase is $4 - 33\times$ faster than EagerM. As for the online phase, *ReHub* is $1 - 5$ orders of magnitude faster than EagerM with this difference amplified for sparser targets, where *ReHub*'s online phase is $447 - 12,642\times$ faster than EagerM. On all datasets and values of D, *ReHub*'s online phase takes less than $1.2ms$ and thus *ReHub* is fast enough for real-time applications, contrary to EagerM's online phase which may require as much as $232ms$.



(a) Index size (Mb) for $ReHub$



(b) Memory requirements of EagerM in comparison to $ReHub$

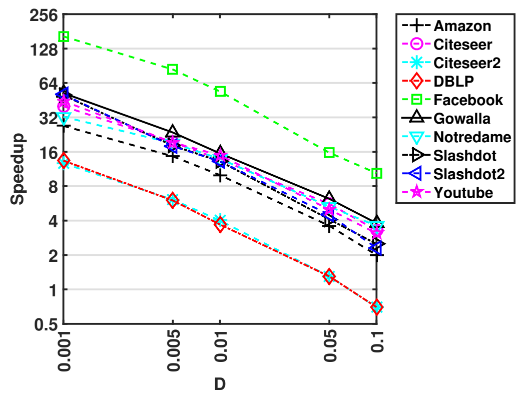
Fig. 8: Memory footprint of *ReHub* for $k = 1$ and varying values of D

	Time (μs)									
	Offline					**Online**				
Dataset	**0.001**	**0.005**	**0.01**	**0.05**	**0.1**	**0.001**	**0.005**	**0.01**	**0.05**	**0.1**
Amazon	16,116	31,798	48,143	142,840	251,600	21.6	33.9	46.5	87.8	109
Citeseer	12,475	25,345	36,862	104,669	176,461	17.3	33.9	50.3	122.7	202
Citeseer2	114,043	238,733	357,143	**1,129,250**	**2,303,340**	219.9	339.1	409.9	652.9	947
DBLP	144,484	361,241	545,826	**1,421,240**	**2,682,460**	290.3	525.1	636.3	881.3	**1,122**
Facebook	29	75	141	587	1,014	0.1	0.2	0.3	0.9	1
Gowalla	4,368	10,757	15,986	44,618	81,588	3.1	10.3	18.6	110.7	207
Notredame	4,757	8,226	11,491	31,752	53,541	2.1	5.5	10.2	35.7	56
Slashdot	1,887	5,199	7,182	21,521	37,483	6.7	19.7	36.5	183.3	369
Slashdot2	2,016	5,712	7,917	23,738	47,431	7.1	22.3	42.3	211.6	442
YouTube	29,648	61,782	105,365	265,353	448,440	27.8	157.7	315.5	**1,151.9**	**2,457**

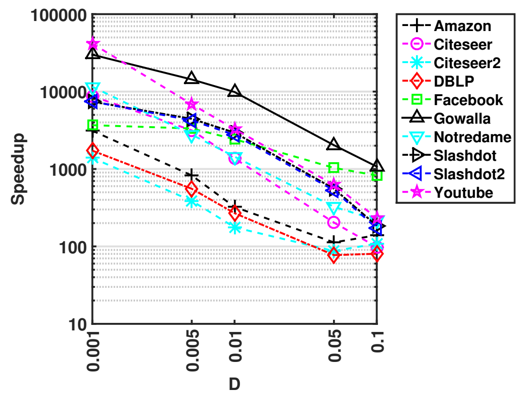
Table VIII: *ReHub* offline and online phase's performance for $k = 4$ and $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$

In terms of memory requirements, Figure 8 reports the memory required for storing the additional data structures for *ReHub* (kNN backward labels, kNN results and the RkNN labels) and the memory requirements of EagerM materialized information (EagerM's materialized information always requires $(5 \times k \times |V|)$ bytes [Yiu et al. 2006]) in comparison to $ReHub$, for the same setting as our previous experiment (i.e., for $k = 1$ and $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$). Note that (i) EagerM will require the original graph and (ii) *ReHub* will require the forward labels (see Section 3.2) for answering RkNN queries. The corresponding memory required for storing those data structures was reported in Table VI. Since, both the original graph and the forward labels may also be used for answering vertex-to-vertex queries, we omit them from the memory comparison to highlight only the overhead of the additional data structures required for answering RkNN queries. Results show that the memory required for the additional data structures for *ReHub* is always less than $13Mb$ even for the worst performing graphs (DBLP, Citeseer2). In comparison to EagerM, *ReHub* requires less memory for sparser targets ($D = 0.001$, $D = 0.005$) and EagerM requires less memory for denser distribution of targets ($D = 0.1$, $D = 0.05$).

We repeat the previous experiment by increasing k to 4. Figure 9 reports the speedup of *ReHub* in comparison to EagerM for the offline and online phases and Table VIII reports the corresponding absolute times (μs) for *ReHub*. Again, we highlight in bold *ReHub*'s offline and online times when they surpass $1s$ and $1ms$ respectively.

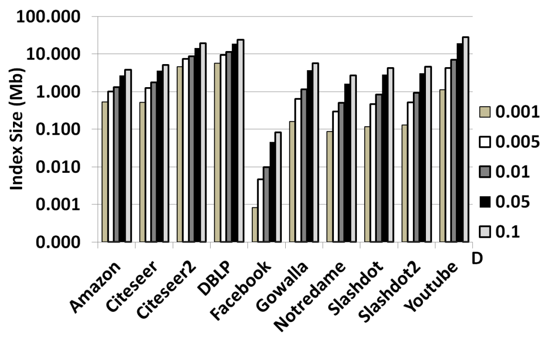


(a) Speedup of *ReHub* in comparison to EagerM for the offline phase

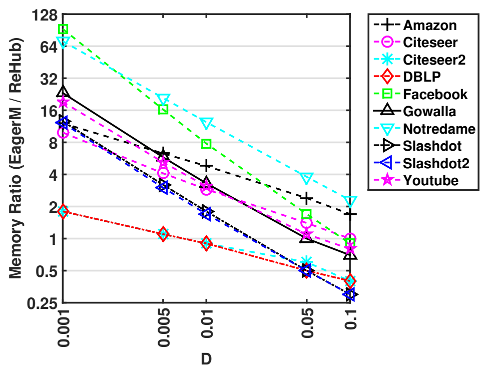


(b) Speedup of *ReHub* in comparison to EagerM for the online phase.

Fig. 9: Offline and online phases of *ReHub* and EagerM for $k = 4$ and varying values of D



(a) Index size (Mb) for $ReHub$



(b) Memory requirements of EagerM in comparison to $ReHub$

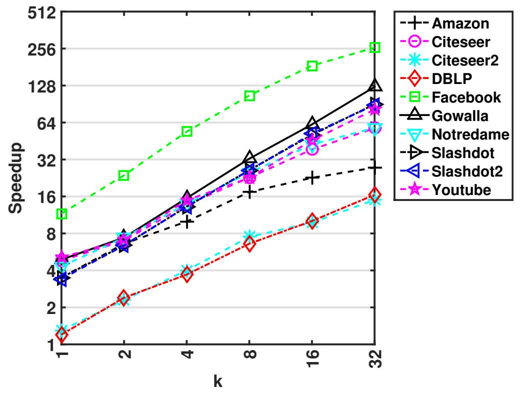
Fig. 10: Memory footprint of *ReHub* for $k = 4$ and varying values of D

Results show that increasing the value of k to 4 further augments the performance difference between *ReHub* and EagerM. Regarding the offline phase, for sparse distribution of targets ($D = 0.001$) *ReHub*'s offline phase is $13 - 167\times$ faster than EagerM and *ReHub*'s online phase is $1,382 - 28,873\times$ faster than EagerM. On all datasets and values of k, *ReHub*'s online phase takes less than $2.5ms$, contrary to EagerM's online phase which may require as much as $1s$ for $D = 0.001$ and the YouTube dataset.

