Industrial and Applications Paper

Scalable Public Transportation Queries on the Database

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

ABSTRACT

Recent scientific literature focuses on answering Earliest Arrival (EA), Latest Departure (LD) and Shortest Duration (SD) queries in (schedule-based) public transportation networks. Unfortunately, most of the existing solutions operate in main memory, making the proposed methods hard to scale for larger instances and difficult to integrate in a multi-user environment. This work proposes PTLDB (Public Transporation Labels on the DataBase), a novel, scalable, pure-SQL framework for answering EA, LD and SD queries, implemented entirely on an open-source database system. Moreover, we formulate four new types of queries targeting public transportation networks, namely the Earliest Arrival and Latest Departure k-Nearest Neighbor (kNN) and One-to-many queries and propose novel ways to efficiently answer them within PTLDB. Our experimentation will show that the proposed solution is fast, scalable and easy to use, making our PTLDB framework a serious contender for use in real-world applications.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS; G.2.2 [**Graph Theory**]: Graph algorithms

General Terms

Algorithms, Design

Keywords

Transportation networks, kNN queries, One-to-many queries

1. INTRODUCTION

Recent algorithmic advances have been very efficient in solving vertex-to-vertex queries on graphs, for a variety of different graph types and instances. For road networks, latest papers focused on supporting additional types of shortest-path (SP) queries, including *one-to-all* (finding SP distances from a source vertex s to all other graph vertices) [8, 15], *one-to-many* (computing the SP distances between the source vertex s and all vertices of a set of targets T) [11, 15], *range* (find all nodes reachable from s within a given timespan) [15], *many-to-many* (calculate a distance table between two sets of vertices S and T) [11] and k-Nearest Neighbor (kNN) queries in [12, 17, 21]. For large-scale networks, the works of [4, 9, 20], proposed methods for solving vertex-to-vertex queries, whereas the works of [16] and [14] deal with more complex queries, such as k-Nearest Neighbor, Reverse k-Nearest Neighbor (RkNN) and one-to-many queries, either on main memory or inside a database. Finally, for schedule-based public transportation networks recent works, either expand existing shortest-path techniques originally used for road networks, or work directly on the provided timetable. Most of the recent methods targeting public transit networks are surveyed in the latest literature overview on transportation networks of [5].

Despite the inherent different characteristics between those three types of graph networks (road, large-scale and public transportation networks), the prevailing technique that excels at all of them is the 2-hop labeling or hub labeling (HL) algorithm of [18],[6], in which we store 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 (i) road networks [2, 3, 10], (ii) undirected, unweighted graphs in [4, 9, 20] and (iii) has also been extended to public transportation networks in [7] and [23]. The HL method has also been applied successfully for one-to-many, many-to-many and kNN queries in road networks [11, 12] and kNN and RkNN queries in the context of large-scale networks in [16].

Although hub labeling is an extremely efficient shortest-path technique on main memory, there are very few works that extend those results for secondary storage. HLDB [13] stores the precomputed labels for road networks in a comercial database system and translated the hub labeling, vertex-to-vertex distance query to plain SQL commands. Moreover, it showed how to answer kNN queries and k-best via points, again with simple SQL queries. The work of [20] proposed HopDB, a C++ customized solution that utilizes secondary storage during preprocessing for large-scale networks. Efentakis et al. proposed the COLD framework [14] that stores hub-labels for large-scale networks in an open-source database engine and answers vertex-to-vertex, one-to-many, kNN and RkNN queries, using SQL commands.

In this work, we focus on timetable, public transporta-

©2016, Copyright is with the authors. Published in Proc. 19th International Conference on Extending Database Technology (EDBT), March 15-18, 2016 - Bordeaux, France: ISBN 978-3-89318-070-7, on OpenProceedings.org. Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0

tion networks and present a novel database framework that may service several route-planning queries on such networks. Our pure-SQL, *PTLDB* (Public Transporation Labels on the DataBase) framework, extends the hub labeling technique for public transportation networks, as presented in Timetable Labeling (TTL) [23] and may answer multiple variations (Earliest Arrival, Latest Departure and Shortest-Duration) of vertex-to-vertex queries, entirely within a database. Moreover, we formulate four new types of route-planning queries for public transit, namely the *Earliest Arrival* and *Latest Departure* k-Nearest Neighbor (kNN) queries and their *one-to-many* versions, i.e., namely the *Earliest Arrival* and *Latest Departure one-to-many* queries. These newly defined public-transit queries may be very useful for a variety of applications utilizing public transport, such as mobile travel guides, transportation and urban planning or geomarketing applications. Our experimentation will show that the proposed *PTLDB* framework may answer all those different types of queries efficiently, while its implementation with an open-source database engine makes *PTLDB* very easy to integrate into existing multi-user, real-world applications. Additionally, each type of query may be answered with a simple SQL command, making our results easily reproducible by anyone. Thus, the *PTLDB framework is the only database solution that focuses on public-transportation networks*, while ensuring excellent performance that is fast-enough for real-time applications.

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

2. BACKGROUND AND RELATED WORK

In this section we will present related work, relative to the hub labeling method for directed weighted graphs $G(V, E, w)$, where V is the set of vertices, $E \subseteq VxV$ is the set of arcs and w is a positive weight function $E \rightarrow R^+$. We will also discuss the latest adaptations of this specific technique for public transportation networks.

2.1 Hub Labeling

In the 2-hop labeling or Hub Labeling (HL) algorithm of [18, 6], 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 vector 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 g, the set $L_f(s) \cap L_b(g)$ must contain at least one hub that is on the shortest $s - g$ path. To find the network distance $dist(s, g)$ between any two vertices s and g, a HL query seeks the hub $v \in L_f(s) \cap L_b(g)$ that minimizes the sum $dist(s, v) + dist(v, g)$. 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, 10]. In the case of large-scale graphs, the Pruned Landmark Labeling (PLL) algorithm of [4] orders vertices by degree and then during preprocessing, performs one BFS per graph vertex, starting from the highest-order / degree vertices. At each iteration, each individual BFS is pruned by using the hub labels calculated from the previous searches. Later, Delling et al. [9] improve the suggested vertex ordering and the storage of the hub labels for maximum compression. The HL method has also been extended to one-to-many, many-to-many and kNN queries on road networks in [11] and [12] respectively. The recent work of [16] has also proposed *ReHub*, a novel main-memory algorithm that extends the Hub Labeling approach to efficiently handle Reverse k-Nearest Neighbor (RkNN) queries on large-scale networks.

Regarding secondary-storage solutions, Jiang et al. [20] propose their HopDB algorithm for constructing an efficient HL index when the given graphs and the corresponding index are too large to fit into main memory. Abraham et al. [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. In [14], Efentakis et al. presented the pure-SQL *COLD* framework (COmpressed Labels on the Database) for answering multiple exact distance queries (vertex-to-vertex, kNN, RkNN and *one-to-many*) for large-scale networks, in a open-source database engine. Despite the fact that each type of query was translated to just few lines of SQL code, experiments have showed that COLD provides excellent performance and is thus, fast enough for real-time applications.

2.2 Public transportation networks

For methods targeting public transportation networks before 2015, one can refer to the latest related literature overview of [5]. In this section, we will mainly focus on the latest research works that appeared in 2015.

Recently, the hub labeling method has also been extended for public transportation networks, in Public Transit Labeling presented in [7] and Timetable Labeling (TTL) presented in [23]. Since our work builds on Timetable Labeling (TTL), we will follow the notation presented there. A schedule-based, public transportation network may be modeled as a multigraph G, where each vertex is associated with a station or stop (i.e, *"distinct locations where one may board a transit vehicle"* [7]) and each arc from a vertex u to a vertex v represents a trip b that departs from stop u at timestamp t_d and arrives at v at timestamp t_a. Thus, each arc e may be represented with a tuple $< u, v, t_d, t_a, b >$, where b, t_d, t_a are the trip, departure time and arrival time of e respectively and $t_a - t_d$ is the corresponding duration of the arc. The arc e is an outgoing arc for vertex-stop u and an incoming arc for vertex-stop v. Thus, there are multiple arcs connecting the same pair of vertices-stops that correspond to the different trips connecting those two stops together.

