Indexing Trajectories for Travel-Time Histogram Retrieval

Robert Waury
Aalborg University
Aalborg, Denmark
rwaury@cs.aau.dk

Christian S. Jensen
Aalborg University
Aalborg, Denmark
csj@cs.aau.dk

Satoshi Koide
Nagoya University
Nagoya, Japan
koide@db.is.i.nagoya-u.ac.jp

Yoshiharu Ishikawa
Nagoya University
Nagoya, Japan
ishikawa@i.nagoya-u.ac.jp

Chuan Xiao
Nagoya University
Nagoya, Japan
chuanx@nagoya-u.jp

ABSTRACT

A key service in vehicular transportation is routing according to estimated travel times. With the availability of massive volumes of vehicle trajectory data, it has become increasingly feasible to estimate travel times, which are typically modeled as probability distributions in the form of histograms. An earlier study shows that use of a carefully selected, context-dependent subset of available trajectories when estimating a travel-time histogram along a user-specified path can significantly improve the accuracy of the estimates. This selection of trajectories cannot occur in a pre-processing step, but must occur online—it must be integrated into the routing itself. It is then a key challenge to be able to select very efficiently the "right" subset of trajectories that offer the best accuracy when the cost of a route is to be assessed. To address this challenge, we propose a solution that applies novel indexing to all available trajectories and that then is capable of selecting the most relevant trajectories and of computing a travel-time distribution based on these trajectories. Specifically, the solution utilizes an in-memory trajectory index and a greedy algorithm to identify and retrieve the relevant trajectories. The paper reports on an extensive empirical study with a large real-world GPS data set that offers insight into the accuracy and efficiency of the proposed solution. The study shows that the proposed online selection of trajectories can be performed efficiently and is able to provide highly accurate travel-time distributions.

1 INTRODUCTION

Vehicular transportation is an important global phenomenon that impacts the lives of virtually all of us. We rely on it for mobility, and we are affected by congestion, accidents, and air and noise pollution. Its influence can be expected to continue into the foreseeable future. For example, in the European Union alone, more than 75% of all freight transport and more than 80% of passenger transport rely on the road networks [8]. The availability of high-resolution GPS trajectories allows for reliable map-matching to a road network. The resulting trajectories are called *network-constrained trajectories* (NCT) and can be used to obtain travel-time estimates for paths in the network, thus making transportation more predictable, safe, and environmentally friendly.

When using such a data set, the most straightforward approach to computing a travel-time estimate for a path is to compute a real-valued estimate for each segment in the path and then sum up these to obtain an estimate for the full path. This approach can be refined by collecting travel-time histograms for each segment and then combine them by means of convolution to obtain a travel-time histogram for the full path. This improves the accuracy of estimates since travel times are better modeled as distributions than real valued. Further, the distributions often do not follow a parameterized distribution, e.g., normal or uniform, and are therefore better estimated with histograms. This segment level approach can also be extended to computing different histograms for different times of day, e.g., the 96 15-minute intervals of the day, to account for changing congestion throughout the day. These histograms can be used as edge weights by routing algorithms to compute better results. All of the above approaches, however, only consider travel-time estimates at the segment level. These approaches fail to take into account factors like the times it takes to pass through intersections, going straight or turning left or right, which are hard to model accurately. An earlier study [26] shows that travel-time estimates for a given path can be improved considerably when they are computed from trajectories that strictly follow the path, as opposed to computing them from segment-level estimates. This type of path-based estimate relies on efficiently processing *strict path queries* (SPQ) as proposed by Krogh et al. [14], which is a query on a trajectory set that only returns trajectories which traversed a given path without detours.

We propose a system that can compute time-varying and personal travel-time histograms for any path in a network based on a large trajectory set. It would be infeasible and impractical to pre-compute and store these time-varying and personal weights for any path in a network before routing occurs. For example, given even a moderately sized road network of a million segments, for all 15-minute windows, nearly a 100 million histograms would be needed to just cover every single segment, with the storage requirements increasing dramatically when considering larger path lengths. We therefore obtain the weights for a path on-the-fly by expressing them as a series of SPQs, which we can efficiently process using our in-memory NCT index. If any of these sub-queries fails to retrieve a sufficient number of matching trajectories, we apply a greedy algorithm that relaxes the SPQ's predicates until the retrieved trajectory set has a specified cardinality. Since performance is crucial in our setting, we also implement a cardinality estimator for SPQs to prevent unnecessary index traversals. We also show that carefully choosing the initial set of SPQs increases the accuracy of the path weights and increases the performance of the query. We perform extensive experiments using a real-world trajectory data set containing 1.4 million trajectories from Northern Denmark, which shows that our approach is suitable for real-time applications.

The main contributions of this paper are the following:

- An adapted NCT index that supports efficient computation of travel-time histograms for SPQs.
- A greedy algorithm that enables efficient processing of any SPQ in periodic time intervals.
- A cardinality estimator for SPQs.
- A detailed analysis of the accuracy and performance of the solution and its components.

The rest of the paper is structured as follows. Section 2 provides an overview of prior work, preliminaries, and a detailed problem description. Section 3 describes the query processing method, while Section 4 details the construction and use of the NCT index. Section 5 outlines the experimental setup and the evaluation metrics. Section 6 reports on the results of the experiments, and Section 7 concludes.

2 PROBLEM FORMULATION

This section provides an overview of prior work, preliminaries, and a problem definition.

2.1 Related Work

We review approaches to travel-time estimation and then, we review network-constrained trajectory indexing with a focus on indexes supporting SPQs.

2.1.1 Travel-Time Estimation. Earlier studies on travel-time estimation compute histograms for single segments [15], which still requires to model turn costs [27], or for short pre-defined paths with considerable traffic [4], which are then convolved at query time. In our approach, travel-times are computed for sub-paths instead of only for individual segments. This approach implicitly handles turn costs within sub-paths, and turn costs only need to be modeled explicitly in-between sub-paths if applicable. Other approaches based on tensor decomposition [25], support vector regression [28], variance-entropy-based clustering [29], or deep neural networks [24] have also been proposed. But they either do not provide travel-time distributions or do not provide estimates for specific paths but only for origin-destination pairs.

2.1.2 NCT Indexing. Several indexes for network-constrained trajectories based on R-trees [5, 6, 9] or B+-trees [20] have been proposed, but they are often only optimized for range queries or nearest neighbor queries.

Two indexes have been proposed to support strict path queries, NETTRA [14] and the SNT-index [12]. NETTRA is a disk-based index designed to answer SPQs with minimal I/O and also supports efficient updates of the index, but may return false positives due to hash collisions. The SNT-index uses data structures adapted from string matching to efficiently identify matching trajectories. This index was originally designed to retrieve all matching trajectory IDs in a given time interval that fulfill the SPQ requirement. We extend it to accommodate travel-time retrieval as well.

2.2 Network Graph & Trajectories

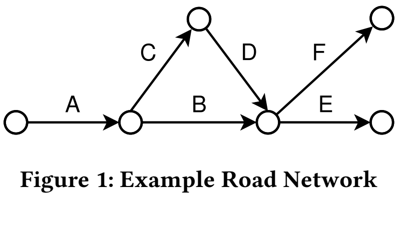


Figure 1: Example Road Network

Table 1: Example of $\mathcal{F}$ and Function *estimateTT*

e	*c*	*z*	*sl*	*l*	*estimateTT*
A	*motorway*	*rural*	110	900	29.5 s
B	*primary*	*city*	50	120	8.6 s
C	*secondary*	*city*	30	40	4.8 s
D	*secondary*	*city*	30	80	9.6 s
E	*primary*	*city*	50	100	7.2 s
F	*primary*	*rural*	80	800	36.0 s

A spatial network is modeled as a directed graph $G = (V, E, \mathcal{F})$, where V is a vertex set, $E \subseteq V \times V$ is a set of edges that represent road segments, and $\mathcal{F} : E \rightarrow Cat \times Z \times SL \times L$ is a set of functions, where Cat is the set of road categories, Z is the set of different types of zones the segments are located in, SL is the set of speed limits in kilometers per hour (or $\frac{1000}{3600}$ meters per second), and L is the set of segment lengths in meters. From this we can derive the function $estimateTT(e_i) = 3.6\frac{\mathcal{F}(e_i).l}{\mathcal{F}(e_i).sl}$ that returns the traversal time in seconds if the segment is traversed at the speed limit. This function is used as a fallback so that we can return a result even if no data is available for a segment. Every edge $e \in E$ has a category that captures the road type of the segment it represents and a zone type describing its location. Figure 1 shows the graph representation of the road network we use in examples. Table 1 shows the mapping of each segment to categories $c \in Cat = \{motorway, primary, secondary\}$ and zones $z \in Z = \{city, rural\}$.

A traversable sequence of segments $P = \langle e_0, e_1, \ldots, e_{l-1}\rangle$ is called a path, with $|P| = l$. A sub-path $\langle e_i, \ldots, e_{j-1}\rangle$, with $0 \le i < j \le l$, of P is denoted as $P[i, j)$. The set of trajectories is given as $\mathcal{T} \subseteq \mathcal{D} \times \mathcal{U} \times \mathcal{S}$, where $\mathcal{D}$ is the set of all trajectory ids, $\mathcal{U}$ is the set of all drivers. Further, $\mathcal{S} : \mathbb{N}_l \rightarrow E \times \mathcal{TS} \times C$ is the domain of functions from the set consisting of the first l natural numbers to the range of triples consisting of an edge $e \in E$, a timestamp $t \in \mathcal{TS}$, and a time duration $TT \in C$. This domain of functions encodes finite sequences of length l.

A trajectory $tr \in \mathcal{T}$ of a user u with the id d is therefore denoted as (d, u, s), where $s \in \mathcal{S}$ is a sequence of 3-tuples:

$$s = \langle (e_0, t_0, TT_0), (e_1, t_1, TT_1), \ldots, (e_{l-1}, t_{l-1}, TT_{l-1})\rangle,$$

where $t_0, .., t_{l-1}$ are the timestamps when a segment was entered with $\forall i \forall j (i < j \Rightarrow t_i < t_j)$, $TT_i > 0$ is the duration of the traversal of e_i, and l is the number of segments traversed.

