COLD. Revisiting Hub Labels on the Database for Large-Scale Graphs

Alexandros Efentakis[1(✉)], Christodoulos Efstathiades[1,2], and Dieter Pfoser[3]

[1] Research Center "Athena", Marousi, Greece
efentakis@imis.athena-innovation.gr
[2] Knowledge and Database Systems Laboratory,
National Technical University of Athens, Zografou, Greece
cefstathiades@dblab.ece.ntua.gr
[3] Department of Geography and GeoInformation Science,
George Mason University, Fairfax, USA
dpfoser@gmu.edu

Abstract. Shortest-path computation is a well-studied problem in algorithmic theory. An aspect that has only recently attracted attention is the use of databases in combination with graph algorithms to compute distance queries on large graphs. To this end, we propose a novel, efficient, pure-SQL framework for answering exact distance queries on large-scale graphs, implemented entirely on an open-source database system. Our COLD framework (COmpressed Labels on the Database) may answer multiple distance queries (vertex-to-vertex, one-to-many, kNN, RkNN) not handled by previous methods, rendering it a complete solution for a variety of practical applications in large-scale graphs. Experimental results will show that COLD outperforms previous approaches (including popular graph databases) in terms of query time and efficiency, while requiring significantly less storage space than previous methods.

1 Introduction

Answering distance queries on graphs is one of the most well-studied problems on algorithmic theory, mainly due to its wide range of applications. Although a lot of recent research focused exclusively on transportation networks (cf. [9] for the most recent overview) the emergence of social networks has generated massive unweighted graphs of interconnected entities. On such networks, the distance between two vertices is an indication of the closeness of their entities, i.e., for finding users closely related to each other or extracting information about existing communities within the social media users. Although we may always use a breadth first search (BFS) to calculate the distance between any two vertices on such graphs, that approach cannot facilitate fast-enough queries on main memory or be easily adapted to secondary storage solutions.

Moreover, most of the excellent preprocessing techniques available for road networks cannot be adapted to large-scale graphs, such as social or collaboration networks. So far, the most promising approach for this type of graphs builds on the 2-hop labeling or hub labeling (HL) algorithm [12,23], in which we store a

C. Claramunt et al. (Eds.): SSTD 2015, LNCS 9239, pp. 22–39, 2015.
DOI: 10.1007/978-3-319-22363-6_2

two-part label $L(v)$ for every vertex v: a forward label $L_f(v)$ and a backward label $L_b(v)$. These labels are then used to very fast answer vertex-to-vertex shortest-path queries. This technique has been adapted successfully to road networks [2–4,15] and quite recently has also been extended to undirected, unweighted graphs [5,14,25]. The HL method has also been applied for one-to-many, many-to-many and kNN queries in road networks [16,17] and kNN and RkNN queries in the context of social networks in [21].

Although hub labeling is an extremely efficient shortest-path computation method using main memory, there are very few works that try to replicate those algorithms for secondary storage. HLDB [18] stores the calculated hub labels for continental road networks in a commercial database system and translates the typical HL distance query between two vertices to plain SQL commands. Moreover, it showed how to efficiently answer kNN queries and k-best via points, again by means of SQL queries. Recently, HopDB [25] proposed a customized solution that utilizes secondary storage also during preprocessing. Unfortunately, both methods have their shortcomings. HLDB has only been tested on road networks and consequently small labels sizes (<100). Its speed would seriously degrade for large-scale graphs due to the much larger label size. HopDB answers only vertex-to-vertex queries and is a customized C++ solution that cannot be used with existing database systems and, hence, has limited practical applicability.

This work presents a database framework that may service multiple distance queries on massive large-scale graphs. Our pure-SQL *COLD* framework (COmpressed Labels on the Database) can answer multiple exact distance queries (point-to-point, kNN) in addition to RkNN and *one-to-many* queries not handled by previous methods, rendering it a complete database solution for a variety of practical massive, large-scale graph problems. Our extensive experimentation will show that COLD outperforms previous solutions, including specialized graph databases, on all aspects (including query performance and memory requirements), while servicing a larger variety of distance queries. In addition, COLD is implemented using a popular, open-source database engine with no third-party extensions and, thus, our results are easily reproducible by anyone.

The outline of the remainder of this work is as follows. Section 2 presents related work. Section 3 describes the novel COLD framework and its implementation details. Experiments establishing the benefits of COLD are provided in Sect. 4. Finally, Sect. 5 gives conclusions and directions for future work.

2 Related Work

Throughout this work we use undirected, unweighted graphs $G(V, E)$ (where V represents vertices and E arcs). A k-Nearest Neighbor (kNN) query seeks the k-nearest neighbors to an input vertex q. The RkNN query (also referred as the monochromatic RkNN query), given a query point q and a set of objects P, retrieves all the objects that have q as one of their k-nearest neighbors according to a given distance function $dist()$. In graph networks, $dist(s, t)$ corresponds to the minimum network distance between the two objects. Formally RkNN$(q) = \{p \in P : dist(p, q)$

$\leq dist(p, p_k)\}$ where p_k is the k-Nearest Neighbor (kNN) of p. Throughout this work, we assume that objects are located on vertices and we always refer to *snapshot* kNN and RkNN queries on graphs, i.e., objects are not moving. Also, similarly to previous works, the term *object density* D refers to the ratio $|P|/|V|$, where P is a set of objects in the graph and $|V|$ is the total number of vertices. Although, there is extensive literature focusing on kNN and RkNN queries in Euclidean space, since our work focuses on graphs we will only describe related work focusing on the latter.

Regarding road networks and kNN queries, G-tree [33] is a balanced tree structure, constructed by recursively partitioning the road network into subnetworks. Unfortunately, this method cannot scale for continental road networks, since it requires several hours for its preprocessing. Moreover, it requires a *target selection phase* to index which tree-nodes contain objects (requiring few seconds) and thus, cannot be used for moving objects. Recently, the work of [17] expanded the graph-separators CRP algorithm of [13] to handle kNN queries on road networks. Unfortunately, (i) CRP also requires a target selection phase and thus, cannot be applied to moving objects and (ii) it may only perform well for objects near the query location. Hence, this solution is also not optimal. The latest work for kNN queries on road networks is the SALT framework [22] which may be used to answer multiple distance queries on road networks, including *vertex-to-vertex* (v2v), single source (one-to-all, range, one-to-many) and kNN queries. This work expands the graph-separators GRASP algorithms of [20] and the ALT-SIMD adaptation [19] of the ALT algorithm and offers very fast preprocessing time and excellent query times. For kNN queries, SALT does not require a target selection phase and hence it may be used for either static or moving objects.