Timetable labelling (TTL) is a extension of Hierarchical Hub Labeling [3] for public transportation networks. The TTL index preprocessing, computes two sets of labels, $L_{in}(v)$ and $L_{out}(v)$, for each vertex-stop v, such that each label in $L_{in}(v)$ (or $L_{out}(v)$) is a tuple describing a "fast" path that ends at (starts from) v. TTL assumes there is a strict vertex ordering, which, defines the relative importance of each vertex with respect to the others. Hence, the rank $r(v)$ of a vertex v is an integer $\in [1, |V|]$. It is assumed that the vertex v ranks lower (i.e., is less important) than u, when $r(v) > r(u)$. When given a timetable graph G and a node order r, the TTL index can be uniquely constructed. The output of TTL preprocessing is two sets of labels, $L_{in}(v)$ and $L_{out}(v)$ for each vertex-stop v, whereas

each label contains tuples $< hub, t_d, t_a, pivot, b >$ representing a fast transit path between stops v and hub, passing through stop $pivot$, using trip b and departing at t_d and arriving at t_a. The pivot information is required for reconstructing the full path, when it is comprised of multiple trips and is set to $null$, when the corresponding tuple describes a direct trip between stops v and hub (i.e., $b \neq null$). Note, that the label sets $L_{in}(v)$ and $L_{out}(v)$ for a vertex v only contain paths between v and vertices of higher order.

The labels calculated during the TTL preprocessing may be used for answering the following three types of queries:

- *Earliest Arrival* (EA). Given two stops s and $g \in G$ and a starting timestamp t, the earliest arrival $EA(s, g, t)$ query seeks the path with the earliest arrival time among those paths that (i) start from s no sooner than t and (ii) end at g.
- *Latest Departure* (LD). Given two stops s and g and an ending timestamp t', the latest departure $LD(s, g, t')$ query seeks the path with the latest departure time among those paths that (i) start from u and (ii) end at v no later than t'.
- *Shortest Duration* (SD). Given two stops s and g, a starting t and an ending t' timestamp, the shortest duration $SD(s, g, t, t')$ query seeks the path with the shortest duration among those that (i) start from s no sooner than t and (ii) end at g no later than t'.

During the query phase, TTL only has to visit the labels of stops s and g (without accessing the original graph G) and select the best solution from three candidate cases: (i) Tuples in $L_{out}(s)$ where $hub = g$, (ii) Tuples in $L_{in}(g)$ where $hub = s$ and (iii) Combining all tuples $l_1 \in L_{out}(s)$ and $l_2 \in L_{in}(g)$ where $l_1.hub = l_2.hub$ and $l_1.ta <= l2.td$. The experimentation in [23], showed that TTL may answer EA, LD and SD queries in less than $30\mu s$, while requiring preprocessing time of less than $17min$ for all datasets.

In this work, based on the lessons learnt from the previous COLD database framework [14], that answers multiple distance queries on large-scale graphs, we will: (i) show how to efficiently store the labels created from Timetable labelling for public transportation networks into a database, (ii) translate the corresponding EA, LD and SD queries into simple SQL commands (iii) formulate four new types of queries for public transportation networks, namely the *Earliest Arrival* and *Latest Departure* k-Nearest Neighbor (kNN) and the *Earliest Arrival* and *Latest Departure one-to-many* queries and (iv) show how to efficiently answer those additional queries, by using simple SQL commands within the same database framework, implemented entirely on a popular, open-source database engine. The resulting *PTLDB* (Public Transporation Labels on the DataBase) framework will be described in the following section.

3. THE PTLDB FRAMEWORK

This section presents the PTLDB (Public Transporation Labels on the DataBase) database framework. *PTLDB* can answer multiple route-planning queries (Earliest Arrival [EA], Latest Departure [LD], Shortest Duration [SD] vertex-to-vertex, EA, LD k-Nearest Neighbor and EA, LD *one-to-many*) for public transportation networks using SQL commands. Since *PTLDB* builds on the Timetable Labeling

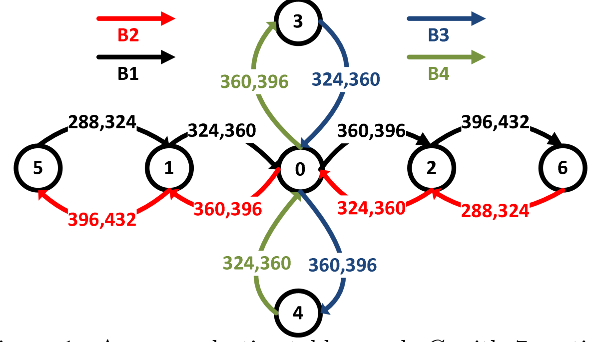


Figure 1: An example timetable graph G with 7 vertices (stops) and 4 trips (each highlighted with a different color). Timestamps are in 100s, i.e., 360=>36,000s (10:00h). Vertex 0 is the highest order vertex, followed by vertices 1,2,3,4

Table 1: The created labels for the example graph G

v	$\mathbf{L_{out}(v)}$ $< \mathbf{hub, t_d, t_a, pivot, b} >$	$\mathbf{L_{in}(v)}$ $< \mathbf{hub, t_d, t_a, pivot, b} >$
0	$< 0, 360, 360, -1, -1 >$	$< 0, 360, 360, -1, -1 >$
1	$< 0, 324, 360, 0, 1 >$	$< 0, 360, 396, 0, 2 >$
	$\mathbf{< 1, 324, 324, -1, -1 >}$	$< 1, 324, 324, -1, -1 >$
	$< 1, 396, 396, -1, -1 >$	$< 1, 396, 396, -1, -1 >$
2	$< 0, 324, 360, 0, 2 >$	$< 0, 360, 396, 0, 1 >$
	$\mathbf{< 2, 324, 324, -1, -1 >}$	$< 2, 324, 324, -1, -1 >$
	$< 2, 396, 396, -1, -1 >$	$< 2, 396, 396, -1, -1 >$
3	$< 0, 324, 360, 0, 3 >$	$< 0, 360, 396, 0, 4 >$
	$\mathbf{< 3, 396, 396, -1, -1 >}$	$< 3, 396, 396, -1, -1 >$
4	$< 0, 324, 360, 0, 4 >$	$< 0, 360, 396, 0, 4 >$
	$\mathbf{< 4, 396, 396, -1, -1 >}$	$< 4, 396, 396, -1, -1 >$
5	$< 0, 288, 360, 1, 1 >$	$< 0, 360, 432, 1, 2 >$
	$< 1, 288, 324, 1, 1 >$	$< 1, 396, 432, 1, 2 >$
	$\mathbf{< 5, 432, 432, -1, -1 >}$	$< 5, 432, 432, -1, -1 >$
6	$< 0, 288, 360, 2, 2 >$	$< 0, 360, 432, 2, 1 >$
	$< 2, 288, 324, 2, 2 >$	$< 2, 396, 432, 2, 1 >$
	$\mathbf{< 6, 432, 432, -1, -1 >}$	$< 6, 432, 432, -1, -1 >$

(TTL) [23], we will explain the basic concepts presented there. We chose PostgreSQL [22] for our implementation, given that it is a popular, open-source RDBMS. Although we use some PostgreSQL-specific data-types (namely arrays) and SQL extensions, we use only features included in its standard installation, without any third-party extensions.

3.1 Vertex-to-Vertex (v2v) queries

The *PTLDB* framework uses the labels generated by the Timetable Labeling (TTL) of [23] for public transportation networks. The respective TTL implementation (and the respective datasets) were made publicly available by the authors at [24]. To highlight the results of this process, the labels for the example timetable graph G of Figure 1 are shown in Table 1. The labels $L_{out}(v)$ and $L_{in}(v)$ for each vertex / stop v is a vector of tuples $< hub, t_d, t_a, pivot, b >$ sorted by hub, t_d (see Section 2). To answer vertex-to-vertex (v2v) queries between two stop s and g, TTL only has to visit the labels $L_{out}(s)$ and $L_{in}(g)$ and select the best solution from three candidate cases: (i) Tuples in $L_{out}(s)$ where $hub = g$, (ii) Tuples in $L_{in}(g)$ where $hub = s$ and (iii) Combining all tuples $l_1 \in L_{out}(s)$ and $l_2 \in L_{in}(g)$ where $l_1.hub = l_2.hub$ and $l_1.ta <= l2.td$. Although selecting between those three cases is trivial for a main memory algorithm, it is complex to adapt in a database context. Thus, we need to generate for every $v \in G$, some extra "dummy" tuples in both

$L_{out}(v)$ and $L_{in}(v)$ with $hub = v$ and $t_d = t_a$ for every DISTINCT (hub, t_d) combination existing in $L_{out}(u)$ and for every DISTINCT (hub, t_a) combination existing in $L_{in}(u)$. Those dummy tuples for our example graph are highlighted in bold. Note that, those dummy tuples are only a small fraction ($< 10\%$) of the total number of tuples and hence, they add minimal overhead to the PTLDB framework's performance. By generating those dummy tuples, we can ensure that each vertex-to-vertex query may be answered by combining exactly one tuple $l_1 \in L_{out}(s)$ and one tuple $l_2 \in L_{in}(g)$ where $l_1.hub = l_2.hub$ and $l_1.ta <= l2.td$, i.e., we unified the three separate cases of TTL query processing into one. Thus, the answer to the $EA(1, 1, 324)$ query is 324 by combining the tuples <1,324,324,...>, present in both $L_{out}(1)$ and $L_{in}(1)$.