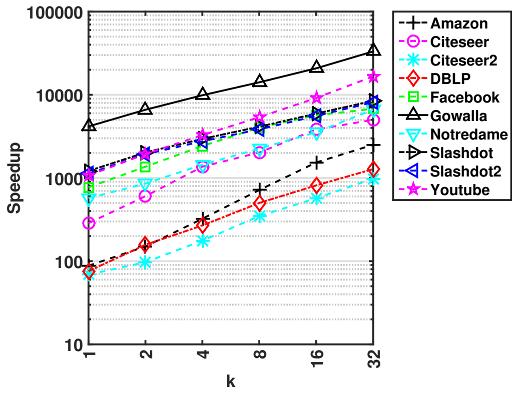
In terms of memory requirements, Figure 10 reports the memory required for storing the additional data structures for *ReHub* (kNN backward labels, kNN results and the RkNN labels) and the memory requirements of EagerM materialized information in comparison to $ReHub$, for the same setting as our previous experiment (i.e., for $k = 4$ and $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$). Results show that the memory required for the additional data structures for *ReHub* is always less than $24Mb$. In comparison to EagerM, *ReHub* requires less memory in most cases for sparser targets ($D = 0.001$,

	Time (μs)											
	Offline						**Online**					
Dataset	**1**	**2**	**4**	**8**	**16**	**32**	**1**	**2**	**4**	**8**	**16**	**32**
Amazon	33,945	39,957	48,143	60,210	97,838	199,641	21.9	30.3	46.5	71.8	113.1	181.7
Citeseer	27,388	33,016	36,862	45,109	68,236	114,299	26.4	38.5	50.3	86.2	124.5	191.8
Citeseer2	248,521	288,211	357,143	464,198	823,735	**1,387,480**	171.2	246.1	409.9	749.3	**1,399.7**	**2252.9**
DBLP	372,245	480,933	545,826	677,085	**1,074,350**	**1,787,630**	232.9	425.8	636.3	**1,100**	**1,724.1**	**2725.7**
Facebook	125	130	141	160	183	315	0.3	0.3	0.3	0.4	0.4	0.4
Gowalla	12,463	14,315	15,986	18,148	20,721	27,191	12.9	15.6	18.6	22.9	24.4	24.6
Notredame	9,829	10,657	11,491	12,728	16,530	30,058	6.1	7.1	10.2	13.2	16.7	21.6
Slashdot	5,659	6,507	7,182	8,126	10,002	15,260	22.9	29.8	36.5	42.2	45.7	49.2
Slashdot2	6,194	7,069	7,917	8,960	11,032	16,943	26.5	33.6	42.3	49.5	51.4	54.9
YouTube	71,925	91,094	105,365	112,468	125,870	186,024	164.0	227.7	315.5	406.9	496.2	553.9

Table IX: *ReHub* offline and online phase's performance for $D = 0.01$ and $k = \{1, 2, 4, 8, 16, 32\}$



(a) Speedup of *ReHub* in comparison to EagerM for the offline phase



(b) Speedup of *ReHub* in comparison to EagerM for the online phase.

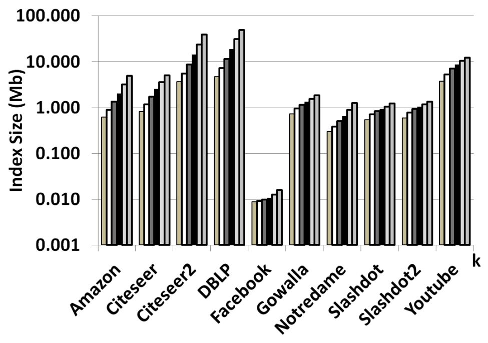
Fig. 11: Offline and online phases of *ReHub* and EagerM for $D = 0.01$ and varying values of k

$D = 0.005$) and EagerM requires less memory for denser targets ($D = 0.1$, $D = 0.05$) and the majority of datasets.

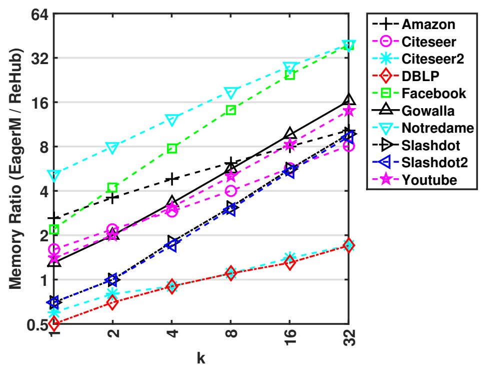
4.2.2. Impact of cardinality k. In our second round of experiments we evaluate the performance of the offline and online phases of *ReHub* and EagerM, in comparison to k. Figure 11 reports the speedup of *ReHub* in comparison to EagerM for the offline and online phases for $D = 0.01$ and $k = \{1, 2, 4, 8, 16, 32\}$. Table IX reports the corresponding absolute times (μs) for *ReHub* for easy reference. Again, we highlight in bold *ReHub*'s offline and online times when they surpass $1s$ and $1ms$ respectively.

Regarding the offline phase, *ReHub* is always faster on all datasets and values of k, with this difference amplified for larger values of k. For $k = 2$, *ReHub*'s offline phase is $2 - 24\times$ faster than EagerM. For $k = 32$, *ReHub*'s offline phase is $15 - 310\times$ faster than EagerM. Regarding the online phase, *ReHub* is $1 - 4$ *orders of magnitude faster than EagerM* with the difference amplified for $k = 32$, whereas *ReHub*'s online phase is $965 - 32,916\times$ faster than EagerM. On all datasets and values of k, *ReHub*'s online phase takes less than $2.7ms$, whereas EagerM's online phase might require as much as $9s$ for the YouTube dataset and $k = 32$.

In terms of memory requirements, Figure 12 reports the memory required for storing the additional data structures for *ReHub* and the memory requirements of Ea-



(a) Index size (Mb) for $ReHub$



(b) Memory requirements of EagerM in comparison to $ReHub$

Fig. 12: Memory footprint of *ReHub* for $D = 0.01$ and varying values of k

	Time (μs)											
	Offline						**Online**					
Dataset	**1**	**2**	**4**	**8**	**16**	**32**	**1**	**2**	**4**	**8**	**16**	**32**
Amazon	307,270	230,203	251,600	294,161	440,271	918128	74	77	109	181	341	826
Citeseer	168,775	160,628	176,461	209,395	315,282	610,787	101	138	202	327	563	**1211**
Citeseer2	**1,926,590**	**1,993,760**	**2,303,340**	**2,560,850**	**3,157,690**	**5,434,290**	629	646	947	**1,449**	**2,460**	**4,506**
DBLP	**2,479,130**	**2,404,150**	**2,682,460**	**2,849,190**	**3,970,470**	**6,062,680**	573	700	**1,122**	**1,932**	**3,418**	**6,642**
Facebook	705	808	1014	1362	2171	4547	1	1	1	1	1	1
Gowalla	65,445	71,988	81,588	94,582	131,292	221,960	90	127	207	297	363	438
Notredame	49,058	48,983	53,541	63,151	91,755	207,321	33	39	56	80	111	158
Slashdot	26,705	33,186	37,483	48,750	70,912	119,654	134	228	369	566	733	892
Slashdot2	31,006	37,075	47,431	54,893	80,008	128,524	155	258	442	660	883	**1,089**
YouTube	459,870	379,139	448,440	622,434	879,281	**1,444,430**	**1,179**	**1,435**	**2,457**	**4,318**	**5,825**	**7,202**

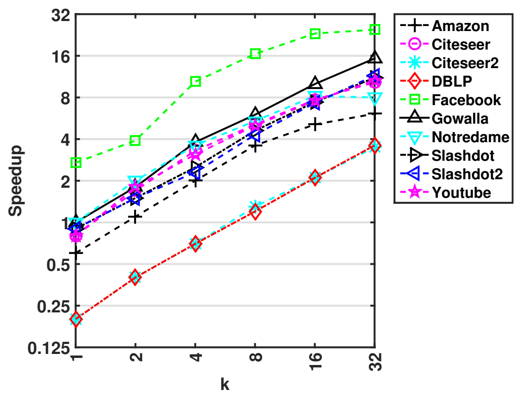
Table X: *ReHub* offline and online phase's performance for $D = 0.1$ and $k = \{1, 2, 4, 8, 16, 32\}$

gerM materialized information in comparison to $ReHub$, for the same setting (i.e., for $D = 0.01$ and $k = \{1, 2, 4, 8, 16, 32\}$). Results show that the memory required for the additional data structures for *ReHub* is less than $50Mb$ even for $k = 32$. In comparison to EagerM, *ReHub* always requires less memory for $k > 4$ with this difference intensified for $k = 32$ where *ReHub* may require $39\times$ less memory than EagerM.

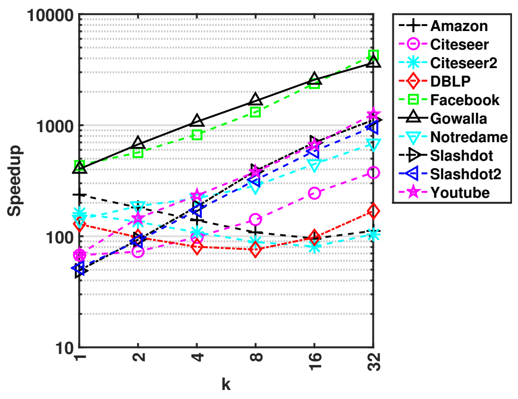
We repeat the previous experiment by increasing the value of D to 0.1, which is the most favorable value for the EagerM algorithm. Figure 13 reports the speedup of *ReHub* in comparison to EagerM for the offline and online phases and Table X reports the corresponding absolute times (μs) for *ReHub*. Again, we highlight in bold *ReHub*'s offline and online times when they surpass $1s$ and $1ms$ respectively.

Although increasing the value of D to 0.1, closes the performance gap between *ReHub* and EagerM, *ReHub*'s offline phase is still faster for $k > 4$. For $k = 32$, *ReHub*'s offline phase is $3 - 28\times$ faster than EagerM. As for the online phase, *ReHub* is significantly faster than EagerM on all cases. For $k = 32$, *ReHub*'s online phase is $79 - 4,392\times$ faster than EagerM. For all datasets and values of k, *ReHub*'s online phase takes less than $7.2ms$, contrary to EagerM's online phase which may require as much as $8s$ for $k = 32$ and the YouTube dataset.