For RkNN queries on road networks, the work of [30] 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 ($110\,K$ arcs) and all precomputed information is stored in a database. Despite the fact that the preprocessing stage for computing the Network Voronoi cells is quite costly, the queries' executions times range from $1.5\,s$ for $D = 0.05$ and $k = 1$, up to $32\,s$ for $k = 20$, rendering this solution impractical for real-time scenarios. Up until recently, the only work dealing with other graph classes (besides road networks) is [32], although it has only been tested on sparse networks, e.g., road networks, grid networks (max degree 10), p2p graphs (avg degree 4) and a very small, sparse co-authorship graph ($4\,K$ nodes). In this work, the conducted experiments for values of $k > 1$ refer only to road networks, therefore the scalability of this work for denser graphs and larger values of k is questionable. Recently, Borutta et al. [10] extended this work for time-dependent road networks, but presented results were not very encouraging. The larger road network tested had $50\,k$ nodes (queries require more than $1\,s$ for $k = 1$) and for a network of $10\,k$ nodes and $k = 8$, RkNN queries take more than $0.3\,s$ (without even adding the I/O cost). In a nutshell, all existing contributions and methods have not been tested on dense, large-scale graphs, cannot scale for increasing k values and their performance highly depends on the object density D.

Our work builds upon the 2-hop labeling or Hub Labeling (HL) algorithm of [12,23] in which, preprocessing stores at every vertex v a forward $L_f(v)$ and a backward label $L_b(v)$. The forward label $L_f(v)$ is a sequence of pairs $(u, dist(v,u))$, with $u \in V$. Likewise, the backward label $L_b(v)$ contains pairs $(w, dist(w,v))$. Vertices u and w are denoted as the *hubs of* v. The generated labels conform to *the cover property*, i.e., for any s and t, the set $L_f(s) \cap L_b(t)$ must contain at least one hub that is on the shortest $s-t$ path. For undirected graphs $L_b(v) = L_f(v)$. To find the network distance $dist(s,t)$ between two vertices s and t, a HL query must find the hub $v \in L_f(s) \cap L_b(t)$ that minimizes the sum $dist(s,v) + dist(v,t)$. By sorting the pairs in each label by hub, this takes linear time by employing a coordinated sweep over both labels. The HL technique has been successfully adapted for road networks in [2–4,15]. In the case of large-scale graphs, the Pruned Landmark Labeling (PLL) algorithm of [5] *produces a minimal labeling for a specified vertex ordering*. In this work, vertices are ordered by degree, whereas the work of [14] improves the suggested vertex ordering and the storage of the hub labels for maximum compression. The HL method has also been used for one-to-many, many-to-many and kNN queries on road networks in [16] and [17] respectively. Our latest work [21] proposed *ReHub*, a novel main-memory algorithm that extends the Hub Labeling approach to efficiently handle RkNN queries. The main advantage of the *ReHub* algorithm is the separation between its costlier offline phase, which runs only once for a specific set of objects and a very fast online phase which depends on the query vertex q. Still, even the costlier offline phase hardly needs more than $1\,s$, whereas the online phase requires usually less than $1\,ms$, making *ReHub* the only RkNN algorithm fast enough for real-time applications and big, large-scale graphs.

Regarding secondary-storage solutions, Jiang et al. [25] propose their HopDB algorithm that suggest an efficient HL index construction when the given graphs and the corresponding index are too big to fit into main memory. The work of [1] introduced the HLDB system, which answers distance and kNN queries in road networks entirely within a database by storing the hub labels in database tables and translating the corresponding HL queries to SQL commands. Throughout this work, we will compare our proposed COLD framework to HLDB, since to the best of our knowledge, it is the only framework that may answer exact distance queries entirely within a database. Moreover, within the COLD framework we also adapt our *ReHub* main-memory algorithm into a database context, so that its online phase may be translated to fast and optimized SQL queries.

3 Contribution

This section presents the *COLD* (COmpressed Labels on the Database) database framework. COLD can answer multiple distance queries (vertex-to-vertex, kNN, RkNN and *one-to-many*) for large-scale graphs using SQL commands. Since COLD builds on HLDB [1] and *ReHub* [21], we will follow the notation and running example presented there, for highlighting the necessary concepts and

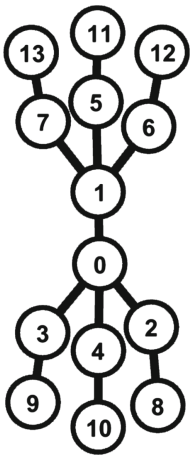


Fig. 1. A sample Graph G

Table 1. The created hub-labels for the sample graph G of Fig. 1

Vertex	Hub Labels (h,d)
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)

challenges for adapting those previous works, (i) in the context of large-scale graphs for [1] and (ii) within the boundaries of a relational database management system (RDBMS) for [21]. To this end, we chose PostgreSQL [29] for our implementation, given that it is a popular, open-source RDBMS. Although we use some PostgreSQL-specific data-types and SQL extensions, we do not use any third-party extensions but only features included in its standard installation.

3.1 Implementation

The COLD framework assumes that we have a correct hub labeling (HL) framework that generates hub-labels for the undirected, unweighted graphs we wish to query. Although COLD will work with any correct HL algorithm, in this work we use the [6] implementation of the PLL algorithm of [5] to generate the necessary labels. To highlight the results of this process, the labels for the undirected, unweighted graph G of Fig. 1 are shown in Table 1. Throughout this work, we will refer to those labels as the *forward labels*. The forward label $L(v)$ for a vertex v is an array of pairs $(u, dist(v,u)$ sorted by hub u. Since our work also focuses on snapshot kNN and RkNN queries, there also some objects $P \in V$ that do not change over time. For our specific running example we assume that $P = \{4, 10, 12\}$ and thus, we highlight the respective entries of Table 1.

