Travel time estimation based on previous experience

Pre-study and prototyping

Ilja Astahovs
Abstract

Travel time depends on various factors, which can be described by data coming from sensors. The author makes an assumption that for the same trip conditions travel time will be the same, and if we can collect enough information on the current trip conditions and find a matching trip which took place in the past, we can estimate the travel time for future trips. The project aim is to design and prototype a system capable of collecting this data, organizing, storing and using it to find matching trips, with the real-time performance being the main consideration. The scope of the system is limited by the needs of a logistic company which wants to be able to track its vehicles and estimate their travel times.

The resulting system is tested in various settings to find out how well it performs. The author identifies the settings which are suitable for the particular implementation and suggests further improvements which are meant to extend the settings.
Acknowledgements

I would like to thank the staff of Acino and especially Jenny Österlund for giving me the opportunity to work for them on this project. Their welcoming attitude and encouragement helped me during the project. I want to extend my thanks to Karl Lindmark and André Lilja for fruitful discussions and good comments concerning my work. Special thanks go to Jan Erik Moström, my supervisor at Umeå University, for his support and feedback.
4.7 Temporal storage of trip statistics
4.8 Trip data interpolation for missed route nodes
4.9 Logging accuracy
4.10 Logging frequency
4.11 Snap-to-node radius

5 Storing data
5.1 Storing trip-related data
5.2 Storing sensor data
5.3 Caching historical trip data
5.4 Storing sensor stations
5.5 Choice of DBMS: SQL vs NoSQL

6 Measuring the similarity of trip conditions

7 Producing travel time estimation

8 Evaluation
8.1 Methodology
  8.1.1 Real-time performance
  8.1.2 Memory consumption
8.2 Results
8.3 Conclusions
  8.3.1 Tracking short trips
  8.3.2 Tracking longer trips
  8.3.3 Performance adjustment

9 Discussion

10 Bibliography

Appendix A: Test results
Appendix B: GUI of the system
Appendix C: Populating the DB with test data
1 Introduction

1.1 Problem statement

Travel time estimation is one of the main features of many Intelligent Transportation Systems, meant to increase quality of service and decrease delay-related costs of logistic companies. Many conventional methods for estimating travel time exist. The data used by these methods is collected from detectors installed along roadways to measure traffic flow, density and speed. Other data sources include video cameras for recognition of licence plates, toll-gates, as well as data of probe vehicles, such as fleet vehicles, equipped with GPS transmitters to track individual vehicles over larger areas.

In this thesis we want to explore another possibility of estimating travel time. We make an assumption that travel time depends on indirect factors such as time of day, day of week, weather factors which impact both road conditions and traffic volume etc. To account for this factors, one acquires data from sensors different from loop detectors, video cameras or probe vehicles. Instead, one can make use of easy-to-access data such as clock, calendar and publicly available data such as weather and road conditions.

We investigate which factors have indirect impact on travel times by reviewing the corresponding literature. We show that the factors which directly impact travel times, such as vehicle flow and density, and thus speed, depend on those indirect factors mentioned above. To describe these factors, we investigate possibilities of acquiring the corresponding real-time sensor data. Given that, we design a system which is capable of acquiring such data local for an arbitrary geographical location. Next, we describe how this data is stored in a database and retrieved. Another important task is to anchor sensor data to a geographical location. To do that, we show how travel data can be recorded while travelling along a route. Finally, based on literature, we decide on how to compare data recorded during different trips along the same route and the current conditions along that route. Our aim is to find a trip for which the conditions were similar to the current conditions.

1.2 Purpose

Our aim is to design a system capable of estimating travel time along a route by assuming that the travel time depends on the various factors which can be measured and recorded for a particular trip and used as a reference the next time we need to travel along the same route during similar conditions. Also, we aim to deliver a functioning prototype to verify the soundness of our design choices.
2 Background

2.1 Factors influencing travel time

A traffic information system capable of estimating travel times should take multiple factors influencing travel time into account [Ran, 2001]. Such factors, in order of importance [Lohmiller and Friedrich, 2012], are as follows: (1) traffic flow, (2) weather conditions, (3) traffic incidents (accidents), (4) time of day, (5) day of week, (6) vehicle type (speed limit for certain vehicle types), (7) traffic composition (public transport and heavy trucks tend to slow down the traffic flow and increase congestion), (8) road construction and maintenance work.

This importance of the top three factors seems to agree with Google’s perception on the topic, which is known to accompany the results produced by its routing service by the following warning:

"These directions are for planning purposes only. You may find that construction projects, traffic, weather, or other events may cause conditions to differ from the map results, and you should plan your route accordingly."

Some of these factors can be considered as primary (demand, weather conditions, accidents), classifying the rest as secondary [Lohmiller and Friedrich, 2012]. In addition, drivers’ individual preferences (as a level of aggressiveness, preferring certain routes etc.) can impact travel times, unless the drivers are required to follow green driving (Swedish: den grøna föraren) guidelines.

2.1.1 Traffic flow

According to [Ran, 2001], the main factors which increase traffic congestion are construction, weather and traffic incidents, so one can combine these other factors when traffic flow data is missing. Also, traffic flow has more impact on travel speed on urban road segments, which means that the weight assigned to this characteristic should depend on the context.

2.1.2 Weather conditions

Meteorological conditions can impact travelling time by decreasing the average driving speed (precipitation directly affects road surface conditions and visibility). Also, certain weather conditions (e.g. lower temperatures) can cause more people switch from walking or biking to taking their cars or travelling by means of public transportation, which increases the traffic congestion [Rondon, 2014]. One study by [Tsapakis et al., 2013] conducted in urban settings suggests that both rain and snow have an impact on travel times and the
delays will depend on its intensity: increasing travel times up to 6.0% for heavy rain, and up to 7.6% for heavy snow. The same study shows that temperature has very small impact on travel times.

Certain weather conditions can also impact traffic safety by impairing visibility and/or road surface conditions.

In the context of travel time estimation, [Ran, 2001] suggests describing weather conditions with four factors: rain, wind, snow and ice, combined with visibility.