The path of trajectory tr is called P_{tr}, and its starting time is $tr.t_0$. The duration function $Dur(tr, P) = TT_0 + TT_1 + \ldots + TT_{l-1}$ returns the sum of all segment traversal times a^P_{tr} of a path P by a trajectory. If a trajectory path P_{tr} does not contain P as a sub-path , $Dur(tr, P)$ is undefined. A trajectory set in our example road network from Figure 1 is shown below:

$$tr_0 : (0, u_1) \rightarrow \langle (A, 0, 3), (B, 3, 4), (E, 7, 4)\rangle$$
$$tr_1 : (1, u_2) \rightarrow \langle (A, 2, 4), (C, 6, 2), (D, 8, 4), (E, 12, 5)\rangle$$
$$tr_2 : (2, u_2) \rightarrow \langle (A, 4, 3), (B, 7, 3), (F, 10, 6)\rangle$$
$$tr_3 : (3, u_1) \rightarrow \langle (A, 6, 3), (B, 9, 3), (E, 12, 4)\rangle$$

2.3 Travel-Time Query

To address the shortcomings of the segment-level approach, we employ the strict path query $Q = spq(P, I, f, \beta)$ that returns a travel-time histogram H. The histogram can be derived from the traversal times of the set of trajectories $\mathcal{T}^P \subseteq \mathcal{T}$ that traverse path P without stops or detours in the time interval I, and fulfill additional filter predicates f:

$$\mathcal{T}^P = \{tr \in \mathcal{T} | \exists i, j\ (P_{tr}[i, j) = P \wedge tr.s.t_i \in I \wedge f(tr))\},$$

where $I = [t_s, t_e)$ denotes a temporal predicate with a size $\alpha = t_e - t_s$ and β is a cardinality requirement for $\mathcal{T}^P$, i.e., we only proceed if $|\mathcal{T}^P| \geq \beta$. If β is omitted all eligible trajectories are retrieved. The temporal predicate can either cover a fixed time interval, e.g., all trajectories from December 1st 2017 until May 1st 2018, or a periodic time-of-day interval denoted as $I^R = \langle \ldots, [t_s - 24\ hours, t_e - 24\ hours), [t_s, t_e), [t_s + 24\ hours, t_e + 24\ hours), \ldots, [t_s + n\ (24\ \ hours), t_e + n\ (24\ \ hours)) \rangle$, e.g., all trajectories from 8:00 until 8:30 on every day. Parameter f is an additional non-temporal filter predicate that trajectories in $\mathcal{T}^P$ have to fulfill, e.g., being from a specified driver.

Using such a query Q for a typical trip path, which can consist of dozens of segments, may not return a sufficient number of trajectories to derive accurate travel-time estimations. To address this problem, we split Q into k sub-queries $\langle Q_1, Q_2, \ldots, Q_k \rangle = \langle spq(P_1, I_1, f_1, \beta), spq(P_2, I_2, f_2, \beta), \ldots, spq(P_k, I_k, f_k, \beta) \rangle$ that return the trajectory sets $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k\}$, where P_i are sub-paths that partition P. These can then be used to compute a set of k histograms $\{H_1, H_2, \ldots, H_k\}$ if $\forall i\ |\mathcal{T}_i| \geq \beta$. Their convolution we call $H = H_1 * H_2 * \ldots * H_k$, where $*$ is the discrete convolution operator and H is a travel-time histogram that covers the full path P. The intuition behind partitioning into k sub-queries is, that different sub-paths often provide better estimates with different predicates, e.g., user predicates mainly improve accuracy outside of cities [26]. Another advantage of partitioning the query is the increased number of eligible trajectories.

How this partitioning into sub-queries is performed and how the sub-queries are processed is discussed in Sections 3 and 4.

An example query for our example trajectory set could be $Q = spq(\langle A, B, E \rangle, [0, 15), u = u_1, 2)$. This would return $\mathcal{T}^P = \{tr_0, tr_3\}$ yielding a histogram with $H = \{[10, 11)\colon 1; [11, 12)\colon 1\}$ since $Dur(tr_0, \langle A, B, E \rangle) = 11$ and $Dur(tr_3, \langle A, B, E \rangle) = 10$. But if a larger cardinality is required, Q could be split into two queries $Q_1 = spq(\langle A, B \rangle, [0, 15), \emptyset, 3)$ and $Q_2 = spq(\langle E \rangle, [0, 15), \emptyset, 3)$ that yield the histograms $H_1 = \{[6, 7)\colon 2; [7, 8)\colon 1\}$ and $H_2 = \{[4, 5)\colon 2; [5, 6)\colon 1\}$, from which the convolution $H = \{[10, 11)\colon 4; [11, 12)\colon 4; [12, 13)\colon 1\}$ can be obtained.

3 QUERY PROCESSING

This section describes the architecture of the system, the processing of travel-time queries, and the greedy algorithm used for relaxing sub-query predicates.

3.1 Architecture

Figure 2 shows the overall system architecture, where boxes with dotted lines indicate pre-existing components, dashed lines indicate modified components, and solid lines indicate new components. At first, a GPS data set is map-matched off-line to trajectories and loaded into the modified SNT-index consisting of a collection of temporal indexes and a spatial index.

Once the trajectory set is loaded, a user is able to dispatch a strict path query Q to the `Sub-query Module` where the query is initially partitioned into k sub-queries by the `Query Partitioner` according to a simple heuristic called π, e.g., sub-paths of a fixed length, or sub-paths that have the same segment category. Each sub-query is then assigned temporal and trajectory filter predicates. Next, the `Cardinality Estimator` uses the `Histogram Store` and the `SNT-Index` to estimate the cardinality $\hat{\beta}$ of the trajectory set $\mathcal{T}_i$ returned by the sub-query $spq(P_i, I_i, f_i, \beta)$. If $\hat{\beta}$ is smaller than the desired cardinality β, the sub-query is modified by the `Sub-query Splitter` using a splitting function σ that relaxes the predicates. If the sub-query's cardinality estimate meets the requirement, it is dispatched to the index, and a trajectory set $\mathcal{T}_i$ is obtained. If $|\mathcal{T}_i| \geq \beta$, it is forwarded to the `Histogram Builder`. If the cardinality is still below the threshold, it is modified again by the `Sub-query Splitter`.

Once all trajectory sets $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k\}$ are obtained, their travel-time sets $\{X_1, X_2, \ldots, X_k\}$ are extracted, with $X_i = \{Dur(tr, P_i) | tr \in \mathcal{T}_i\}$. From those, a set of histograms $\mathcal{H} = \{H_1, H_2, \ldots, H_k\}$ is computed, and they are convolved into a single histogram $H = H_1 * H_2 * \ldots * H_k$ that estimates the travel-time distribution for the complete path P.

3.2 Partitioning Methods

For the initial partitioning of queries, we propose five different methods. We use the query $Q = spq(P, I, f, \beta)$ with path $P = \langle A, C, D, E \rangle$ from the network in Figure 1 as example. The initial periodic time interval I_i^R is identical for all sub-queries and is always chosen so that $t_e - t_s = \alpha_{min}$, where α_{min} is the minimum time interval size, which is chosen by the system. The predicate f is also initially identical for all sub-queries but may be modified by the splitting method (cf. Section 3.3).

3.2.1 Regular (π_p). The regular partitioning creates sub-queries for paths of length p, i.e., every query is partitioned into $k = \lceil \frac{l}{p} \rceil$ sub-queries, i.e., the sub-queries $\pi_p(Q) = \langle spq(P[0, p), I_1^R, f_1, \beta), spq(P[p, 2p), I_2^R, f_2, \beta), \ldots, spq(P[p\lfloor \frac{l}{p} \rfloor, l), I_k^R, f_k, \beta) \rangle$ are created. In our experiments we chose π_1, π_2 and π_3, which for our example path yield the paths $\langle \langle A \rangle, \langle C \rangle, \langle D \rangle, \langle E \rangle \rangle$, $\langle \langle A, C \rangle, \langle D, E \rangle \rangle$, and $\langle \langle A, C, D \rangle, \langle E \rangle \rangle$, respectively.

3.2.2 Segment Category (π_C). The segment type partitioning creates partitions of sub-paths with identical segment categories, i.e., two neighboring segments e_i and e_{i+1} are split unless $\mathcal{F}(e_i).c = \mathcal{F}(e_{i+1}).c$. For our example query, this results in the sub-paths $\langle \langle A \rangle, \langle C, D \rangle, \langle E \rangle \rangle$.

3.2.3 Zone Type (π_Z). The zone type partitioning creates partitions of sub-paths within the same zone type, i.e., two neighbouring segments e_i and e_{i+1} are split unless $\mathcal{F}(e_i).z = \mathcal{F}(e_{i+1}).z$. For our example query, this results in the sub-paths $\langle \langle A \rangle, \langle C, D, E \rangle \rangle$.

3.2.4 Zone Type & Segment Category (π_{ZC}). The zone type and segment category partitioning creates partitions of sub-paths within the same zone type and segment category combination, i.e., two neighboring segments e_i and e_{i+1} are split unless $\mathcal{F}(e_i).z = \mathcal{F}(e_{i+1}).z \wedge \mathcal{F}(e_i).c = \mathcal{F}(e_{i+1}).c$. For our example query, this results in the sub-paths $\langle \langle A \rangle, \langle C, D \rangle, \langle E \rangle \rangle$.

3.2.5 None (π_N). No initial partitioning is attempted, and the query is processed according to one of the splitting strategies described below. For our example query, this results in the single sub-path $\langle \langle A, C, D, E \rangle \rangle$.

3.3 Splitting Methods

If a sub-query $spq(P, I, f, \beta)$ does not return the desired cardinality, it is modified by a splitting function σ described in Procedure 1 that takes a query and the list of time interval sizes $A = \langle \alpha_1, \ldots, \alpha_n \rangle$, with $\forall i \forall j\ (i < j \Rightarrow \alpha_i < \alpha_j)$, and $\alpha_1 = \alpha_{min}$ and $\alpha_n = \alpha_{max}$ as arguments.