Vertex-to-Vertex (v2v) Queries. To find the network distance $dist(s,t)$ between two vertices s and t, a HL query must find the hub $v \in L(s) \cap L(t)$

Table 2. The *forward* table used in HLDB for the sample graph G

v	hub	dist
...	...	...
2	0	1
2	2	0
...	...	...
7	0	2
7	1	1
7	7	0
...	...	...

Table 3. The *forwcold* table used for COLD for the sample graph G

v	hubs	dists
...	...	...
2	$\{0, 2\}$	$\{1, 0\}$
...	...	...
7	$\{0, 1, 7\}$	$\{2, 1, 0\}$
...	...	...

Code 1.1. V2v query for HLDB

```
SELECT MIN(n1.dist+n2.dist)
FROM forward n1, forward n2
WHERE n1.v = s
AND n2.v = t
AND n1.hub = n2.hub;
```

Code 1.2. V2v query for COLD

```
SELECT MIN(n1.d+n2.d) FROM
/* Expand hubs, dists arrays */
(SELECT UNNEST(hubs) AS hub,
UNNEST(dists) AS d
FROM forwcold WHERE v = s) n1,
(SELECT UNNEST(hubs) AS hub,
UNNEST(dists) AS d
FROM forwcold WHERE v = t) n2
WHERE n1.hub=n2.hub;
```

that minimizes the sum $dist(s, v) + dist(v, t)$. For our sample graph G, the minimum distance between e.g., vertices 2 and 7 is $d(2, 7) = 3$, using the hub 0. To translate this HL query into SQL commands, in HLDB [1] forward labels are stored in a database table denoted *forward* where the labels of vertex v are stored as triples of the form $(v, hub, dist(v, hub))$ (see Table 2). The table *forward* has the combination of (v, hub) as the primary key and is clustered according to those columns, so that "*all rows corresponding to the same label are stored together to minimize random accesses to the database*" [1]. Then we can find the distances between any two vertices s and t by the SQL query of Code 1.1.

Although the HLDB vertex-to-vertex (v2v) query is very simple, there is one major drawback. For such a query, HLDB has to fetch from secondary storage the subset of $|L(s)| + |L(t)|$ rows with common hubs. Although this is practical for road networks where the forward labels have less than 100 hubs per vertex [3], it cannot scale for large-scale graphs where the forward labels have thousand of hubs per vertex. Moreover, on such graphs the *forward* DB table and the corresponding primary key index will become too large, which is also an important disadvantage. To this end, we take advantage of the fact that PostgreSQL features an array data type that allows columns of a DB table to be defined as variable-length arrays. Hence, in COLD we store hubs and distances for a vertex (both ordered by hub) as arrays in two separate columns (i.e., hubs and dists) in a single row. The resulting *forwcold* compressed DB table is shown

in Table 3. This approach not only emulates exactly how labels are stored on main-memory for fast v2v queries but also has considerable advantages: (i) The *forwcold* DB table has exactly $|V|$ rows (ii) The *forwcold* DB table has the column v as primary key without needing a composite key. This alone facilitates faster queries. Moreover the size of the corresponding index will be much smaller. In fact, our experimentation will show that the primary-key index for *forwcold* may be $> 4,400\times$ smaller than the index size of HLDB. (iii) For a v2v query, COLD needs to access exactly two rows, regardless of the sizes of $|L(s)|$ and $|L(t)|$. This way, we efficiently minimized the secondary-storage utilization, even working inside a database. The resulting SQL query for COLD is shown in Code 1.2. There we exploit the fact that PostgreSQL "*guarantees that parallel unnesting*" for hubs and distances for each nested query "*will be in sync*", i.e., each pair (hub, dist) is expanded correctly since for the same v the respective arrays have the same number of elements[1].

Additional Queries Overview. For answering more complex (kNN, RkNN and *one-to-many*) distance queries on a HL framework for a set of objects P, we need to build some additional data structures from the forward labels (for undirected graphs). Then to answer the respective query we only need to combine the forward labels $L(q)$ of query vertex q, with the respective data structure explained in the following. Those data structures are summarized in Table 4.

Table 4. Necessary data structures for the sample graph G, $P = \{4, 10, 12\}$ and *one-to-many*, kNN and RkNN queries

Hub	Backward Labels (to-many) [16]	kNN Backward Labels (k=2) [1]	RkNN Backward Labels (k=1) [21]	Obj	kNN Result (k=1) (Obj., dist) [21]
0	(4,1), (10,2), (12,3)	(4,1), (10,2)	(4,1), (12,3)	**4**	(10,**1**)
1	(12,2)	(12,2)	(12,2)		
4	(4,0), (10,1)	(4,0),(10,1)	(4,0), (10,1)	**10**	(4,**1**)
6	(12,1)	(12,1)	(12,1)		
10	(10,0)	(10,0)	(10,0)	**12**	(4,**4**)
12	(12,0)	(12,0)	(12,0)		

For answering *one-to-many* queries, i.e., calculate distances between a source vertex q and all objects in P, we need to build the *backward labels-to-many* by basically ordering the forward labels of the objects by hub [16] and then by distance for the same hub. For kNN queries we only need to keep at most the k-best pairs (of smallest distances) per hub from the backward labels-to-many to create the kNN *backward labels* [1]. In our specific example, the kNN backward labels for $k = 2$ and hub 0, do not contain the pair $(12, 3)$. Finally, for RkNN queries, we must first calculate the kNN *Results* (i.e., the NN of the object 4 is the object 10 with distance 1) and then we build the RkNN backward labels, based

[1] http://stackoverflow.com/a/23838131.

on the observation that *"we need to access those pairs from the backward labels-to-many to a specific object, if and only if those distances are equal or smaller than the distance of the* kNN *of this object"* [21]. In our specific example, the RkNN backward labels for $k = 1$ and hub 0, do not contain the pair (10,2) since the NN of object 10 (the object 4) is within distance 1. Although for our small graph the differences between the individual data structures seem minimal, for larger graphs those differences become very prominent. This was also showcased by the theoretical analysis provided in [21] which showed that backward labels-to-many will have on average $D \cdot |HL|$ pairs, the kNN backward labels have at most $k \cdot |V|$ pairs and the RkNN backward labels have on average $\varepsilon \cdot D \cdot |HL|$ pairs where ε may be < 0.01 for specific datasets and experimental settings. Moreover, Efentakis et al. [21] have shown how these additional data structures may be constructed from the forward labels in main-memory, requiring less than few seconds, even for the larger tested datasets.