2.1.3 Time of day and day of week

Several authors [Lohmiller and Friedrich, 2012; Ran, 2001; Rondon, 2014] point out that amount of traffic is different depending on time of day and day of week. Commuters are the main cause of traffic congestion during morning and evening rush hours. Analogously, the volume of commuters, public transport and vehicles owned by businesses is higher during the weekdays. On a large scale, certain days of year (such as public holidays) or days when a certain event occurs (e.g. large sport- or cultural events) an increase in traffic takes place.

2.2 Data sources

A sensor, in a sense relevant for this thesis, is defined by [www.oxforddictionaries.com, 2015] as:

"A device which detects or measures a physical property and records, indicates, or otherwise responds to it."

Most of the factors impacting travel times are described by the data produced by sensors such as inductive-loop detector, magnetic sensors, video image processors and other traffic flow sensors, [U.S. Department of Transportation, 2006]. rain gauge and snow gauge to measure amount of precipitation [SMHI, 2015], thermometer [SMHI, 2015] etc. These sensors can be found spread across the country along the roads, and are regarded as fixed sensors. Recently, a new type of sensors has emerged, making use of massive spread of GPS and cellphone devices it can employ what is called crowd-sourcing, i.e. to collect data from a large amount of mobile agents (here: moving vehicles). From this data, one can build live-maps showing the current traffic congestion [Bak, 2013].

For this thesis, the data which corresponds to the factors mentioned above can be acquired free of charge from Swedish Transport Administration (STA, Swedish: Trafikverket) and Swedish Meteorological and Hydrological Institute (SMHI). Both of these information providers have open access to their data via REST API. In addition to weather-related information (like road surface temperature and road condition), STA also provides information on traffic accidents and maintenance work. Worth mentioning, STA also provides travel time estimation, which covers though only some selected areas and is not very useful for this project. Previously, a successful attempt of fetching traffic information from STA and using it on a mobile platform was done by [Jonsson and Svensson, 2010].
2.3 Asynchronous data flow

We want the system to be able to process requests from multiple users at the same time. Generally, there are multiple ways to achieve this in software, e.g. using polling, multiple threads etc. For this project, we chose the Node.js runtime for the back-end, which is built upon the idea of asynchronous I/O. Doing I/O operations, especially making calls to external servers, can take arbitrary time to execute, or even fail eventually. To allow such behaviour, Node.js makes these kind of operations run in parallel with other methods. When the method completes, it is required to execute a callback function to signal the completion, optionally passing the resulting data as an argument.

"Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices".

Node.js is written in C and can run code written in JavaScript to create networking applications. With its package manager, npm, it implements a modular approach to import additional functionality to projects, so called modules. These modules facilitate processing requests from a client and so called routing (express) interacting with a MySQL database (node-mysql), providing a HTTP server (http), communicating back and forth with clients in real-time (sockets), making HTTP calls such as GET (request), simplifying dealing with nested callbacks by organizing methods which should be executed consequently in series (async) and provide other useful functionality.

The environment created by Node.js consists of a single process having a single thread, which runs the Event Loop. Prior to executing any method, the method, together with the corresponding callback function, is put into the Event Queue. The Event Loop constantly polls the Event Queue to check for code waiting to be executed. Depending on whether it is a blocking operation or not, the method will be passed for execution to the underlying ThreadPool or will be executed inside the Event Loop. The resulting data from the execution and the corresponding callback function used to process the results will be then put back to the Event Queue. This architecture is depicted by Figure 1 [Dahl, 2009, Hughes-Croucher and Wilson, 2015].

![Figure 1: Node.js architecture](http://misclassblog.com/interactive-web-development/node-js/)

An alternative approach to allow concurrency without relying on the event loop, is creating a new thread for each connection. The downside of this approach is that thread switching takes additional CPU time and maintaining the threads costs additional memory [Dahl, 2009].

---

1 https://nodejs.org/
2 https://expressjs.com/
3 The original picture was found at http://misclassblog.com/interactive-web-development/node-js/.
We want the users to be able to send to the server small bits of GPS data frequently. We want the server to quickly respond to these requests with small responses containing routing information and travel time estimation. Also, we favour the scalability of the solution, since we want to be able manage an arbitrary number of vehicles. According to [McLaughlin, 2011], Node.js is well-suited for these tasks.
3 System design

3.1 Requirements for the system

The following set of requirements was defined for the system during the planning phase:

1. MUST The system must know the current route for each active vehicle.
2. MUST Any time a GPS position is received from some vehicle, the system must estimate travel time for the route travelled by this vehicle.
3. MUST For testing purposes, the system must be able to simulate receiving the historical data for the active vehicles.
4. MUST For each route, the historical data must be stored in a DB.
5. MUST Travel time estimation must rely on historical data.
6. MUST There must be at least a minimalistic GUI to present the results.
7. MAY If the travel time estimation for the vehicle does not match the corresponding arrival frame, the system may generate an alert.
8. MAY The design of the system may provide easy integration into the ProCon system.

3.2 System architecture

A brief overview of the system design is presented by Figure 2. The main parts are: the main server, the database, the routing service, the internal sensor data providers, a number of thin clients and the external sensor data providers.

The main server is the heart of the system. It accepts and maintains connections from clients, handles incoming route requests and delegates route calculation to the underlying modules, communicates with data providers and fetches sensor data both from the external and internal sources, tracks the users along their routes and collects their trip data to be able to provide route estimations. The PostgreSQL/PostGIS database is used for storing the road graph corresponding to the region covered by the system (Sweden), historical trip data and coordinates of the available data stations. The Routing Service, being an extension to the database, is used to find routes between given positions.

Besides acting as an interface to the database and the Routing Service to produce a routing table, RoutePlanner initializes the data structures necessary for speeding-up sensor data collection and historical trip data analysis. It provides capabilities for finding best match among historical trip conditions, and estimating travel time along the given route.

TripLogger is responsible for logging trip conditions as the user transmits its GPS coordinates. It does its best to collect trip data for every road segment found in the Routing Table.
UserManager maintains a list of all connected clients together with their session data. The client side of the system is designed to be thin storing only some minimal information about its state. The reason for that is to minimize its energy consumption by minimizing the number of tasks it needs to perform, minimize hardware requirements and risk of failures. All it needs is to be able to connect to the main server via the WebSocket protocol and transmit its current GPS location and the desirable destination.

User Manager maintains a list of all connected users together with their IDs.