Regarding memory requirements, Figure 14 reports the memory required for storing the additional data structures for *ReHub* and the memory requirements of EagerM materialized information in comparison to $ReHub$, for the same setting (i.e., for $D = 0.1$

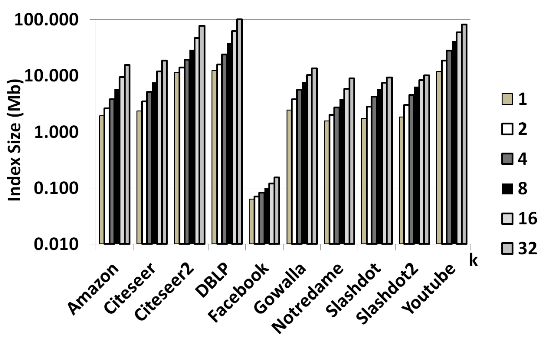


(a) Speedup of *ReHub* in comparison to EagerM for the offline phase

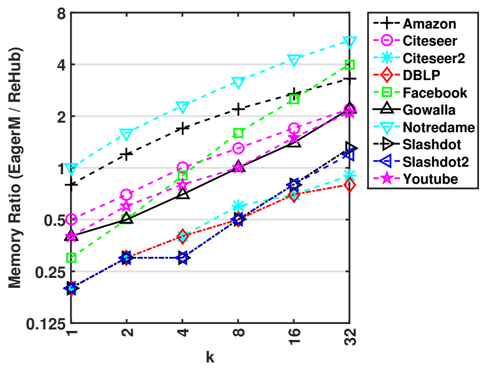


(b) Speedup of *ReHub* in comparison to EagerM for the online phase.

Fig. 13: Offline and online phases of *ReHub* and EagerM for $D = 0.1$ and varying values of k



(a) Index size (Mb) for $ReHub$



(b) Memory requirements of EagerM in comparison to $ReHub$

Fig. 14: Memory footprint of *ReHub* for $D = 0.1$ and varying values of k

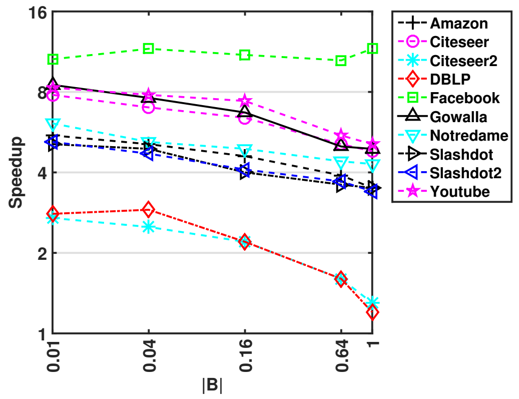
and $k = \{1, 2, 4, 8, 16, 32\}$). Results show that the memory required for the additional data structures for *ReHub* is less than $101Mb$ even for $k = 32$. In comparison to EagerM, *ReHub* always requires less memory for $k > 8$ and EagerM requires less memory for most datasets and $k \leq 2$.

4.2.3. Impact of target distribution. In our third experiment, we evaluate the impact of targets distribution to the performance of the offline and online phases of *ReHub* and EagerM. Again, we keep the density of targets steady at $D = 0.01$ and for $k = 1$ we experiment with different values of $|B|$ represented as percent of the total graph vertices. Figure 15 reports the speedup of *ReHub* in comparison to EagerM for the offline and online phases for $D = 0.01$, $k = 1$ and $|B| = \{0.01, 0.04, 0.016.0.64, 1\}$. Table XI reports the corresponding absolute times (μs) for *ReHub* for easy reference.

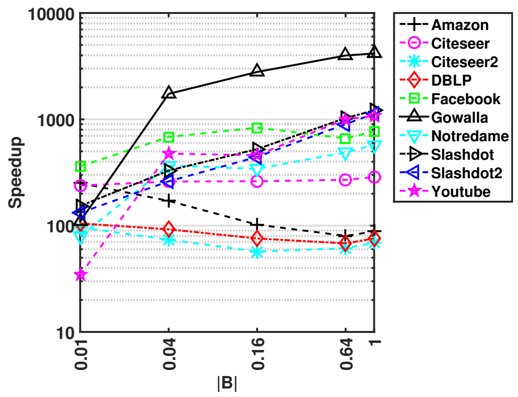
Regarding the offline phase, *ReHub* is always faster on all datasets and values of $|B|$, with this difference amplified for smaller values of $|B|$ (more concentrated targets). This is due to the fact that when targets are closely together, the more closer are the

	Time (μs)									
	Offline					**Online**				
Dataset	**0.01**	**0.04**	**0.16**	**0.64**	**1**	**0.01**	**0.04**	**0.16**	**0.64**	**1**
Amazon	20,590	22,768	27,227	30,914	33,945	7.0	9.6	15.9	20.5	21.9
Citeseer	14,574	17,287	20,728	24,572	27,388	7.2	12.3	17.0	23.2	26.4
Citeseer2	114,165	130,117	144,524	200,649	248,521	81.1	107.7	136.4	157.9	171.2
DBLP	179,615	196,323	229,409	332,143	372,245	81.0	112.4	141.0	212.2	232.9
Facebook	91	90	107	134	125	0.2	0.2	0.2	0.3	0.3
Gowalla	6,387	6,862	8,223	10,990	12,463	7.3	7.3	8.7	12.0	12.9
Notredame	6,151	7,348	8,392	9,532	9,829	4.4	5.2	6.4	6.5	6.1
Slashdot	3,635	3,999	4,724	5,254	5,659	8.0	10.1	13.9	20.9	22.9
Slashdot2	3,868	4,352	5,089	5,714	6,194	8.4	11.0	16.1	22.3	26.5
YouTube	41,454	45,322	48,880	66,930	71,925	47.7	65.5	111.9	136.4	164.0

Table XI: *ReHub* offline and online phase's performance for $D = 0.01$, $k = 1$ and varying values of $|B|$



(a) Speedup of *ReHub* in comparison to EagerM for the offline phase

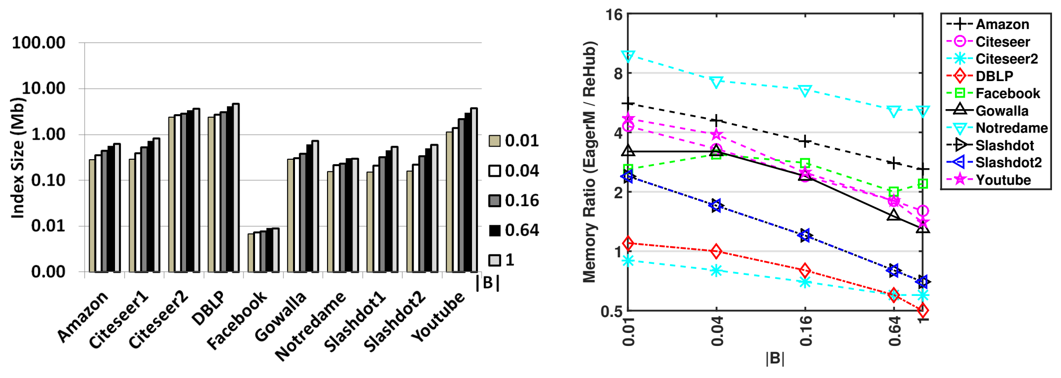


(b) Speedup of *ReHub* in comparison to EagerM for the online phase.

Fig. 15: Offline and online phases of *ReHub* and EagerM for D=0.01, k=1 and varying values of $|B|$

kNN of each target and hence the size of the RkNN labels is smaller (see Section 3.1.3). Considering the online phase, *ReHub* is always at least $17\times$ faster than EagerM for all values of $|B|$ with the difference amplified for $|B| = 1$ (random targets), whereas *ReHub*'s online phase is on average $830\times$ faster than EagerM.

Considering memory requirements, Figure 16 reports the memory required for storing the additional data structures for *ReHub* and the memory requirements of EagerM materialized information in comparison to $ReHub$, for the previous setting (i.e., for $D = 0.01$, $k = 1$ and varying values of $|B|$). Results show that the memory required for the additional data structures for *ReHub* remains less than $5Mb$. In fact, more skewed distributions of targets (i.e., smaller values of $|B|$) favor *ReHub* more and thus *ReHub* may require $5\times$ less memory than EagerM for $|B| = 0.01$. Conclusively, *ReHub*'s online phase never takes more than $7.2ms$ on all experiments, regardless of the values of D, k and $|B|$ and is therefore orders of magnitude faster than EagerM which may require up to $9s$ for $k = 32$, while requiring less memory than EagerM on many cases. Therefore for a static set of targets, *ReHub*'s *online phase is orders of magnitude faster than either Eager or EagerM*, and is thus, the only RkNN solution fast enough for real-time applications.