***k*NN Queries.** To translate the HL kNN query into SQL, HLDB stores kNN *backward labels* in a separate DB table denoted *knntab* that stores triples of the form $(hub, dist, obj)$ (see Table 5). The respective table *knntab* has the combination of $(hub, dist, obj)$ as a composite primary key and is clustered according to those columns. Note that in HLDB, we cannot use the combination of $(hub, dist)$ as a primary key, because especially in large scale graphs we will have a lot of distance ties even for k-entries for the same hub. Then we can can answer a kNN query from vertex q by the SQL query of Code 1.3. Again, the kNN HLDB query has the same drawbacks as before, i.e., it has to retrieve $|L(q)|$ rows from *forward* and $k \cdot |L(q)|$ rows from *knntab* tables, for a total of $(k+1) \cdot |L(q)|$ rows retrieved from secondary storage. Moreover in a database, it makes sense to create one large *knntab* table for the maximum value $kmax$ of k (e.g., for $k = 16$) that may be serviced by the DB framework and that same table will be used for all kNN queries up to $k = kmax$. In that case, the HLDB framework will have to retrieve $(kmax + 1) \cdot |L(q)|$ rows for every kNN query regardless of the value of k.

To remedy the HLDB drawbacks, COLD creates the *knncold* DB table (Table 6) that has the columns $(hub, dist, objs)$, whereas objects are grouped and ordered per hub and distance (the column *objs* is an array). Although for our sample graph G, the DB tables *knntab* and *knncold* seem identical, COLD's method offers several advantages: (i) We can now use the combination of $(hub, dist)$ as a primary key, which makes the respective index significantly

Table 5. The *knntab* table used in HLDB for the sample graph G, $k = 2$ and $P = \{4, 10, 12\}$

hub	dist	obj
0	1	4
0	2	10
1	2	12
...	...	...

Table 6. The *knntab* table used in COLD for the sample graph G, $k = 2$ and $P = \{4, 10, 12\}$

hub	dist	objs
0	1	{4}
0	2	{10}
1	2	{12}
...	...	...

smaller and faster and (ii) In case of many distance ties (common to large-scale graphs) and one large *knncold* DB table that services all kNN queries for values of k up to the maximum value $kmax$, we only need to fetch the first k-*objs* entries (i.e., `objs[1:k]`) per hub and dist, which makes the later sorting faster (see Code 1.4).

Code 1.3. kNN query for HLDB

```
SELECT MIN(n1.dist+n2.dist),
n2.obj FROM
forward n1, knntab n2
WHERE n1.v = q
AND n1.hub = n2.hub
GROUP BY n2.obj
ORDER BY MIN(n1.dist+n2.dist)
LIMIT k;
```

Code 1.4. kNN query for COLD

```
SELECT MIN(n1.d+n2.dist),
UNNEST(objs) AS obj FROM
(SELECT UNNEST(hubs) AS hub,
UNNEST(dists) AS d
FROM forwcold WHERE v = q) n1,
/* k-entries per hub,dist */
(SELECT hub, dist,objs[1:k]
FROM knncold) n2
WHERE n1.hub=n2.hub
GROUP BY obj
ORDER BY MIN(n1.d+n2.dist)
LIMIT k;
```

One-to-Many Queries. Similar to how COLD handles kNN queries, for one-to-many queries, COLD stores the *backward labels-to-many* in a new *objcold* DB table that has an identical format to *knncold*, i.e., it has three columns $(hub, dist, objs)$ whereas objects are grouped and ordered per hub and distance. *Objcold* also uses the combination of $(hub, dist)$ as a primary key. The resulting *one-to-many* query (Code 1.5) is quite similar to COLD's kNN query, but (i) it operates on the larger *objcold* DB table (ii) It does not have the `ORDER BY ... LIMIT k` clause and (iii) We use the entire *objs* array per hub and distance instead of `objs[1:k]`. Note that HLDB cannot possibly support such queries because it will need to retrieve on average $|L(q)|$ rows from the *forward* table and a total of $|L(q)| \cdot D \cdot (|HL|/|V|)$ [21] rows from the corresponding *objlab* table, which will be prohibitively slow for very large datasets.

Table 7. The *knnres* table used in COLD for RkNN queries, the sample graph G, $k = 1$ and $P = \{4, 10, 12\}$

obj	dists	objs
4	{1}	{10}
10	{1}	{4}
12	{4}	{4}

R*k*NN Queries. For RkNN queries, COLD stores the RkNN backward labels in a separate *revcold* DB table that has an identical format to previous *knncold* and *objcold* DB tables, i.e., three columns $(hub, dist, objs)$ where objects are grouped and ordered per hub and distance and the combination of $(hub, dist)$ used as a primary key. COLD also stores the kNN *Results*, i.e., the kNN of all objects in another *knnres* DB table that has the format $(obj, dists, objs,)$ where obj is the primary key and *objs* and *dists* are arrays (both ordered by distance) (Table 7). Therefore the kNN of object p is the `objs[k]` within distance `dists[k]` of the respective row for p. Again it makes sense to build a *knnres* DB table for a max value of $kmax$ that may service RkNN queries for varying values of k. As a result, during the

Code 1.5. *One-to-many* COLD query

```
SELECT MIN(n1.d+n2.dist),
UNNEST(objs) AS obj FROM
(SELECT UNNEST(hubs) AS hub,
UNNEST(dists) AS d
FROM forwcold
WHERE v = q) n1,
objcold n2
WHERE n1.hub=n2.hub
GROUP BY obj;
```

Code 1.6. RkNN query for COLD

```
SELECT n3.id2,n3.dist FROM
/* n3 subquery is a modified
 one-many-query to revcold */
(SELECT MIN(n1.d+n2.dist) AS d3,
UNNEST(objs) AS obj FROM
(SELECT UNNEST(hubs) AS hub,
UNNEST(dists) AS d
FROM forwcold WHERE v = q) n1,
revcold n2
WHERE n1.hub=n2.hub
GROUP BY obj
ORDER BY obj,MIN(n1.d+n2.dist)
) n3,
/* Join with knnres table */
(SELECT obj, dists[k] AS dist
FROM knnres) n4
WHERE n3.obj=n4.obj
AND n3.d3<=n4.dist
ORDER BY n3.obj;
```

RkNN COLD query, we will have to use an additional `JOIN` between the *revcold* and *knnres* DB tables. The resulting query is shown in Code 1.6.

