Noname manuscript No.
(will be inserted by the editor)

Hub Labels on the database for large-scale graphs with the COLD framework

Alexandros Efentakis · Christodoulos Efstathiades · Dieter Pfoser

Received: date / Accepted: date

Abstract Shortest-path computation on graphs is one of the most well-studied problems in algorithmic theory. An aspect that has only recently attracted attention is the use of databases in combination with graph algorithms, so-called distance oracles, to compute shortest-path queries on large graphs. To this purpose, we propose a novel, efficient, pure-SQL framework for answering exact distance queries on large-scale graphs, implemented entirely on an open-source database engine. Our COLD framework (COmpressed Labels on the Database) may answer multiple distance queries (vertex-to-vertex, one-to-many, k-Nearest Neighbors, Reverse k-Nearest Neighbors, Reverse k-Farthest Neighbors and Top-k Range) not handled by previous methods, rendering it a complete database solution for a variety of practical large-scale graph applications. Our experimentation shows that COLD outperforms existing approaches (including popular graph databases) in terms of query time and efficiency, while requiring significantly less storage space than these methods.

A. Efentakis
Research Center "Athena"
Artemidos 6 & Epidavrou, Marousi 15125, Greece
E-mail: efentakis@imis.athena-innovation.gr

C. Efstathiades
Department of Computer Science and Engineering,
European University Cyprus
E-mail: c.efstathiades@euc.ac.cyr

D. Pfoser
Department of Geography and GeoInformation Science, George Mason University
Exploratory Hall, Rm 2203, 4400 University Drive, MS 6C3 Fairfax, VA, 22032
E-mail: dpfoser@gmu.edu

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. [10] for a 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, such an approach cannot facilitate fast-enough queries on main memory or be easily adapted to secondary storage solutions.

Moreover, most of the 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 [28,14,29], in which the preprocessing stage stores a 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 [2,3,19,5], undirected, unweighted graphs in [6,17,33] and schedule-based, public-transportation networks in [15, 48,22]. The HL method has also been applied to one-to-many, many-to-many and k-Nearest Neighbors (kNN) queries on road networks [18,20] and Reverse k-Nearest Neighbor (RkNN) queries in the context of social networks in [26].

With the emergence of big data analytics, implementing analysis methods as part of a commercial database is attractive. It would allow developers to leverage the power of an existing system and the expressiveness of a database language such as SQL to create new types of data analysis methods. In the case of route planning respective methods have been implemented as database distance oracles [43,21]. Data is stored and queried completely in SQL rather than having a standalone module that runs outside the database. The benefits are that the database system automatically provides an external memory implementation of the algorithm to support scaling. Hub labeling methods present an excellent case for a database-based implemementation, as the preprocessing step generates a quantitative representation of possible shortest paths that can be stored and queried in a database. However, there still has been only a limited number of works that try to replicate those algorithms for secondary storage. HLDB [21] 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 [33] proposed a customized solution that utilizes secondary storage also during preprocessing. Unfortunately, both these methods have their inherent 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 extends the results originally presented in [23] and proposes a database framework that may service multiple exact distance queries on massive large-scale graphs. Our pure-SQL *COLD* framework (COmpressed Labels on the Database) can answer multiple exact distance queries (vertex-to-vertex, k-Nearest Neighbors) in addition to Reverse k-Nearest Neighbors 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. In comparison to our original work of [23], this work extends the HL technique to solve *Reverse k-Farthest Neighbors* and *Top-k Range* queries and shows how these specific queries may be efficiently answered within the COLD framework. 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 Section 4. Finally, Section 5 gives conclusions and directions for future work.

2 Related work

Throughout this work we use undirected, unweighted graphs $G(V, E)$ (where V and E represent vertices and arcs respectively). Since we also deal with additional types of queries (e.g., k-Nearest Neighbor queries) there is also a static set of targets P that are assumed to be located on vertices. Throughout this work, we always refer to *snapshot* queries, i.e, targets are not moving or changing. Also, similarly to previous works, the term *target density* D refers to the ratio $|P|/|V|$, where P is the set of targets in the graph and $|V|$ is the total number of vertices. Although there is extensive literature focusing on certain queries (e.g., kNN, RkNN) in Euclidean space, since our work focuses on graphs we will mainly describe related work focusing on the latter. Since the COLD framework builds on the Hub Labeling technique, we will also describe related work referring to this specific shortest-path method.

2.1 Hub Labeling

Our work builds upon the 2-hop labeling or Hub Labeling (HL) algorithm of [28, 14, 29] in which, the preprocessing stage stores at every vertex v a for-

ward $L_f(v)$ 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 [2,3,19,5]. In the case of large-scale graphs, the Pruned Landmark Labeling (PLL) algorithm of [6] *produces a minimal labeling for a specified vertex ordering.* In this work, vertices are ordered by degree, whereas the work of [17] improves the suggested 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 [18] and [20] respectively. Recently the HL technique has also been extended to schedule-based, public-transportation networks in [15,48].

2.2 k-Nearest Neighbor (kNN) queries

A k-Nearest Neighbor (kNN) query seeks the k-nearest neighbors to a query vertex q, according to the network distance. Considering road networks and kNN queries, G-tree [51] is a balanced tree structure, constructed by recursively partitioning the road network into sub-networks. Then the best-first algorithm is applied on this G-tree index structure to answer kNN queries. 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 targets (requiring few seconds) and thus, cannot be used for moving targets.

Several efforts focussed on kNN queries on road networks using well-known algorithmic shortest-path techniques. The work of [20] expanded the graph-separators CRP algorithm of [16] to handle kNN queries on road networks. Likewise, the work of [37], extended Contraction Hierarchies (CH) [31] in the context of kNN queries in road networks. Unfortunately, both methods have their inherent disadvantages: (i) Since the work of [37] builds on CH, it requires several minutes for preprocessing continental road networks. Hence, it cannot be used for dynamic, live-traffic scenarios. (ii) Both approaches also require a target selection phase and thus, they cannot be applied in the case of moving targets. (iii) Their performance degrades for sparser targets, which is the typical case for a specific type of targets or small vehicle-fleets. Hence, those solutions are also not optimal. To remedy those shortcomings, the SALT framework [27] requires a few seconds for preprocessing continental road networks and is thus suitable for dynamic, live-traffic road networks. Moreover, it may also be used to answer multiple distance queries on road networks, in-

cluding *vertex-to-vertex* (v2v), single source (one-to-all, range, one-to-many) and kNN queries. That work expanded the graph-separators GRASP [25] algorithms and the ALT-SIMD adaptation [24] of the ALT algorithm and offers 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 targets, offering comparable query times to [20, 37].

2.3 Reverse k-Nearest Neighbor (RkNN) queries

The Reverse k-Nearest Neighbor (RkNN) query (also referred as the monochromatic RkNN query), given a query point q and a set of targets P, retrieves all the targets 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 two vertices. Formally RkNN(q) = $\{p \in P : dist(p, q) \leq dist(p, p_k)\}$ where p_k is the k-Nearest Neighbor of p.

For RkNN queries on road networks, the work of [42] 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. In addition, he preprocessing stage for computing the Network Voronoi cells is quite costly. Up until recently, the only work dealing with other graph classes (besides road networks) is [50], 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 ($4K$ 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. [11] extended this work for time-dependent road networks, but the presented results were not very encouraging.

The latest work of the co-authors [26] 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 targets and a very fast online phase which depends on the query vertex q. Still, even the costlier offline phase hardly needs more than $1s$, whereas the online phase requires usually less than $1ms$, making *ReHub* the only RkNN algorithm fast enough for real-time applications on big, large-scale graphs.