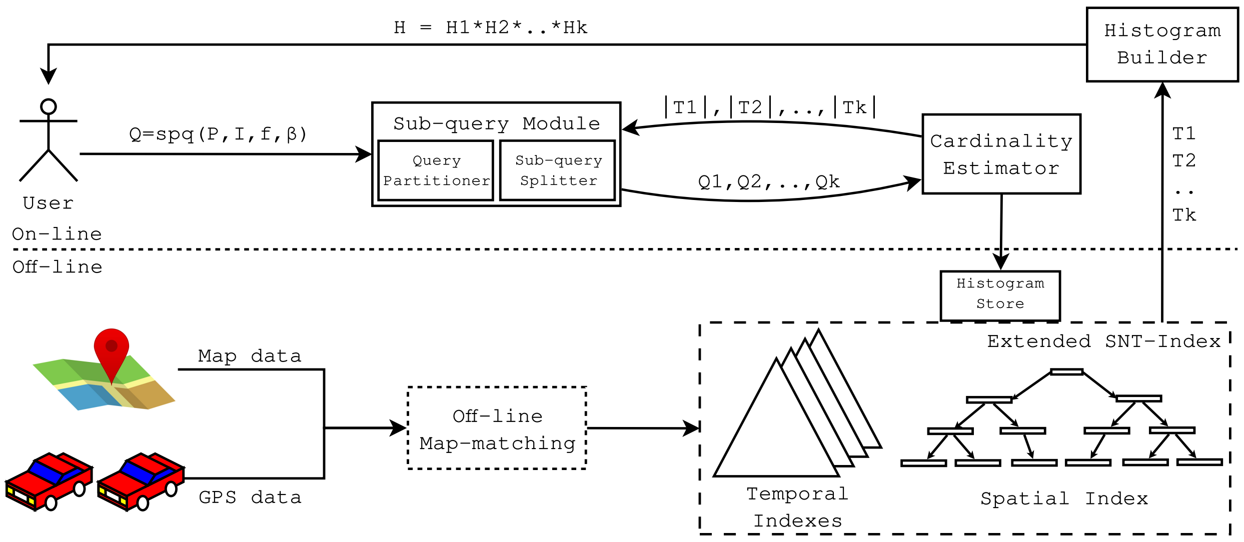


Figure 2: Overall Architecture

At first the procedure tries to increase the sample size by increasing the size of the time interval for the path by choosing the next largest size from the list A and widening the periodic interval with $widen([t_s, t_e)^R, \alpha_{i+1}) = [t_s - \frac{\alpha_{i+1}-\alpha_i}{2}, t_e + \frac{\alpha_{i+1}-\alpha_i}{2})^R$. After A has been exhausted, the path is split, and two new sub-queries with the smallest allowed time interval size α_{min} are created.

We propose two types of splitting and again use the path $P = \langle A, C, D, E\rangle$ in examples.

σ_R Regular splitting cuts the path in half, i.e., $P_1 = P[0, \lfloor \frac{l}{2} \rfloor)$ and $P_2 = P[\lfloor \frac{l}{2} \rfloor, l)$, so splitting the example path P results in $P_1 = \langle A, C\rangle$ and $P_2 = \langle D, E\rangle$.

σ_L Longest prefix splitting creates two sub-paths $P_1 = P[0, m)$ and $P_2 = P[m, l)$, with $1 \le m < l$, where the maximum value for m for which $|\mathcal{T}^{P_1}| \ge \beta$ holds is chosen.

If a sub-path cannot be split further, any non-temporal filter predicates are dropped (Line 10). As a fallback, all temporal filters and the β parameter are dropped as well, i.e., for a single segment, all available trajectories are considered in the fixed time interval $[0, t_{max})$ (Line 12).

Procedure 1 Modify a sub-query *spq* to increase sample size (σ):

```
Input: Sub-query $spq(P, I, f, \beta)$, time interval sizes $A$
Output: a sequence of sub-queries $\langle Q_1, \dots, Q_k\rangle$
 1: $\alpha_i \leftarrow t_e - t_s$
 2: if $\alpha_i < \alpha_{max}$ then
 3:     $I'^R \leftarrow widen(I^R, \alpha_{i+1})$
 4:     return $\langle spq(P, I'^R, f, \beta)\rangle$
 5: else if $|P| > 1$ then
 6:     $m \leftarrow split(P)$
 7:     $I'^R \leftarrow shrink(I^R, \alpha_{min})$
 8:     return $\langle spq(P[0, m), I'^R, f, \beta), spq(P[m, l), I'^R, f, \beta)\rangle$
 9: else if $f \ne \emptyset$ then
10:     return $\langle spq(P, I^R, \emptyset, \beta)\rangle$
11: else
12:     return $\langle spq(P, [0, t_{max}), \emptyset)\rangle$
13: end if
```

4 THE INDEX

This section describes the SNT-index and how we adapt and optimize it to support travel-time queries using an example trajectory set.

4.1 SNT-Index

Koide et al. [12] proposed the SNT-index for strict path queries using the FM-index as a spatial index and a forest of B+-trees as a temporal index. The advantage of the FM-index over R-tree-based methods is that by representing the trajectory set $\mathcal{T}$ as a string T and adapting a method from substring matching, evaluating spatial queries is only dependent on the size of the spatial network ($|E|$) and not on the size of the trajectory set ($|\mathcal{T}|$). In addition, it can be established from just the FM-index whether a given path is traversed at all, often saving a costly temporal index traversal. While the original index returns a set of trajectory ids given the query $spq(P, I)$, where P is the path and I is a time interval, our index returns the traversal times of the trajectories for P, which can be stored in a histogram. Sections 4.1.1 and 4.1.2 recap the previously described SNT-index and the remaining section describes our modifications to it to facilitate the efficient retrieval of travel-times.

4.1.1 The Spatial FM-Index. For our example we are indexing the trajectory set $\mathcal{T} = \{tr_0, tr_1, tr_2, tr_3\}$ introduced earlier.

To index the trajectories, we first need to compute the trajectory string T from the alphabet $\Sigma = E \cup \{\$\}$ where the symbol \$ denotes the end of a trajectory and where $\forall e \in E\,(e > \$)$ and $T = P_{tr_0}\$P_{tr_1}\$\dots\$P_{tr_{n-1}}\$$, $\forall tr \in \mathcal{T}$. With our example trajectory set, this yields the trajectory string $T = ABE\$ACDE\$ABF\$ABE\$$.

From this trajectory string, we compute an array S of all suffixes of T, where $S[i] = T[i, n)$, where $0 \le i < n = |T|$. These suffixes are then sorted lexicographically to obtain the *suffix array SA* as shown in Figure 3, where $SA[j]$ contains the index of the j-th smallest suffix. From SA, we can then compute the *inverse suffix array ISA* where $SA[j] = i$ and $ISA[i] = j$ [17]. Every substring (or in our case, subpath) P of length l therefore has a range of ISA values $R(P) = [st, ed)$ that is defined as $R(P) = \{i \mid S[SA[i]]\,[0, l) = P\}$, e.g., the ISA range of the path $\langle A\rangle$ is $R(\langle A\rangle) = [4, 8)$ since four trajectories contain A and they

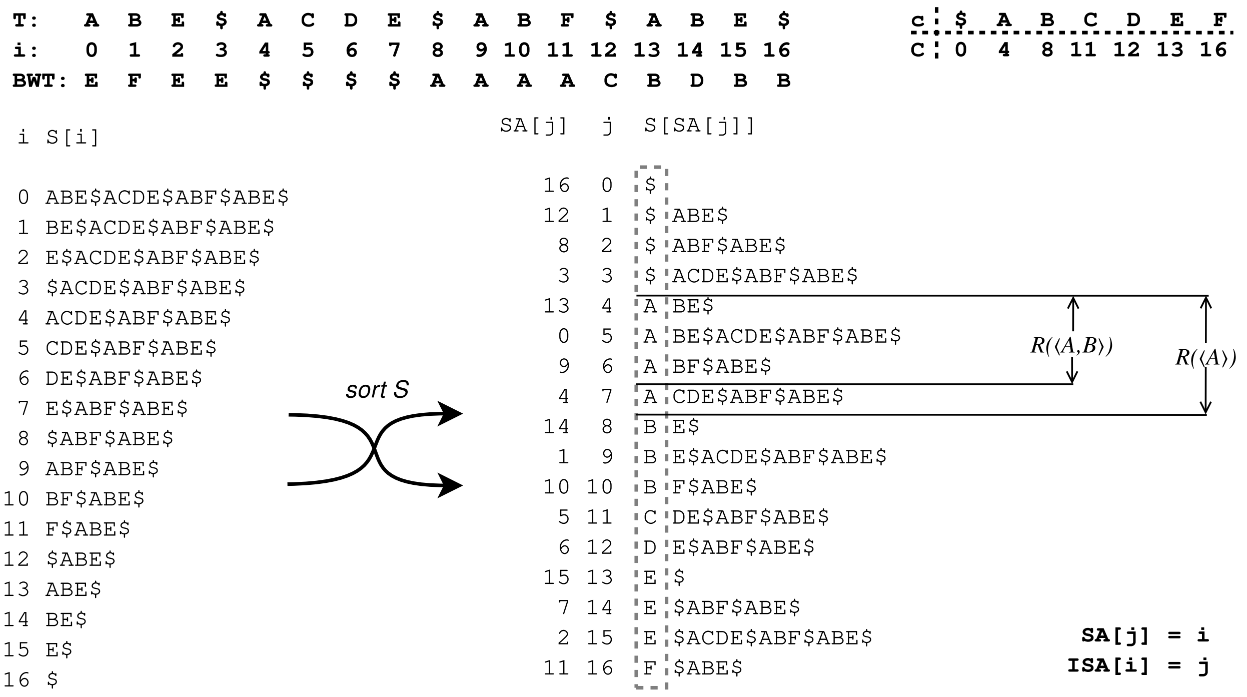


Figure 3: The Suffix Array and Burrows-Wheeler-Transform

Procedure 2 Calculate ISA range $[st, ed)$ for a path P of length l (*getISARange*):

```
Input: Burrows-Wheeler transform $T_{bwt}$ of the trajectory string
    $T$, symbol counts $C$, path $P = p_0 \ldots p_{l-1}$
Output: ISA range $[st, ed)$ that matches $P$
 1: $c \leftarrow p_{l-1}$
 2: $st \leftarrow C[c]$
 3: $ed \leftarrow C[c+1]$
 4: for $i \leftarrow 2$ to $l$ do
 5:     $c \leftarrow p_{l-i}$
 6:     $st \leftarrow C[c] + rank_c(T_{bwt}, st)$
 7:     $ed \leftarrow C[c] + rank_c(T_{bwt}, ed)$
 8:     if $st \geq ed$ then
 9:         return $[0, 0)$
10:     end if
11: end for
12: return $[st, ed)$
```

appear at the start of the suffixes $S[SA[st]]$ to $S[SA[ed-1]]$, and the range for $R(\langle A, B\rangle)$ is $[4, 7)$ as only three trajectories traverse this path.