We see that even the more complex RkNN query in COLD requires just a few lines of SQL code that will work on any recent PostgreSQL version without any need of third-party extensions or specialized index structures. In fact, all DB tables in COLD, use only standard B-tree primary key indexes, without any modifications. To satisfy this strict requirement, we effectively compressed the index sizes by grouping rows per vertex (*forcold* table) or object (*knnres* table), or by hub and distance for *knncold*, *objcold* and *rknncold*. And although we used PostgreSQL specific SQL extensions for expanding the stored arrays, latest versions of other databases (e.g., Oracle) support similar array data-types. Hence, it would be quite easy to port COLD to other database vendors as well.

This section detailed the COLD framework in terms of design and implementation. COLD can answer multiple distance queries (v2v, kNN, RkNN and one-to-many) based on data stored in an off-the-shelf relational database. We also presented the actual queries used and the way the necessary data structures are stored within the database, so that our results are easily reproducible. Although we focused on query efficiency, it is important to note that once we create the *forcold* table, all the adjoining DB tables within COLD may also be created using SQL commands (resulting queries were omitted due to space restrictions). This fact also shows that COLD is truly a pure-SQL framework for servicing multiple distance queries on large-scale graphs. We also provided the necessary theoretical details as to why the COLD framework will outperform existing solutions. This will be further quantified in the following section.

4 Experimental Evaluation

To assess the performance of COLD on various large-scale graphs, we conducted experiments on a workstation with a 4-core Intel i7-4771 processor clocked at 3.5G Hz and 32 Gb of RAM, running Ubuntu 14.04. We compare our COLD framework with a custom implementation of HLDB in PostgreSQL and with *Neo4j*, a well-known, popular graph database.

We use the same network graphs as our previous work of [21] that are taken from the Stanford Large Network Dataset Collection [26] and the 10th Dimacs Implementation Challenge website [8]. All graphs are undirected, unweighted and strongly connected. We used collaboration graphs (DBLP, Citeseer1, Citeseer2) [24], social networks (Facebook [28], Slashdot1 and Slashdot2 [27]), networks with ground-truth communities (Amazon, Youtube) [31], web graphs (Notre Dame) [7] and location-based social networks (Gowalla) [11]. The graphs' average degree is between 3 and 37 and the PLL algorithm creates $26 - 4,457$ labels per vertex, requiring $0.03 - 5,946\,s$ for the hub labels' construction (see Table 8).

Table 8. Networks graphs statistics

Graph	| V |	| E |	Avg degr	| HL | / | V |	PLL Preproc. Time (s)
Facebook	4,039	88,234	22	26	0.03
NotreDame	325,729	1,090,108	3	55	6
Gowalla	196,591	950,327	5	100	13
Youtube	1,134,890	2,987,624	3	167	123
Slashdot1	77,360	469,180	6	204	11
Slashdot2	82,168	504,230	6	216	13
Citeseer1	268,495	1,156,647	4	408	110
Amazon	334,863	925,872	3	689	230
DBLP	540,486	15,245,729	28	3,628	5,720
Citeseer2	434,102	16,036,720	37	4,457	5,946

COLD and HLDB were implemented in PostgeSQL 9.3.6, 64 bit with reasonable settings (8192 Mb *shared buffers*, 64 Mb *temp buffers*). We also used Neo4j Server v2.1.5. The Neo4j queries were formulated using *Cypher*, Neo4j's declarative query language and we report query times as they were returned by the server. Although Cypher may theoretically facilitate *one-to-many* queries (besides vertex-to-vertex), testing Neo4j with our datasets and the same number of target vertices we tested COLD with, resulted in a "`java.lang.StackOverflowError`". Providing the server with additional resources[2] had no positive effect and thus there are no results for *one-to-many* queries and Neo4j.

[2] http://neo4j.com/developer/guide-performance-tuning/.

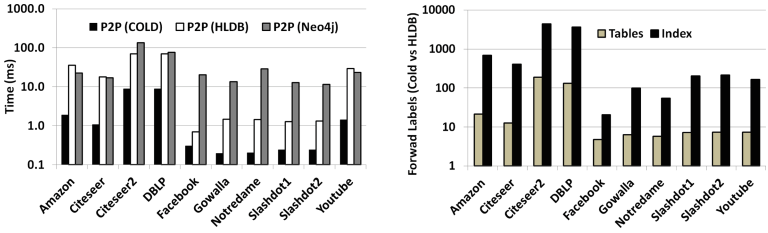
We conducted experiments belonging to four query types: (i) *vertex-to-vertex*, (ii) kNN, (iii) RkNN and (iv) *one-to-many*. For each experiment, we used 10,000 random start vertices, reporting the average running time. Before each experiment, we restart the PostgreSQL and Neo4j servers for clearing their internal cache and we also clear the operating system's cache for accurate benchmarking. All charts are plotted in logarithmic scale.

4.1 Performance on HDD

In our first round of experiments, we ran experiments on an HDD, specifically a SATA3 Seagate Barracude ST3000DM001 7200rpm with 64Mb cache.

Vertex-to-vertex. Fig. 2(a) shows results for vertex-to-vertex (v2v) queries for COLD, HLDB and Neo4j. Results show that COLD is consistently 2 - 20.7× faster than HLDB, with this difference amplified for the Citeseer1, Amazon and Youtube datasets (16.8, 19.1 and 20.7 respectively). Moreover, COLD is also 9 - 143× (for the *Gowalla* dataset) faster than Neo4j, which exhibits stable performance for all datasets, but is slower from both COLD and HLDB. For all datasets, COLD requires less than 9ms for answering v2v queries.