2.4 Reverse k-Farthest Neighbor (RkFN) queries

Reverse k-nearest neighbor queries have a natural counterpart called Reverse k-Farthest (RkFN) neighbors, where we are interested in knowing the points that are least influenced by a query point q. These are those points that get assigned q as their farthest neighbor. RkFN queries haver numerous applications in location-based services, marketing, outlier detection, clustering and

profile-based management. For example, in site planning when given a potential location for a hazardous waste site, the reverse farthest neighbors of the site are the ones that are least affected by the site. Other problems mentioned are Painter's algorithm, a popular rendering approach, which gives priority to furthest targets from the view point. An object that has many RFNs (i.e., is among the furthest targets from many view points) is likely to be accessed first for rendering (cf. [47]). In relation to social networks RFN queries are important to identify an audience that has not been "reached" yet, i.e., which is something that is relevant when monitoring the effect of social media marketing campaigns. Kumar et al. [34] introduce RFN problems and present an approximate approach to compute RFNs. It requires pre-calculation of so called FN-ball of points. Liu et al. [38] proposed the PIV algorithm that constructs metric indexes and employs triangle inequality to do pruning. All these techniques require expensive pre-computation and, therefore, are not suitable for dynamic data sets or for arbitrary value of k. Recently, Wang et al. [47] is the latest among several papers discussing RkFN queries. Their method works for arbitrary values of k and relies on several pruning phases, with the main phase being based on k-depth contours [12].

While all the above methods work on Euclidean space, Tran et. al. [46] address RFN queries on road networks based on Network Voronoi Diagrams and a precomputation of network distances. A Network Voronoi Diagram (NVD) differs from a Euclidean space Voronoi Diagram in that every point in each Voronoi cell is closer to the generator point of the cell based on minimum network distances, instead of the Euclidean distances between the points. While considerable, this specific work does not report the pre-computation cost for the NVD construction. The query times range from seconds for $k = 1$, to thousands of seconds for larger values of k.

2.5 Top-k range queries

The *top-k range* reporting problem finds the best few targets among only a subset of the dataset satisfying a range predicate. For example, a user of a hotel database may be interested in discovering the k-best rated hotels whose prices are in a designated range. Likewise, for promotion purposes, the manager of a company may want to find the k-salesmen with the best performance, among those salesmen whose salaries are in a certain range. Top-k range queries have also been studied in the context of information retrieval [4], OLAP [39], and data streams [32]. Recently, there is also significant work [44,45] regarding the theoretical aspects of the top-k range reporting problem.

When extended to spatial or network databases, the top-k range query $top - k\ range(q, k, [a, b))$ may be naturally extended to find the k-nearest neighbors of the query location or vertex between the subset of targets $\in P$ with their spatial or network distance from the query location (vertex) lying in the $[a, b)$ interval. The spatial / network $top - k\ range(q, k, [a, b))$ query has numerous use-cases, such as finding the k-closest hotels to the city cen-

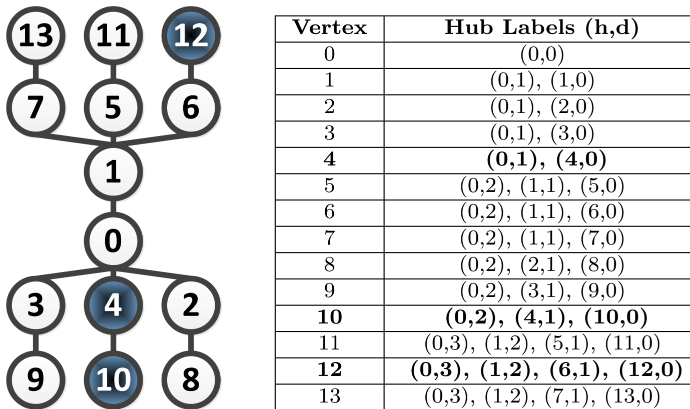
ter or the airport that are however between $2km$ and $4km$ away from it, to avoid the corresponding noise or inflated prices. In the context of collaboration networks, a $top - k\ range(q, k, [a, b))$ may be used to identify k-scientific authors belonging to a particular institution that have collaborated with one of the co-authors of researcher A (and therefore have a hop-distance > 2 from the vertex associated with researcher A). To the best of our knowledge, the network top-k range variation has not been tackled with hub labels before.

2.6 Secondary storage

Regarding secondary-storage solutions, Jiang et al. [33] propose the 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, HLDB 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 [26] into a database context, so that its online phase may be translated to a fast and optimized SQL query. Based on the results presented in our original work of [23], the recent work of [22] also extended the Timetable Labelling [48] method for public-transportation networks, in the context of databases.

3 The COLD framework

This section presents the *COLD* (COmpressed Labels on the Database) database framework. COLD can answer multiple exact distance queries (vertex-to-vertex, k-Nearest Neighbor, Reverse k-Nearest Neighbor and *one-to-many*) for large-scale graphs using SQL commands. Considering that COLD builds on HLDB [1] and *ReHub* [26], we will follow the notation and running example presented there, for highlighting the necessary concepts and 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 [26]. To this end, we chose PostgreSQL [41] 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. In comparison to our original work of [23], this section has the additional Sections 3.1.5 and 3.1.7 that show how (i) *Top-k range* and Reverse k-Farthest Neighbors (RkFN) may be answered by a hub labeling framework on main memory and (ii) how these specific queries may be handled inside COLD, providing the exact implementation details, including the required database table formats and the corresponding SQL query code.



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)

Fig. 1 & Table 1: A sample Graph G and the created hub-labels

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 [7] implementation of the PLL algorithm of [6] to generate the necessary labels. To highlight the results of this process, the labels for the undirected, unweighted graph G of Figure 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 a vector of pairs $(u, dist(v, u))$ sorted by hub u. Since our work also focuses on snapshot kNN, RkNN, RkFN and Top-k range queries, there are also some targets $P \in V$ that do not change over time. For our specific running example we assume that $P = \{4, 10, 12\}$ and thus, we highlight the respective vertices and entries in Figure 1 and Table 1, respectively.

3.1.1 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 $v \in L(s) \cap L(t)$ 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.

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

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: V2v query for HLDB

```
1 SELECT MIN(n1.dist+n2.dist)
2 FROM
3 FORWARD n1,
4 FORWARD n2
5 WHERE n1.v = s
6   AND n2.v = t
7   AND n1.hub = n2.hub;
```

Code 2: V2v query for COLD

```
1 SELECT MIN(n1.d+n2.d)
2 FROM
3   /* Expand hubs, dists arrays */
4   (SELECT UNNEST(hubs) AS hub,
5           UNNEST(dists) AS d
6    FROM forwcold
7    WHERE v = s) n1,
8   (SELECT UNNEST(hubs) AS hub,
9           UNNEST(dists) AS d
10   FROM forwcold
11   WHERE v = t) n2
12 WHERE n1.hub=n2.hub;
```

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 vertex-to-vertex 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 the primary key without needing a composite key. This alone facilitates faster queries. Moreover the size of the corresponding PK index will be significantly 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 vertex-to-vertex 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

Table 4: Necessary data structures for the sample graph G, $P = \{4, 10, 12\}$ and *one-to-many*, k-Nearest Neighbor and Reverse k-Nearest Neighbor queries

Hub	**Labels-to-many [18]**	***k*NN Backward Labels (k=2) [1]**	**R*k*NN Labels (k=1) [26]**	**Obj.**	**kNN Result (k=1) (Obj., dist) [26]**
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 COLD is shown in Code 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].

3.1.2 Additional queries overview

For answering more complex (kNN, RkNN and *one-to-many*) distance queries on a HL framework for a set of targets P, we need to build some additional data structures from the forward labels (for undirected graphs). Then, for answering 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.