Figure 2: System overview.
4 Acquiring data

4.1 GPS tracking system

In commercial applications, a GPS tracking system can be used for people-, vehicle-, or other asset tracking. Depending on where a GPS tracking system stores the recorded data, GPS tracking can be classified into passive and active tracking. A passive GPS tracking system stores the data in the internal memory of the GPS tracking unit, and allows manually downloading it at some later moment, or downloads it automatically at the specific points along the trip. This kind of system can be used for later route analysis. On the other hand, in an active GPS tracking system the data is transmitted to a server in real time as soon as a new GPS location is logged. The real time approach allows, for instance, depicting the current location of the moving vehicle on a map [Bertagna, 2015].

In our application, the GPS tracking is used for the following purposes:

- to match the vehicle’s position and the destination with the coordinates of nodes in the road graph to be able to find the two positions;
- to determine the vehicle’s position relatively to the data-emitting sensors to figure out the closest sensor;
- for tracking the vehicles position along its trip to be able to tell whether the vehicle is following the route or not.

4.2 Routing

Since the system is required to provide travel time estimation, we need to know which route the user is going to take to the destination, thus the routing functionality is needed. Two alternative approaches were identified: (1) to use the Google Maps service to get driving directions and, what is more important, GPS coordinates describing the points along the route; (2) to use the data provided by OpenStreetMaps together with some routing library to calculate a path between two points in a routing graph.

4.2.1 Google Maps Directions

Google Maps JavaScript API provides Directions service to acquire available routes between two geographical locations. It returns a DirectionsResult object containing available routes. Figure 3 shows its structure. There, overview polyline and polyline are two encoded JSON objects containing 2D points describing the resulting route. The difference between these two is as follows: overview polyline is a scalable representation of the route, which scales depending on the length of the route, i.e. its number of points remains within certain
range no matter how long the corresponding route is, as shown by Table 1. On the other hand, *polyline* does not scale, which results in a large amount of data and processing time.

![Figure 3: DirectionsResult object.](image)

<table>
<thead>
<tr>
<th>Number of points in <em>overview_polyline</em> object.</th>
<th>km</th>
<th>overview_polyline</th>
<th>polyline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Umeå, Erskoda - Umeå, Umedalen</td>
<td>13.5</td>
<td>187</td>
<td>448</td>
</tr>
<tr>
<td>Umeå - Skellefteå</td>
<td>134</td>
<td>284</td>
<td>3640</td>
</tr>
<tr>
<td>Umeå - Sundsvall</td>
<td>258</td>
<td>245</td>
<td>7690</td>
</tr>
<tr>
<td>Umeå - Stockholm</td>
<td>630</td>
<td>221</td>
<td>16718</td>
</tr>
</tbody>
</table>

Table 2 and a corresponding Figure 4 shows how much storage space is required to store different amount of nodes. Since we index our routes geospatially by the locations of its nodes, indexing takes additional space. One can draw a conclusion, that amount of storage space is linearly proportional to the number of nodes. Also, in our case, indexing takes about 1.5 times more space than nodes itself.

Figure 5 and Figure 6 show how node density looks in reality. In case of *polyline* object, density of nodes is dictated by the road curvature. This raises a question: which node density is suitable for our system? In our system, we use nodes for two purposes: (1) to be able to tell if a vehicle is following the predefined route, (2) to anchor sensor data to a certain location. Thus we make an assumption that node density is related to how frequently turns occur on a road and how dense sensors are spread. There is no need for multiple nodes on a relatively short road segment with no turns since: (1) the vehicle cannot change its route, (2) the conditions are likely to be the same along the whole segment. Figure 7 shows the density of weather stations, where we fetch our data from, around city of Umeå. This image was produced by parsing a JSON file containing all available weather stations provided by SHMI. The same way, it has been experimentally verified that there are ca 25 weather stations along E4 highway between Sundsvall and Skellefteå, which corresponds to a ca 400 kilometers long road segment. The density for this particular road is thus 1 weather station per 16 kilometers.

The sensor density may vary a lot depending on type of data this sensor produces. For instance, traffic flow sensors may be spread much densely.

### 4.2.2 OpenStreetMaps

*OpenStreetMaps* (OSM) provides free maps over the whole world. It is based on crowdfounded data collected by its community members and available to download for free for everyone. The data include description of road network (motorways, roads, cycleways, footways etc.), public transport- and railway network, amenities, buildings etc. There is an API provided for editing or fetching data from OSM. As of this writing, OSM does not have its own routing API, but it provides all the data required for use with a third party
Figure 4: Table size depending on number of nodes.

Figure 5: Acceptable node density when extracting route nodes from *overview.polyline*. Road segment length: ca 200 m. Number of nodes: 6.
Figure 6: Redundant node density when extracting route nodes from polyline. Road segment length: ca 200 m. Number of nodes: ca 25.

Figure 7: Density of weather stations around city of Umeå.
<table>
<thead>
<tr>
<th>number of nodes</th>
<th>table size, kB</th>
<th>indexes size, kB</th>
<th>total size, kB</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>8</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>187</td>
<td>16</td>
<td>40</td>
<td>56</td>
</tr>
<tr>
<td>284</td>
<td>24</td>
<td>48</td>
<td>72</td>
</tr>
<tr>
<td>448</td>
<td>32</td>
<td>72</td>
<td>104</td>
</tr>
<tr>
<td>3640</td>
<td>248</td>
<td>416</td>
<td>664</td>
</tr>
<tr>
<td>7690</td>
<td>520</td>
<td>864</td>
<td>1384</td>
</tr>
<tr>
<td>16718</td>
<td>1120</td>
<td>1840</td>
<td>2960</td>
</tr>
</tbody>
</table>

4.2.3 pgRouting

`pgRouting` is an extension for PostgreSQL/PostGIS which provides API for finding routes and solving similar problems. It implements Dijkstra’s and A* search algorithms for finding shortest path given a topology network as input data. In this case, the routing network topology should include the structural description of the underlying graph including: identifier for each edge in the graph, identifier of the source- and target nodes, as well as cost (optionally, reverse cost) for each edge. Additionally, to use A* one must provide coordinates for each node belonging to the graph. [pgRouting Project, 2015]

4.2.4 Creating a routing graph

To prepare the data required for pgRouting, the following steps were taken:

1. The portion of Planet.osm data dump was obtained, we were only interested in obtaining data related to Sweden. Depending on the tools, we use data in .pbf format to build a topology network and the same data in .osm format to fetch nodes describing all the available roads in Sweden.
2. By using `osm2po` utility [Moeller, 2015], which is multi-platform, routing network data was extracted from the .pbf file and put into a sequence of SQL statements. We used these to populate our PostgreSQL database with the routing information. The result is a table containing a graph whose edges are the road segments, together with IDs for the nodes constituting the graph’s vertices and the corresponding costs to travel along these edges, in both directions [Boston Geographic Information Systems, 2015]. Table 3 shows how the routing table is stored in the DB. Figure 8 shows that in this table, the edges correspond to the road segments between road junctions, represented by solid circles. The small circles are points depicting the road curvature. These are stored as polylines belonging to each edge in the graph and are not directly involved in the route search. However, the length of these polylines provide us the length of the corresponding edges. On Linux, one can use `osm2pgrouting` tool for the same purpose [pgRouting Project, 2015].