Figure 2(b) shows the difference in memory size for the DB tables *forcold* (COLD) and *forward* (HLDB) and their respective primary-key (PK) indexes. Results show that the size of the PK index in COLD is $3,600$ - $4,444$× smaller than for HLDB (for DBLP and Citeseer2 respectively). As expected, the difference in index sizes is almost identical to the $|HL|/|V|$ ratio, since *forcold* table has $|V|$ rows and *forward* has $|HL|$ rows. Likewise, the corresponding tables are 131 - 188× smaller for COLD. Thus, the techniques used for compressing the forward labels in COLD clearly achieve a considerable reduction in memory size, rendering our proposed framework suitable for real-world scenarios.



(a) Vertex-to-vertex query times

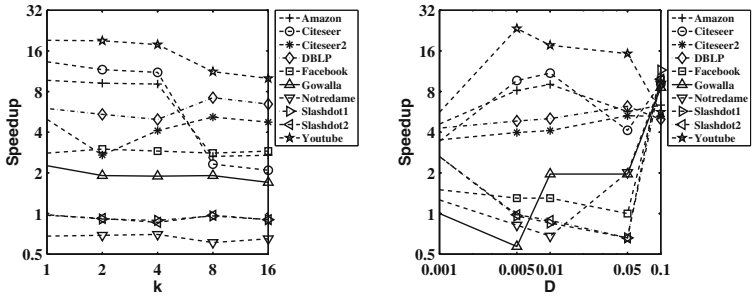
(b) Memory size's difference between COLD and HLDB

Fig. 2. Experiments on HDD for *vertex-to-vertex*

***k*NN.** Fig. 3(a) shows the speedup of COLD compared to HLDB in the case of kNN queries for $D = 0.01$ and $k = \{1, 2, 4, 8, 16\}$. As described in Sect. 3.1,

we have created two DB tables for each framework (COLD, HLDB), one for $kmax = 4$ and one for $kmax = 16$. Then the DB table for $kmax = 4$ is used for answering kNN queries for $k = 1$, $k = 2$ and $k = 4$ and the kNN table for $kmax = 16$ is used for answering kNN queries for $k = 8$ and $k = 16$. Results show that for $k = 1$, COLD is 5 - 19× faster for the five largest datasets (Amazon, Citeseer,Citeseer2, DBLP. Youtube) and although this speedup degrades for larger values of k, COLD remains consistently 2 - 10× faster even for $k = 16$. For the smaller datasets, performance between COLD and HLDB is quite similar, with COLD performing better on Facebook and Gowalla, while HLDB performs only marginally better for Slashdot1, Slashdot2 and Notredame. In all cases, COLD answers kNN queries for all datasets in less than $26ms$ even for $k = 16$.

In our second set of kNN experiments, we assess the performance of COLD vs HLDB for varying values of D. For each value for D, we have build separate versions of *knntab* (HLDB) and *knncold* (COLD) DB tables for $D \cdot |V|$ objects selected at random from each dataset and $kmax = 4$. Figure 3(b) shows results for $k = 4$ and $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$. Again, for the five largest datasets COLD is consistently 3.4 - 23.4× faster than HLDB, whereas even for the smaller datasets, COLD is consistently 8.6 - 11.5× faster than HLDB for the largest value of D (for $D = 0.1$). Moreover, COLD may answer kNN queries for $k = 4$ on all datasets and all values of D in less than $14ms$.

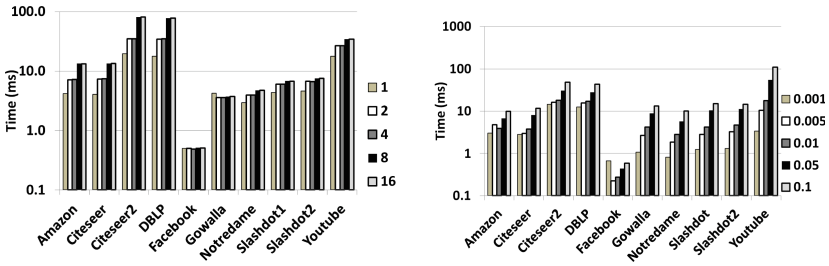


(a) kNN Speedup of COLD vs HLDB for $D = 0.01$ and varying values of k

(b) Speedup of COLD vs HLDB for $k = 4$ and varying values of D

Fig. 3. kNN Experiments on HDD for COLD and HLDB

R*k*NN. For RkNN experiments, we only report COLD's performance, since there is no other SQL framework that supports these queries. In out first experiment, we report the performance of COLD for $D = 0.01$ and $k = \{1, 2, 4, 8, 16\}$. For all those queries we have built one version of the *knnres* DB table for $kmax = 16$ (see Sect. 3.1) and 3 separate *revcold* tables for $kmax = \{1, 4, 16\}$. As expected, for RkNN queries and $k = 1$ we use the *revcold* table built for



(a) COLD RkNN query times for $D = 0.01$ and varying values of k

(b) COLD RkNN query times for $k = 1$ and varying values of D

Fig. 4. RkNN Experiments on HDD for COLD

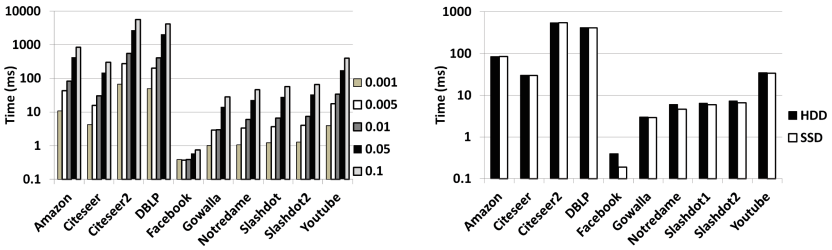
$kmax = 1$, for $k = 2$, $k = 4$ we use the *revcold* table built for $kmax = 4$ and for $k = 8$, $k = 16$ we use the *revcold* table built for $kmax = 16$. Figure 4(a) presents the results. In all cases, COLD provides excellent query times that are below 20 ms for $k = 1$ in all datasets and never exceed 82 ms even for $k = 16$.

In our second set of RkNN experiments, we assess the performance of COLD for varying values of D. Figure 4(b) presents results for $k = 1$ (as this is the typical case for RkNN queries) and $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$. Results show that although COLD's performance degrades for larger values of D, RkNN query times are below 49ms for all datasets and values of D, with the exception of Youtube and $D = 0.1$ (109.3 ms). Thus, COLD offers excellent and stable performance in RkNN queries for all all datasets and tested values of k and D.