For answering *one-to-many* queries, i.e., calculate distances between the query vertex q and all targets in P, we need to build the *labels-to-many* by basically ordering the forward labels of the targets by hub [18] and then by distance for the same hub. For k-Nearest Neighbor queries we only need to keep at most the k-best pairs (of smallest distances) per hub from the 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 Reverse k-Nearest Neighbor queries, we must first calculate the *kNN Results* (i.e., the NN of the target 4 is the target 10 with distance 1) and then we build the RkNN labels, based on the observation that *"we need to access those pairs from the labels-to-many to a specific target, if and only if those distances are equal or smaller than the distance of the kNN of this target"* [26]. In our specific example, the RkNN labels for $k = 1$ and hub 0, do not contain the pair (10,2) since the NN of target 10 (the target 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 prominent. This was also showcased by the theoretical analysis provided in [26] which showed that the 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 labels have on average $\varepsilon \cdot D \cdot |HL|$ pairs where $\varepsilon < 0.01$ for specific datasets and experimental settings. Moreover, Efentakis et al. [26] have shown how these additional data structures may be

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

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}
...	...	...

constructed from the forward labels in main-memory, requiring less than few seconds, even for the larger tested datasets.

3.1.3 k-Nearest Neighbor (kNN) queries

To translate the HL k-Nearest Neighbor query into SQL, HLDB stores the *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 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 targets 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 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 4).

Code 3: kNN query for HLDB

```
1 SELECT MIN(n1.dist+n2.dist),
2        n2.obj
3 FROM
4 FORWARD n1,
5         knntab n2
6 WHERE n1.v = q
7   AND n1.hub = n2.hub
8 GROUP BY n2.obj
9 ORDER BY MIN(n1.dist+n2.dist)
10 LIMIT k;
```

Code 4: kNN query for COLD

```
1 SELECT MIN(n1.d+n2.dist),
2        UNNEST(objs) AS obj
3 FROM
4   (SELECT UNNEST(hubs) AS hub,
5           UNNEST(dists) AS d
6    FROM forwcold
7    WHERE v = q) n1,
8   /* k-entries per hub,dist */
9   (SELECT hub,
10          dist,
11          objs[1:k]
12   FROM knncold) n2
13 WHERE n1.hub=n2.hub
14 GROUP BY obj
15 ORDER BY MIN(n1.d+n2.dist)
16 LIMIT k;
```

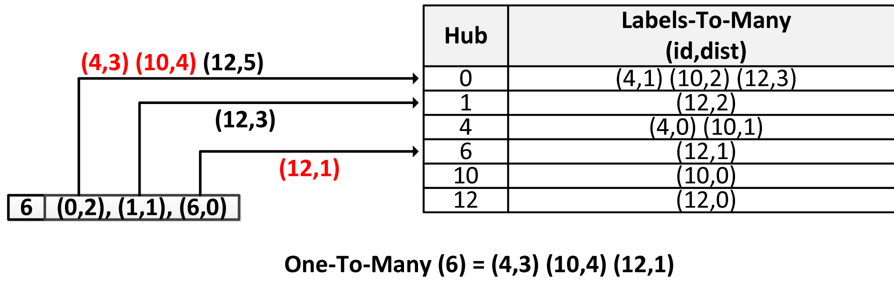


Fig. 2: An example one-to-many query from query vertex 6

3.1.4 One-to-many queries

Similar to how COLD handles k-Nearest Neighbor queries, for one-to-many queries (see Figure 2 for an example), COLD stores the *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 targets are grouped and ordered per hub and distance. The *Objcold* DB table (see Table 7) also uses the combination of $(hub, dist)$ as a primary key. The resulting *one-to-many* SQL query (Code 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 $|L(q)| \cdot D \cdot (|HL|/|V|)$ [26] rows from the corresponding *objlab* table, which will be prohibitively slow for datasets with large $|HL|/|V|$ ratios.

Table 7: The *objcold* table used in COLD for One-To-Many queries, the sample graph G, $k = 1$ and $P = \{4, 10, 12\}$

hub	dist	objs
0	1	{4}
0	2	{10}
0	3	{12}
1	2	{12}
4	0	{4}
4	1	{10}
...	...	...

Code 5: *One-to-many* COLD query

```
1 SELECT MIN(n1.d+n2.dist),
2        UNNEST(objs) AS obj
3 FROM
4   (SELECT UNNEST(hubs) AS hub,
5           UNNEST(dists) AS d
6    FROM forwcold
7    WHERE v = q) n1,
8      objcold n2
9 WHERE n1.hub=n2.hub
10 GROUP BY obj;
```

3.1.5 Top-k range queries

Since, to the best of our knowledge, there is no previous work that answers top-k range queries for graphs with hub labels, we will: (i) demonstrate how this specific query may be solved in main memory and then (ii) how we can answer this query efficiently within our COLD database framework.

Regarding a main memory solution, once we build the *labels-to-many* data structure (see Column 2 of Table 4) the $top - k\ range(q, k, [a, b))$ query may be solved with the pseudocode provided in procedure Top-k Range. In this solution, we basically run a modified one-to-many query from the query vertex q where we store the distances from the query vertex to the target vertices in a hash map (denoted *hmap* in the pseudocode) if and only if those distances are smaller than the upper bound b of the top-k range query (see Line 8 in the pseudocode). If not, since the labels-to-many are ordered by hub and distance, we can move on to the next hub (Line 9). Note, than we cannot use the lower bound a to prune this calculation, since we need to calculate correct shortest-path distances to the target vertices.

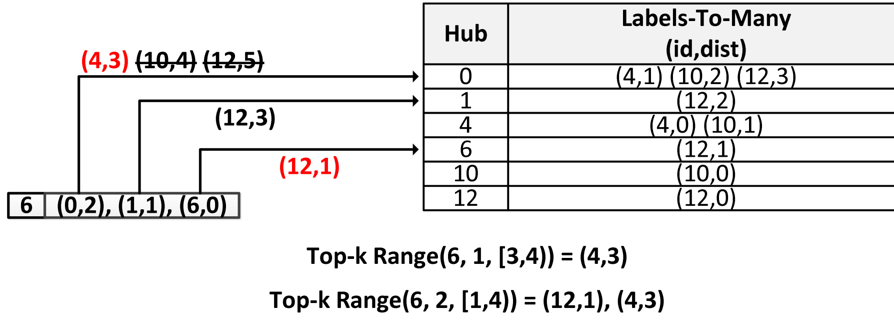


Fig. 3: Example top-k range queries from query vertex 6

TOP-K RANGE $(q, k, a, b, forwLabels, LabelsToMany, PQueue)$

1 $hmap = empty\ HashMap(id, distance)$
2 **for** $i = 0$ **to** $forwLabels[q].size$
3 $\quad hub = forwLabels[q][i].hub$
4 $\quad d = forwLabels[q][i].dist$
5 $\quad$ **for** $j = 0$ **to** $LabelsToMany[hub].size$
6 $\quad\quad id = LabelsToMany[hub][j].id$
7 $\quad\quad d2 = d + LabelsToMany[hub][j].dist$
8 $\quad\quad$ **if** $d2 \geq b$
9 $\quad\quad\quad$ **Break**
10 $\quad\quad$ **if** $hmap[id] == \text{NIL} \mid\mid d2 < hmap[id]$
11 $\quad\quad\quad hmap[id] = d2$
12 $PQueue = empty\ Bounded\ Priority\ Queue(distance, id)\ of\ size\ k$
13 **for** each $(id, distance)\ pair \in hmap$
14 $\quad$ **if** $distance \geq a$
15 $\quad\quad PQueue.push(distance, id)$
16 **return** $PQueue$

Once we calculate those correct shortest-path distances to all target vertices that are smaller than b (Lines 1-11 in the pseudocode) we use a vector-based bounded priority queue (denoted $PQueue$ in the pseudocode) of size k to store the top-k range results. Then, we have to loop through all the entries of the $hmap$ and push each $(id, distance)$ pair for which $distance \geq a$. This customized push operation checks if the priority queue has already k items and keeps only the best k-entries (smallest distances) from the query vertex. This process is highlighted in Figure 3 that shows the results of two top-k range queries from vertex 6 for different ranges.