The ISA range of a path can be obtained efficiently from two data structures that comprise the FM-index:

C	an array that stores the number of lexicographically smaller characters in the trajectory string for every member of the alphabet Σ, e.g., $C['B'] = 8$ since there exist 8 characters in T that are lexicographically before $'B'$.
T_{bwt}	the Burrows-Wheeler transform [3] of the trajectory string T that is defined as $T_{bwt}[i] = T[SA[i]-1]$, with $0 \leq i < \lvert T\rvert$. For our example, this yields the string *EFEE$$$$AAAA–CBDBB*.

We define the rank operation $rank_c(T_{bwt}, i)$ that counts the occurrences of the character c in $T_{bwt}[0, i)$. As an example of computing the ISA range, we compute the range for the path $P = \langle A, B\rangle$ as described in Procedure 2. At first, the segment $c \leftarrow B$ is set in Line 1, and $st \leftarrow 8$ and $ed \leftarrow 11$ are initialized in Lines 2 and 3. For the first (and in this case the only) iteration of the loop from Line 4 to 11, $c \leftarrow A$, $st \leftarrow 4 + rank_A(T_{bwt}, 8)$, and $ed \leftarrow 4 + rank_A(T_{bwt}, 11)$, which yields the ISA range $[4, 7)$, since the ranks are 0 and 3, respectively.

The Burrows-Wheeler transform is stored in a wavelet tree to enable rank queries in $O(log|\Sigma|)$ time [10]. Therefore, obtaining the ISA range $[st, ed)$ of any path P can be performed in $O(|P|\ log\ |\Sigma|)$ time, which does not depend on the size of T.

4.1.2 The Temporal Indexes.

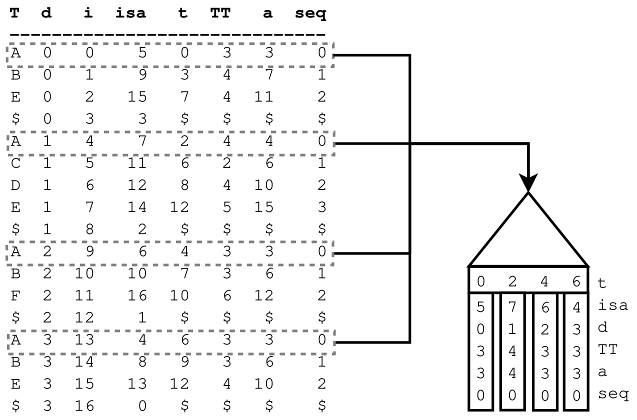
4.1.2 The Temporal Indexes. The temporal indexes $F = \{\Phi_e | e \in E\}$ contain a B+-tree for every segment in the network. Each tree indexes the records $r \in \Phi$ by the timestamp t when a trajectory entered the segment. A leaf node entry r for a timestamp t contains the ISA index (*isa*) and the trajectory identifier (d).

The original SNT-index is only capable of retrieving the trajectory ids, which would then have to be processed in turn to obtain the traversal times of the query path.

4.1.3 Extensions to the SNT-Index. To support travel-time histogram construction directly using the SNT-index, we add the following information to each leaf node in a temporal index:

- The traversal time TT of the segment in seconds.
- The sequence number *seq* of the segment in the trajectory.
- The sum of the travel-times $a_{seq} = \sum_{i=0}^{seq} TT_i$ from the start of the trajectory and up to and including the segment.

Figure 4 shows the contents of the temporal index Φ_A of segment A for our example trajectory set where each leaf is a record r, mapping a timestamp t to a tuple (isa, d, TT, a, seq). Furthermore, we add an associative container U that maps every trajectory id d to its respective user id u to check the filter predicate f. With those fields, we can build a hash table during the scan of the index of the first segment with the trajectory id and sequence number as the key (d, seq) and the aggregate of the preceding segment of the trajectory $(a_0 - TT_0)$, as value as described in Procedure 3. The sequence number is included to guard against



T	d	i	isa	t	TT	a	seq
A	0	0	5	0	3	3	0
B	0	1	9	3	4	7	1
E	0	2	15	7	4	11	2
$	0	3	3	$	$	$	$
A	1	4	7	2	4	4	0
C	1	5	11	6	2	6	1
D	1	6	12	8	4	10	2
E	1	7	14	12	5	15	3
$	1	8	2	$	$	$	$
A	2	9	6	4	3	3	0
B	2	10	10	7	3	6	1
F	2	11	16	10	6	12	2
$	2	12	1	$	$	$	$
A	3	13	4	6	3	3	0
B	3	14	8	9	3	6	1
E	3	15	13	12	4	10	2
$	3	16	0	$	$	$	$

0	2	4	6	t
5	7	6	4	isa
0	1	2	3	d
3	4	3	3	TT
3	4	3	3	a
0	0	0	0	seq

Figure 4: Extended Temporal Index

trajectories with circular paths. The spatial filtering is performed with the ISA range $[st, ed)$ obtained from Procedure 2 during the index scan in Line 3. The filter predicate f can be evaluated in constant time with the associative container U. The cardinality parameter β is used to reduce the processing time since not all eligible trajectories are necessary to obtain a good estimate, and the *buildMap* procedure terminates as soon as β trajectories are found (Line 6). When scanning the temporal index of the last segment in the query, we can obtain the traversal time of the query path by $a_{l-1} - (a_0 - TT_0)$ as described in Procedure 4.

4.2 Travel-Time Query

When used together, the previous three procedures make it possible to obtain the set of travel times for any path, as shown in Procedure 5, to answer the sub-query $spq(P, I, f, \beta)$. To obtain all trajectories that traversed a path P during a given time interval I, an ISA range is first obtained from the FM-index in Line 1. If a non-empty range is returned, a range scan on the index of the first (Line 6) and last segment (Line 11) of the path are performed for I and filtered by the ISA index in the leafs. If no matching trajectories exist or no periodic time interval with more than β trajectories is found (Line 7) the query returns the empty set. If the sub-query provided by Procedure 1 has a fixed time interval, the query is processed regardless of β. If that still yields no trajectories, an estimate based on the speed limit of the segment is provided (Line 13).

Procedure 6 shows how a full query is partitioned and processed. For longer trips, the periodic interval I_i^R is adapted with the shift-and-enlarge procedure (Line 4) suggested by Dai et al. [4], that shifts the beginning of the interval t_s by the sum of all previous minimums $S_i = \sum_{j=1}^{j=i-1} H_j^{min}$ and enlarges it by the sum of all previous ranges $R_i = \sum_{j=1}^{j=i-1}(H_j^{max} - H_j^{min})$.

4.3 Optimizations

4.3.1 CSS-Trees. The cache sensitive search tree (CSS-tree) proposed by Rao and Ross is a low memory overhead pointerless index that speeds up searches in sorted arrays [21]. In our system, we use it as an append-only replacement for the temporal B+-tree forest (cf. Section 4.1.2) to speed up Procedures 3 and 4 and to reduce memory consumption. Furthermore, its ability to efficiently compute the size of a key range in logarithmic time is used to improve the accuracy of the cardinality estimator (cf. Section 4.4). The CSS-tree is optimized to reduce the number of cache misses during a search by using the processor's cache line size as its node size. Since it indexes sorted arrays, only appends can be performed efficiently. We deem this an acceptable trade off because inserting additional trajectories would also require a re-computation of the entire FM-index, making the index mostly suited for batch updates.