One-to-Many. Again, COLD is the only SQL framework that supports *one-to-many queries*. Figure 5(a) presents the corresponding results for varying values of D ($D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$). COLD answers such queries in less than a second for all datasets and values of D, except the Citeseer2 and DBLP datasets (those with the highest $|HL|/|V|$ ratio) that require 5601 ms and 4170 ms respectively, for $D = 0.1$. For such high values of D, the *one-to-many* query reaches the complexity of an *one-to-all* query and as expected, it cannot be any faster on a secondary storage device. Note that even specialized graph databases like Neo4j cannot support this type of queries for more than a 1,000 target objects, whereas *COLD answers one-to-many queries to 110,000 target objects in the Youtube dataset in* 401 ms *with a simple SQL query.*

4.2 Performance on SSD

Having established the performance characteristics of COLD in the HDD, in our second round of experiments, we repeat some of the previous experiments, using a SSD to measure the impact of the secondary-storage device type to results. The SSD used is a SATA3 Crucial CT512MX100SSD1 MX100 512 GB 2.5".



(a) One-to-Many experiments for COLD varying values of D

(b) COLD One-to-Many HDD vs SSD

Fig. 5. One-to-many experiments for COLD

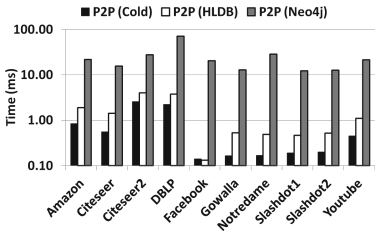


Fig. 6. SSD *vertex-to-vertex*

Vertex-to-vertex. Although the usage of SSD favors HLDB more than COLD (see Fig. 6), COLD is consistently 1.6 - 3.2× faster than HLDB (except Facebook, the smallest of datasets). The SSD has almost no impact on Neo4j and thus, COLD is now 11-171× faster than $Neo4j$ on all datasets. Note, than on the SSD, COLD requires less than $0.9\,ms$ for all datasets and v2v queries, except the Citeseer2 and DBLP datasets (those with the highest $|HL|/|V|$ ratio). But even then, vertex-to-vertex queries still require less than $2.6\,ms$ for COLD.

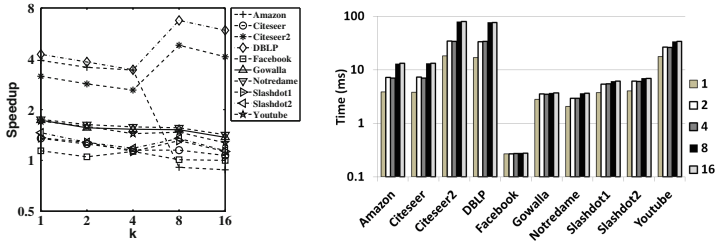
***k*NN.** Fig. 7(a) shows the performance speedup of COLD compared to HLDB in the case of kNN queries running on the SSD, for $D = 0.01$ and varying value of k. Again, although the SSD lowers the performance gap between COLD and HLDB, COLD is still faster on all datasets (except Facebook). In fact, COLD is 2.6 - 6.75× faster than HLDB for the high $|HL|/|V|$ ratio datasets (Citeseer2, HLDB) requiring less than $24.6\,ms$ even for $k = 16$.

R*k*NN. Fig. 7(b) presents the results of the RkNN query time performance on COLD for $D = 0.01$ and varying value of k. Results show that SSD usage accelerates COLD by only 20 % at most, which clearly demonstrates that COLD effectively minimized secondary storage utilization and thus adding a better secondary-storage medium provides minimal benefits for RkNN queries.

One-to-Many. Finally, Fig. 5(b) compares *one-to-many* queries on HDD and SSD for COLD. Again, the SSD usage accelerates COLD by only 2- 30 %, which further confirms the optimal secondary storage utilization of COLD.

4.3 Summary

Our experimentation has shown that our proposed COLD framework outperforms previous state-of-the-art HLDB in all performance benchmarks, including



(a) kNN Speedup of COLD vs HLDB for $D = 0.01$ and varying values of k

(b) COLD RkNN query times for $D = 0.01$ and varying values of k

Fig. 7. kNN and RkNN SSD performance

query performance, memory size and scalability. Using HDDs, COLD is $2--21\times$ faster for *vertex-to-vertex* queries and $5--19\times$ faster for kNN queries and the largest datasets. Using SSDs, COLD is $1.6--3.2\times$ faster than HLDB for *vertex-to-vertex* and up to $6.75\times$ faster for kNN queries. COLD also requires up to $4,444\times$ less storage space (indexes) and up to $188\times$ less storage space (DB tables) used for storing forward labels. Even specialized graph databases like Neo4j are outperformed by COLD, which is up to $143\times$ faster. Most importantly COLD may service additional (RkNN, one-to-many) queries, not handled by any other previous secondary-storage solutions, while providing excellent query times and optimal secondary-storage utilization even on standard hard drives.

5 Conclusions

This work presented COLD, a novel SQL framework for answering various exact distance queries for large-scale graphs on a database. Our results showed that COLD outperforms existing solutions (including specialized graph databases) on all levels, including query performance, secondary storage utilization and scalability. Moreover, COLD also answers RkNN and one-to-many queries, not handled by previous methods. This establishes COLD as a competitive database-driven framework for querying large-scale graphs. The paper gives the design and implementation details of COLD using a popular, open-source database system along with the actual SQL queries used in our implementation. This should allow for a simple replication of our results and encourage other researchers to expand the COLD framework towards handling more complex queries and test-cases.

Acknowledgements. This work was partially supported by EU (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund and the EU/Greece funded KRIPIS Action: MEDA Project. D. Pfoser's work was partially supported by the NGA NURI grant HM02101410004.