---

Table 3: Road network graph produced by *osm2po* as its stored in sweden_routing. **id** identifies an edge, **source** and **target** identify the nodes as the edge is traversed in forward direction, **km** is the length of the edge in kilometres, **lng1** and **lng2**, **lat1** and **lat2** are coordinates of source- and target nodes, respectively. **geom**/**way** is an encoded polyline describing the shape of the road which corresponds to the edge.

<table>
<thead>
<tr>
<th>id</th>
<th>source</th>
<th>target</th>
<th>km</th>
<th>lng1</th>
<th>lat1</th>
<th>lng2</th>
<th>lat2</th>
<th>geom/way</th>
</tr>
</thead>
</table>

Figure 8: Road network graph produced by *osm2po*.

Thus, we imported the routing network topology data into our PostgreSQL database. Now, to find routes, we can make a request to the DB calling the corresponding function from pgRouting API, namely *pgr_dijkstra*.

4.2.5 Storing the routes

Internally, the routes are stored as objects containing a list of edges as shown by Figure 9.

4.3 Representing sensors

From the perspective of the system described in this thesis, we can generally regard as a sensor any characteristic that can describe a physical location or a moment in time. Thus we can think of a clock as being a sensor since it can classify a moment in time as morning or evening rush, or a day as a weekday or a weekend. To allow such generalisation, we can first classify the sensors related to the system as:

- **locals**, which are only valid for the geographic location (or area) where these sensors are installed, and
- **globals**, which are valid globally for any geographic location covered by the system.
Figure 9: Route object. See Section 5.3 and Figure 16 for a detailed explanation on the history[].

4.4 Collecting the trip statistics

At any given time, trip conditions along any road segment can be described by a set of attributes which correspond to the values of the sensors covering that segment. Thus, one can construct a vector:

\[ X_{i,j} = \begin{bmatrix} x_{1,j} \\ x_{2,j} \\ \vdots \\ x_{n,j} \end{bmatrix}, \quad (4.1) \]

where \( x_{1,j}, x_{2,j}, \ldots, x_{n,j} \) are values of \( n \) sensors describing trip conditions for a road segment corresponding to the \( i \)-th edge during \( j \)-th trip. When logging a trip along \( j \)-th trip, this vector corresponds to sensor_data[] array belonging to edge[i] as depicted by Figure 10.

One of the first decisions to make was about what to anchor trip statistics data to. Since some of our sensors describe different conditions valid for certain location or area, a sound idea is to anchor our observation to these. Also, because we want to store travel times, which as we believe are related to trip conditions, we choose to anchor trip conditions to intermediate points which make a route. Trip statistics and travel time are supposed to be collected continuously as a vehicle is travelling along the route.

How often should we record trip data? Assume, a route is split into \( n \) segments, and we collect travel time and trip conditions for each segment. We consider the following aspects:

- We want to be able to calculate travel time for the whole route. To achieve this, we only need a single segment \( (n = 1) \), i.e. we subtract value of the time-stamp which we recorded at the beginning of the route from the one we recorded when we have completed the route. For this approach to be useful, the trip conditions have to be considered constant along the whole route. While some of our travel factors (for instance, day of week) are likely to remain constant during a single trip (we do not
account for overnight traffic), many of the sensors will produce different values for different points, unless the route is so short that it is bounded by an area covered by a single set of sensors.

- We want to have as complete picture of trip conditions as possible, so missing some existing sensors is undesirable. To achieve this, we can track the available sensors and fetch the data when we come close enough to a sensor. Thus, the frequency of logging is dictated by density of the sensors.

- We want the data collected for one route to be re-usable for travel time estimations for other routes which include, partly or completely, this route. To achieve this, we want to split routes into segments in such a way that if two routes overlapped, they would share common segments. This implies splitting the routes at the road junctions and logging at the same locations when the same roads are traversed.

Given this, the following conclusions can be drawn: (1) we need a graph containing all routes so that we can store travel times and the corresponding trip conditions for every edge (route segment), (2) we might need to split the segments further to be able to cover all sensors. Our aim is to log for every node along the given route.

![Trip Data Structure](image)

**Figure 10:** Trip data structure used for temporal storage of trip data on the server.

### 4.4.1 Time-wise GPS logging

For every point a vehicle passes through along the route, the system does its best to record the trip data. To decide which point to log for, the system needs to get the edge which is nearest to the vehicle’s current position. By fetching the nearest edge, the system can figure out whether the vehicle is following the current route or not. If it is, the system picks the source or the destination node for that edge and tries to create a log entry for this point (see Figure 11). The vehicle’s GPS position gets recorded and matched with sensor data only when the vehicle is close enough to a route point. If the vehicle comes closer to the point for which the system already fetched the sensor data previously during the ongoing trip, the sensor data and the corresponding time-stamp will be updated and overwritten. This behaviour is illustrated by Figure 12 (the symbols used by this and other figures are explained by Table 4).
Figure 11: Logging trip.