Code 6: Top-k range query for COLD

```
1 SELECT UNNEST(ids) AS id2,
2        MIN(n1.distance+n2.distance)
3 FROM
4   (SELECT id,
5           UNNEST(hubs) AS hub,
```

```
6           UNNEST(distances) AS distance
7     FROM forwcold
8     WHERE id=q) n1,
9    (SELECT hub,
10           distance,
11           ids[1:k]
12    FROM objcold) n2
13 WHERE n1.hub=n2.hub
14   AND n1.distance+n2.distance<b
15 GROUP BY id2
16 HAVING MIN(n1.distance+n2.distance)>=a
17 ORDER BY MIN(n1.distance+n2.distance),id2
18 LIMIT k;
```

Theorem 1 *The proposed top-k range algorithm is correct.*

Proof The labels-to-many and the forward labels of query vertex q suffice to calculate correct shortest-path distances to all target vertices, as shown by [18]. Since we only allow shortest-path distances that are smaller than b to enter the *hmap*, Lines 1-11 store correct shortest-path distances to target vertices only when they are smaller than b. Then, by using a bounded priority queue of size k and by not allowing distances smaller than a (Line 14) to enter this priority queue, we can guarantee that at the end of this procedure, the bounded priority queue has the k-smallest $(distance, id)$ pairs with shortest-path distances between bounds a and b.

To answer top-k range queries within COLD, we can directly use the *forwcold* (see Table 3) and *objcold* DB tables (see Table 7) without needing any additional DB tables. The corresponding SQL query is shown in Code 6. It is obvious that the corresponding query is highly optimized, since it only retrieves k-entries per $(hub, distance)$ combination of the objcold DB table (see Line 12) and prunes the search space according to upper bound b (Line 5).

3.1.6 Reverse k-Nearest Neighbor (RkNN) queries

For RkNN queries, COLD stores the RkNN 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 targets 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 targets 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). Therefore the kNN of target 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 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 7.

We see that even the more complex Reverse k-Nearest Neighbor query in COLD requires just a few lines of SQL code that will work in any recent PostgreSQL version without any need of third-party extensions or specialized index

Table 8: 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}

Code 7: RkNN query for COLD

```
1 SELECT n3.id2,
2        n3.dist
3 FROM
4 /* n3 subquery is a modified
5  one-many-query to revcold */
6   (SELECT MIN(n1.d+n2.dist) AS d3,
7           UNNEST(objs) AS obj
8    FROM
9      (SELECT UNNEST(hubs) AS hub,
10             UNNEST(dists) AS d
11       FROM forwcold
12       WHERE v = q) n1,
13         revcold n2
14    WHERE n1.hub=n2.hub
15    GROUP BY obj
16    ORDER BY obj,
17             MIN(n1.d+n2.dist)) n3,
18   /* Join with knnres table */
19   (SELECT obj,
20           dists[k] AS dist
21    FROM knnres) n4
22 WHERE n3.obj=n4.obj
23   AND n3.d3<=n4.dist
24 ORDER BY n3.obj;
```

structures. This also holds true for the similar Reverse k-Farthest Neighbor query, detailed in the following section.

3.1.7 Reverse k-Farthest Neighbor (RkFN) queries

Again, to the best of our knowledge, there is no previous work that solves Reverse k-Farthest Neighbor (RkFN) queries for graphs, using the calculated hub labels. Thus, we will: (i) show how this query type may be solved in main memory and then (ii) how we can answer this specific type of queries efficiently within our COLD database framework.

Regarding main memory, our solution expands the methodology originally proposed for RkNN queries in our previous work of [26]. Analogously to the *ReHub* algorithm proposed for RkNN queries, for answering RkFN queries we need to calculate the Reverse k-Farthest Neighbors of all target vertices during the offline phase. The corresponding offline phase hence, depends only on the

target vertices and the value of k. During the following online phase, we need to perform an one-to-many query from the query vertex q to the target vertices and then check if the query vertex belongs to each target's Reverse k-Farthest Neighbors set. The corresponding pseudocode for the online and offline phase of the proposed *ReFar* algorithm in shown in procedures OfflinePhaseReFar and OfflinePhaseReFar respectively.

OfflinePhaseReFar $(k, P, |P|, forwLabels, LabelsToMany)$

1 Initialize $PQueue(|P|)$
2 **for** $i = 0$ **to** $|P|$
3 $\quad hmap = empty\ HashMap(id, distance)$
4 $\quad$ OneToMany$(P_i, forwLabels[P_i], LabelsToMany, hmap)$
5 $\quad PQueue[i] = empty\ Bounded\ Max - PQueue(distance, id)\ of\ size\ k$
6 $\quad$ **for** each $(id, distance)\ pair \in hmap$
7 $\quad\quad PQueue[i].push(distance, id)$
8 **return** $PQueue$

OnlinePhaseReFar $(q, k, P, |P|, forwLabels, LabelsToMany, PQueue)$

1 $results = empty\ vector(id, distance)$
2 $hmap = empty\ HashMap(id, distance)$
3 OneToMany$(q, forwLabels[q], LabelsToMany, hmap)$
4 **for** $i = 0$ **to** $|P|$
5 $\quad$ **if** $hmap[id] \leq PQueue[id][k].distance$
6 $\quad\quad results.pushBack(id, hmap[id])$
7 **return** $results$

Considering the offline phase, after building the *labels-to-many* data structure, we need to perform a total of $|P|$ one-to-many queries (one from each target), using a hash map (denoted *hmap* in the pseudocode) for temporarily storing the results of each individual query. Then, by using a $|P|$-sized vector of bounded max-priority queues of size k, we store the k-largest shortest path $(distance, id)$ combinations per target. Note, that it is necessary to perform each individual one-to-many query to first calculate correct shortest-path distances from each target before keeping the k-largest shortest path $(distance, id)$ combinations per target. In the online phase, we use again the *labels-to-many* data structure (see Column 2 of Table 4) and we perform an one-to-many query from the query vertex q and similarly store the corresponding results on a hash map. Then we loop at this hash map and we store the $(id, distance)$ pair combinations where the calculated distance is greater or equal to the k-farthest neighbor of target id (see Line 5).

Theorem 2 *The proposed* ReFar *algorithm for RkFN queries is correct.*

Proof The forward labels of each target and the labels-to-many suffice to calculate correct shortest-path distances from each target to all other target vertices, as shown by [18]. Then we can calculate the Reverse k-Farthest Neighbors of each target by keeping the k-largest of those $(distance, id)$ combinations.

Table 9: The *farres* table used in COLD for RkFN queries, the example graph G, $k = 1$ and $P = \{4, 10, 12\}$

obj	dists	objs
4	{4}	{12}
10	{5}	{12}
12	{5}	{10}

Thus, the offline phase of *ReFar* is correct. Then, during the online phase by using the forward labels of the query vertex and the labels-to-many we can calculate correct shortest-path distances from the query vertex to each target. Keeping those $(id, distance)$ combinations with distances $\geq$ the distance of the k-Farthest neighbor of target with $ID = id$ gives us the Reverse k-Farthest Neighbors of query vertex q. Thus, the online phase of *ReFar* is also correct.

For our specific graph G, $P = \{4, 10, 12\}$ and $k = 1$, *ReFar*'s offline phase correctly outputs that $RFN(4) = (12, 4)$, $RFN(10) = (12, 5)$ and $RFN(12) = (10, 5)$. Similarly to how COLD handles RkNN queries and the corresponding *knnres* DB table used there, COLD stores the RkFN results in a *farres* 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) (see Table 9). Therefore the RkFN of target p is the `objs[k]` within distance `dists[k]` of the respective row for p. Again, we build a *farres* DB table for a max value of *kmax* that may service RkFN queries for varying values of k. The corresponding SQL query for the online phase of *ReFar* and our COLD framework is shown in Code 8. There we see that even for the Reverse k-Farthest Neighbors queries, the corresponding SQL code is very simple within our COLD framework.