Procedure 3 Create a mapping of trajectory identifier and sequence number (d, seq) to an antecedent travel time *diff* for the first β trajectories matching the predicates (*buildMap*):

```
Input: Temporal index $\Phi$ of the first segment of the query path,
    ISA range $[st, ed)$, time interval $I$, predicate $f$, and cardinality
    parameter $\beta$
Output: a mapping of $(d, seq)$ to $(a - TT)$
1:  $M \leftarrow \emptyset$
2:  for all $r \in \Phi$ do
3:      if $r.t \in I \wedge st \leq r.isa < ed \wedge f(r.d)$ then
4:          $diff \leftarrow r.a - r.TT$
5:          $M \leftarrow M \cup \{(r.d, r.seq) \rightarrow diff\}$
6:          if $|M| \geq \beta$ then
7:              return $M$
8:          end if
9:      end if
10: end for
11: return $M$
```

Procedure 4 Compute the travel times for all eligible trajectories over the path identified in the *buildMap* function (*probeMap*):

```
Input: Temporal index $\Phi$ of the last segment of the query path,
    path length $l$, and probe table $M$
Output: a list of travel times $X$
1: $X \leftarrow \emptyset$
2: for all $r \in \Phi$ do
3:     $b \leftarrow M[(r.d, r.seq + 1 - l)]$
4:     if $b \neq \emptyset$ then
5:         $X \leftarrow X \cup \{r.a - b.diff\}$
6:     end if
7: end for
8: return $X$
```

Procedure 5 Retrieve all travel-times $X = \langle x_0, ..., x_{\beta-1} \rangle$ of trajectories in I that meet predicate f for a path P (*getTravelTimes*):

```
Input: Burrows-Wheeler transform of the trajectory string $T_{bwt}$,
    temporal indexes $F$, symbol counts $C$, path $P = p_0...p_{l-1}$,
    time interval $I$, predicate $f$, and cardinality parameter $\beta$
Output: a set of travel-times $X$
1:  $[st, ed) \leftarrow getISARange(T_{bwt}, C, P)$
2:  if $st \geq ed$ then
3:      return $\emptyset$
4:  end if
5:  $\Phi_0 \leftarrow F[p_0]$
6:  $M \leftarrow buildMap(\Phi_0, [st, ed), I, f, \beta)$
7:  if $|M| < \beta$ and $isPeriodic(I)$ then
8:      return $\emptyset$
9:  end if
10: $\Phi_{l-1} \leftarrow F[p_{l-1}]$
11: $X \leftarrow probeMap(\Phi_{l-1}, l, M)$
12: if $X = \emptyset$ and $|P| = 1$ then
13:     return $\{estimateTT(p_0)\}$
14: end if
15: return $X$
```

Procedure 6 Compute a histogram H for the query $spq(P, I, f, \beta)$ (*tripQuery*):

Input: Burrows-Wheeler transform of the trajectory string T_{bwt}, temporal indexes F, symbol counts C, query $spq(P, I, f, \beta)$, time interval sizes A, partitioning method π, and splitting method σ

Output: a histogram H

```
1: $\langle Q_1, \ldots, Q_k \rangle \leftarrow \pi(Q); \mathcal{H} \leftarrow \emptyset$
2: for all $Q_i \in \langle Q_1, \ldots, Q_k \rangle$ do
3:     if $isPeriodic(I_i)$ and $i > 1$ then
4:         $I_i \leftarrow [t_s + S_i, t_e + R_i)^R$
5:     end if
6:     $X_i \leftarrow getTravelTimes(P_i, I_i, f_i, \beta)$
7:     if $X_i \neq \emptyset$ then
8:         $\mathcal{H} \leftarrow \mathcal{H} \cup createHistogram(X_i)$
9:     else
10:        $\langle Q_{i+1}, \ldots, Q_k \rangle \leftarrow \langle Q_{i+1}, \ldots, Q_k \rangle \cup \sigma(Q_i)$
11:    end if
12: end for
13: $H \leftarrow H_1$
14: for all $i > 1 \wedge H_i \in \mathcal{H}$ do
15:     $H \leftarrow H * H_i$
16: end for
17: return $H$
```

4.3.2 Temporal Partitioning. Temporal partitioning of the SNT-index was originally proposed here [13], but not evaluated. It allows more efficient updates to the index without necessitating a complete re-computation of the FM-index which does not efficiently support updates or appends. Partitioning requires to split the trajectory set $\mathcal{T}$ into $\mathcal{T}_1, \ldots, \mathcal{T}_W$, where $\forall i < j \; (\nexists tr_i \in \mathcal{T}_i \; (\forall tr_j \in \mathcal{T}_j \; (tr_i.t_0 \geq tr_j.t_0)))$. From those trajectory sets, W trajectory strings $T^1, \ldots, T^W$ are then computed, and Procedure 2 is modified return a collection of ISA ranges from the Burrows-Wheeler transforms $T^1_{bwt}, \ldots, T^W_{bwt}$ using separate segment counters $C^1, \ldots, C^W$. Temporal partitioning also requires adding the partition identifier w to every leaf in the temporal indexes since every partition's FM-index can return a different ISA range for the same path.

4.4 Cardinality Estimator

Cardinality estimators are widely used in DBMSs to improve query plans. In our case, we want to avoid costly scans of our temporal indexes if the required sample size β cannot be met. We require a function $card(Q)$ that returns an estimate $\hat{\beta}$ for the cardinality of the return trajectory set $\mathcal{T}$ and if $\hat{\beta} < \beta$, we apply the split function σ to Q without running a costly query. The cardinality estimator relies on a time-of-day histogram for every segment and fast computation of the ISA range $[st, ed)$, which is enabled by Procedure 2. The exact count of all trajectories traversing a path $c_P = ed - st$ is efficiently retrieved. After that, the selectivity of the temporal filters needs to be estimated. The easiest way is to assume a uniform distribution throughout the day and to divide the size of a periodic interval by the length of the day, which yields the time-of-day selectivity:

$$sel_{tod} = sel(P, I^R = [t_s, t_e)^R) = \frac{t_e - t_s}{24 \; hours} \tag{1}$$

The uniformity assumption, however, usually does not hold so, the selectivity estimate can be improved by maintaining a time-of-day histogram H_e for every segment e. Then the selectivity can be estimated using the following formula:

$$sel(P, I^R = [t_s, t_e)^R) = \frac{B(H_{e_0}, [t_s, t_e))}{B(H_{e_0}, [0, 24 \; hours))}, \tag{2}$$

where $B(H, [t_s, t_e))$ counts the elements of all buckets in H in the range $[t_s, t_e)$. In addition to being constrained by the time-of-day a user might wish to limit the query to a certain time frame, e.g., only considering trajectories within the past year. The selectivity can be estimated naively with the following formula:

$$sel_{tf} = sel(P, I = [t_s, t_e)) = \frac{t_e - t_s}{F[e_0]_{max} - F[e_0]_{min}}, \tag{3}$$

where $F[e_0]_{min}$ and $F[e_0]_{max}$ are the earliest and latest traversal times of segment e_0. When using the CSS-tree, the number of entries for which $t_s \leq t < t_e$ can be obtained exactly in logarithmic time and sel_{tf} can be computed exactly. To compute the selectivity of user predicates sel_u, we use the default of $\frac{1}{10}$ suggested by Selinger et al. [22]. To obtain the estimate for a query, we combine these selectivity factors to obtain our estimate $\hat{\beta} = sel_{tod} * sel_{tf} * sel_u * c_P$.

We define five different modes for the cardinality estimator:

ISA only uses the size of the ISA range c_P as estimate $\hat{\beta}$

BT-Fast uses formulas 1 and 3 to estimate the selectivity

BT-Acc uses formulas 2 and 3 to estimate the selectivity

CSS-Fast uses formula 1 and a fast lookup in the CSS-tree to estimate the selectivity

CSS-Acc uses formula 2 and a fast lookup in the CSS-tree to estimate the selectivity

5 EXPERIMENTAL SETUP

This section describes the data set and quality metrics we use to evaluate our system.

5.1 Datasets

5.1.1 OpenStreetMap. Our network graph is based on the OpenStreetMap data of the road network of Northern Denmark, which contains around 750,000 road segments. When converted to a spatial network graph, this graph has around 1.46 million directed edges [19]. Each edge represents a direction on a segment and has one of 17 different segment categories. This categorization is available for all OpenStreetMap maps and makes segment category-based partitioning possible for other map-matched trajectory datasets as well. The OpenStreetMap data also includes the speed limits for many segments, which we use as a fallback if no trajectory data is available. If the speed limit is not known, we use the median of all known speed limits of its segment category.

5.1.2 Zone Dataset. To distinguish rural and urban areas, we use the zoning map published by the Danish Business Authority [7] that consists of 4,259 zone geometries, each of which assigns one of three categories to an area:

- *city*: segments within city limits
- *rural*: segments in rural areas
- *summer house*: segments in areas zoned for summer house usage

A spatial join is used to assign a zone type to every segment in the map. A fourth category that we call *ambiguous* is assigned to segments located in more than one zone type.

5.1.3 ITSP Dataset. The "ITS Platform" dataset contains over 1.1 billion GPS points sampled at 1 Hertz collected from 458 vehicles in Aalborg and the surrounding region during the period from May 2012 to December 2014 [1].

In a preprocessing step, the GPS points are map-matched [18] to obtain in excess of 79 million segment traversals that form around 1.4 million trajectories, where a new trajectory is created if more than a 180 seconds have elapsed since the last GPS point. The map-matching algorithm also discards GPS points at the beginning and end of a trip if too few points are matched to the start and end segments of the trajectory. This is done so the durations of the segment traversals are meaningful. Each GPS record contains the trajectory ID, the vehicle ID, a segment ID, the time and date the segment was entered (minute resolution), and the time on segment (second resolution). Since the cars in our dataset are privately owned, we treat the vehicle ID as the user ID. The segment IDs are derived from the unique mapping of OpenStreetMap segment key and the driving direction. The time on a segment is also computed during the preprocessing step.

5.2 Query

We derive our query set Q from a random sample $\mathcal{T}_S \subset \mathcal{T}$ from our trajectory set:

$$Q = \{spq(P_{tr}, I_{tr}, f, \beta) | tr \in \mathcal{T}_S\},$$

with either $f = \{u = tr.u\}$ or $f = \emptyset$ if no user filters are used and different values of β being used in the experiments. For the time interval I_{tr}, either the periodic time interval $I^R_{tr} = [tr.s.t_0 - \frac{\alpha_{min}}{2}, tr.s.t_0 + \frac{\alpha_{min}}{2})^R$ or or the fixed time interval $I_{tr} = [0, tr.s.t_0)$ is used.

For the interval size we use the values 15 *min*, 30 *min*, 45 *min*, 60 *min*, 90 *min*, and 120 *min*.

5.3 Accuracy Metric

5.3.1 sMAPE. To evaluate the accuracy of the retrieved traversal times, we use the symmetric mean absolute percentage error [2] of the sum of the means of all sub-paths.

$$\text{sMAPE} = \frac{100\%}{|Q|} \sum_{i=1}^{|Q|} \frac{|\sum_{j=1}^{k} \bar{X}_j - a_{tr_i}|}{\frac{1}{2}(\sum_{j=1}^{k} \bar{X}_j + a_{tr_i})},$$

where k is the respective number of sub-queries of each query $Q \in Q$ and $\bar{X}_j$ is the travel-time mean retrieved with the sub-query.