Figure 12: Logging trip data. **Step 1:** the vehicle’s reported position is too far away, the logging request is ignored. **Step 2:** the vehicle has entered the proximity of the route node, the request is recorded. **Step 3:** the vehicle is even even more close now to the route node, the request is recorded by overwriting the previously recorded value. **Step 4:** the vehicle starts moving away from the route node, the request is ignored. See Table 1 for the symbol explanation.
Table 4 The symbols and their meanings.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a node along the route for which the system will try to log trip data,</td>
</tr>
<tr>
<td></td>
<td>an external sensor station (a data station),</td>
</tr>
<tr>
<td></td>
<td>the logging radius,</td>
</tr>
<tr>
<td></td>
<td>a trip point for which the data logging has been accepted,</td>
</tr>
<tr>
<td></td>
<td>a trip point for which the data logging has been rejected since it is</td>
</tr>
<tr>
<td></td>
<td>outside the logging radius.</td>
</tr>
</tbody>
</table>

4.5 Fetching current sensor data for a given route

For every node belonging to a route, we can get a matching location on the map. For every such location, we want to fetch data from the nearest data station. The process is illustrated by the Figure 13.

1. The vehicle’s driver chooses a destination to go to. Together with the vehicle’s current position, it is put into a `find route` request and sent to the server.

2. The server calculates the route between source and destination and parses the resulting structure. For every node, request the external factors from the corresponding data providers.

3. The data providers reply with the requested data.

4. The server stores the received data in the route route structure and uses it to find the matching sequences in the history.

Figure 13: Fetching data from data stations. See Table 4 for the symbol explanation.
4.6 Locating the closest data station

To be able to quickly locate the closest data station for every registered information source is crucial for the performance of the system. As vehicles travel along their routes, they may move closer or further from their current data stations. At some point, a vehicle can move far enough from the current data station so it becomes necessary to switch to another data station. Depending on the density of data stations, and the nature of the corresponding sensors, one chooses how often to re-locate the closest data station. The simplest way to find the closest data station is too delegate this task to the database module.

4.7 Temporary storage of trip statistics

Every vehicle tracked by the system knows the route it is expected (but not forced) to follow. During the trip, the system continuously matches the position of the vehicle with the route to find the closest node belonging to the route, if the route is within a certain radius, trip statistics will be recorded for this node. For that, we need to be able to find the nearest node quickly, in a non-sequential fashion. We take the same approach adopted by PostGIS for indexing its 2-dimensional data, i.e. R-Trees [PostGIS Manual, 2015].

Generally, R-Trees are used to organize d-dimensional data into a tree by partitioning the data set into d-dimensional rectangles. The leaves of this tree are actual data, its parent nodes are Minimum Bounding Rectangles (MBRs) which spatially contain the data, its parent, in turn, are even larger MBRs. This hierarchy repeats all the way upwards until one reaches the root of the tree. For each R-Tree, there is a minimum (m) and maximum (M) number of entries for each node to contain, including the root [Guttman, 1984]. Comparing to linear time \( O(n) \) required for sequential search, searching in R-Tree is \( O(M \log_M n) \), on average. The actual performance depends on the partition, since MBRs are allowed to overlap. Also, R-Tree search can be thought of as a “filtering mechanism” to narrow down the search range [Course papers at Bowdoin College, 2008]. Bearing this in mind, we record and temporary store trip statistics the way it is shown by Algorithm 1. For R-Tree functionality, RBush was used, JavaScript library compatible with Node.js.

4.8 Trip data interpolation for missed route nodes

During the trip, data logging can fail for some route nodes. Possible causes can be: the GPS transmitter’s hardware failure, connection loss between the GPS transmitter and the server, logging frequency is too low or the vehicle’s speed is too high in relation to each other to be able to cover each route node. Also, for routes produced by route-planning services, for some route segments, the distribution of the route nodes can be specially dense. For instance, to represent roundabouts precisely enough, because of its curvature, one needs to place the corresponding nodes close to each other. See Figure 14 for how Google Maps handles roundabouts.

\[\text{http://en.wikipedia.org/wiki/R-tree}\]
\[\text{https://github.com/mourner/rbush}\]
Algorithm 1: Tracking the position of the vehicle to record trip conditions along its route.

**Input:** $nodes = \{n_0, n_1, \cdots, n_i\}$ is a set of nodes belonging to the route, in sequential order, together with space-holders to store trip conditions for every node; $rect$ is a minimal bounding rectangle to include a circle around current position of the vehicle with radius, $r$, being the snap-to-node radius.

**Result:** an array containing trip statistics for each node along the given route.

Initialize an $rt$, a R-Tree data structure, by bulk-copying $\{\{n_0.lat, n_0.lng, 0\}, \{n_1.lat, n_1.lng, 1\}, \ldots, \{n_i.lat, n_i.lng, i\}\};$

**while** the vehicle follows the route and has not reached the destination **do**

\[
\text{pos} \leftarrow \text{current position of the vehicle};
\]

\[
\text{selectedNodes} \leftarrow \text{nodes contained within rect for the current pos};
\]

**foreach** node $j$ in the selectedNodes **do**

\[
\text{dist}_j \leftarrow \text{Euclidean distance between pos and node}_j;
\]

\[
\text{jNearest} \leftarrow \text{index } j \text{ of the smallest dist}_j;
\]

\[
\text{nodes}_{j}.data \leftarrow \text{trip conditions for nodes}_j;
\]

To estimate travel time for a selected route, we need trip data to be available for every node and every segment along the route. For robustness, the system allows the traveller to miss logging trip data for some nodes along the route. Except for the very first and the very last node, the system can linearly interpolate the trip data for the intermediate nodes.

Given a sequence of route nodes $n_0, n_1, n_2$, assume we could successfully snap trip data to $n_0$ and $n_2$, leaving $n_1$ with no recorded trip data by the end of the trip. Denote $\alpha_0$ and $\alpha_2$ the corresponding values for some sensor for the first and the last node, respectively. The value for the missing $\alpha_1$ can be approximated as follows:

\[
\alpha_1 = (\alpha_2 - \alpha_0) \cdot \frac{\text{dist}(n_0, n_1)}{\text{dist}(n_0, n_2)},
\]

where $\text{dist}(n_0, n_1)$ is the distance between the nodes $n_0$ and $n_1$, and $\text{dist}(n_0, n_2)$ is distance between the nodes $n_0$ and $n_2$. Similarly, one can interpolate trip data for arbitrary many route nodes.

### 4.9 Logging accuracy

Accuracy of a GPS receiving device depends on many factors, among other things, on the surroundings of the vehicle. Urban areas as well as other obstructed areas (valleys etc.) are known to decrease the number of directly "visible" GPS satellites and to cause reflections and weakening of the GPS signals, both resulting in lower accuracy.