Code 8: RkFN query for COLD

```
1 SELECT id2,
2        mindist
3 FROM
4   (SELECT UNNEST(ids) AS id2,
5           MIN(n1.distance+n2.distance) AS mindist
6    FROM
7      (SELECT id,
8              UNNEST(hubs) AS hub,
9              UNNEST(distances) AS distance
10      FROM forwcold
11      WHERE id=q) n1,
12      objcold n2
13   WHERE n1.hub=n2.hub
14   GROUP BY id2
15   ORDER BY MIN(n1.distance+n2.distance)) s2a,
16  (SELECT id,
17          distances[k] AS distfar
18   FROM farres) s2b
19 WHERE id2=id
20   AND s2a.mindist>=distfar
21 ORDER BY mindist, id2;
```

On an additional note, 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 target (*knnres*, *farres* tables), or by hub and distance for *knncold*, *objcold* and *rknncold*. Likewise, 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 (vertex-to-vertex, kNN, RkNN RkFN, top-k range 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 for clarity). This fact also shows that COLD is truly a pure-SQL framework for servicing multiple exact 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 3.5GHz and 32Gb 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 works of [26, 23] that are taken from the Stanford Large Network Dataset Collection [35] and the 10th Dimacs Implementation Challenge website [9]. All graphs are undirected, unweighted and connected. We used collaboration graphs (DBLP, Citeseer1, Citeseer2) [30], social networks (Facebook [40], Slashdot1 and Slashdot2 [36]), networks with ground-truth communities (Amazon, Youtube) [49], web graphs (Notre Dame) [8] and location-based social networks (Gowalla) [13]. The graphs' average degree is between 3 and 37 and the PLL algorithm creates $26 - 4,457$ labels per vertex, requiring $0.03 - 5,946s$ for the hub labels' construction (see Table 10). To approximate each graph's diameter we also used the largest distance encountered in the corresponding hub labels that provides a lower bound on the suggested diameter.

COLD and HLDB were implemented in PostgeSQL 9.3.6, 64bit with reasonable settings (8192Mb *shared buffers*, 64Mb *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

Table 10: Networks graphs statistics

Graph	|V|	|E|	Avg degree	|HL| / |V|	PLL Preproc. Time (s)	Diameter
Facebook	4,039	88,234	22	26	0.03	5
NotreDame	325,729	1,090,108	3	55	6	27
Gowalla	196,591	950,327	5	100	13	10
Youtube	1,134,890	2,987,624	3	167	123	14
Slashdot	77,360	469,180	6	204	11	7
Slashdot2	82,168	504,230	6	216	13	8
Citeseer	268,495	1,156,647	4	408	110	28
Amazon	334,863	925,872	3	689	230	39
DBLP	540,486	15,245,729	28	3,628	5,720	14
Citeseer2	434,102	16,036,720	37	4,457	5,946	25

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.

We conducted experiments belonging to the following query types: (i) *vertex-to-vertex*, (ii) k-Nearest Neighbors (kNN), (iii) Reverse k-Nearest Neighbors (RkNN) and (iv) *one-to-many*. In comparison to our original work of [23] this section also presents detailed results about the additional top-k range and reverse k-farthest queries, presented in the newly introduced Sections 4.1.4, 4.1.6, 4.2.4 and 4.2.6. For each experiment, we used 1,000 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.

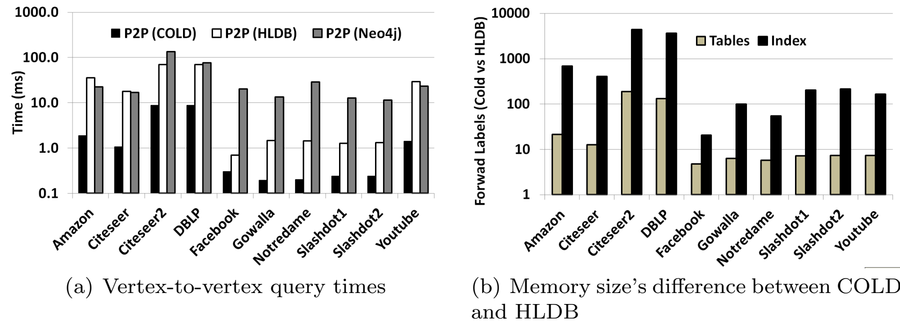
4.1 Performance on HDD

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

4.1.1 Vertex-to-vertex queries

Figure 4(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

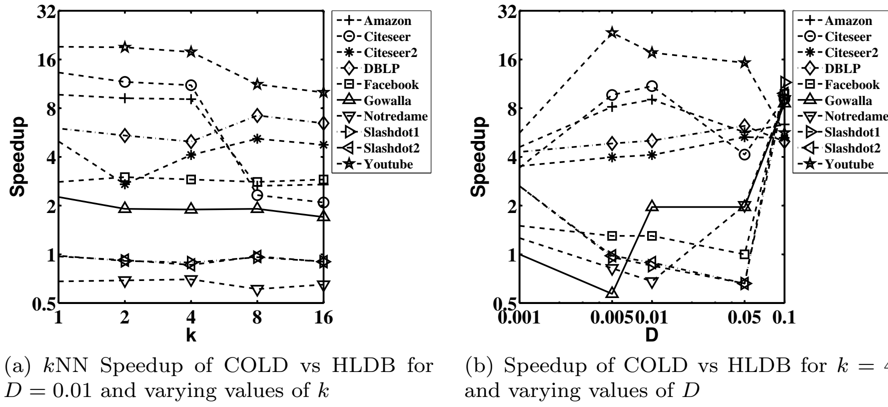
[2] http://neo4j.com/developer/guide-performance-tuning/



(a) Vertex-to-vertex query times

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

Fig. 4: *Vertex-to-vertex* queries for COLD, HLDB and Neo4j on the HDD



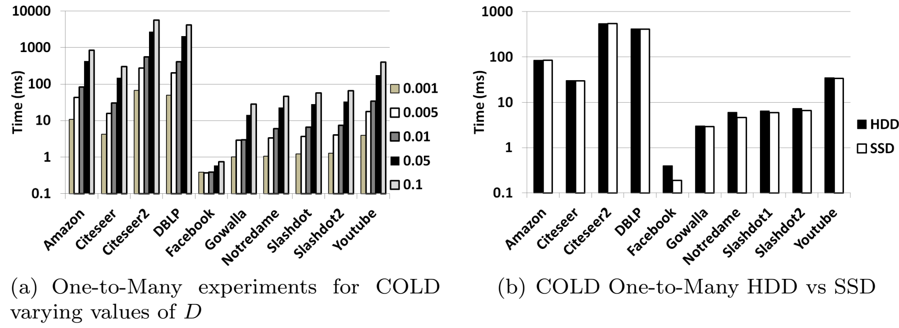
(a) kNN Speedup of COLD vs HLDB for $D = 0.01$ and varying values of k

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

Fig. 5: kNN Experiments for COLD and HLDB on the HDD

for all datasets, but is slower from both COLD and HLDB. For all datasets, COLD requires less than $9ms$ for answering vertex-to-vertex queries.

Figure 4(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 $3,600$ - $4,444\times$ 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 - $188\times$ 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 on large-scale graphs.