(a) Index size (Mb) for $ReHub$

(b) Memory requirements of EagerM in comparison to $ReHub$

Fig. 16: Memory footprint of *ReHub* for k=1, D=0.01 and varying values of $|B|$

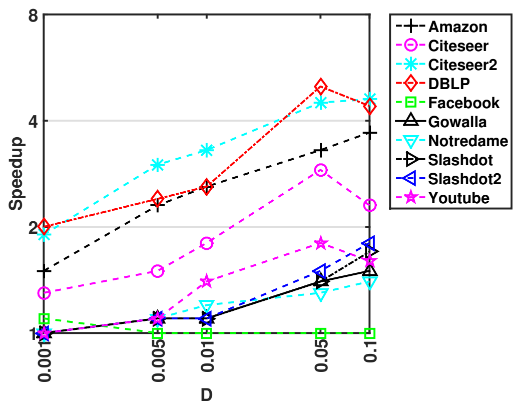
4.3. Comparison to NaiveToMany and theoretical insights

In the previous section we have shown that *ReHub* vastly outperforms EagerM for a static set of targets. In Section 3.1.3, we have also stated that RkNN queries may also be answered by keeping the two initial stages (i.e., the kNN backward labels construction and the batch kNN calculations from targets) of *ReHub*'s offline phase untouched and then construct the *labels-to-many* data structure. Then, we could run a *one-to-many* query using the labels-to-many, while checking which of the calculated distances to a target are smaller than the distance of the kNN of this specific target. We referred to this RkNN alternative solution as the *NaiveToMany* algorithm. In this section, we will compare *ReHub* and the NaiveToMany algorithm and provide additional information about *ReHub*'s online phase complexity to highlight how the *ReHub*'s online phase performs in comparison to the theoretical bounds presented in Section 3.3.

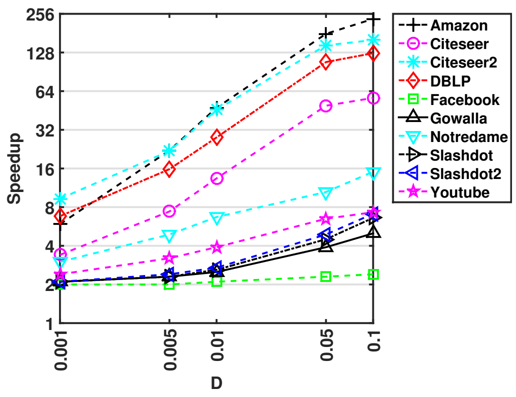
4.3.1. Impact of target density D. In our first round of experiments we evaluate the performance of the offline and online phases of *ReHub* in comparison with the NaiveToMany algorithm for varying values of target density D. Figure 17 reports the speedup of *ReHub* in comparison to NaiveToMany for the offline and online phases for $k = 1$ and $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$.

Results show that *ReHub*'s offline phase is always faster than NaiveToMany with this difference amplified for denser targets (i.e., larger values of D), where *ReHub*'s offline phase is $2 - 4\times$ faster for the largest datasets. The same pattern is even more prominent for the online phase, where *ReHub* is $52 - 221\times$ faster for $D = 0.1$ and the datasets with the highest $|HL|/|V|$ ratio (i.e., Amazon, Citeseer, Citeseer2, DBLP). Considering that large values of D is the least favourable case for *ReHub* compared to the previous Eager and EagerM algorithms, this clearly showcases how *ReHub* is clearly a much better alternative than the NaiveToMany algorithm.

In terms of memory requirements, Figure 18(a) reports the size of the RkNN labels in comparison to the labels-to-many (i.e., the inverse of the variable ε introduced in Section 3.3). Results show that the construction of the RkNN labels is very efficient, since RkNN labels may be $35 - 115\times$ smaller than the labels-to-many for $D = 0.1$ and the datasets with the highest $|HL|/|V|$ ratio. Overall, the variable ε decreases with increasing values of D. Figure 18(b) shows the total memory required for storing the corresponding data structures (kNN backward labels, kNN results) and the RkNN labels (*ReHub*) or labels-to-many (NaiveToMany) for the two algorithms. Again, *ReHub*

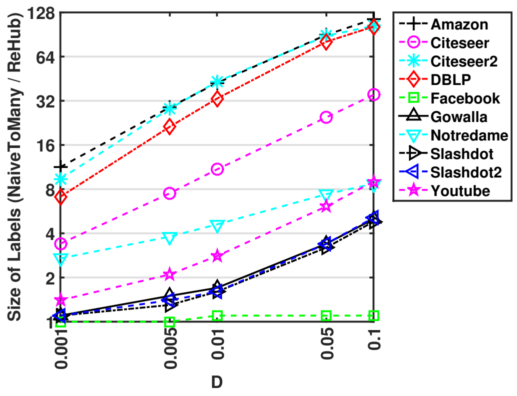


(a) Speedup of *ReHub* in comparison to NaiveToMany for the offline phase

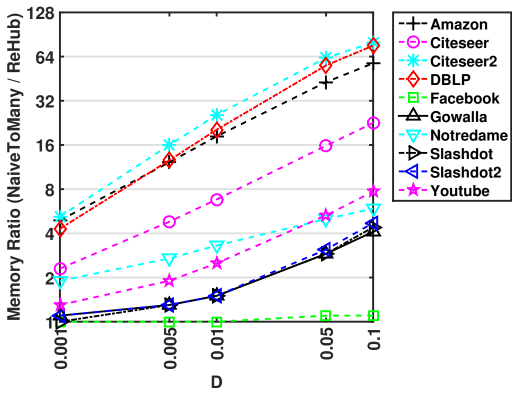


(b) Speedup of *ReHub* in comparison to NaiveToMany for the online phase.

Fig. 17: Offline and online phases of *ReHub* and NaiveToMany for $k = 1$ and varying values of D



(a) Size of RkNN and to-Many labels for $ReHub$.



(b) Memory requirements of NaiveToMany in comparison to $ReHub$.

Fig. 18: Memory footprint of *ReHub* vs NaiveToMany for k=1 and varying values of D

occupies significantly less memory, requiring $22-80\times$ less memory for $D = 0.1$ and the datasets with the highest $|HL|/|V|$ ratio.

In Section 3.3 we have shown that the complexity of the online phase of *ReHub* is expected to be between $\varepsilon \cdot D \cdot (\frac{|HL|}{|V|})^2$ (best case) and $\varepsilon \cdot D \cdot |HL|$ (worst case). Likewise the complexity of the online phase of the NaiveToMany algorithm is the same as the one-to-many query and hence between $D \cdot (\frac{|HL|}{|V|})^2$ and $D \cdot |HL|$. Figure 19 shows the number of $(id, dist)$ pairs accessed during the online phase by the two algorithms, in comparison to $\varepsilon \cdot D \cdot (\frac{|HL|}{|V|})^2$ (lower bound for *ReHub*) and $D \cdot |HL|$ (upper bound for NaiveToMany). Results show that for the datasets with the highest $|HL|/|V|$ ratio and for $D = 0.1$, *ReHub*'s performance is very close to its lower bound, while the corresponding NaiveToMany performance converges to its upper bound. Thus, for those datasets the complexity of *ReHub*'s online phase is significantly better than the Naive-

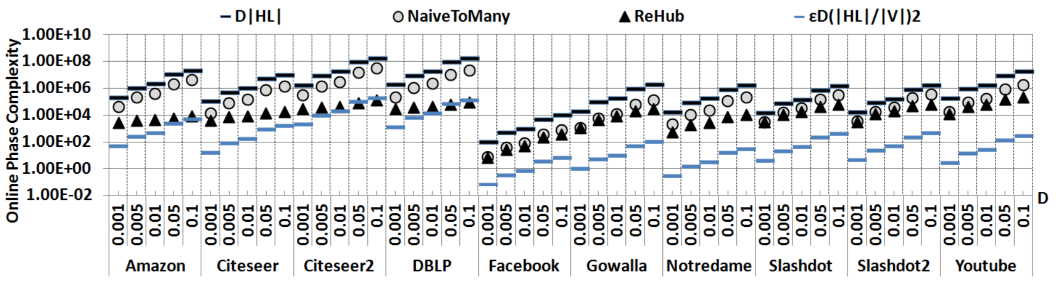
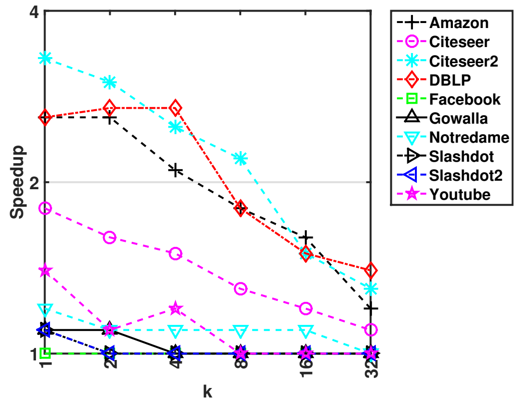
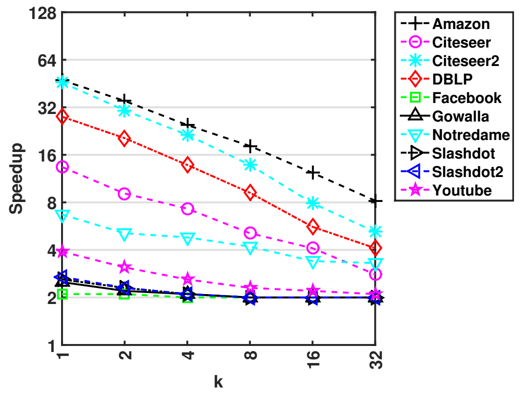


Fig. 19: Online phase complexity of *ReHub* and NaiveToMany in comparison to theoretical bounds for $k = 1$ and varying values of D



(a) Speedup of *ReHub* in comparison to NaiveToMany for the offline phase



(b) Speedup of *ReHub* in comparison to NaiveToMany for the online phase.