Based on some accuracy tests [Shaner, 2013], we can expect modern smartphones with an integrated GPS chip-set to provide $\pm 3 - 5m$ accuracy for open areas. Connecting an external GPS receiver to a smartphone can improve the accuracy for open areas to be less

---

4http://wiki.openstreetmap.org/wiki/Accuracy_of_GPS_data
than ±3m. A high-accuracy GPS receiver can increase the accuracy even further, reaching ±1m error. Older studies [Modsching et al., 2006] focusing on consumer-level external GPS receivers, provide us a baseline for our further assumptions: ±2m for open areas and ±15m for urban streets. Recently, one blogger [McCormick, 2014] reported that combining a smartphone with a non-expensive consumer-level GPS receiver would bring the accuracy under ±10m, for both open- and obstructed areas. The announced accuracy for this kind of GPS receivers, from its manufacturer, is ±2.5m [Dual, 2015].

The results of these surveys has to be taken into account when deciding on snapping-to-route radius. We assume that it is realistic to expect the GPS receiving device on the client side to have accuracy of ±10 meters for most types of areas.

4.10 Logging frequency

How frequently a travelling vehicle transmits its GPS position to the server is important for processing the trip data. Lower frequencies can cause missing trip data for some of the route nodes. Higher frequencies produce excessive data flow which can impact the performance of the system. Also, because the client-side of the application is supposed to be run on a mobile device, higher GPS update rates can be energy-inefficient.

GPS tracking units used for tracking fleets usually update at 1, 2 or 5 minutes intervals 5. For consumers, GPS tracking receivers with rates of 1Hz and 0.2Hz (i.e. 1 second- and 5 seconds intervals, respectively) [Garmin International, Inc., 2015] and 1Hz [Dual, 2015] are available.

Also, logging frequency directly affects the value of the snapping-to-route radius - higher frequencies reduce the radius.

---

5http://en.wikipedia.org/wiki/GPS_tracking_unit
4.11 Snap-to-node radius

Given the route nodes can be either stand-alone (e.g., being part of a long straight road segment) or form a group of nodes (in case of roundabouts as depicted by [14], we want the system to do its best to record trip data for every stand-alone node while allowing it to skip most nodes belonging to a group and apply the interpolation instead.

To derive a snap-to-road radius to be just large enough to guarantee that a route node will not be missed provided the GPS receiver is functioning to its promises, we take into account the distance, in meters, travelled by the vehicle within one update interval of the GPS device (denoted by $d$) and its accuracy, in meters, (denoted by $e$), as depicted by Figure 15. Thus,

$$ r = \frac{d}{2} + e, \quad \text{(4.3)} $$

where $r$ is the snap-to-road radius, i.e. the maximum distance from the current position of the vehicle to a route node for the trip data to be assigned to this node.

Given GPS update rate is $f$ Hz and vehicle’s speed is $v$ km/h,

$$ d = \frac{v \cdot 1000 \cdot f}{3600} \quad \text{(4.4)} $$

Table 5 shows different values for snap-to-node radius depending on GPS update rate and vehicle’s speed.

<table>
<thead>
<tr>
<th>Speed (km/h)</th>
<th>1 update per second (1 Hz)</th>
<th>1 update per 5 seconds (0.2 Hz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>19.2</td>
<td>35.8</td>
</tr>
<tr>
<td>50</td>
<td>21.9</td>
<td>49.7</td>
</tr>
<tr>
<td>70</td>
<td>24.7</td>
<td>63.6</td>
</tr>
<tr>
<td>80</td>
<td>26.1</td>
<td>70.6</td>
</tr>
<tr>
<td>90</td>
<td>27.5</td>
<td>77.5</td>
</tr>
</tbody>
</table>

To be able to match the trip data to every stand-alone route node, we want to choose the snap-to-node radius to be the maximum value over all the speeds for the chosen GPS update rate. Thus, given 1 Hz rate, we set the snap-to-node radius to 28 meters. For the logging rate optimal with the respect to the performance of the system, see page 37.
Figure 15: Snap-to-node radius. \( d \) denotes the distance travelled by the vehicle within one update interval of the GPS device and \( e \) is its accuracy. See Table 4 for the symbol explanation.
5 Storing data

As suggested by [van der Veen et al., 2012], indexing should be always used when possible both for RDBMS and NoSQL databases. Even though it puts a slight overhead on write operations and storage, it dramatically increases the performance of reads.

5.1 Storing trip-related data

To allow collection of data from an arbitrary number of sensors during a trip, the trip-related data is stored in two separate tables. One table is used for storing fixed amount of data for each edge in the routing table, while a separate table contains an arbitrary number of records for each edge, each record containing data from a data sensor available for this edge. Table 6 presents the schema for storing travel time for edges recorded during different trips. Table 7 shows the schema for storing the sensor data corresponding to those trips for each edge.

Table 6 times

<table>
<thead>
<tr>
<th>tripid:integer</th>
<th>edgeid:integer</th>
<th>ttime:integer</th>
</tr>
</thead>
</table>

Table 7 sensor_data

<table>
<thead>
<tr>
<th>tripid:integer</th>
<th>edgeid:integer</th>
<th>sensorid:char(15)</th>
<th>value:real</th>
</tr>
</thead>
</table>

5.2 Storing sensor data

When deciding on how to store sensor data, we have to consider the following factors:

- storage space
- time to make a request
- time to process a response

5.3 Caching historical trip data

To be able quickly access historical trip data for the given route, the trip data retrieved from the DB is stored for every edge belonging to the route as shown by Figures 9 and 16. For every edge, the history contains sensor data and the corresponding travel time.
Figure 16: History data cached for every edge along the actual route.

Putting together all trip history for the \(i\)-th edge yields the matrix \(5.1\):

\[
\begin{bmatrix}
X_{i,1} & X_{i,2} & \cdots & X_{i,j}
\end{bmatrix} = \\
\begin{bmatrix}
x_{1,1} & x_{1,2} & \cdots & x_{1,j} \\
x_{2,1} & x_{2,2} & \cdots & x_{2,j} \\
\vdots & \vdots & \vdots & \vdots \\
x_{n,1} & x_{n,2} & \cdots & x_{n,j}
\end{bmatrix},
\]

(5.1)

where \(j\) is the total number of trips through the \(i\)-th edge for which there are trip conditions recorded. This matrix corresponds to history[] array (see Figure 16) describing travel conditions for all trips ever made along edge[i] (see Figure 9).