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

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

Fig. 6: One-to-many experiments for COLD on the HDD

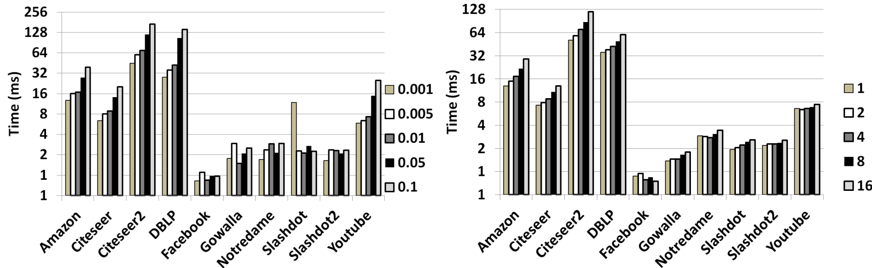
4.1.2 k-Nearest Neighbor (kNN) queries

Figure 5(a) shows the speedup of COLD compared to HLDB in the case of k-Nearest Neighbor queries for $D = 0.01$ and $k = \{1, 2, 4, 8, 16\}$. As described in Section 3.1.3, 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 smallest 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 and 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 5(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 $14ms$.

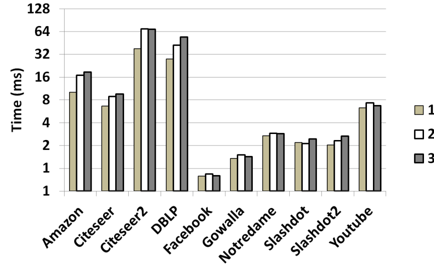
4.1.3 One-to-Many queries

COLD is the only SQL framework that supports *one-to-many queries*. Figure 6(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



(a) Top-k Range experiments for $k = 4$, second tertile and varying values of D

(b) Top-k Range experiments for $D = 0.01$, second tertile and varying values of k



(c) Top-k Range experiments for $D = 0.01$, $k = 4$ and varying ranges

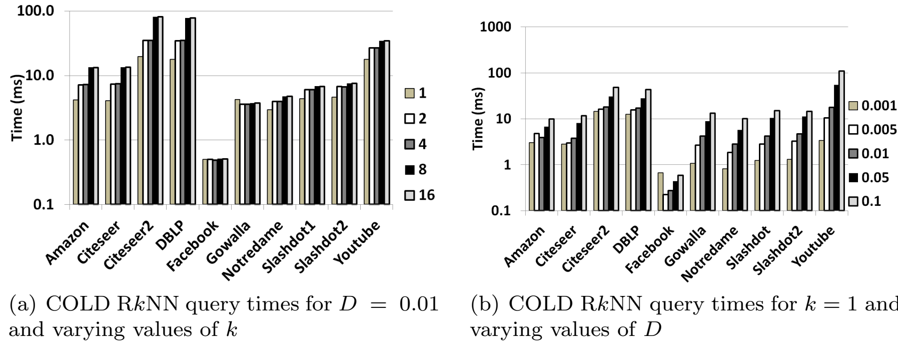
Fig. 7: Top-k Range experiments for COLD on the HDD

(those with the highest $|HL|/|V|$ ratio) that require $5601ms$ and $4170ms$ 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* $401ms$ *with a simple SQL query.*

4.1.4 Top-k Range queries

Again, COLD is the only SQL framework that supports top-k range queries. Since we have approximated the diameter of tested graphs (see Column 7 or Table 10) we have split the graph distances in 3 equal tertiles, hence creating 3 same-sized ranges per graph. The first range is $[0, 1st\ tertile)$, the second range is $[1s\ tertile, 2nd\ tertile)$ and the third range is $[2nd\ tertile, diameter + 1)$, whereas $1st\ tertile = 1/3\ graph\ diameter$ and $2nd\ tertile = 2/3\ graph\ diameter$.

Figure 7(a) presents the corresponding results for $k = 4$, the second tertile and varying values of D ($D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$). Although larger values of D correspond to slower queries, COLD still answers top-k range queries in less than $170ms$ for all datasets and values of D. Figure 7(b) presents results for $D = 0.01$, the second tertile and varying values of k



(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

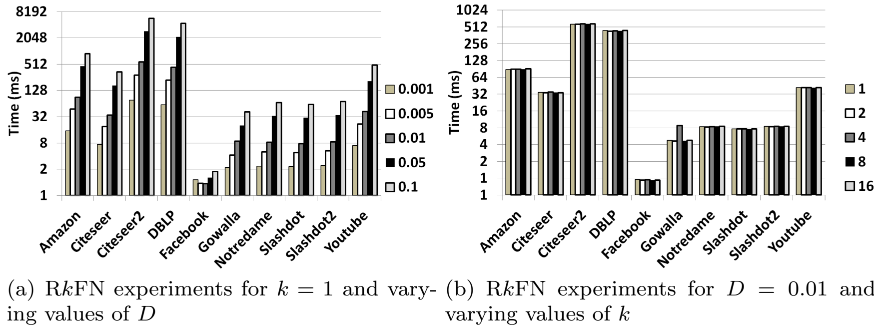
Fig. 8: RkNN Experiments on the HDD for COLD

($k = \{1, 2, 4, 8, 16\}$). Again, larger values of k translate to slower queries but COLD can still answer top-k range queries in less than $60ms$ for all datasets and values of k (except Citeseer2 and k=16 that requires $120ms$). Finally, Figure 7(c) presents results for $D = 0.01$, $k = 4$ and different ranges that correspond to the aforementioned tertiles of possible graph distances. As expected, moving to a larger tertile slows down queries but COLD can still answer top-k range queries in less than $70ms$ for all datasets and tertiles. Regarding the first tertile, top-k range queries may be answered in less than $39ms$.

4.1.5 Reverse k-Nearest Neighbor (RkNN) queries

For Reverse k-Nearest Neighbor experiments, we only report COLD's performance, since there is no other SQL framework that supports this specific type of 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 Section 3.1.6) 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 $kmax = 1$, 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 8(a) presents the results. In all cases, COLD provides excellent query times that are below $20ms$ for $k = 1$ in all datasets and never exceed $82ms$ even for $k = 16$.

In our second set of Reverse k-Nearest Neighbor experiments, we assess the performance of COLD for varying values of D. Figure 8(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 $D = 0.1$ ($109.3ms$). Thus, COLD offers excellent and stable performance regarding RkNN queries for all datasets and tested values of k and D.



(a) RkFN experiments for $k = 1$ and varying values of D

(b) RkFN experiments for $D = 0.01$ and varying values of k

Fig. 9: Reverse k-Farthest Neighbors experiments for COLD on the HDD

4.1.6 Reverse k-Farthest Neighbor (RkFN) queries

Again, COLD is the only SQL framework that supports Reverse k-Farthest Neighbor queries. Figure 9(a) presents the corresponding results for $k = 4$ and varying values of D ($D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$). As expected, the performance of RkFN queries is quite similar to the performance of *one-to-many* queries despite the additional `JOIN` operation with the *farres* DB table (see Section 3.1.7) and thus the corresponding performance decreases with increasing values of D. For all datasets and values of D, RkFN queries take less than $600ms$, except the Citeseer2 and DBLP datasets and $D \geq 0.05$. Figure 9(b) presents results for $D = 0.01$ and varying values of k ($k = \{1, 2, 4, 8, 16\}$). Results show that the value of k has minimal impact on query times, since the main bottleneck of the Reverse k-Farthest Neighbor queries is performing the initial one-to-many query before the `JOIN` operation. On all datasets and values of k, COLD RkFN queries take less than $584ms$.

4.2 Performance on SSD

Having established the performance characteristics of COLD in the HDD, in our second round of experiments, we repeat 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 512GB 2.5".