5.3.2 Weighted Error. The weighted error, which we derive from sMAPE, considers the accuracy of the sub-query results and weighs them according to their fraction of the path length.

$$wE = \frac{100\%}{|Q|} \sum_{i=1}^{|Q|} \sum_{j=1}^{k} w_j \frac{|\bar{X}_j - a^{P_j}_{tr_i}|}{\frac{1}{2}(\bar{X}_j + a^{P_j}_{tr_i})},$$

with $w_j = \frac{\sum^{e \in P_j} \mathcal{F}(e).l}{\sum^{e \in P} \mathcal{F}(e).l}$, where P is the query path and P_j is the sub-query path.

5.3.3 Log-Likelihood. To evaluate the quality of the histograms, we compute the average log-likelihood of the travel-times a_{tr_i} with a discrete probability density function derived from the result histogram H_i.

For each trajectory with a result histogram H with a bucket width h, we compute the average log-likelihood $log\mathcal{L}$:

$$\frac{1}{|Q|} \sum_{i=1}^{|Q|} log\mathcal{L}(a_{tr_i}, H_i),$$

where the likelihood $\mathcal{L}(x, H)$ is defined by the discrete probability density function

$$p_H(x) = \gamma f(x, H) + (1 - \gamma)U(x),$$

where $U(x)$ is a uniform distribution defined for $[t_{min}, t_{max})$, $0 < \gamma < 1$ and

$$f(x, H) = \frac{B(H, [\lfloor \frac{x}{h} \rfloor, \lfloor \frac{x}{h} \rfloor + h))}{B(H, [t_{min}, t_{max}))}.$$

The smoothing with the uniform distribution $U(x)$ is performed so that $p_H(x)\ \forall x \in [t_{min}, t_{max})$ never reaches zero.

5.3.4 Q-Error. To evaluate the accuracy of the cardinality estimator, we use the q-error proposed by Moerkotte et al. [16]. To estimate the quality of our cardinality estimate $\hat{\beta}$, we compare it to the actual cardinality of the retrieved trajectory set $n = |\mathcal{T}|$. For every estimate, we obtain the q-error $q = max(\hat{\beta}'/n', n'/\hat{\beta}')$ with $n' = max(n, 1)$ and $\hat{\beta}' = max(\hat{\beta}, 1)$. This is done to handle estimations for empty sets as proposed by Stefanoni et al. [23]. The q-error shows the difference in orders of magnitude between the real cardinality and the estimate.

6 EVALUATION

This section reports on the experimental results. For all experiments, a query set Q is generated from the trajectory set $|\mathcal{T}_S| =$ 6,942, which is a random 1% sample of all trajectories in $\mathcal{T}$ that occur after the 8th of September 2013, the median of the timestamps in the ITSP data set. This is to ensure that every query has more than a year of trajectory data available. On average, the paths of the query set have a length of 13.7 kilometers, consist of 55 segments, and last 800 seconds.

In our study we evaluate three types of queries:

Temporal Filters that use a periodic time interval and no user filter ($spq(P_{tr}, I^R_{tr}, \emptyset, \beta)$)

User Filters that use a periodic time interval and a user filter ($spq(P_{tr}, I^R_{tr}, \{u = tr.u\}, \beta)$)

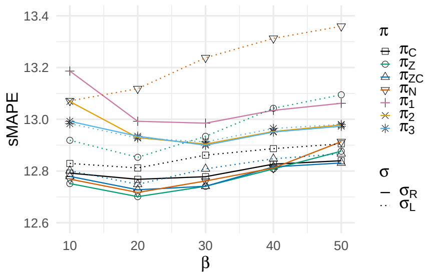
SPQ Only that use a fixed time interval and no user filter ($spq(P_{tr}, [0, t_{max}), \emptyset, \beta)$)

6.1 Qualitative Assessment

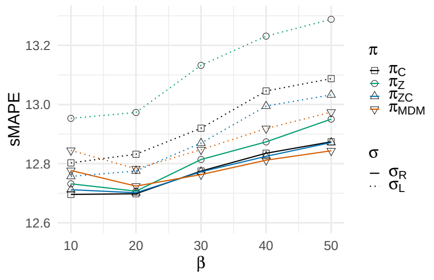
Figures 5 to 8 show the results of accuracy measured with sMAPE, the weighted error, and the log-likelihood and the average sub-query length. The figures show the results for different types of partitioning and splitting methods and filter predicates.

The regular partitioning method π_p (cf. Section 3.2.1) is used as a baseline with $p = 1, 2,$ and 3 because they are the sub-path lengths for which histograms can still be pre-computed at a reasonable overhead and because no known histogram-based methods perform better. For the user filter queries, we also evaluate the π_{MDM} method that partitions queries like π_C but only applies user filters to sub-queries with paths on main roads like motorways or other major roads connecting cities. This partitioning method is derived from the results of a previous study of travel-time estimation methods [26].

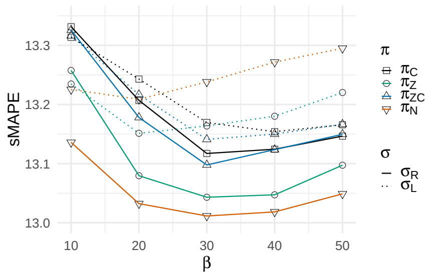
Figure 5a shows the average error for seven different partitioning methods with temporal filters. Here, π_1 performs worst, followed π_2 and π_3, and they achieve their highest accuracy at $\beta = 30$. If only the speed limits are used to estimate the travel time, sMAPE is 34.3% and if all available trajectories for each segment are used, the error is 13.8%. The partitioning methods based on the segment category and/or zone (π_C, π_Z, and π_{ZC})



(a) Temporal Filters

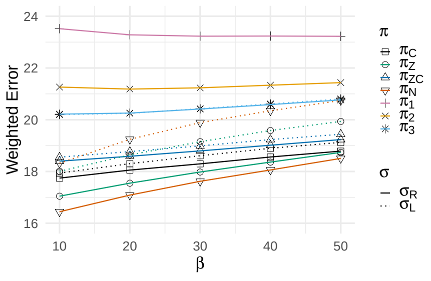


(b) User Filters

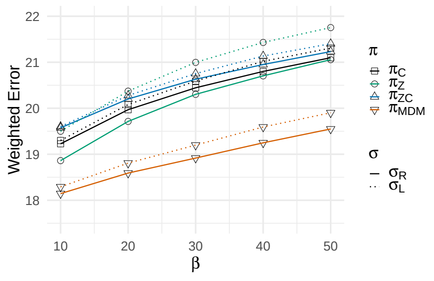


(c) SPQ Only

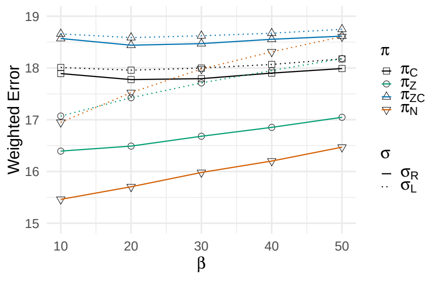
Figure 5: sMAPE



(a) Temporal Filters

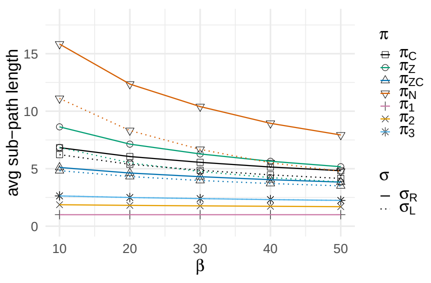


(b) User Filters

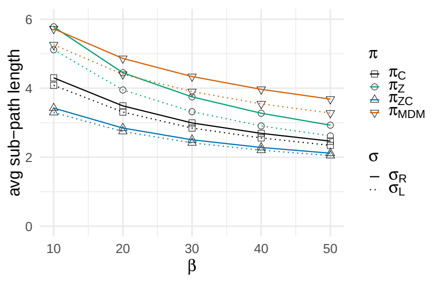


(c) SPQ Only

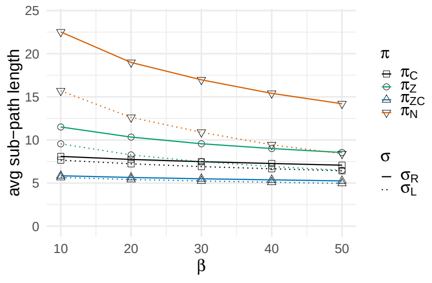
Figure 6: Weighted Error



(a) Temporal Filters

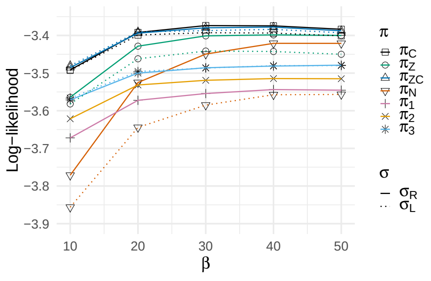


(b) User Filters

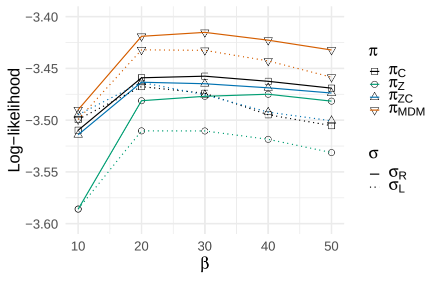


(c) SPQ Only

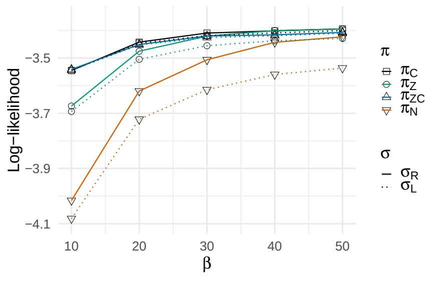
Figure 7: Sub-query Path Length



(a) Temporal Filters



(b) User Filters



(c) SPQ Only

Figure 8: Log-Likelihood

together with π_N achieve very similar accuracy. Here, the accuracy peaks at $\beta = 20$. Category-based partitioning is the most stable in terms of accuracy, and zone-based partitioning provides the overall best result. The queries using user filters shown in Figure 5b perform equally well, but with the exception of π_Z do not degrade as much with higher values of β and also obtain their lowest error at $\beta = 20$ and exhibit very similar accuracy to the queries without user filters. The SPQ Only methods in Figure 5c methods did not manage to outperform the baseline because it does not use periodic intervals that can observe changing congestion, e.g., longer travel-times during rush hours. In nearly all cases with regular splitting (σ_R) achieves considerably better accuracy than longest prefix splitting (σ_L). In most cases