Code 1: Vertex-to-vertex (v2v) queries for *PTLDB*

```
WITH outp AS
  (SELECT UNNEST(hubs) AS hub,
          UNNEST(tds) AS td,
          UNNEST(tas) AS ta
   FROM lout WHERE v=s),
inp AS
  (SELECT UNNEST(hubs) AS hub,
          UNNEST(tds) AS td,
          UNNEST(tas) AS ta
   FROM lin WHERE v=g)
/* For EA queries */
SELECT MIN(inp.ta)
/* For LD queries */
SELECT MAX(out.td)
/* For SD queries */
SELECT MIN(inp.ta-outp.td)
FROM outp,
     inp
WHERE outp.hub=inp.hub AND outp.ta<=inp.td
  /* For EA,SD queries */
  AND outp.td>=t
  /* For LD,SD queries */
  AND inp.ta<=t'
```

After generating the additional dummy tuples for simplifying the TTL vertex-to-vertex queries, we need to store the respective $L_{out}(v)$ and $L_{in}(v)$ labels in the database, as two separate DB tables denoted *lout* and *lin*, respectively. Similar to the previous work COLD [14], 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 *PTLDB* we store hubs, departure timestamps t_d and arrival timestamps t_a for a vertex (all ordered by hub, t_d) as arrays in three separate columns (i.e., hubs, tds and tas) in a single row. The resulting *lout* and *lin* DB tables are shown in Tables 2 and 3. Similar to COLD, this approach has considerable advantages: (i) The *lout* and *lin* DB tables have exactly $|V|$ rows (ii) Each of those DB tables has the column v as primary key, minimizing the size of the respective index. (iii) For any v2v query, *PTLDB* needs to access exactly two rows, regardless of the sizes of $|L_{out}(s)|$ and $|L_{in}(g)|$, thus minimizing the secondary-storage utilization, even working inside a database. In case of timetables changing depending on the weekday (e.g., weekdays vs weekends) or the time of the year (e.g., on holidays) in *PTLDB* we would need to have different versions of the *lout* and *lin* DB tables, for servicing each different period. Also note that in *PTLDB*, we do not need to store the pivot or the trip information, since if we wanted to reconstruct the full path, it would make more sense to store on the database the expanded path for each tuple generated by the TTL preprocessing. After all, this is another advantage of databases, also suggested by previous efforts like [1].

The resulting SQL commands for all types (Earliest Arrival, Latest Departure and Shortest Duration) of vertex-to-vertex queries for *PTLDB* are shown in Code 1, where the user may choose between the lines 12, 14 and 16 and lines 21, 23 depending on the specific type of query. We use Common Table Expressions (CTEs) for greater readability and we exploit the fact that PostgreSQL "*guarantees that parallel unnesting*" for *hubs*, *tds* and *tas* for each nested query "*will be in sync*", i.e., each tuple $< hub, td, ta >$ is expanded correctly since for the same v the respective arrays have the same number of elements[1]. It is obvious that the *PTLDB* vertex-to-vertex query is very simple, since it is implemented with just a few lines of SQL code and at the same time, it is highly optimized since it only has to fetch only one row from each *lout* and *lin* DB tables.

THEOREM 3.1.1. *The PTLDB v2v query is correct.*

PROOF. By adding the dummy tuples to the *lout* and *lin* DB tables, we can guarantee that the solution to any vertex-to-vertex query is a combination of one tuple $l1 \in lout$ and one tuple $l2 \in lin$ with $l1.hub = l2.hub$ and $l1.ta <= l2.td$ (Line 19). Considering the fact that PostgreSQL guarantees correct unnesting of the hubs, tds and tas arrays (line 2-4, 7-9) for the respective rows for u and v and the extra conditions specific for each type of query (Lines 12,14 and 16 and lines 21, 23) then the resulting *PTLDB* vertex-to-vertex query is correct. □

3.2 EA and LD kNN queries

The k-Nearest Neighbour (kNN) query, either for Euclidean space or for network databases, is a very well-studied problem in database systems due to its wide range of applications. Unfortunately, it has not been extended yet, to public transportation networks. To this propose, we formulate the Earliest Arrival and Latest Departure k-Nearest Neighbour (kNN) queries for schedule-based timetable networks, according to the following definitions:

- *Earliest Arrival kNN query* (EA-kNN). Given a stop q, a set of target-stops T and a starting timestamp t, the earliest arrival $EA{-}\mathrm{k}NN(q, T, t, k)$ query seeks the k-distinct stops $\in T$ with the earliest arrival time among the paths that (i) start from q no sooner than t and (ii) end in any stop $\in T$.

- *Latest Departurel kNN query* (LD-kNN). Given a stop q, a set of target-stops T and an ending timestamp t, the latest departure $LD{-}\mathrm{k}NN(q, T, t, k)$ query seeks the k-distinct stops $\in T$ with the latest departure time from stop q from among the paths that (i) start from q and (ii) end in any stop $\in T$ no later than t.

Both these type of queries may be used in a wide range of useful applications, such as an tourist deciding to visit the nearest Point of Interest (POI) using public transport

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

Table 2: The *lout* table used in *PTLDB* for the example graph G

v	hubs	tds	tas
...	...	...	...
1	$\{0,1,1\}$	$\{324,324,396\}$	$\{360,324,396\}$
...	...	...	...
4	$\{0,4\}$	$\{324,396\}$	$\{360,396\}$
...	...	...	...

Table 3: The *lin* table used for *PTLDB* for the example graph G

v	hubs	tds	tas
...	...	...	...
1	$\{0,1,1\}$	$\{360,324,396\}$	$\{396,324,396\}$
...	...	...	...
4	$\{0,4\}$	$\{360,396\}$	$\{396,396\}$
...	...	...	...

Table 4: The *ea_knn_naive* table for the example graph G, $T = \{4,6\}$ and $k = 1$

hub	td	vs	tds
0	360	{4,~~6~~}	{396,~~432~~}
2	396	{6}	{432}
4	396	{4}	{396}
6	432	{6}	{432}

(EA-kNN) or how a city visitor may determine his remaining time for finishing his breakfast, before reaching one of his preferred POI-destinations by 11:00 (LD-kNN). To the best of our knowledge, *this is the first time that these queries have been formalized for public transit networks* and therefore we are not aware of any previous approach tackling them. Throughout this work we assume that targets are not changing, which is a reasonable assumption for public transportation networks, since, e.g., for location based services we already know the stops that are located near attractive POIs or the most visited city-landmarks. In the following sections, we will show how to efficiently solve those queries within *PTLDB*. For our example graph G (Figure 1) and the remainder of this section, we assume that target stops are 4 and 6, i.e., $T = \{4,6\}$

3.2.1 Implementation

In this section, we will mainly discuss EA-kNN queries. The solution to LD-kNN queries will be directly analogous. Typically, to solve kNN queries with the hub labeling method, we need to group the L_{in} tuples of the targets by hub [1, 16, 14] and keep the k-best entries per hub. Unfortunately, this approach cannot be extended directly to public transportation networks, due to the condition $l_1.ta <= l2.td$ that must always hold. A naive solution to this problem, would be to group the L_{in} tuples of the targets per *hub* and t_d instead, and again keep the best distinct k-entries per *hub*, t_d ordered by t_a, with ties broken arbitrarily. The results of this process would then be stored in the DB table *ea_knn_naive*, with the data structure shown in Table 4. As seen there, for $k = 1$, $hub = 0$ and $td = 360$ we only need to keep the best entry that corresponds to $v = 4$ and $td = 396$. The primary key of *ea_knn_naive* table will be the (hub, td) combination. After building this table, the EA-kNN(q, T, t, k) query may be solved by the SQL of Code 2, that combines the row of *lout* DB table that corresponds to vertex q, with the *ea_knn_naive* table. Therefore the $EA{-}kNN(0, \{4,6\}, 360, 1)$ will have the correct answer $(4, 396)$, i.e., the NN of vertex 0 for departure time 360 or later is the vertex 4 with arrival time 396. As showcased in [14], it makes sense to create one large *ea_knn_naive* 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 this case, we only need to retrieve k-entries per (hub, td) combination and thus we only expand $vs[1:k]$ and $tas[1:k]$ (Lines 14-15) for $k < kmax$.