4.2.1 Vertex-to-vertex queries

Although the usage of SSD favors HLDB more than COLD (see Figure 10), 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 $0.9ms$ for all datasets and v2v queries, except

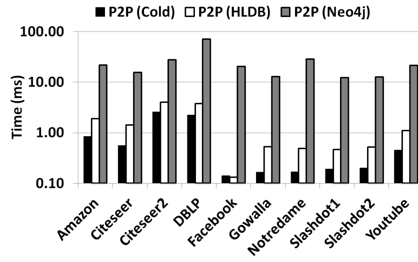
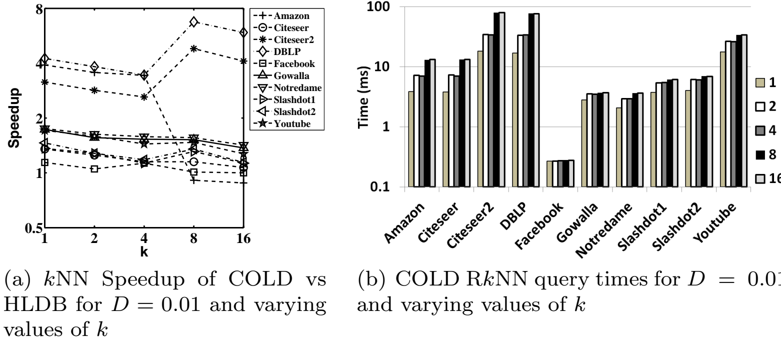


Fig. 10: *Vertex-to-vertex* queries for COLD, HLDB and Neo4j on the SSD



(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. 11: kNN and RkNN query performance on the SSD

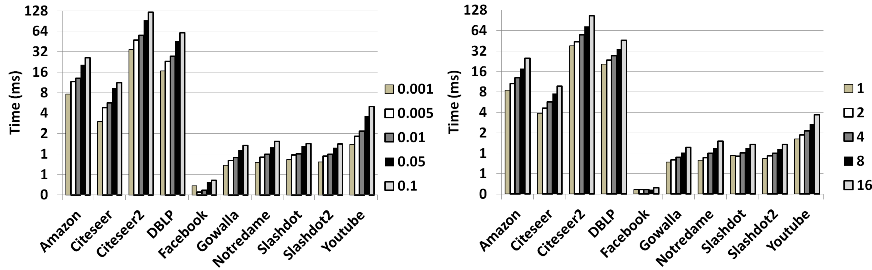
the Citeseer2 and DBLP datasets (those with the highest $|HL|/|V|$ ratio). But even then, vertex-to-vertex queries still require less than $2.6ms$ for COLD.

4.2.2 k-Nearest Neighbor (kNN) queries

Figure 11(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 $24.6ms$ even for $k = 16$.

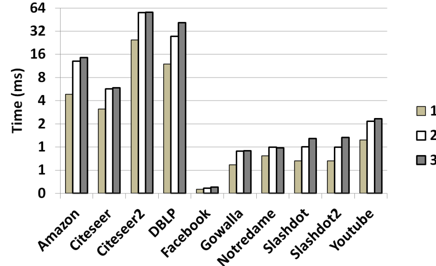
4.2.3 One-to-Many queries

Figure 6(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.



(a) Top-k Range experiments for $k = 4$, second tertile and varying values of D

(b) Top-k Range experiments for $D = 0.01$, second tertile and varying values of k



(c) Top-k Range experiments for $D = 0.01$, $k = 4$ and varying ranges

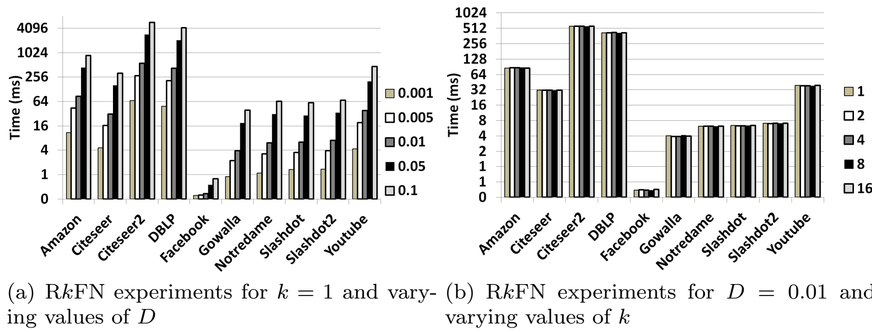
Fig. 12: Top-k Range experiments for COLD on the SSD

4.2.4 Top-k Range queries

Figure 12(a) presents the corresponding results for top-k range queries on the SSD for $k = 4$, the second tertile and varying values of D. The usage of SSD accelerates top-k range queries by 45 - 56% and thus, COLD answers such queries in less than $121ms$ for all datasets and values of D on the SSD. Figure 12(b) presents results for the SSD and $D = 0.01$, the second tertile and varying values of k ($k = \{1, 2, 4, 8, 16\}$). Likewise, the usage of SSD accelerates top-k range queries by 37 - 52% and hence, COLD can answer those queries on the SSD in less than $106ms$ for all datasets. Finally, Figure 7(c) presents results on the SSD, for $D = 0.01$, $k = 4$ and different ranges that correspond to the three tertiles of possible graph distances. Here, the usage of SSD accelerates top-k range queries by 43 - 61% and hence, this specific type of queries takes less than $56ms$ for all datasets and ranges.

4.2.5 Reverse k-Nearest Neighbor (RkNN) queries

Figure 11(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



(a) RkFN experiments for $k = 1$ and varying values of D

(b) RkFN experiments for $D = 0.01$ and varying values of k

Fig. 13: Reverse k-Farthest Neighbors experiments for COLD on the SSD

effectively minimized secondary storage utilization and thus adding a better secondary-storage medium provides only moderate benefits for RkNN queries.

4.2.6 Reverse k-Farthest Neighbor (RkFN) queries

We repeat the previous Reverse k-Farthest Neighbor experiments on the SSD. Figure 13(a) presents the corresponding results for $k = 4$ and varying values of D on the SSD. Similar to one-to-many queries, the impact of SSD is not very significant, since it only accelerates RkFN queries by 8 - 31%. Figure 13(b) presents results for $D = 0.01$ and varying values of k ($k = \{1, 2, 4, 8, 16\}$). Results show that again, the impact of SSD is not very significant, since it only accelerates Reverse k-Farthest Neighbor queries by only 17 - 21%.

4.3 Summary

Our experimentation has shown that our proposed COLD framework outperforms previous state-of-the-art HLDB in all performance benchmarks, including query performance, memory size and scalability. Using HDDs, COLD is 2 - 21× faster for *vertex-to-vertex* queries and 5 - 19× faster for kNN queries and the largest datasets. Using SSDs, COLD is 1.6 - 3.2× faster than HLDB for *vertex-to-vertex* and up to 6.75× faster for kNN queries. COLD also requires up to $4,444\times$ less storage space (indexes) and up to 188× 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× 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. Regarding the additional queries introduced in this paper in comparison to our original work of [23], top-k range queries may be answered in $170ms$ for all tested values of D, k and ranges. The usage of SSD further drops down this time to $121ms$. In the case of reverse k-farthest queries,

COLD may answer RkFN queries for all tested datasets and values of D and k in less than $600ms$ on the HDD (except the Citeseer2 and DBLP datasets and $D \geq 0.05$), whereas the usage of SSD lowers this time to $563ms$. Therefore, COLD is the only framework that may simultaneously answer all those different variations of queries with a few lines of SQL code, while providing performance which is fast enough for real-world applications.

5 Conclusions

This work presented COLD, a novel SQL framework for answering multiple 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 Reverse k-Nearest Neighbors, Reverse k-Farthest Neighbors, one-to-many and top-k range queries, not handled by any other secondary storage solution. This establishes COLD as a competitive database-driven framework for querying large-scale graphs.

This paper gives the complete 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 additional queries and use-cases.

Acknowledgements

This work was partially supported by the project "Research Programs for Excellence 2014-2016 / CitySense-ATHENA R.I.C." and the EU/Greece funded KRIPIS Action: MEDA Project. D. Pfoser's work was partially supported by the NGA NURI grant HM02101410004.