A similar picture to the sMAPE results can be seen for the temporal filter queries in Figure 6a, with π_N having the lowest weighted error. If only the speed limits are used to make an estimate, the weighted error is 36.9%, and if all available trajectories for each segment are used, the error is 24.0%. For the user filter queries in Figure 6b, only π_{MDM} manages to consistently outperform the baseline. The SPQ only queries shown in Figure 6c show the lowest error with the coarsest partitioning methods. The low error of the SPQ only methods is due to sub-query results being weighted according to the length of the sub-paths and not relative to the share of travel time. Estimates for paths on long segments with high speed limits, e.g., motorways, exhibit already low estimation errors and also tend to improve the most when custom predicates are used [26]. In all cases, σ_L has a higher error than σ_R. Figure 7 shows the average path lengths of the final sub-queries. We can see that there is an inverse relationship between the weighted error and the sub-query paths. We can also see that π_Z provides the coarsest partitioning with the exception of π_N, which initially provides none.

Figures 8a to 8c show the average log-likelihood with $f(x, H)$ derived form a histogram with a bucket size of $h = 10s$ and different values for β and $\gamma = 0.99$. The queries with only temporal filters and π_Z and π_{ZC} return the most accurate histograms, and the coarser the partitioning method the less accurate the histograms are with low sample sizes. Among the User Filter queries π_{MDM} consistently outperforms the other three partitioning methods. The queries run with π_N do not even outperform the baseline for $\beta < 30$. In all cases, σ_L performs worse than σ_R. We evaluated several values for γ (from 0.90 to 0.99) but the qualitative results did not change.

6.2 Efficiency

The index is implemented in C++17 and compiled with g++ 7.2.0 with the `-O3 -march=native` flags. For the performance test, the SNT-index with a CSS-forest and only a single partition is used. The FM-index is implemented using `sdsl-lite`'s integer-alphabet Huffman-shaped wavelet tree implementation, and the suffix array is computed with Yuta Mori's `sais-lite` library [17]. The performance test ran on a server with AMD Opteron 6376 processors and 512 GiB RAM. For the processing time, the average runtime in milliseconds of 6,942 queries is reported in Figure 9.

The temporal filter queries shown in Figure 9a perform very similar to the baseline, with π_C and π_{ZC} being slightly faster than regular partitioning. The combination of π_C and σ_L has been omitted in the figure for reasons of scaling since the results are in the range of 50 to 65 ms. In Figure 9b, it can be seen that the average user filter query takes around 4 to 5 times longer than the temporal filter queries; and with π_{MDM}, queries only take around twice as long since it applies non-temporal predicates only selectively. SPQ only queries have much lower processing times than do the other two query types, and all consistently outperform the baseline. The reason for their low processing times can be seen in Figure 7c and Procedure 5. Since their sub-queries tend to cover comparatively long paths, SPQ only queries need to perform considerably fewer temporal index scans than the other query types, which need to be split into more sub-queries to fulfill the cardinality requirements. The average runtime of π_N with σ_L, which is between 30 to 35 ms, has been omitted in Figure 9c. In all cases σ_L performed poorly in comparison to σ_R.

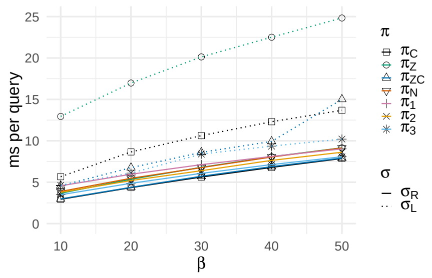
6.3 Temporal Partitioning

Figure 10 shows the effect of the temporal partitioning defined in Section 4.3.2 on memory consumption and setup time. The figures show the results for partition sizes of 7, 30, 90, and 365 days, resulting in a 138, 33, 11, and 3 partitions, respectively. We also examine the case where only one partition is used (FULL). Where applicable, the performance of the index with a B+-forest (BT) is reported as well. For the in-memory B+-trees, we use the `btree_multimap` from Google's `cpp-btree` library [11]. Figure 10a shows the memory consumption of the different index components, where *Forest* is the memory consumption of the CSS-tree or B+-tree forest, respectively. The size of the forest is not impacted by different partition sizes, but if the partition feature is removed from the index, the memory saved in the tree leafs by omitting the partition identifier w is around 300 MiB for our data set. We can also see that the in-memory B+-tree forest has slightly higher memory requirements than the CSS-forest. The associative container U used to enable user filtering (user) is also not affected by the partitioning and takes up around 65 MB for our data set. The two data structures that comprise the FM-index, the wavelet tree (WT) and the segment counter (C), are affected considerably by partitioning. The segment counter grows linearly with the number of partitions from less than 6 MB to nearly 600 MB since a separate segment count needs to be maintained for every wavelet tree. The compression rate of the wavelet tree degrades considerably with smaller trajectory strings, which for the 7 day partitioning are only a few MBs per partition as opposed to several hundred in a single partition and it grows from around 280 MB to over 4 GB. The memory requirements of the time-of-day histograms required for the cardinality estimator are affected considerably if a histogram is maintained for every non-empty partition for every segment, and the memory required for the histograms soon exceeds the amount required for the index. Figure 10b shows the memory consumption for three different bucket sizes h (1, 5, and 10 minutes).

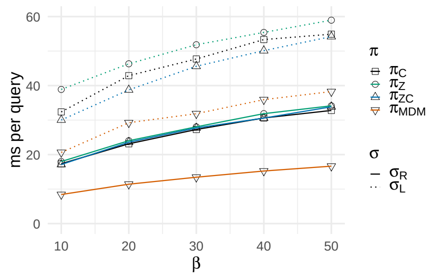
The setup times for the index shown in Figure 10c are not significantly affected by the different partition sizes or tree types and always remain between 425 and 475 seconds. For the setup, the trajectory and map data are loaded from disk.

6.4 Cardinality Estimator

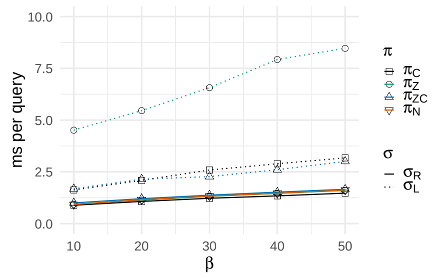
Figure 10 shows the results for the cardinality estimator. In all cases the results for partitioning method π_Z with regular splitting and $\beta = 20$ are shown. Figure 11a shows the q-error of the five different cardinality estimator modes. Here, 5,000 queries are run, after which their cardinalities n are compared with estimate $\hat{\beta}$. The simplest estimate using just the ISA range is on average off by an order of magnitude. The four other modes provide considerably more reliable estimates, with the histogram



(a) Temporal Filters

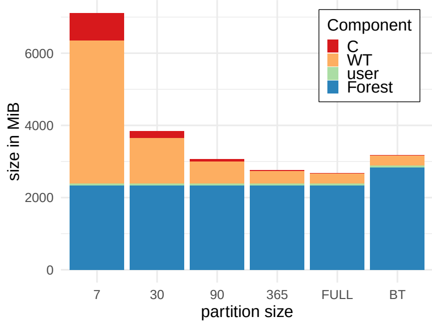


(b) User Filters

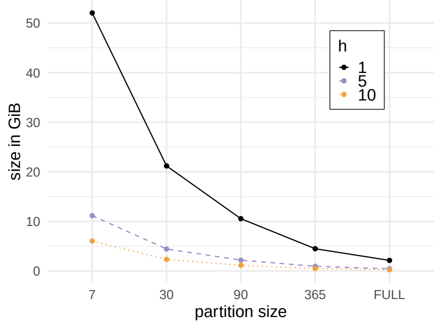


(c) SPQ Only

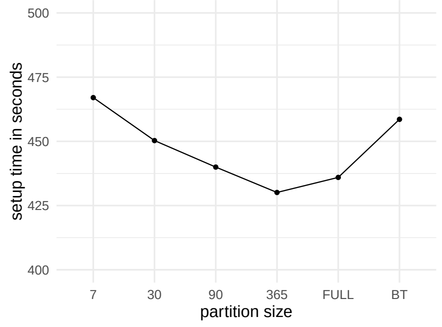
Figure 9: Processing Time



(a) Index Memory Consumption

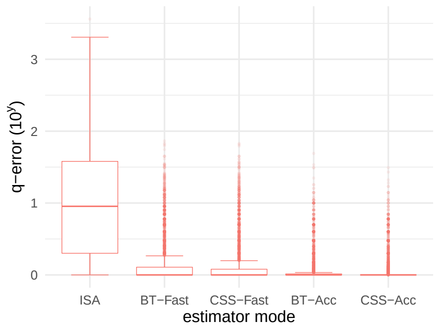


(b) Histogram Memory Consumption

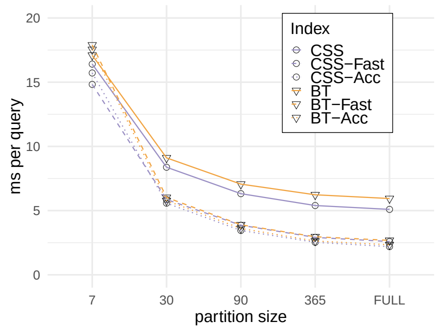


(c) Setup Time

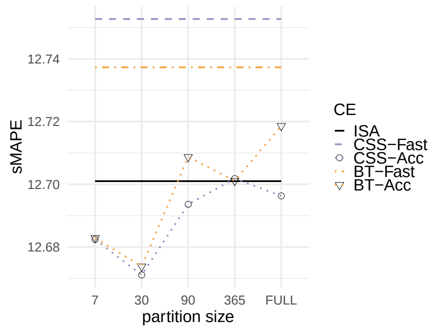
Figure 10: Temporal Partitioning



(a) Q-Error



(b) Runtime



(c) Effect on Accuracy

Figure 11: Cardinality Estimator

based methods performing better than the fast ones and the CSS-tree based methods performing slightly better than their B+-tree counterparts.