References

1. Abraham, I., Delling, D., Fiat, A., Goldberg, A.V., Werneck, R.F.: Hldb: Location-based services in databases. In: SIGSPATIAL GIS. ACM, November 2012
2. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: A hub-based labeling algorithm for shortest paths in road networks. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 230–241. Springer, Heidelberg (2011)
3. Abraham, I., Delling, D., Goldberg, A.V., Werneck, R.F.: Hierarchical hub labelings for shortest paths. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 24–35. Springer, Heidelberg (2012)
4. Akiba, T., Iwata, Y., Kawarabayashi, K., Kawata, Y.: 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, 5 January 2014, pp. 147–154 (2014)
5. Akiba, T., Iwata, Y., Yoshida, Y.: 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, pp. 349–360 (2013)
6. Akiba, T., Iwata, Y., Yoshida, Y.: Pruned landmark labeling (2015). https://github.com/iwiwi/pruned-landmark-labeling
7. Albert, R., Jeong, H., Barabási, A.-L.: The diameter of the world wide web, CoRR (1999). http://cond-mat/9907038
8. Bader, D.A., Meyerhenke, H., Sanders, P., Wagner, D. (eds.): Graph Partitioning and Graph Clustering. Contemporary Mathematics, vol. 588. American Mathematical Society, Providence (2013)
9. Bast, H., Delling, D., Goldberg, A.V., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., Werneck, R.F.: Route planning in transportation networks. CoRR, abs/1504.05140 (2015)
10. Borutta, F., Nascimento, M.A., Niedermayer, J., Kröger, P.: Monochromatic rknn queries in time-dependent road networks. In: Proceedings of the Third ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems, MobiGIS 2014 pp. 26–33, New York, NY, USA. ACM (2014)
11. Cho, E., Myers, S.A., Leskovec, J.: 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, 21–24 August 2011, pp. 1082–1090 (2011)
12. Cohen, E., Halperin, E., Kaplan, H., Zwick, U.: Reachability and distance queries via 2-hop labels. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002, pp. 937–946. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2002)
13. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable route planning. In: Pardalos, P.M., Rebennack, S. (eds.) SEA 2011. LNCS, vol. 6630, pp. 376–387. Springer, Heidelberg (2011)
14. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Robust distance queries on massive networks. In: Schulz, A.S., Wagner, D. (eds.) ESA 2014. LNCS, vol. 8737, pp. 321–333. Springer, Heidelberg (2014)
15. Delling, D., Goldberg, A.V., Werneck, R.F.: Hub label compression. In: Demetrescu, C., Marchetti-Spaccamela, A., Bonifaci, V. (eds.) SEA 2013. LNCS, vol. 7933, pp. 18–29. Springer, Heidelberg (2013)

16. Delling, D., Goldberg, A.V., Werneck, R.F.F.: Faster batched shortest paths in road networks. In: ATMOS, pp. 52–63 (2011)
17. Delling, D., Werneck, R.F.: Customizable point-of-interest queries in road networks. In: 21st SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2013, Orlando, FL, USA, 5–8 November 2013, pp. 490–493 (2013)
18. Delling, D., Werneck, R.F.: Better bounds for graph bisection. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 407–418. Springer, Heidelberg (2012)
19. Efentakis, A., Pfoser, D.: Optimizing landmark-based routing and preprocessing. In: CTS: 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science, 5 November 2013, Orlando, FL, USA, p. 25 (2013)
20. Efentakis, A., Pfoser, D.: GRASP. extending graph separators for the single-source shortest-path problem. In: Schulz, A.S., Wagner, D. (eds.) ESA 2014. LNCS, vol. 8737, pp. 358–370. Springer, Heidelberg (2014)
21. Efentakis, A., Pfoser, D.: ReHub. Extending hub labels for reverse k-nearest neighbor queries on large-scale networks (2015). arXiv preprint http://arXiv:1504.01497
22. Efentakis, A., Pfoser, D., Vassiliou, Y.: SALT. a unified framework for all shortest-path query variants on road networks. In: Bampis, E. (ed.) SEA 2015. LNCS, vol. 9125, pp. 298–311. Springer, Heidelberg (2015)
23. Gavoille, C., Peleg, D., Pérennes, S., Raz, R.: Distance labeling in graphs. In: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2001, pp. 210–219. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2001)
24. Geisberger, R., Sanders, P., Schultes, D.: Better approximation of betweenness centrality. In: Munro, J.I., Wagner, D. (eds.) ALENEX, pp. 90–100. SIAM (2008)
25. Jiang, M., Fu, A.W., Wong, R.C., Xu, Y.: Hop doubling label indexing for point-to-point distance querying on scale-free networks. PVLDB **7**(12), 1203–1214 (2014)
26. Leskovec, J., Krevl, A.: SNAP Datasets: Stanford large network dataset collection, June 2014. http://snap.stanford.edu/data
27. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Math. **6**(1), 29–123 (2009)
28. McAuley, J.J., Leskovec, J.: 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 3–6 December 2012, Lake Tahoe, Nevada, United States, pp. 548–556 (2012)
29. PostgreSQL. The world's most advanced open source database (2015). http://www.postgresql.org/
30. Safar, M., Ibrahimi, D., Taniar, D.: Voronoi-based reverse nearest neighbor query processing on spatial networks. Multimedia Syst. **15**(5), 295–308 (2009)
31. Yang, J., Leskovec, J.: Defining and evaluating network communities based on ground-truth. In: 12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, 10–13 December 2012, pp. 745–754 (2012)
32. Yiu, M.L., Papadias, D., Mamoulis, N., Tao, Y.: Reverse nearest neighbors in large graphs. IEEE Trans. Knowl. Data Eng. **18**(4), 540–553 (2006)
33. Zhong, R., Li, G., Tan, K.-L., Zhou, L.: G-tree: An efficient index for knn search on road networks. In: Proceedings of the 22nd ACM International Conference on Conference on Information Knowledge Management, CIKM 2013, pp. 39–48. ACM, New York, NY, USA (2013)

http://www.springer.com/978-3-319-22362-9

Advances in Spatial and Temporal Databases
14th International Symposium, SSTD 2015, Hong Kong, China, August 26-28, 2015. Proceedings
Claramunt, C.; Schneider, M.; Wong, R.C.-W.; Xiong, L.; Loh, W.-K.; Shahabi, C.; Li, K.-J. (Eds.)
2015, XIV, 522 p. 235 illus., Softcover
ISBN: 978-3-319-22362-9