Code 2: EA-kNN naive query for *PTLDB*

```
1  WITH n1 AS
2    (SELECT v, hub, td, ta
3     FROM
4       (SELECT v AS v,
5               UNNEST(hubs) AS hub,
6               UNNEST(tds) AS td,
7               UNNEST(tas) AS ta
8        FROM lout
9        WHERE v=q) n1a
10    WHERE td >=t)
11 SELECT v2,MIN(n2.ta)
12 FROM n1,
13   (SELECT hub, td,
14           UNNEST(vs[1:k]) AS v2,
15           UNNEST(tas[1:k]) AS ta
16    FROM ea_knn_naive) n2
17 WHERE n1.hub=n2.hub
18   AND n2.td>=n1.ta
19 GROUP BY v2
20 ORDER BY MIN(n2.ta), v2
21 LIMIT k;
```

THEOREM 3.2.1. *The naive EA-kNN query is correct.*

PROOF. The naive EA-kNN query joins the $l1$ tuples in q row of DB table *lout*, with the $l2$ tuples of *ea_knn_naive* DB table with $l1.hub = l2.hub$ and $l1.ta <= l2.td$. Since for each individual (hub, td) combination the *ea_knn_naive* DB table stores the top-k (earliest arrival) entries, this ensures that the naive EA-kNN query provides correct results. ☐

Although the EA-kNN naive query is very simple, it cannot scale well for large metropolitan networks. In a realistic setting, multiple buses or trains leave the same *hub* every few minutes and therefore for each hub we will have multiple t_d entries. Thus, the size of *ea_knn_naive* DB table and its primary key index will vastly increase (even after keeping only the best k-entries per hub, t_d). The number of rows that should be joined will also grow, making queries too slow for real-time applications. Hence, we must further group entries per hub and create a condensed *knn_ea* DB table. Ideally, each tuple contained in the q row of *lout* DB table should be joined with only a single row in the *knn_ea* table, during a EA-kNN query. To achieve that, we can group hub entries per hour of departure, i.e., making a separate entry per hub and hour of departure for the available timestamp ranges of t_d in *lin* DB table. The resulting *knn_ea* table will have the data structure showcased in Table 5 and the combination of $hub, dephour$ as a primary key. For a specific hub and hour, e.g., $hub = 0, dephour = 10$, (i) the columns $tds{-}exp$, $vs{-}exp$ and $tas{-}exp$, contain ALL tuples of *lin*

Table 5: The *knn_ea* table data structure

hub	dephour	vs	tas	tds-exp	vs-exp	tas-exp
0	0	top-k entries (v) $hub = 0, hour \geq 1$	top-k entries (ta) $hub = 0, hour \geq 1$	all entries (td) $hub = 0, 0 \leq hour \leq 1$	all entries (v) $hub = 0, 0 \leq hour \leq 1$	all entries (ta) $hub = 0, 0 \leq hour \leq 1$
0	1	top-k entries (v) $hub = 0, hour \geq 2$	top-k entries (ta) $hub = 0, hour \geq 2$	all entries (td) $hub = 0, 1 \leq hour \leq 2$	all entries (v) $hub = 0, 1 \leq hour \leq 2$	all entries (ta) $hub = 0, 1 \leq hour \leq 1$
...	...	...	...	...	...	...
1	0	top-k entries (v) $hub = 1, hour \geq 1$	top-k entries (ta) $hub = 1, hour \geq 1$	all entries (td) $hub = 1, 0 \leq hour \leq 1$	all entries (v) $hub = 1, 0 \leq hour \leq 1$	all entries (ta) $hub = 1, 0 \leq hour \leq 1$
1	1	top-k entries (v) $hub = 1, hour \geq 2$	top-k entries (ta) $hub = 1, hour \geq 2$	all entries (td) $hub = 1, 1 \leq hour \leq 2$	all entries (v) $hub = 1, 1 \leq hour \leq 2$	all entries (ta) $hub = 1, 1 \leq hour \leq 1$

for targets T, with $hub = 0$ and t_d between 10:00 and 11:00 ordered by t_d, whereas (ii) the columns vs and tas contain only the best top-k (earliest arrival) distinct entries for targets T and $hub = 0$, $t_d \geq$11:00.

Thus, the optimized EA-kNN query must implement those two separate cases: (i) Expanding the $l2$ tuples for a specific hub between e.g., 10:00 and 11:00 contained in DB columns $tds-exp$, $vs-exp$ and $tas-exp$ (still checking that $l1.ta <= l2.td$ for those entries) and (ii) Expanding the $l3$ tuples that leave the specific hub after 11:00. As showcased earlier, both cases are included in a single row per hub of the *knn_ea* DB table. The resulting query is shown in Code 3. For combining those aforementioned cases we still have to use the UNION operator (Line 30) and for increasing performance, the JOIN between the *lout* and *knn_ea* DB tables happens AFTER expanding the *lout* tuples for q row and BEFORE expanding the tuples in *knn_ea* DB table (Lines 1-18). Note, that if each row in *lout* DB table contains on average $|L_{out}|/|V|$ tuples, then the optimized EA-kNN query will always access at most $|L_{out}|/|V|$ rows from the *knn_ea* DB table, thus minimizing secondary storage utilization.

THEOREM 3.2.2. *The construction of the knn_ea DB table and the corresponding EA-kNN query are correct.*

PROOF. For the EA-kNN query (q, T, t), assume there is a tuple $l1 =< h, td, ta >$ with $td \geq t$ for the query vertex q, included in the *lout* DB table. The EA-kNN query will join this tuple with exactly one row in the *knn_ea* DB table for which $hub = h$ and $dephour = FLOOR(ta/3600)$. This specific *knn_ea* row contains (i) all $l2$ tuples of targets T for $hub = h$ and $dephour$ between $FLOOR(ta/3600)$ and $FLOOR(ta/3600)+1$ (we still need to check for those entries that $l1.ta <= l2.td$) but (ii) only the best top-k (earliest arrival) distinct l3 tuples for targets T that leave $hub = h$ after $FLOOR(ta/3600) + 1$. There is no need to search for any other tuples for $hub = h$ after $FLOOR(ta/3600) + 1$ because all other tuples will have worst arrival time than the $l3$ tuples. Also, there is no need to look for any other tuples for $hub = h$ before $FLOOR(ta/3600)$, because the $l1$ trip arrives at hub h after $FLOOR(ta/3600)$. Since the EA-kNN query will similarly join all tuples in *lout* DB that leave query vertex q after timestamp t, the resulting EA-kNN query is correct. □

Considering the choice of an hour as the tuning parameter for grouping the *knn_ea* DB table entries, we could have chosen any other valid time interval to that purpose. In fact, we have even experimented with other intervals (smaller or larger than an hour) but (i) when smaller intervals are used, the respective *knn_ea* table has more rows (which makes queries slower) and (ii) when larger intervals are used (e.g., 3 hours) the number of tuples stored in $tds - exp$, $vs - exp$ and $tas - exp$ columns vastly increases, which counteracts the benefit of the smaller number of total rows. Thus, a time interval of an hour seems like the best compromise between those two scenarios and worked well for all tested datasets. However, it can be tuned according to the specific use-cases and user needs for a particular public transit network.

In the case of LD-kNN queries, the corresponding *knn_ld* table will have a similar data structure to the *knn_ea* table. There are some main differences though: (i) Entries are grouped by hub and hour of ARRIVAL, i.e., for a specific hub and hour, e.g., $hub = 0, arrhour = 10$, the columns $tds-exp$, $vs-exp$ and $tas-exp$, contain all tuples of lin for targets T, with $hub = 0$ and t_a between 10:00 and 11:00 ordered by t_d. Likewise, the combination of $hub, arrhour$ will be the primary key. (ii) The columns vs and tds (and not tas as before) contain only the best top-k (latest departure) distinct entries for targets T and $hub = 0$, $t_a \leq$10:00. The corresponding LD-kNN query (see Code 4) will also be slightly different (e.g., `MIN(n3.ta)`, `MIN(n2.ta)` will be replaced by `MAX(n1_td)` or the `DESC` ordering) but it will still offer the same performance benefits as before.

3.3 EA and LD One-to-many queries

Similar to kNN, we formulate the Earliest Arrival and Latest Departure One-to-many queries for schedule-based timetable networks, according to the following definitions:

- *Earliest Arrival One-to-many query* (EA-OTM). Given a stop q, a set of target-stops T and a starting timestamp t, the earliest arrival $EA - OTM(q, T, t)$ query seeks the earliest arrival time for all target-stops for trips that start from q no sooner than t.
- *Latest Departurel One-to-many query* (LD-OTM). Given a stop q, a set of target-stops T and an ending timestamp t, the latest departure $LD - OTM(q, T, t)$ query seeks the latest departure times for trips starting from q and end in any stop $\in T$ no later than t.