### 5.4 Storing sensor stations

Given vehicle’s position, we want to be able to locate the nearest sensor station to fetch sensor data from. Thus, we need to store in the database position for every station as presented by Table 8. Also, to make the search faster, the table is indexed by the position.

<table>
<thead>
<tr>
<th>sensor_station id:char(15)</th>
<th>pos:geography</th>
</tr>
</thead>
</table>

Every sensor station (in STA’s terms: measurement site) declares the following information about itself [Trafikverket, 2015]:

- record ID, e.g. "SE_STA_VVIS227",
- name, e.g. "Arlanda",
- location, a pair of latitude and longitude, 59.6434669, 17.893177,
- a list of specific characteristics, which are types of available sensors, e.g. "temperatureInformation", "trafficFlow", see Table 9 for the full list.
- other information which the system does not use: the road and county numbers which correspond to the site’s location etc.

By using the service provided by STA, we can fetch a file in JSON format containing a list of all the sensor stations together with a list of sensors available for each station and their
current values. We process this file by picking only those types of sensors which we are interested in.

<table>
<thead>
<tr>
<th>Table 9 Sensor types declared by STA.</th>
</tr>
</thead>
<tbody>
<tr>
<td>humidityInformation</td>
</tr>
<tr>
<td>individualVehicleMeasurements</td>
</tr>
<tr>
<td>pollutionInformation</td>
</tr>
<tr>
<td>precipitationInformation</td>
</tr>
<tr>
<td>pressureInformation</td>
</tr>
<tr>
<td>radiationInformation</td>
</tr>
<tr>
<td>roadSurfaceConditionInformation</td>
</tr>
<tr>
<td>temperatureInformation</td>
</tr>
</tbody>
</table>

5.5 Choice of DBMS: SQL vs NoSQL

Database management systems can be classified by its data model as either relational (RDBMS) or non-relational (NoSQL) DBMSs. RDBMS organizes the data into tables (relations), where each row (tuple) describes an object or concept with columns being its properties (attributes). Also, one can specify key attribute to make each row unique and use it to create links between tables. The tables can be manipulated using SQL language [Elmasri and Navathe, 2011]. Examples of RDBMS include MySQL, PostgreSQL and Oracle. The assumptions behind the design of RDBMS favors fixed table structure and data density. Changing the structure of a table by adding new columns is a costly operation and creates empty columns, therefore it should be avoided. Also, RDBMS are known to scale well vertically but not horizontally. This means one can only increase the capacity of the system to certain extent by adding new hardware (to increase storage or performance), but as data volume goes beyond what can be stored on a single machine, data distribution across several machines impacts performance. This is due to data consistency and integrity constraints adopted by RDBMS [Shashank, 2011].

On the other hand, NoSQL databases do not have fixed structure forced by schemas. They store data in various formats: key/value pairs (Dynamo), document stores (CouchDB, MongoDB), and hierarchical (graph) structures (Neo4J). Loose structure of NoSQL databases allows insertion of new attributes quickly without having to alter the schema. Adding new attributes does not increase sparsity, since empty values are not stored [Shashank, 2011] (in contrast to RDBMS), which makes NoSQL a good candidate for Agile development where new attributes can be added at low cost at any stage of development.

As mentioned by [Shashank, 2011], RDBMS have been the first choice for many developers due to its known stability. On the other hand, according to [Stonebraker, 2010], many consider switching to NoSQL technologies because of performance and flexibility. The author identifies four major sources of lower performance for RDBMS: logging, locking, latching and buffer management. He suggests that it is possible to eliminate these factors. A study conducted by [van der Veen et al., 2012] compares performance of RDBMS (PostgreSQL) and NoSQL (Cassandra and MongoDB) for handling sensor data. It highlights the differ-

\[^{1}\text{http://www.mongodb.com/nosql-explained}\]
ences in performance depending on whether writes/reads are single or multiple. Specifically for collecting and analysing sensor data, write requests to a database from these sensors are considered to be single and small, while read request requires transmitting large amount of data.

For our system, one can argue if this is the case. When collecting our trip data, we are free to make a choice on whether to buffer data received from the user along the trip and write it at once when the trip is finished or to write trip data for every node continuously. This choice depends on the performance of the system: if the amount of data generated during the trip is so large that writing it to the database all at once will introduces long delays, doing it continuously during the whole trip is a better option. Thus, we will postpone this decision until we can test the system.

The same study \cite{van_der_Veen_2012}, suggests that the performance of the databases in question depends on whether the requests are done from a single client or multiple clients.

As by design of our system, all heavy workload related to finding routes and calculation of similarity between trips is done on the server side. All communication with the database is done through a single module, \textit{DBManager}, therefore we assume there is a single database user present in our system. For a single client, \cite{van_der_Veen_2012} concludes that PostgreSQL beats the NoSQL databases when performing multiple reads, while Cassandra and MongoDB show best results for multiple writes and single writes, respectively. For single reads, MongoDB shows best performance.

Since our system deals with geospatial data, a database with support for geospatial functionality is highly desirable. Not only it allows to delegate queries as ”Find the nearest data station” or ”Find all routes which bring me to a location no further than 50 meters from a requested destination”, thus saving development time, but also provides geospatial indexing (e.g. by position) to speed-up these requests. Also, usually it implements a number of geospatial datatypes, specified by \textit{GeoJSON} standard. Many databases are known to support it. PostgreSQL can be extended by \textit{PostGIS} to enable spatial data types and relationships. PostGIS implements the \textit{Simple Features} standard specified by \textit{Open Geospatial Consortium} and provides a reach set of functionalities \cite{PostGIS_2015}. MySQL implements some of the spatial functionality mentioned above, but with certain limitations (this is valid for MySQL 5.6 and earlier), which can result in worse precision if compared to PostGIS \cite{Rubin_2013}. There exist many other geospatial extensions for different RDBMS, and some RDBMS provide the functionality natively.