References

1. I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, and R. F. Werneck. Hldb: Location-based services in databases. In *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, pages 339–348, 2012.
2. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In *Proc. 10th International Symposium on Experimental Algorithms (SEA)*, pages 230–241. 2011.
3. I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *Proc. 20th Annual European Symposium on Algorithms (ESA)*, pages 24–35. 2012.
4. P. Afshani, G. S. Brodal, and N. Zeh. Ordered and unordered top-k range reporting in large data sets. In *Proc. Twenty-second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 390–400, 2011.
5. T. Akiba, Y. Iwata, K. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *Proc. 16th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 147–154, 2014.

6. T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 349–360, 2013.
7. T. Akiba, Y. Iwata, and Y. Yoshida. Pruned landmark labeling [online]. `https://github.com/iwiwi/pruned-landmark-labeling`, 2015.
8. R. Albert, H. Jeong, and A.-L. Barabási. The diameter of the world wide web. *CoRR*, cond-mat/9907038, 1999.
9. D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Proc. 10th DIMACS Implementation Challenge Workshop Graph Partitioning and Graph Clustering*, 2013.
10. H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. Route planning in transportation networks. *CoRR*, abs/1504.05140, 2015.
11. F. Borutta, M. A. Nascimento, J. Niedermayer, and P. Kröger. Monochromatic rknn queries in time-dependent road networks. In *Proc. Third ACM SIGSPATIAL International Workshop on Mobile Geographic Information Systems*, pages 26–33, 2014.
12. M. A. Cheema, Z. Shen, X. Lin, and W. Zhang. A unified framework for efficiently processing ranking related queries. In *Proc. 17th International Conference on Extending Database Technology (EDBT)*, pages 427–438, 2014.
13. E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proc. of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1082–1090, 2011.
14. E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proc. Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 937–946, 2002.
15. D. Delling, J. Dibbelt, T. Pajor, and R. Werneck. Public transit labeling. In *Proc. 14th International Symposium on Experimental Algorithms(SEA)*, pages 273–285. 2015.
16. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable route planning. In *Proc. 10th International Conference on Experimental Algorithms (SEA)*, pages 376–387, 2011.
17. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Robust distance queries on massive networks. In *Proc. 22th Annual European Symposium on Algorithms (ESA)*, pages 321–333, 2014.
18. D. Delling, A. V. Goldberg, and R. F. Werneck. Faster batched shortest paths in road networks. In *Proc. 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS)*, 2011.
19. D. Delling, A. V. Goldberg, and R. F. Werneck. Hub label compression. In *Proc. 12th International Symposium on Experimental Algorithms (SEA)*, pages 18–29, 2013.
20. D. Delling and R. F. Werneck. Customizable point-of-interest queries in road networks. *IEEE Trans. Knowl. Data Eng.*, 27(3):686–698, 2015.
21. D. Delling and R. F. F. Werneck. Better bounds for graph bisection. In *Proc. 20th Annual European Symposium on Algorithms (ESA)*, pages 407–418, 2012.
22. A. Efentakis. Scalable public transportation queries on the database. In *Proc. 19th International Conference on Extending Database Technology (EDBT)*, pages 527–538, 2016.
23. A. Efentakis, C. Efstathiades, and D. Pfoser. COLD. revisiting hub labels on the database for large-scale graphs. In *Proc. 14th International Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 22–39. 2015.
24. A. Efentakis and D. Pfoser. Optimizing landmark-based routing and preprocessing. In *Proc. 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science (CTS)*, 2013.
25. A. Efentakis and D. Pfoser. GRASP. extending graph separators for the single-source shortest-path problem. In *Proc. 22th Annual European Symposium on Algorithms (ESA)*, pages 358–370. 2014.
26. A. Efentakis and D. Pfoser. Rehub: Extending hub labels for reverse k-nearest neighbor queries on large-scale networks. *J. Exp. Algorithmics*, 21:1.13:1–1.13:35, 2016.
27. A. Efentakis, D. Pfoser, and Y. Vassiliou. Salt. a unified framework for all shortest-path query variants on road networks. In *Proc. 14th International Symposium on Experimental Algorithms (SEA))*, pages 298–311. 2015.

28. C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. In *Proc. Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SODA '01, pages 210–219, 2001.
29. C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. *J. Algorithms*, 53(1):85–112, 2004.
30. R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *Proc. 10th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 90–100, 2008.
31. R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proc. 7th International Workshop on Experimental Algorithms (WEA)*, pages 319–333, 2008.
32. H.-P. Hung, K.-T. Chuang, and M.-S. Chen. Efficient process of top-k range-sum queries over multiple streams with minimized global error. *IEEE Transactions on Knowledge & Data Engineering*, (10):1404–1419, 2007.
33. M. Jiang, A. W. Fu, R. C. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *PVLDB*, 7(12):1203–1214, 2014.
34. Y. Kumar, R. Janardan, and P. Gupta. Efficient algorithms for reverse proximity query problems. In *Proc. 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 39:1–39:10, 2008.
35. J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.
36. J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.
37. B. Liao, L. H. U, M. L. Yiu, and Z. Gong. Beyond millisecond latency $k$nn search on commodity machine. *IEEE Trans. Knowl. Data Eng.*, 27(10):2618–2631, 2015.
38. J. Liu, H. Chen, K. Furuse, and H. Kitagawa. An efficient algorithm for reverse furthest neighbors query with metric index. In *Proc. 21st International Conference on Database and Expert Systems Applications (DEXA): Part II*, pages 437–451, 2010.
39. Z. Luo, T. W. Ling, C.-H. Ang, S. Y. Lee, and B. Cui. Range top/bottom k queries in olap sparse data cubes. In *Proc. 12th International Conference on Database and Expert Systems Applications (DEXA)*, pages 678–687, 2001.
40. J. J. McAuley and J. Leskovec. Learning to discover social circles in ego networks. In *Proc. 26th Annual Conference on Neural Information Processing Systems*, pages 548–556, 2012.
41. PostgreSQL. The world's most advanced open source database. `http://www.postgresql.org/`, 2016.
42. M. Safar, D. Ibrahimi, and D. Taniar. Voronoi-based reverse nearest neighbor query processing on spatial networks. *Multimedia Systems*, 15(5):295–308, 2009.
43. J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *IEEE Trans. on Knowl. and Data Eng.*, 22(8):1158–1175, 2010.
44. C. Sheng and Y. Tao. Dynamic top-k range reporting in external memory. In *Proc. 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 121–130, 2012.
45. Y. Tao. A dynamic i/o-efficient structure for one-dimensional top-k range reporting. In *Proc. 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 256–265, 2014.
46. Q. T. Tran, D. Taniar, and M. Safar. Transactions on large-scale data- and knowledge-centered systems i. chapter Reverse K Nearest Neighbor and Reverse Farthest Neighbor Search on Spatial Networks, pages 353–372. Springer-Verlag, 2009.
47. S. Wang, M. A. Cheema, X. Lin, Y. Zhang, and D. Liu. Efficiently computing reverse k furthest neighbors. In *Proc. 32nd IEEE International Conference on Data Engineering (ICDE)*, pages 1110–1121, 2016.
48. S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou. Efficient route planning on public transportation networks: A labelling approach. In *Proc. 2015 ACM SIGMOD International Conference on Management of Data*, pages 967–982, 2015.
49. J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. In *Proc. 12th IEEE International Conference on Data Mining (ICDM)*, pages 745–754, 2012.

50. M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse nearest neighbors in large graphs. *IEEE Transactions on Knowledge and Data Engineering*, 18(4):540–553, 2006.
51. R. Zhong, G. Li, K.-L. Tan, and L. Zhou. G-tree: An efficient index for knn search on road networks. In *Proc. 22nd ACM International Conference on Conference on Information Knowledge Management (CIKM)*, pages 39–48. ACM, 2013.