Again the EA-OTM and LD-OTM queries have a wide range of useful applications, such as transportation planning (e.g., find faraway stops) or geomarketing applications (e.g., nearby what stop one must build a franchise store to be more easily reachable by clients). To the best of our knowledge, *this is also the first time that these queries have been formalized for public transportation.* How to efficiently solve them within *PTLDB*, will be shown in the following.

For answering the EA-OTM query, we need to build a new *otm_ea* DB table that will have the data structure showcased in Table 6. In the *otm_ea* table the columns *hub*, *dephour* (= hour of departure), $tds - exp$, $vs - exp$ and $tas - exp$ are identical to the *knn_ea* DB table and

Code 3: EA-kNN and EA-OTM queries for *PTLDB*

```
1 WITH n1 AS
2   (SELECT v,hub, td, ta
3    FROM
4      (SELECT v,
5              UNNEST(hubs) AS hub,
6              UNNEST(tds) AS td,
7              UNNEST(tas) AS ta
8       FROM lout
9       WHERE v=q) n1a
10   WHERE td >=t),
11     n1b AS
12  (SELECT n1bb.*,
13          n1.ta AS n1_ta
14          n1.td AS n1_td
15   /* EA-$k$NN query */
16   FROM knn_ea n1bb,n1
17   /* EA-OTM query */
18   FROM otm_ea n1bb,n1
19   WHERE n1bb.hub=n1.hub
20     AND n1bb.dephour=FLOOR(n1.ta/3600))
21 SELECT v2,MIN(ta)
22 FROM (
23        (SELECT v2,MIN(n3.ta) AS ta
24         FROM
25           (SELECT
26           /* EA-$k$NN query */
27           UNNEST(tas[1:k]) AS ta,
28           UNNEST(vs[1:k]) AS v2
29           /* EA-OTM query */
30           UNNEST(tas) AS ta,
31           UNNEST(vs) AS v2
32            FROM n1b) n3
33         GROUP BY v2
34         ORDER BY MIN(n3.ta),v2
35         /* EA-$k$NN query */
36         LIMIT k
37         )
38      UNION
39        (SELECT n2.v2,MIN(n2.ta) AS ta
40         FROM
41           (SELECT n1_ta,
42                   UNNEST(tds_exp) AS td,
43                   UNNEST(vs_exp) AS v2,
44                   UNNEST(tas_exp) AS ta
45            FROM n1b) n2
46         /* Check for l1.ta<=l2.td */
47         WHERE n1_ta<=n2.td
48         GROUP BY n2.v2
49         ORDER BY MIN(n2.ta),v2
50         /* EA-$k$NN query */
51         LIMIT k
52         )) S53
53 GROUP BY v2
54 ORDER BY MIN(ta),v2
55 /* EA-$k$NN query */
56 LIMIT k;
```

Code 4: LD-kNN and LD-OTM queries for *PTLDB*

```
1 WITH n1 AS
2   (SELECT v,hub,td,ta
3    FROM
4      (SELECT v,
5              UNNEST(hubs) AS hub,
6              UNNEST(tds) AS td,
7              UNNEST(tas) AS ta
8       FROM lout
9       WHERE v=q) n1a),
10     n1b AS
11  (SELECT n1bb.*,
12          n1.ta AS n1_ta,
13          n1.td AS n1_td
14   /* LD-$k$NN query */
15   FROM knn_ld n1bb,n1
16   /* LD-OTM query */
17   FROM otm_ld n1bb,n1
18   WHERE n1bb.hub=n1.hub
19     AND n1bb.arrhour=FLOOR(t/3600))
20 SELECT v2,MAX(td)
21 FROM (
22        (SELECT v2,MAX(n3.n1_td) AS td
23         FROM
24           (SELECT n1_td, n1_ta,
25                   /* LD-$k$NN query */
26                   UNNEST(tds[1:k]) AS td,
27                   UNNEST(vs[1:k]) AS v2
28                   /* LD-OTM query */
29                   UNNEST(tds) AS td,
30                   UNNEST(vs) AS v2
31            FROM n1b) n3
32         WHERE n3.td>=n1_ta
33         GROUP BY v2
34         ORDER BY MAX(n3.n1_td) DESC, v2
35         /* LD-$k$NN query */
36         LIMIT k
37         )
38      UNION
39        (SELECT n2.v2,MAX(n2.n1_td) AS td
40         FROM
41           (SELECT n1_td,n1_ta,
42                   UNNEST(tds_exp) AS td,
43                   UNNEST(vs_exp) AS v2,
44                   UNNEST(tas_exp) AS ta
45            FROM n1b) n2
46         WHERE n2.td>=n1_ta
47           AND n2.ta<=t
48         GROUP BY n2.v2
49         ORDER BY MAX(n2.n1_td) DESC, v2
50         /* LD-$k$NN query */
51         LIMIT k
52         )) S53
53 GROUP BY v2
54 ORDER BY MAX(td) DESC, v2
55 /* LD-$k$NN query */
56 LIMIT k;
```

Table 6: The *otm_ea* table data structure

hub	dephour	vs	tas	tds-exp	vs-exp	tas-exp
0	0	best entry per target (v) $hub = 0, hour \geq 1$	best entry per target (ta) $hub = 0, hour \geq 1$	all entries (td) $hub = 0, 0 \leq hour \leq 1$	all entries (v) $hub = 0, 0 \leq hour \leq 1$	all entries (ta) $hub = 0, 0 \leq hour \leq 1$
0	1	best entry per target (v) $hub = 0, hour \geq 2$	best entry per target (ta) $hub = 0, hour \geq 2$	all entries (td) $hub = 0, 1 \leq hour \leq 2$	all entries (v) $hub = 0, 1 \leq hour \leq 2$	all entries (ta) $hub = 0, 1 \leq hour \leq 1$
...	...	...	...	...	...	...
1	0	best entry per target (v) $hub = 1, hour \geq 1$	best entry per target (ta) $hub = 1, hour \geq 1$	all entries (td) $hub = 1, 0 \leq hour \leq 1$	all entries (v) $hub = 1, 0 \leq hour \leq 1$	all entries (ta) $hub = 1, 0 \leq hour \leq 1$
1	1	best entry per target (v) $hub = 1, hour \geq 2$	best entry per target (ta) $hub = 1, hour \geq 2$	all entries (td) $hub = 1, 1 \leq hour \leq 2$	all entries (v) $hub = 1, 1 \leq hour \leq 2$	all entries (ta) $hub = 1, 1 \leq hour \leq 1$

the combination $(hub, dephour)$ again serves as the primary key. The only difference is in the *vs* and *tas* columns, where we must store the best tuple (earliest arrival) per vertex for the following hours, instead of only the top-k entries over all targets (as in *knn_ea* table), i.e., the *vs* and *tas* columns will store at-most $|V|$ tuples per row, instead of only k. Although this makes the resulting EA-OTM query slower, its SQL implementation is practically identical to the EA-kNN query (see Code 3). We only need to replace the *knn_ea* table with *otm_ea* (Line 16), remove the `LIMIT k` clauses (Lines 36,51,56) and use `UNNEST(tas), UNNEST(vs)` (Lines 30,31) instead of the lines 27,28.

Likewise, for LD-OTM queries we need to build the corresponding *otm_ld* DB table that follows the same structure as *knn_ld* DB table, except that in the *vs* and *tds* columns, we must store the best tuple (latest departure) per vertex for the PREVIOUS hours, instead of only the top-k entries over all targets (as in *knn_ld* table). The respective LD-OTM query is very similar to the previous LD-kNN query, as showcased in Code 4.

Conclusively, for any query vertex q (containing on average $|L_out|/|V|$ tuples), then the proposed kNN and *One-to-many* queries will always access at most $|L_out|/|V|$ rows from the respective *knn* or *otm* DB tables. Thus, it will be hard to achieve better secondary storage utilization inside a database. It is important to note that once we load the TTL labels and create the *lout* and *lin* DB tables, all the auxiliary DB tables within *PTLDB* (namely the *knn_ea*, *knn_ld*, *otm_ea* and *otm_ld*) may also be created by simple SQL commands (the corresponding queries were omitted due to space restrictions). This fact also demonstrates that *PTLDB* is truly a pure-SQL framework for servicing multiple route-planning queries on public transportation graphs. In the following experimentation section, we will benchmark *PTLDB*'s performance for various real-world datasets.