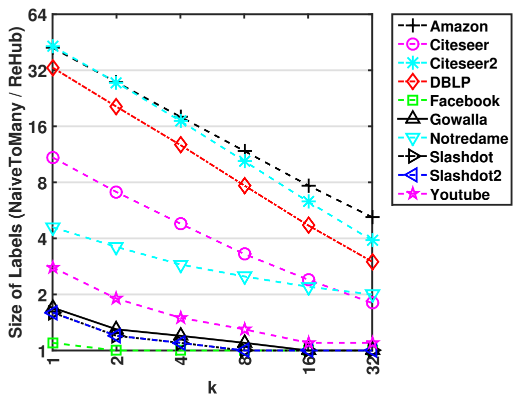
Fig. 20: Offline and online phases of *ReHub* and NaiveToMany for $D = 0.01$ and varying values of k

ToMany algorithm, while in the smaller datasets although *ReHub* still performs better, the performance gap between the two algorithms is not that prominent.

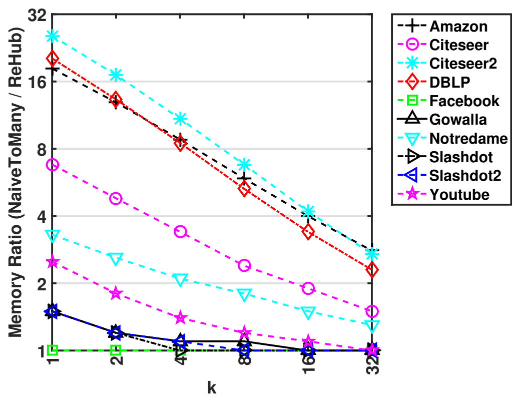
4.3.2. Impact of cardinality k. In our second round of experiments we evaluate the performance of the offline and online phases of *ReHub* and NaiveToMany, in comparison to k. Figure 20 reports the speedup of *ReHub* in comparison to EagerM for the offline and online phases for $D = 0.01$ and $k = \{1, 2, 4, 8, 16, 32\}$.

For the offline phase, *ReHub* is always faster on all datasets and values of k, with this difference increased for smaller values of k. For $k = 1$, *ReHub*'s offline phase is $3\times$ faster for the datasets with the highest $|HL|/|V|$ ratio (Amazon, Citeseer2, DBLP). As expected, this advantage diminishes for larger values of k, since for those values of k *ReHub* will converge to the NaiveToMany algorithm. Considering the online phase, *ReHub* is $30 - 43\times$ faster than NaiveToMany for $k = 1$ and the datasets with the highest $|HL|/|V|$ ratio, while for $k = 32$, *ReHub* is still $3 - 7\times$ faster for the same datasets. Again, considering that small values of k is the least favourable case for *ReHub* in comparison to the previous Eager and EagerM algorithms, again our results demonstrate that *ReHub* is a much better solution than the NaiveToMany algorithm.

In terms of memory requirements, Figure 21(a) reports the size of the RkNN labels in comparison to the labels-to-many (i.e., the inverse of the variable ε). Results show



(a) Size of RkNN and to-Many labels for $ReHub$.



(b) Memory requirements of NaiveToMany in comparison to $ReHub$.

Fig. 21: Memory footprint of ***ReHub*** vs NaiveToMany for D=0.01 and varying values of k

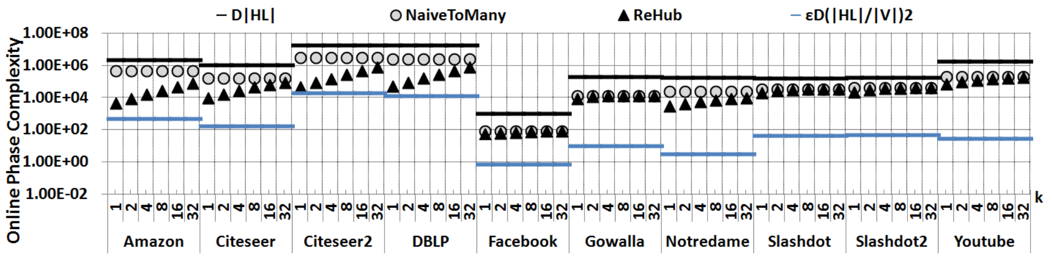
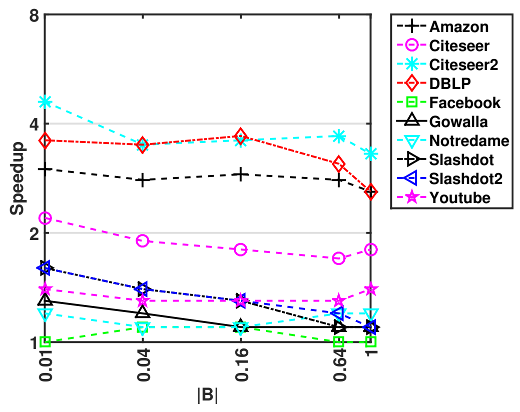


Fig. 22: Online phase complexity of ***ReHub*** and NaiveToMany in comparison to theoretical bounds for $D = 0.01$ and varying values of k

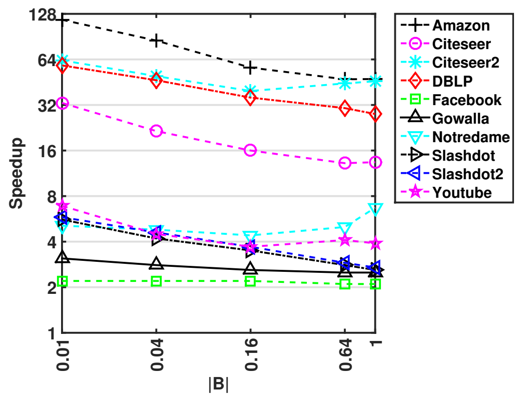
that for $k = 1$, RkNN labels may be $11 - 42\times$ smaller than the labels-to-many and the datasets with the highest $|HL|/|V|$ ratio. Overall, the variable ε increases with increasing values of k. Figure 21(b) shows the total memory required for storing the corresponding data structures (kNN backward labels, kNN results) and the RkNN labels (***ReHub***) or the labels-to-many (NaiveToMany) for the two algorithms. Again, ***ReHub*** occupies significantly less memory, requiring $6 - 26\times$ less memory for $k = 1$ and the datasets with the highest $|HL|/|V|$ ratio.

Finally, Figure 22 shows the number of $(id, dist)$ pairs accessed during the online phase by the two algorithms, in comparison to $\varepsilon \cdot D \cdot (\frac{|HL|}{|V|})^2$ (lower bound for ***ReHub***) and $D \cdot |HL|$ (upper bound for NaiveToMany). Results show that for the datasets with the highest $|HL|/|V|$ ratio and for small values of k, ***ReHub***'s performance is close to the theoretical lowest bound, while the corresponding NaiveToMany performance converges to the corresponding upper bound. For the remaining datasets, the complexity of ***ReHub***'s online phase is marginally better than the NaiveToMany algorithm.

4.3.3. Impact of target distribution. In our third experiment, we evaluate the impact of target distribution to the performance of the offline and online phases of ***ReHub*** and NaiveToMany. We keep the density of targets steady at $D = 0.01$ and for $k = 1$ we experiment with different values of $|B|$ represented as percent of the total graph vertices.

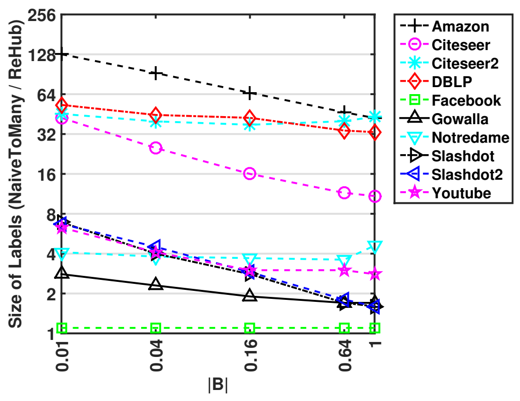


(a) Speedup of *ReHub* in comparison to NaiveToMany for the offline phase

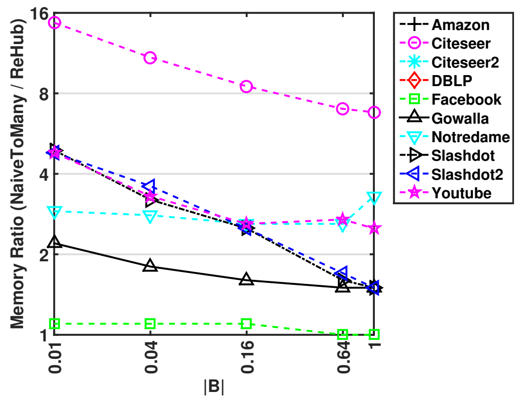


(b) Speedup of *ReHub* in comparison to NaiveToMany for the online phase.

Fig. 23: Offline and online phases of *ReHub* and NaiveToMany for D=0.01, k=1 and varying values of $|B|$



(a) Size of RkNN and to-Many labels for $ReHub$.