Since the selectivity estimates of the estimators might underestimate cardinalities of queries, a query might be split despite covering a sufficient sample size. This may affect the quality of the overall travel-time estimate. Figure 11c, however, shows that the effects on quality are minuscule compared to the baseline (ISA) and might even yield slight improvements in accuracy.

Figure 11b shows that partitioning as well as using the cardinality estimators can impact performance significantly. For single, yearly, and quarterly partitions, the query performance changes little, and use of the cardinality estimator reduces query processing times by around 50%. For smaller partitions, however, the effects of using the cardinality estimator diminish; and with weekly partitioning, the B+-tree version of the index performs worse with the estimators. The histogram-based CSS-tree version (CSS-Acc) performs worse than the fast version (CSS-Fast), which is most likely due to the amount of time-of-day histograms that have to be scanned to obtain the selectivity sel_{tod}.

6.5 Implications

Overall our data shows that after a certain β is reached no significant gains in accuracy are obtained by increasing it further indicating smaller result sets obtained from fewer SPQs of long paths provide more accurate estimates than larger result sets obtained with short paths. One can also see that evaluating non-temporal predicates comes with a considerable overhead and for the user predicates provides no improvement in quality over the purely temporal methods. If such methods are however applied selectively (e.g. π_{MDM}) the performance overhead is mitigated and the accuracy improves. The naive regular splitting method does not only achieve better accuracy but also has a considerably shorter runtime, making it more suitable for a real-time queries. The CSS-tree version of the index is as least as fast or faster than the B+-tree-based version, but the improvements become less noticeable when using the index in conjunction with a cardinality estimator. CSS-trees reduce the memory consumption of the index as well and improve the accuracy of the cardinality estimator with their efficient range lookups. We have also shown that temporal partitioning of the index is viable in some cases, but that using time-of-day histograms to estimate the selectivity of periodic time intervals, despite slight improvements in estimator accuracy and query performance, is hardly worth the memory overhead for the evaluated data set. Additional experiments with larger data sets may offer additional insight into this trade off, but no larger trajectory data sets with user information were available to us. Our results show that modifications aimed at improving query performance often also improve the accuracy of the estimates.

7 CONCLUSIONS & OUTLOOK

Travel-time estimations in road networks can be improved considerable by utilizing large NCT data sets not only to provide estimates on a segment level, but also for full paths in the network. To our knowledge no current system supports these path-based estimations which cannot rely on pre-computations. We therefore propose a system that computes travel-time estimations based on trajectories selected at runtime and is able to improve upon the accuracy of existing histogram-based methods by expressing them as a series of strict path queries and adapting their predicates automatically to ensure accurate estimates. The SPQs are processed by our adapted SNT-index which is able to retrieve the traversal times for any path directly from the index. We have shown that the queries can be processed fast enough for real-time applications by utilizing specialized in-memory data structures and cardinality estimators tailored to SPQs. We evaluate our system with a large real-world trajectory data set and find that optimizing queries for performance is not preclusive of accuracy.

Our proposed system leaves several avenues of future work. The current greedy algorithm used for identifying a suitable partitioning and splitting of an SPQ is based on fairly simple heuristics and could be augmented by more sophisticated machine learning methods to improve accuracy of estimations. Approaches that use different values of the parameter β for each sub-query, e.g., smaller sample size requirements in rural zones, could be evaluated. While the processing time of single query might not considerably improve through parallelization the overall query throughput of the system most likely could, making it suitable for online routing applications that support a large number of users. Our approach also does not fully address the issue of data sparseness apart from providing relaxing the predicates if their selectivity is too low. Several approaches to solving the problem of data sparseness have been suggested [25, 30] and could be combined with our system to provide time-dependent travel-time estimates for paths where data is sparse.

ACKNOWLEDGMENTS

This research was supported in part by a grant from the Obel Family Foundation and by the DiCyPS project.

REFERENCES

[1] Aalborg University. 2018. ITS Platform. http://www.itsplatform.dk/.
[2] J Scott Armstrong. 1985. *Long-Range Forecasting: From Crystal Ball to Computer* (2 ed.). John Wiley & Sons, Inc.
[3] Michael Burrows and David J Wheeler. 1994. *A Block-sorting Lossless Data Compression Algorithm.* Technical Report. Digital Equipment Corporation.
[4] Jian Dai, Bin Yang, Chenjuan Guo, Christian S Jensen, and Jilin Hu. 2016. Path Cost Distribution Estimation Using Trajectory Data. *Proceedings of the VLDB Endowment* 10, 3 (2016), 85–96.
[5] Victor Teixeira De Almeida and Ralf Hartmut Güting. 2005. Indexing the Trajectories of Moving Objects in Networks. *GeoInformatica* 9, 1 (2005), 33–60.
[6] Zhiming Ding. 2008. UTR-tree: An Index Structure for the Full Uncertain Trajectories of Network-Constrained Moving Objects. In *9th IEEE International Conference on Mobile Data Management.* IEEE, 33–40.
[7] Erhvervsstyrelsen. 2018. zonekort. http://kort.plandata.dk/spatialmap.
[8] Eurostat. 2018. Transport, volume and modal split. https://ec.europa.eu/eurostat/web/transport/data/main-tables.
[9] Elias Frentzos. 2003. Indexing Objects Moving on Fixed Networks. In *8th International Symposium on Spatial and Temporal Databases.* Springer, 289–305.
[10] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms.* Springer, 326–337.
[11] Google Inc. 2011. cpp-btree. https://code.google.com/archive/p/cpp-btree/.
[12] Satoshi Koide, Yukihiro Tadokoro, and Takayoshi Yoshimura. 2015. SNT-index: Spatio-temporal index for vehicular trajectories on a road network based on substring matching. In *1st International ACM SIGSPATIAL Workshop on Smart Cities and Urban Analytics.* ACM, 1–8.
[13] Satoshi Koide, Yukihiro Tadokoro, Takayoshi Yoshimura, Chuan Xiao, and Yoshiharu Ishikawa. 2018. Enhanced Indexing and Querying of Trajectories in Road Networks via String Algorithms. *ACM Transactions on Spatial Algorithms and Systems* 4, 1 (2018), 1–41.
[14] Benjamin Krogh, Nikos Pelekis, Yannis Theodoridis, and Kristian Torp. 2014. Path-based Queries on Trajectory Data. In *22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, 341–350.
[15] Yu Ma, Bin Yang, and Christian S Jensen. 2014. Enabling Time-Dependent Uncertain Eco-Weights For Road Networks. In *Workshop on Managing and Mining Enriched Geo-Spatial Data.* ACM, 1–6.
[16] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing Bad Plans by Bounding the Impact of Cardinality Estimation Errors. *Proceedings of the VLDB Endowment* 2, 1 (2009), 982–993.
[17] Yuta Mori. 2008. SAIS: An implementation of the induced sorting algorithm.
[18] Paul Newson and John Krumm. 2009. Hidden Markov Map Matching Through Noise and Sparseness. In *17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, 336–343.
[19] OpenStreetMap contributors. 2014. Planet dump retrieved from https://planet.osm.org. https://www.openstreetmap.org.
[20] Iulian Sandu Popa, Karine Zeitouni, Vincent Oria, Dominique Barth, and Sandrine Vial. 2010. PARINET: A Tunable Access Method for in-Network Trajectories. In *26th IEEE International Conference on Data Engineering.* IEEE, 177–188.
[21] Jun Rao and Kenneth A Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *25th International Conference on Very Large Databases.* 78–89.
[22] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access Path Selection in a Relational Database Management System. In *ACM SIGMOD International Conference on Management of Data.* ACM, 23–34.
[23] Giorgio Stefanoni, Boris Motik, and Egor V Kostylev. 2018. Estimating the Cardinality of Conjunctive Queries over RDF Data Using Graph Summarisation. In *27th International World Wide Web Conference.* 1043–1052.
[24] Dong Wang, Junbo Zhang, Wei Cao, Jian Li, and Yu Zheng. 2018. When Will You Arrive? Estimating Travel Time Based on Deep Neural Networks. In *32nd AAAI Conference on Artificial Intelligence.* 2500–2507.
[25] Yilun Wang, Yu Zheng, and Yexiang Xue. 2014. Travel Time Estimation of a Path using Sparse Trajectories. In *20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM, 25–34.
[26] Robert Waury, Christian S Jensen, and Kristian Torp. 2018. Adaptive Travel-Time Estimation: A Case for Custom Predicate Selection. In *19th IEEE International Conference on Mobile Data Management.* IEEE, 96–105.
[27] Stephan Winter. 2002. Modeling Costs of Turns in Route Planning. *GeoInformatica* 6, 4 (2002), 345–361.
[28] Chun-Hsin Wu, Jan-Ming Ho, and Der-Tsai Lee. 2004. Travel-Time Prediction With Support Vector Regression. *IEEE Transactions on Intelligent Transportation Systems* 5, 4 (2004), 276–281.
[29] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. 2010. T-Drive: Driving Directions Based on Taxi Trajectories. In *18th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, 99–108.
[30] Fangfang Zheng and Henk Van Zuylen. 2013. Urban link travel time estimation based on sparse probe vehicle data. *Transportation Research Part C: Emerging Technologies* 31 (2013), 145–157.