4. EXPERIMENTAL EVALUATION

To assess the performance of *PTLDB* on various public transportation 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. In our experiments, we use the same 11 public transportation networks from [19] , as in Timetable Labeling (TTL) [23], where "*each dataset records the timetable of the public transportation network of a major city or country on a weekday*" [23]. The characteristics of these networks and the necessary TTL preprocessing (using the vertex ordering files provided by its authors) for creating the labels are presented in Table 7. The graphs' average degree is between 53 and 413 and the TTL algorithm creates $630 - 7,230$ tuples per vertex, requiring $4.5 - 353.6s$ for the labels' construction.

Table 7: Public transportation graphs statistics

Graph	|V|	|E|	Avg degr.	|HL|/|V|	TTL Preproc. Time (s)
Austin	2K	317K	119	1,600	11.3
Berlin	12K	2,081K	153	1,734	184.7
Budapest	5K	1,446K	252	2,486	54.4
Denver	10K	7,11K	75	1,190	27.3
Houston	10K	1,113K	113	2,196	72.6
Los Angeles	15K	1928K	127	2,572	194.5
Madrid	4K	1,913K	413	7,230	338.5
Roma	9K	2,281K	258	4,370	353.6
Salt Lake City	6K	330K	53	630	4.5
Sweden	51K	4,072K	76	775	179.1
Toronto	10K	3,300K	305	2,987	262.1

PTLDB was implemented in PostgeSQL 9.3.6, 64bit with the same settings used in [14] (8192Mb *shared buffers* and 64Mb *temp buffers*). We conducted experiments belonging to the following query types: (i) Earliest Arrival (EA), Latest Departure (LD) and Shortest Duration (SD) *vertex-to-vertex*, (ii) Earliest Arrival (EA) and Latest Departure kNN and (iii) Earliest Arrival (EA), Latest Departure (LD) *one-to-many*. For each experiment, we used 1,000 random start vertices (and goal vertices for vertex-to-vertex queries), reporting the average running time. Starting timestamps for EA and SD queries are randomly selected from the first quarter of timestamp ranges, whereas ending timestamps for LD and SD queries are randomly selected from the fourth quarter of timestamp ranges, to ensure that in the majority of the cases we actually get trip results that service a particular type of query. Contrarily, selecting timestamps randomly from all available ranges would significantly lower query times, since a significant percent of those queries would provide no results (no trip would fill the suggested criteria). Before each experiment, we restart the PostgreSQL server for clearing its internal query cache and we also clear the operating system's cache for accurate benchmarking. All kNN and one-to-many charts are plotted in logarithmic scale. Note that (i) *PTLDB* is the only pure-SQL framework tailored for servicing public transportation queries and (ii) to the best of our knowledge there is no previous work or any working system trying to tackle EA, LD kNN and one-to-many queries for such networks. Thus, we only present our results, since there is no previous secondary-storage work for comparison.

4.1 Performance on HDD

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

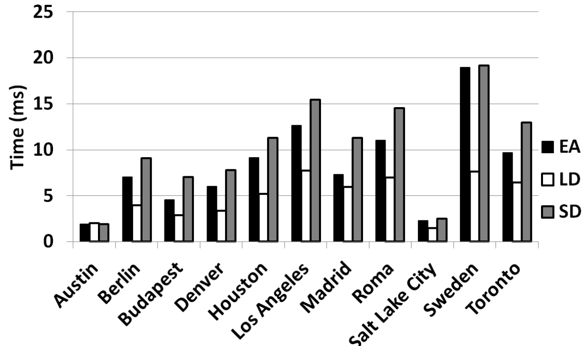


Figure 2: EA, LD and SD v2v queries on a HDD

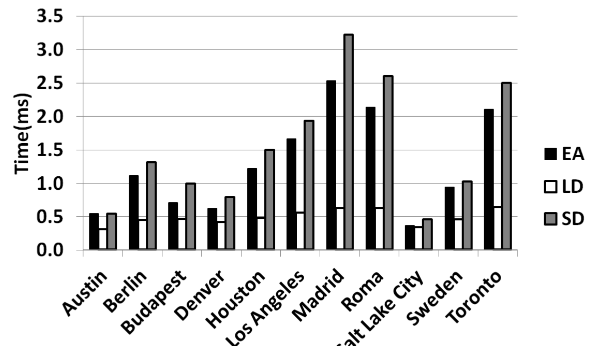


Figure 7: EA, LD and SD v2v queries on a SSD

4.1.1 Vertex-to-vertex queries

Figure 2 shows results for vertex-to-vertex (v2v) queries for *PTLDB*. Results show that LD queries are 35% faster than EA queries, because in the LD queries we select timestamps from the fourth quarter of timestamp ranges where there are less trips than the beginning of the day (as in EA queries). SD queries are on average 26% slower than EA queries, due to the increased complexity of the query. In all cases, EA and SD queries take less than $19.2ms$ and LD queries take less than $7.7ms$, which is an considerable achievement, since even main memory solutions (before TTL) would require a few ms for vertex-to-vertex queries and the suggested datasets [23]. Moreover, we may answer such queries with a simple SQL command inside a database, which ensures scalability, regardless of the numbers of users or the size of the datasets and with a performance that is fast enough for real-time online applications.

4.1.2 kNN queries

In this section, we provide the *PTLDB*'s results for EA and LD kNN queries. Similar to previous works [14], we will experiment with varying values of k and *target density* D, i.e., the ratio $|T|/|V|$, where T is the set of target-stops in the graph and $|V|$ is the total number of vertices. As explained in Section 3.2.1, for database frameworks it makes sense to create large *knn* tables for the maximum value $kmax$ of k that will be serviced by the respective framework. Thus, we have created two versions of kNN DB tables for *PTLDB*, one for $kmax = 4$ and one for $kmax = 16$. Then, the kNN 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 $k = 16$.

In our first set of kNN experiments, we compare our optimized EA-kNN and LD-kNN queries (see Codes 3, 4) in comparison to the corresponding naive kNN implementation (Code 2) for varying values of k. Results are presented in Figure 3. Results show that the optimized versions are $11 - 53\times$ faster than their naive counterparts. Thus, it really pays off to group tuples in the *knn_ea* (and *knn_ld*) DB tables by departure (arrival) hour. For the remainder of the paper, we will only provide results for the optimized EA-kNN and LD-kNN queries, since those queries provide significantly superior performance.

Figure 4 shows the absolute times of optimized EA-kNN and LD-kNN queries for the same scenario, i.e., for $D = 0.01$ and varying values of k. Results show that the EA-kNN queries require $<64ms$ for all values of k (except the highest ratio $|HL|/|V|$ dataset of Madrid and $k = 8, 16$). LD-kNN queries are even faster, requiring less than $32ms$ on all cases.

In our third set of kNN experiments, we assess the performance of *PTLDB* for varying values of D. For each value for D, we have build separate versions of *knn_ea* and *knn_ld* DB tables for $D \cdot |V|$ targets selected at random from each dataset and $kmax = 4$. Figure 5 shows results for $k = 4$ and $D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$. Results show, that although *PTLDB*'s performance degrades for larger values of D, kNN queries may still be answered in less than $128ms$ (with the exception of Madrid for EA queries and Toronto for LD queries and $D = 0.1$). For the smaller datasets (Austin, Berlin, Budapest, Denver, Houston, Los Angeles, Salt Lake City, even Sweden) kNN queries always take less than $32ms$. Moreover, EA-kNN queries are more robust to increasing values of D than LD-kNN queries that perform significantly worse for denser targets (i.e., for $D = 0.1$). Conclusively, the *PTLDB* framework provides excellent kNN query performance for all values of D and k.