(b) Memory requirements of NaiveToMany in comparison to $ReHub$.

Fig. 24: Memory footprint of *ReHub* vs NaiveToMany for k=1, D=0.01 and varying values of $|B|$

Figure 23 reports the speedup of *ReHub* in comparison to NaiveToMany for the offline and online phases for $D = 0.01$, $k = 1$ and $|B| = \{0.01, 0.04, 0.016.0.64, 1\}$.

Regarding the offline phase, *ReHub* is consistently faster on all datasets and values of $|B|$, with this difference slightly amplified for smaller values of $|B|$. This is due to the fact that when targets are closely together, the more closer are the kNN of each target and hence the size of the RkNN labels is smaller in comparison to the labels-to-many (see Section 3.1.3). For the online phase, *ReHub* is $58 - 101\times$ faster than NaiveToMany for $|B| = 0.01$ and the datasets with the highest $|HL|/|V|$ ratio, while for $|B| = 1$, *ReHub* remains $13 - 45\times$ faster for the same datasets.

Figure 24(a) reports the size of the RkNN labels in comparison to the labels-to-many (i.e., the inverse of the variable ε). Results show that the variable ε decreases with decreasing values of $|B|$ (denser distribution of objects). Figure 24(b) shows the total

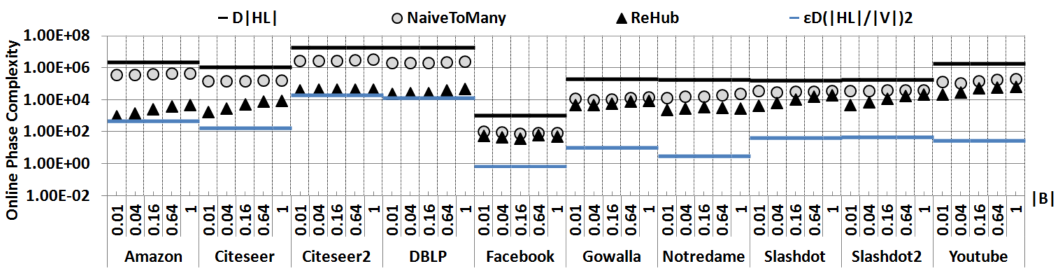


Fig. 25: Online phase complexity of *ReHub* and NaiveToMany in comparison to theoretical bounds for $D = 0.01$, $k = 1$ and varying values of $|B|$

memory required for storing the corresponding data structures (kNN backward labels, kNN results) and the RkNN labels (*ReHub*) and labels-to-many (NaiveToMany) for the two algorithms. Again, *ReHub* occupies significantly less memory, requiring $15 - 30\times$ less memory for $B = 0.01$ and the datasets with the highest $|HL|/|V|$ ratio.

Finally, Figure 25 shows the number of $(id, dist)$ pairs accessed during the online phase by the two algorithms, in comparison to $\varepsilon \cdot D \cdot (\frac{|HL|}{|V|})^2$ (lower bound for *ReHub*) and $D \cdot |HL|$ (upper bound for NaiveToMany). Results show that for the datasets with the highest $|HL|/|V|$ ratio and for small values of $|B|$ (i.e., for $|B| \leq 0.04$), *ReHub*'s performance is very close to the theoretical lowest bound, while the corresponding NaiveToMany performance converges to the corresponding upper bound.

Conclusively, *ReHub* not only outperforms the previous state-of-the-art Eager and EagerM algorithms but also exhibits optimal performance, especially for the least favourable cases (i.e., for small values of k and large values of D). Hence, it will be very hard to provide a better hub labeling solution for RkNN queries than *ReHub*.

4.4. The All-RkNN problem

As stated in Section 3.5, for the All-RkNN variation (i.e., precomputing the RkNN for every graph vertex $v \in V$ for a static set of targets P and a known value of k), we can use either *ReHub* or EagerM. For both algorithms that would require to run the offline phase once and then perform the online phase for every graph vertex, for a total of $|V|$ iterations. In this section, we will provide the approximate query times for running the online phase for $|V|$ iterations for EagerM and *ReHub*, based on the average query times reported on the previous sections. We have omitted the offline phase in those calculations, since it will have negligible impact on the calculated times. We use the aforementioned approximation, because running the complete experiments for EagerM would require several months that would be infeasible.

Figure 26 shows the calculated query times (in minutes) for running the online phase for *ReHub* and EagerM for a total of $|V|$ iterations for $k = 1$ and varying values of D (i.e., for $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$). Results show that, *ReHub* is the only viable solution for the *All-RkNN problem*. For all datasets and values of D, *ReHub* would require less than $9.3min$, while EagerM would require more than 3 days (73 hours) for the YouTube dataset and $D = 0.001$. Even for $D = 0.1$ which is the most favourable case for EagerM, the corresponding calculation would still require more than 21 hours. For Citeseer2 and DBLP and for $D = 0.001$, EagerM would require 5 and 12 hours respectively, while *ReHub* would only require less than $1.1min$ on both cases.

Figure 27 shows the calculated query times for running the online phase for *ReHub* and EagerM for a total of $|V|$ iterations for $D = 0.01$ and varying values of k (i.e., for $k = \{1, 2, 4, 8, 16, 32\}$). Here, results favour *ReHub* even more. For all datasets and

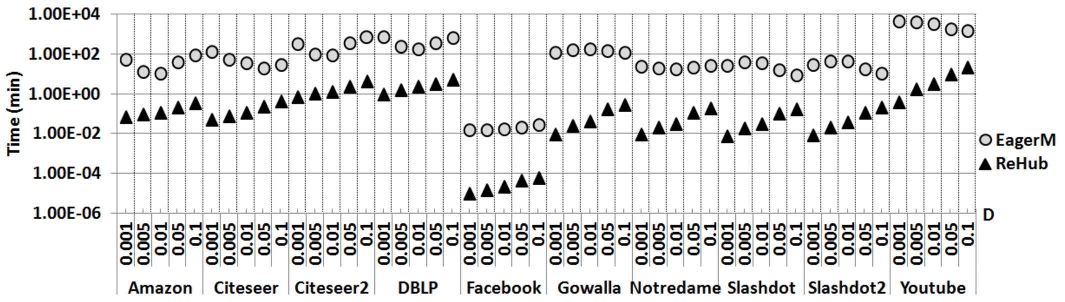


Fig. 26: All-RkNN preprocessing time with *ReHub* and EagerM for $k = 1$ and varying values of D

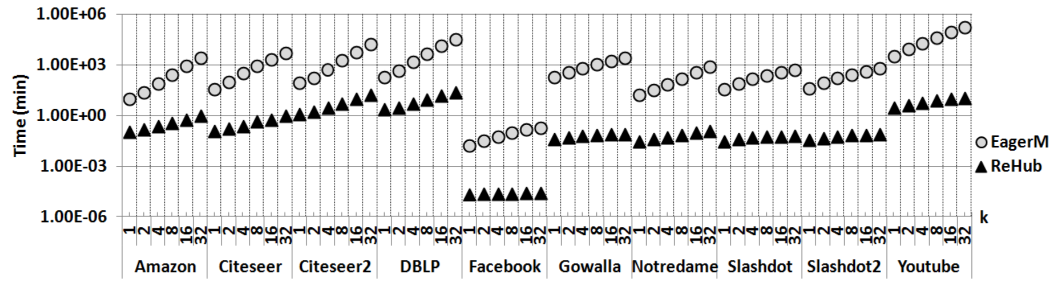


Fig. 27: All-RkNN preprocessing time with *ReHub* and EagerM for $D = 0.01$ and varying values of k

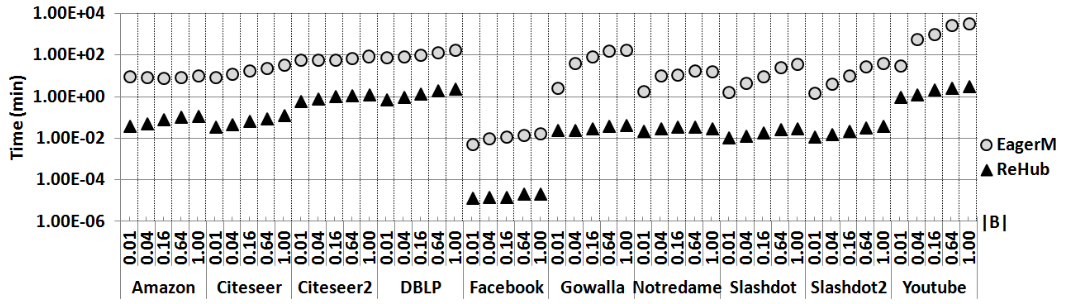
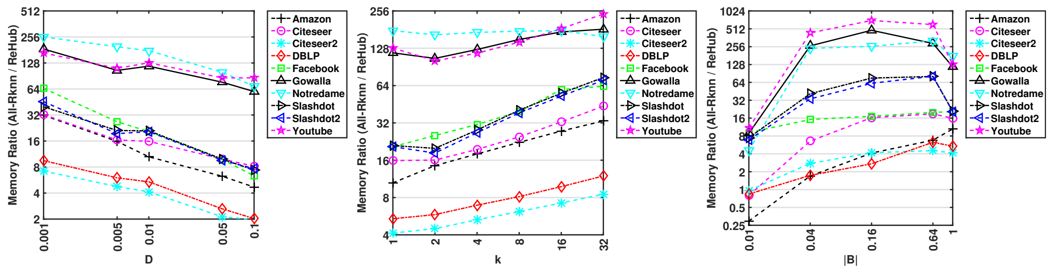


Fig. 28: All-RkNN preprocessing time with *ReHub* and EagerM for $D = 0.01$, $k = 1$ and varying values of $|B|$

values of k, *ReHub* would require less than $25min$, while for $k = 32$ *EagerM would require 123, 23 and 11 days* for the YouTube, DBLP and Citeseer2 datasets respectively. Hence, *ReHub* is the only practical solution for solving the All-RkNN query variation for increasing values of k.

Finally, Figure 28 shows the calculated query times (in minutes) for running the online phase for *ReHub* and EagerM for a total of $|V|$ iterations for $k = 1$, $D = 0.01$ and varying values of $|B|$ (i.e., for $|B| = \{0.01, 0.04, 0.16, 0.64, 1\}$). Once again, *ReHub* would never require more than $3min$ for all datasets and values of $|B|$, while EagerM would require $2.2days$ for the YouTube dataset and $|B| = 1$. Conclusively, *ReHub* is the only RkNN algorithm fast enough to make the computation of the RkNN of all graph



(a) For $k = 1$ and varying values of D

(b) For $D = 0.01$ and varying values of k

(c) For $k = 1$, $D = 0.01$ and varying values of $|B|$

Fig. 29: Memory requirements of All-RkNN in comparison to $ReHub$

vertices for large-scale networks practical, requiring less than a few minutes for all tested values of D, k and $|B|$.

4.4.1. Memory requirements. Our previous experimentation has clearly shown that ***ReHub*** is the only practical solution for precomputing the RkNN for every graph vertex $v \in V$ for a static set of targets and a known value of k. This section will compare the memory requirements of storing those RkNN results for all graph vertices (i.e., the ***RkNN results vector*** - see Section 3.5) with the necessary ***ReHub*** data structures (kNN results and the RkNN labels) for answering RkNN queries. Note that this comparison is already unfair for ***ReHub*** because the All-RkNN variation will still require all ***ReHub***'s data structures (including the forward labels) during its offline phase. For fairness, we also excluded the forward labels from the comparison, because the forward labels for both algorithms should be kept in main memory, in case that the set of targets P or the value of k changes. However, since running the complete All-RkNN experiments even by ***ReHub*** for all datasets, values of k, D and $|B|$ would require several days, we approximated the number of RkNN results returned per vertex by the same 1000 RkNN queries used in the previous sections and hence the size of the ***RkNN results vector*** is approximated by $5 \cdot |V| \cdot \overline{|RkNN|}$ bytes (since for storing each RkNN of a graph vertex requires 5 bytes). Figure 29 shows the corresponding results.

Figure 29(a) shows that for $k = 1$ and varying values of D, ***ReHub*** requires $2 - 256\times$ less memory than the All-RkNN variation. Although, this difference decreases with larger values of D, for the YouTube dataset ***ReHub*** still requires $86\times$ less memory, even for $D = 0.1$. Likewise, Figure 29(b) shows that for $D = 0.01$ and varying values of k, ***ReHub*** requires $2 - 241\times$ less memory than the All-RkNN variation, with this difference amplified with larger values of k. Interestingly enough, on the YouTube dataset and $k = 32$ the All-RkNN storage *takes as much as 2.5GB*, whereas ***ReHub*** only requires $10Mb$. Finally, Figure 29(c) shows that for $k = 1$, $D = 0.01$ and varying values of $|B|$, ***ReHub*** requires up to $708\times$ less memory than the All-RkNN variation, with this difference amplified for $|B|$ between 0.16 and 0.64.

Hence, although ***ReHub***'s online phase will always be slower than the All-RkNN solution, ***ReHub*** requires significantly less main memory. On the practical scenario of a static graph (with no vertex or edge updates) where targets may change in infrequent intervals or applications in which users perform multiple concurrent RkNN queries over different sets of targets, ***ReHub*** would typically require less than $1s$ (for its offline phase) when targets change, whereas in the All-RkNN variation the system will have to stay offline for a few minutes, rendering the corresponding solution totally impractical. Thus, ***ReHub*** is still the most practical solution for real-world applications, combining fast query performance and applicability.

4.5. Summary

Our extensive experimentation has shown that *ReHub* exhibits excellent query performance and requires very small additional memory for all tested networks, regardless of the target density, the cardinality k of the RkNN result or the distribution of targets. In comparison to previous works, *ReHub* clearly outperforms all previous solutions (Eager, EagerM) tested on large-scale graphs on the majority of cases, especially for increasing values of k, where the performance of previous state-of-the-art methods degrades polynomially. Especially for static sets of targets, the online phase of *ReHub* is orders of magnitude faster than EagerM, making *ReHub* the only RkNN algorithm fast enough for real-time applications on large-scale networks. Moreover, we have demonstrated that *ReHub* is the only practical solution for precomputing the RkNN of every graph vertex, requiring less than a few minutes on all cases. In addition, Efentakis et al. [Efentakis et al. 2015a] have showed that the online phase of *ReHub* may be easily translated to a simple SQL query on a open-source database engine, making *ReHub* the only RkNN solution that may also be used on a pure-SQL context, for even greater versatility and scalability.

Moreover, we showed that *ReHub* can easily handle networks where the size of the created labels are more than three thousand hubs per vertex (e.g., Citeseer2, DBLP) and hence, the proposed algorithm will be even more efficient and faster when applied to sparser graph classes (e.g., road networks), where the size of the created labels are less than a few hundred hubs per vertex for well behaving metrics (e.g., travel times). We have also provided theoretical bounds for *ReHub*'s performance and have demonstrated how *ReHub* actually performs in practice, in comparison to those bounds. Compared to the specialized RkNN solutions presented in road networks [Safar et al. 2009; Borutta et al. 2014] *ReHub may handle two orders of magnitude larger, denser networks* (those previous methods were only tested on very small road networks of 110k arcs and 50k vertices respectively), *may scale easily for* $k = 32$, where previous secondary storage methods have only been tested for up to $k = 8$ [Borutta et al. 2014] or $k = 20$ [Safar et al. 2009]. But even then, e.g., for $k = 8$ those methods require more than $300ms$ [Borutta et al. 2014], whereas for similarly small networks (e.g. Gowalla) *ReHub*'s offline phase requires $< 20ms$ and the online phase $< 0.02ms$. Even for larger networks, the online phase typically requires less than $1ms$, i.e., *ReHub is at least 3 orders of magnitude faster*. Therefore, it is safe to say that *ReHub* will be the best overall RkNN algorithm for any network where HL algorithms typically perform well.

5. CONCLUSION AND FUTURE WORK

This work introduced *ReHub*, a novel main-memory algorithm that extends the Hub Labeling approach to efficiently handle RkNN queries on large-scale graphs. Our experimentation showed that *ReHub* provides excellent query performance, has minimal additional memory requirements and scales very well with the network size, the target density, the target distribution, the size of the labels and the cardinality of the reverse k-nearest neighbor result. Given these results, *ReHub* clearly outperforms all previous methods on the majority of cases and is thus, the best overall and most complete solution for RkNN queries with the added advantage that it is the only solution which is fast enough for use in real-time applications. Moreover, our experimentation has shown that *ReHub* is the only practical solution for precomputing the RkNN of every graph vertex, requiring less than a few minutes on all cases, while previous solutions would require days for the same computation. As later extension papers have already demonstrated, *ReHub*'s online phase may be easily translated to a simple SQL query, for use in cases where a pure secondary storage database solution is preferable.

Directions for future work are to extend *ReHub* towards handling targets updates, i.e., vertices may be added or deleted from the targets' set. Not having to redo the offline phase from scratch for such updates will significantly increase the practical applicability of the algorithm. Also testing our results on directed graphs and road networks will further showcase the algorithm's performance with respect to a wider range of graph classes, additional hub labelling algorithms and domains.

REFERENCES

Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. 2012a. HLDB: Location-based Services in Databases. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems (SIGSPATIAL '12)*. ACM, New York, NY, USA, 339–348. DOI:http://dx.doi.org/10.1145/2424321.2424365

Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2011. A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks. In *Experimental Algorithms*, PanosM. Pardalos and Steffen Rebennack (Eds.). Lecture Notes in Computer Science, Vol. 6630. Springer Berlin Heidelberg, 230–241. DOI:http://dx.doi.org/10.1007/978-3-642-20662-7_20

Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2012b. Hierarchical Hub Labelings for Shortest Paths. In *Algorithms ESA 2012*, Leah Epstein and Paolo Ferragina (Eds.). Lecture Notes in Computer Science, Vol. 7501. Springer Berlin Heidelberg, 24–35. DOI:http://dx.doi.org/10.1007/978-3-642-33090-2_4

Takuya Akiba, Yoichi Iwata, Ken-ichi Kawarabayashi, and Yuki Kawata. 2014. Fast Shortest-path Distance Queries on Road Networks by Pruned Highway Labeling. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2014, Portland, Oregon, USA, January 5, 2014*. 147–154. DOI:http://dx.doi.org/10.1137/1.9781611973198.14

Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, USA*. 349–360. DOI:http://dx.doi.org/10.1145/2463676.2465315

Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2015. Pruned Landmark Labeling [Online]. https://github.com/iwiwi/pruned-landmark-labeling. (2015).