4.1.3 One-to-Many queries

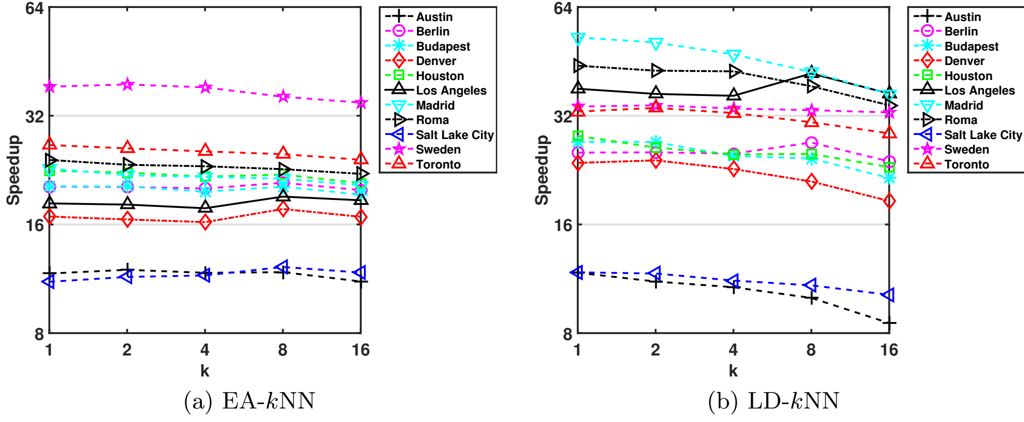
In our third round of experiments, we assess the performance of *PTLDB* for one-to-many queries. Figure 6 presents the corresponding results for varying values of D ($D = \{0.001, 0.005, 0.01, 0.05, 0.1\}$). *PTLDB* answers EA-OTM queries in less than $512ms$ for all datasets and values of D (except the Madrid and Toronto datasets that require $1084ms$ and $751ms$ respectively for $D = 0.1$). For LD-OTM queries *PTLDB* requires less than $256ms$ for all datasets and values of D (except the Madrid, Roma and Toronto datasets that require $303ms$, $325ms$ and $349ms$ respectively for $D = 0.1$). Note, that for such high values of D, the *one-to-many* query almost degrades to the *one-to-all* query and hence, it cannot get any faster on a secondary storage device.

4.2 Performance on SSD

Having established *PTLDB*'s performance in the HDD, we repeat most previous experiments on a SSD (a SATA3 Crucial CT512MX100SSD1 MX100 512GB 2.5") to measure how the secondary-storage device type impacts results.

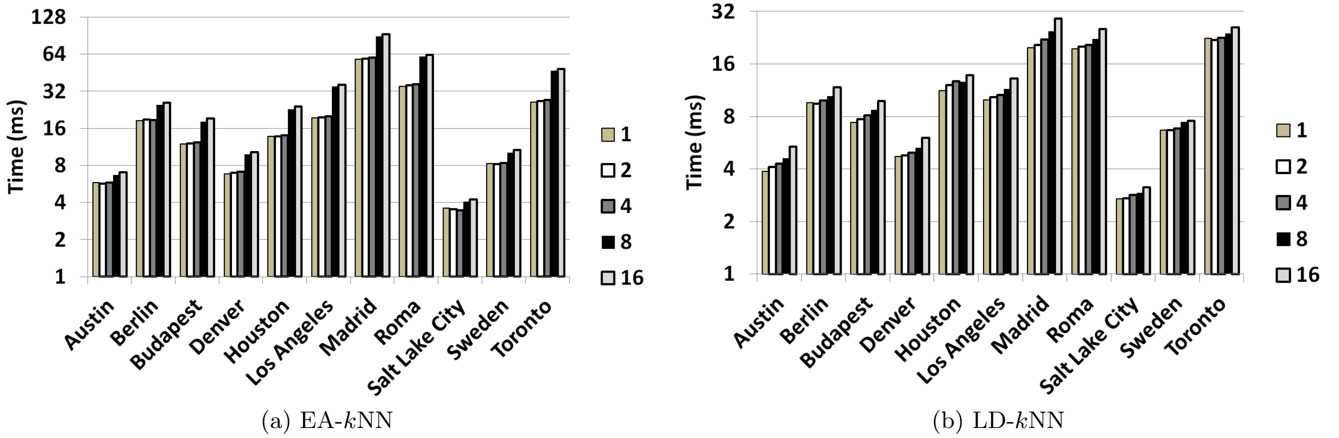
4.2.1 Vertex-to-vertex queries

Results for all variants of vertex-to-vertex queries on the SSD are shown in Figure 7. Results show that by using the SSD, *PTLDB* is 3 - 20× faster for EA, 6 - 17× faster for LD and 3 - 19× faster for SD vertex-to-vertex queries. Thus, EA queries may now be answered in less than $2.5ms$, SD queries may now be answered in less than $3.2ms$ and LD queries may now be answered in less than $0.6ms$. Conclusively, the usage



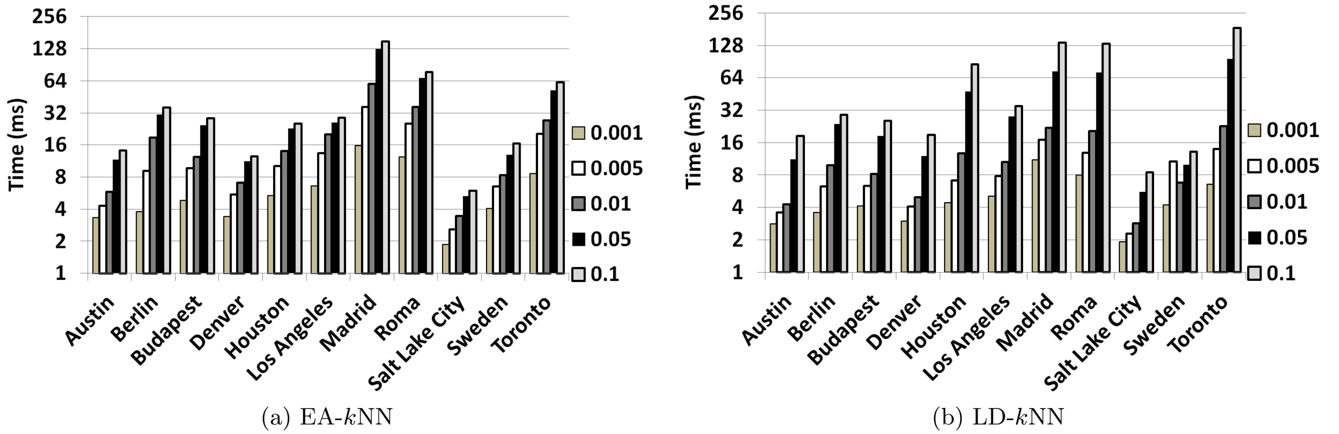
(a) EA-kNN (b) LD-kNN

Figure 3: Speedup of optimized kNN queries, in comparison to the naive versions for $D = 0.01$ and varying values of k



(a) EA-kNN (b) LD-kNN

Figure 4: kNN queries for $D = 0.01$ and varying values of k



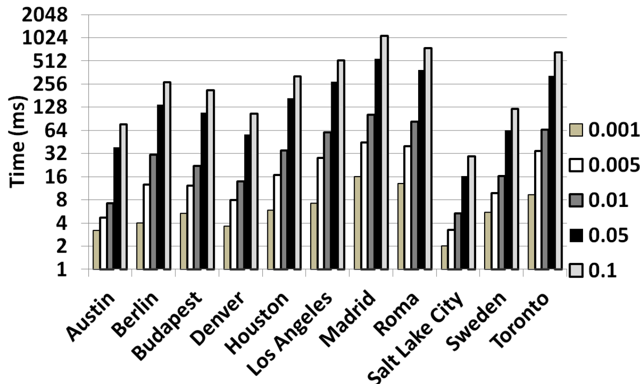
(a) EA-kNN (b) LD-kNN

Figure 5: kNN queries for $k = 4$ and varying values of D

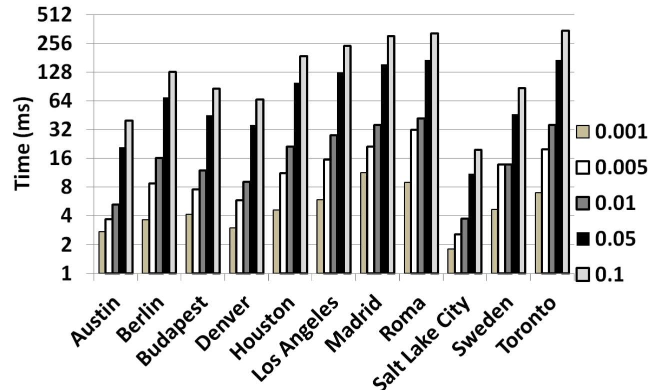
of SSD benefits significantly all vertex-to-vertex variations within *PTLDB* and therefore *PTLDB* may easily be used for public-transit real-time applications and such queries, since query times always require less than $3.2ms$.

4.2.2 *kNN and One-to-many queries*

In this section, we repeated all the kNN and one-to-many experiments performed in Sections 4.1.2 and 4.1.3 on the SSD. Results for kNN queries, $D = 0.01$ and varying values of k are presented in Figure 8. Results show that for kNN queries, the usage of the SSD does not provide any further benefits (in fact sometimes the SSD performs slightly worse), meaning that in *PTLDB* we have effectively minimized secondary storage utilization for kNN queries and thus, adding a faster storage medium adds no performance benefits. The same pattern was encountered on all experiments, for different values of D or k, including the respective one-to-many queries and therefore the resulting figures are omitted.