Rka Albert, Hawoong Jeong, and Albert-Lszl Barabsi. 1999. The diameter of the world wide web. *CoRR* cond-mat/9907038 (1999). http://dblp.uni-trier.de/db/journals/corr/corr9907.html\#cond-mat-9907038

David A. Bader, Henning Meyerhenke, Peter Sanders, Christian Schulz, Andrea Kappes, and Dorothea Wagner. 2014. Benchmarking for Graph Clustering and Partitioning. In *Encyclopedia of Social Network Analysis and Mining*, Reda Alhajj and Jon Rokne (Eds.). Springer New York, 73–82. DOI:http://dx.doi.org/10.1007/978-1-4614-6170-8_23

Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. 2015. Route Planning in Transportation Networks. *CoRR* abs/1504.05140 (2015). http://arxiv.org/abs/1504.05140

Felix Borutta, Mario A. Nascimento, Johannes Niedermayer, and Peer Kröger. 2014. Monochromatic RkNN Queries in Time-dependent Road Networks. In *Proceedings of the Third ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems (MobiGIS '14)*. ACM, New York, NY, USA, 26–33. DOI:http://dx.doi.org/10.1145/2675316.2675317

Muhammad Aamir Cheema, Wenjie Zhang, Xuemin Lin, Ying Zhang, and Xuefei Li. 2012. Continuous Reverse K Nearest Neighbors Queries in Euclidean Space and in Spatial Networks. *The VLDB Journal* 21, 1 (Feb. 2012), 69–95. DOI:http://dx.doi.org/10.1007/s00778-011-0235-9

Eunjoon Cho, Seth A. Myers, and Jure Leskovec. 2011. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*. 1082–1090. DOI:http://dx.doi.org/10.1145/2020408.2020579

Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2002. Reachability and Distance Queries via 2-hop Labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 937–946. http://dl.acm.org/citation.cfm?id=545381.545503

Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. 2015. Public Transit Labeling. In *Experimental Algorithms*, Evripidis Bampis (Ed.). Lecture Notes in Computer Science, Vol. 9125. Springer International Publishing, 273–285. DOI:http://dx.doi.org/10.1007/978-3-319-20086-6_21

Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. 2011a. PHAST: Hardware-Accelerated Shortest Path Trees. In *25th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011, Anchorage, Alaska, USA, 16-20 May, 2011 - Conference Proceedings*. 921–931. DOI:http://dx.doi.org/10.1109/IPDPS.2011.89

Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. 2013a. PHAST: Hardware-accelerated shortest path trees. *J. Parallel Distrib. Comput.* 73, 7 (2013), 940–952. DOI:http://dx.doi.org/10.1016/j.jpdc.2012.02.007

Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. 2014. Robust Distance Queries on Massive Networks. In *Algorithms - ESA 2014*, AndreasS. Schulz and Dorothea Wagner (Eds.). Lecture Notes in Computer Science, Vol. 8737. Springer Berlin Heidelberg, 321–333. DOI:http://dx.doi.org/10.1007/978-3-662-44777-2_27

Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2011b. Faster Batched Shortest Paths in Road Networks. In *ATMOS 2011 - 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems, Saarbrücken, Germany, September 8, 2011*. 52–63. DOI:http://dx.doi.org/10.4230/OASIcs.ATMOS.2011.52

Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. 2013b. Hub Label Compression. In *Experimental Algorithms*, Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela (Eds.). Lecture Notes in Computer Science, Vol. 7933. Springer Berlin Heidelberg, 18–29. DOI:http://dx.doi.org/10.1007/978-3-642-38527-8_4

Daniel Delling and Renato F. Werneck. 2015. Customizable Point-of-Interest Queries in Road Networks. *IEEE Trans. Knowl. Data Eng.* 27, 3 (2015), 686–698. DOI:http://dx.doi.org/10.1109/TKDE.2014.2345386

Alexandros Efentakis. 2016. Scalable Public Transportation Queries on the Database. In *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-18, 2016.* 527–538. DOI:http://dx.doi.org/10.5441/002/edbt.2016.50

Alexandros Efentakis, Christodoulos Efstathiades, and Dieter Pfoser. 2015a. COLD. Revisiting Hub Labels on the Database for Large-Scale Graphs. In *Advances in Spatial and Temporal Databases*, Christophe Claramunt, Markus Schneider, Raymond Chi-Wing Wong, Li Xiong, Woong-Kee Loh, Cyrus Shahabi, and Ki-Joune Li (Eds.). Lecture Notes in Computer Science, Vol. 9239. Springer International Publishing, 22–39. DOI:http://dx.doi.org/10.1007/978-3-319-22363-6_2

Alexandros Efentakis and Dieter Pfoser. 2014. GRASP. Extending Graph Separators for the Single-Source Shortest-Path Problem. In *Algorithms - ESA 2014*, Andreas S. Schulz and Dorothea Wagner (Eds.). Lecture Notes in Computer Science, Vol. 8737. Springer Berlin Heidelberg, 358–370. DOI:http://dx.doi.org/10.1007/978-3-662-44777-2_30

Alexandros Efentakis, Dieter Pfoser, and Yannis Vassiliou. 2015b. SALT. A Unified Framework for All Shortest-Path Query Variants on Road Networks. In *Experimental Algorithms*, Evripidis Bampis (Ed.). Lecture Notes in Computer Science, Vol. 9125. Springer International Publishing, 298–311. DOI:http://dx.doi.org/10.1007/978-3-319-20086-6_23

Fletcher Foti, Paul Waddell, and Dennis Luxen. 2012. A Generalized Computational Framework for Accessibility: From the Pedestrian to the Metropolitan Scale. In *Proceedings of the 4th Transportation Research Board Conference on Innovations in Travel Modeling (ITM), Tampa, Florida, USA*.

Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. 2001. Distance Labeling in Graphs. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '01)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 210–219. http://dl.acm.org/citation.cfm?id=365411.365447

Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. 2004. Distance Labeling in Graphs. *J. Algorithms* 53, 1 (Oct. 2004), 85–112. DOI:http://dx.doi.org/10.1016/j.jalgor.2004.05.002

Robert Geisberger. 2011. *Advanced Route Planning in Transportation Networks*. Ph.D. Dissertation. Institute of Theoretical Informatics, Algorithmics II - Karlsruhe Institute of Technology (KIT).

Robert Geisberger, Peter Sanders, and Dominik Schultes. 2008a. Better Approximation of Betweenness Centrality.. In *ALENEX*, J. Ian Munro and Dorothea Wagner (Eds.). SIAM, 90–100. http://dblp.uni-trier.de/db/conf/alenex/alenex2008.html\#GeisbergerSS08

Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. 2008b. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings*. 319–333. DOI:http://dx.doi.org/10.1007/978-3-540-68552-4_24

Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. 2012. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science* 46, 3 (2012), 388–404. DOI:http://dx.doi.org/10.1287/trsc.1110.0401

Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Yanyan Xu. 2014. Hop Doubling Label Indexing for Point-to-Point Distance Querying on Scale-Free Networks. *PVLDB* 7, 12 (2014), 1203–1214. http://www.vldb.org/pvldb/vol7/p1203-jiang.pdf

Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. 2007. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007*. DOI:http://dx.doi.org/10.1137/1.9781611972870.4

Flip Korn and S. Muthukrishnan. 2000. Influence Sets Based on Reverse Nearest Neighbor Queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD '00)*. ACM, New York, NY, USA, 201–212. DOI:http://dx.doi.org/10.1145/342009.335415

Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data. (June 2014).

Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. 2009. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics* 6, 1 (2009), 29–123. DOI:http://dx.doi.org/10.1080/15427951.2009.10129177

Ling Liu and M. Tamer Özsu (Eds.). 2009. *Encyclopedia of Database Systems*. Springer US.

Julian J. McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States.* 548–556. http://papers.nips.cc/paper/4532-learning-to-discover-social-circles-in-ego-networks

Kurt Mehlhorn and Peter Sanders. 2008. *Algorithms and Data Structures: The Basic Toolbox*. Springer, Berlin.

Maytham Safar, Dariush Ibrahimi, and David Taniar. 2009. Voronoi-based reverse nearest neighbor query processing on spatial networks. *Multimedia Systems* 15, 5 (2009), 295–308. DOI:http://dx.doi.org/10.1007/s00530-009-0167-z

Sibo Wang, Wenqing Lin, Yi Yang, Xiaokui Xiao, and Shuigeng Zhou. 2015. Efficient Route Planning on Public Transportation Networks: A Labelling Approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 967–982. DOI:http://dx.doi.org/10.1145/2723372.2749456

Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*. 745–754. DOI:http://dx.doi.org/10.1109/ICDM.2012.138

Man Lung Yiu, D. Papadias, Nikos Mamoulis, and Yufei Tao. 2006. Reverse nearest neighbors in large graphs. *Knowledge and Data Engineering, IEEE Transactions on* 18, 4 (April 2006), 540–553. DOI:http://dx.doi.org/10.1109/TKDE.2006.1599391