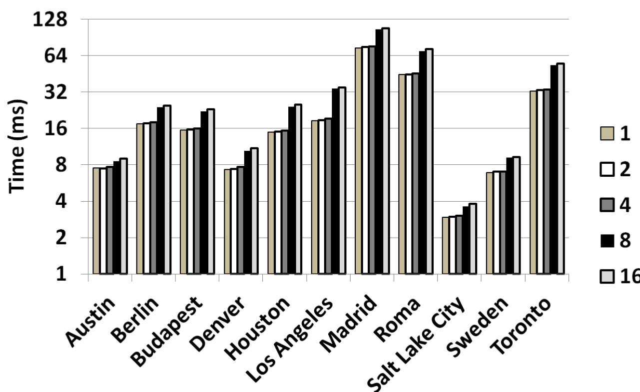


(a) EA-OTM

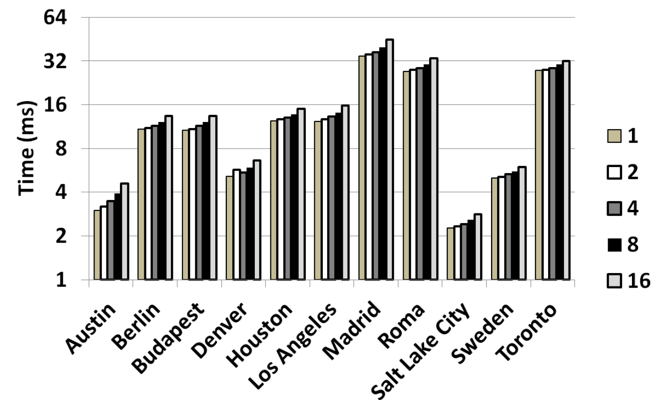


(b) LD-OTM

Figure 6: One-to-many queries for varying values of D



(a) EA-kNN



(b) LD-kNN

Figure 8: kNN queries for $D = 0.01$ and varying values of k on the SSD

4.3 Summary

Our experimentation has shown that our proposed *PTLDB* framework provides excellent performance for all public transportation planning queries. Using HDDs, *PTLDB* may answer vertex-to-vertex queries in less than $19.2ms$ For SSDs, this time drops down to $3.2ms$. For the newly formulated EA and LD kNN queries, *PTLDB* requires less than $64ms$ and $32ms$, for $k = 16$ and $D = 0.01$ for the vast majority of the tested datasets. Even the EA and LD One-to-many queries require less than $512ms$ and $256ms$ respectively, for most datasets and varying values of D. Regarding memory requirements, *PTLDB* is very modest, since all DB tables and primary key indexes, including the *lout*, *lin*, *knn_ea*, *knn_ld* (for all values of D and $kmax = 4, 16$) and the *otm_ea*, *otm_ld* tables (for all available values of D) for all tested datasets, require less than $12GB$. Hence, *PTLDB* may easily scale to even significantly larger datasets. Overall, not only *PTLDB* is the only pure-SQL framework tailored for multiple public-transportation queries, offering excellent performance for real-time applications but the simplicity of its SQL queries, makes its integration with existing real-world applications very easy and seamless.

5. CONCLUSION

This work presented *PTLDB*, a novel SQL framework for answering multiple route-planning queries for public transportation graphs on a database. Our results showed that *PTLDB* provides excellent query performance, minimum secondary storage utilization and maximum scalability. Moreover, we have extended kNN and one-to-many queries for public transportation networks and proposed how to efficiently answer them within *PTLDB*, with a few lines of SQL code. This establishes *PTLDB* as a competitive database-driven solution for querying public transportation networks.

The paper gives the complete design and implementation details of *PTLDB* using a popular, open-source database engine, along with the exact SQL queries used in our implementation. This easily allows the replication of our results and might provide the necessary foundation for other researchers to expand the *PTLDB* framework towards handling additional types of queries and novel use-cases. In terms of future work, currently the *PTLDB* framework aims at optimizing travel times, without taking the number of transfers as an additional optimization criterion. Integrating this additional constraint would further improve the use-cases and marketability of the *PTLDB* framework.

Acknowledgments

This work was partially funded by the project "Research Programs for Excellence 2014-2016 / CitySense-ATHENA R.I.C." The author would also like to thank the authors of Timetable Labeling (TTL) [23].

6. REFERENCES

[1] I. Abraham, D. Delling, A. Fiat, A. V. Goldberg, and R. F. Werneck. Hldb: Location-based services in databases. In *SIGSPATIAL GIS*. ACM, November 2012.

[2] I. Abraham, D. Delling, A. Goldberg, and R. Werneck. A hub-based labeling algorithm for shortest paths in road networks. In P. Pardalos and S. Rebennack, editors, *Experimental Algorithms*, volume 6630 of *Lecture Notes in Computer Science*, pages 230–241. Springer Berlin Heidelberg, 2011.

[3] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In L. Epstein and P. Ferragina, editors, *Algorithms – ESA 2012*, volume 7501 of *Lecture Notes in Computer Science*, pages 24–35. Springer Berlin Heidelberg, 2012.

[4] T. Akiba, Y. Iwata, and Y. Yoshida. 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*, pages 349–360, 2013.

[5] 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.

[6] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 937–946, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[7] D. Delling, J. Dibbelt, T. Pajor, and R. Werneck. Public transit labeling. In E. Bampis, editor, *Experimental Algorithms*, volume 9125 of *Lecture Notes in Computer Science*, pages 273–285. Springer International Publishing, 2015.

[8] D. Delling, A. V. Goldberg, A. Nowatzyk, and R. F. Werneck. Phast: Hardware-accelerated shortest path trees. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 921–931, Washington, DC, USA, 2011.

[9] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Robust distance queries on massive networks. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 321–333, 2014.

[10] D. Delling, A. V. Goldberg, and R. F. Werneck. Hub label compression. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, pages 18–29, 2013.

[11] D. Delling, A. V. Goldberg, and R. F. F. Werneck. Faster batched shortest paths in road networks. In *ATMOS*, pages 52–63, 2011.

[12] D. Delling and R. F. Werneck. Customizable point-of-interest queries in road networks. In *21st SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2013, Orlando, FL, USA, November 5-8, 2013*, pages 490–493, 2013.

[13] D. Delling and R. F. F. Werneck. Better bounds for graph bisection. In *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, pages 407–418, 2012.

[14] A. Efentakis, C. Efstathiades, and D. Pfoser. Cold. revisiting hub labels on the database for large-scale graphs. In C. Claramunt, M. Schneider, R. C.-W. Wong, L. Xiong, W.-K. Loh, C. Shahabi, and K.-J. Li, editors, *Advances in Spatial and Temporal Databases*, volume 9239 of *Lecture Notes in Computer Science*, pages 22–39. Springer International Publishing, 2015.

[15] A. Efentakis and D. Pfoser. GRASP. Extending graph separators for the single-source shortest-path problem. In A. S. Schulz and D. Wagner, editors, *Algorithms - ESA 2014*, volume 8737 of *Lecture Notes in Computer Science*, pages 358–370. Springer Berlin Heidelberg, 2014.

[16] A. Efentakis and D. Pfoser. Rehub. extending hub labels for reverse k-nearest neighbor queries on large-scale networks. *CoRR*, abs/1504.01497, 2015.

[17] A. Efentakis, D. Pfoser, and Y. Vassiliou. Salt. a unified framework for all shortest-path query variants on road networks. In E. Bampis, editor, *Experimental Algorithms*, volume 9125 of *Lecture Notes in Computer Science*, pages 298–311. Springer International Publishing, 2015.

[18] C. Gavoille, D. Peleg, S. Pérennes, and R. Raz. Distance labeling in graphs. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '01, pages 210–219, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[19] GoogleTransitDataFeed. PublicFeeds. List of publicly-accessible transit data feeds [online]. `https://code.google.com/p/googletransitdatafeed/wiki/PublicFeeds`, 2015.

[20] 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.

[21] B. Liao, L. U, M. Yiu, and Z. Gong. Beyond millisecond latency k nn search on commodity machine. *Knowledge and Data Engineering, IEEE Transactions on*, 27(10):2618–2631, Oct 2015.

[22] PostgreSQL. The world's most advanced open source database [online]. `http://www.postgresql.org/`, 2015.

[23] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou. Efficient route planning on public transportation networks: A labelling approach. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 967–982, New York, NY, USA, 2015. ACM.

[24] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou. Timetable labelling [online]. `http://sourceforge.net/projects/ttl2015/`, 2015.