MongoDB provides implementation of GeoJSON types\footnote{http://docs.mongodb.org/manual/applications/geospatial-indexes/} and spatial functionality, allowing geospatial indexing and functionality as ”intersection, within and nearness” \cite{Chodorow_2013}.
6 Measuring the similarity of trip conditions

For every route segment, for its starting and ending node, we store a number of sets of parameters which describe the weather conditions for the trips done previously along this route. Given a set of parameters describing current weather conditions, we want to find K nearest neighbours for this set among all the available sets. We calculate the average travel time over these trips and use it as an estimation of travel time along the corresponding segment. This problem can be solved by using The K-Nearest Neighbour Algorithm, as suggested by [Keller et al., 1985], which is presented, with modifications, by Algorithm 2.

Algorithm 2: The modified K-Nearest Neighbour Algorithm. For a route segment, finds K pairs of similar condition vectors for trips through this segment.

**Data:**

$V=\{v_1, v_2, \cdots, v_n\}$ is a set of vectors, one vector per trip, where each vector contains sensor values for the first node of a route segment.

$W=\{w_1, w_2, \cdots, w_n\}$ is a set of vectors, one vector per trip, where each vector contains sensor values for the second node of the same route segment.

**Input:** $x$ and $y$, vectors containing current sensor values for the first and the second nodes of the same route, respectively; $K$, number of similar trip conditions to find.

**Result:** K pairs of similar condition vectors

Initialize i=1;

repeat

$\quad d_1 \leftarrow$ distance between $v_i$ and $x$;

$\quad d_2 \leftarrow$ distance between $w_i$ and $y$;

$\quad d \leftarrow \frac{d_1 + d_2}{2}$

if $i \leq K$ then

Include $x_i$ and $y_i$ in the set of K-nearest pairs of neighbours;

else

if $d$ for this pair $(v_i, w_i)$ is less than $d$ for the the previous pair of nearest vectors

then

Delete farthest in the set of K-nearest pair of neighbours;

Include the pair $(v_i, w_i)$ in the set of K-nearest pair of neighbours.

end

end

Increment i;

until K-nearest neighbours found;

To compute the distance between two vectors, as mentioned in the algorithm, different methods, generally known as distance functions, exist. One faces two challenges when
picking a particular method: (1) combining continuous and categorical (nominal) data and (2) dealing with high dimensionality of data \cite{Li and Li 2010}. There exist different distance metrics suited for continuous and categorical data. For continues data, some notable examples include Euclidean distance (Formula 6.1) and Minkowski distance (Formula 6.2), which is a generalization of Euclidean distance. For Minkowski distance, when \( r \) is 1, the case is known as Manhattan distance. For categorical data, Overlap metric (Formula 6.3) is widely \cite{Li and Li 2010} used for machine learning. For binary data, this metric is called the Hamming distance.

\[
d(x,y) = \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2} \tag{6.1}
\]

\[
d(x,y) = \sqrt{\sum_{i=1}^{k} |x_i - y_i|^r} \tag{6.2}
\]

\[
d(x,y) = \sum_{i=1}^{k} \sigma(x_i,y_i), \text{ where } \sigma(x_i,y_i) = 0 \text{ if } x_i = y_i, \text{ 1 otherwise.} \tag{6.3}
\]

\[\text{[Wang and Sun, 2012] categorize distance metrics as linear and non-linear. The linear distance is calculated on data mapped linearly into some space, while a non-linear mapping is applied to data to calculate the non-linear distance. Next, we will present two approaches which use different types of distance metrics.}\]

\[\text{[Giraud-Carrier and Martinez, 1995] give background information on how to choose an appropriate distance function. They highlight that the function suitable for homogeneous data sets may produce inconsistent results when applied to heterogeneous data sets. Also, they show that it is important to normalize the data prior to applying a distance function to it. Relating to our system, we cannot meaningfully compare raw values of air temperature (for Sweden, values on the scale } -50^\circ \text{C to } 50^\circ \text{C) and precipitation (0 mm to 300 mm). To overcome this, we can apply normalization (also known as feature scaling) either during the data preprocessing step by the formula shown by Formula 6.4} \text{[Aksoy and Haralick, 2000], or inside the distance function, as will be shown below. Either ways, the data is mapped linearly.}\]

\[
X_i = \frac{X_i - X_{Min}}{X_{Max} - X_{Min}} \tag{6.4}
\]

The authors propose a distance metric for heterogeneous data, which applies either of two distance functions to every pair of attributes depending on whether these attributes are linear or nominal and then sum the distances together. Their distance functions are based on Euclidean distance. In their paper, they propose a more elaborated formula to create rules by ”mapping a set of points, rather than a single point, to a given output value”. For this project, we simplify their formula for Distance for HETerogeneous Spaces (DHET), a metric suitable for sets of heterogeneous data, and use only its parts as shown by Formula 6.5. In the terms used by this project, the symbols used in the formula can be interpreted as

\[1\text{http://xlinux.nist.gov/dads/HTML/manhattanDistance.html}\]

\[2\text{http://xlinux.nist.gov/dads/HTML/HammingDistance.html}\]
follows: $x$ and $y$ are two vectors containing sensor data for the same route segment collected during two different trips. $x_i$ and $y_i$ are values for the sensor $i$ for the trips corresponding to $x$ and $y$, respectively. range$(i)$ is the range of values produced by the sensor $i$. One can see that both functions $\eta$ (Formula 6.6) and $\lambda$ (Formula 6.7) produce values belonging to the range $[0, 1]$, which addresses the normalization requirement.

$$DHET_{simplified}(x, y) = \sqrt{\sum_{i=1}^{n} \left\{ \frac{\eta(x_i, y_i)}{\left[\lambda(x_i, y_i)\right]^2} \right\} \text{if attribute } i \text{ is nominal}} \quad \text{where} \quad (6.5)$$

$$\eta(x_i, y_i) = \left\{ \begin{array}{ll} 1 & \text{if } x_i \neq y_i \\ 0 & \text{otherwise} \end{array} \right., \quad (6.6)$$

$$\lambda(x_i, y_i) = \frac{|x_i - y_i|}{\text{range}(i)}. \quad (6.7)$$

[Qingqing Mu and Li, 2010] present another approach both to the normalization and the choice of the distance function. Applied to load forecasting of power systems, they assume the load to be affected by the day type and the weather and base their estimations on finding days with similar conditions. To normalize the data, they use an index-mapping database, which maps most of the data to the interval $[0, 1]$, while some data can be mapped to an extended interval $[0, a], a > 1$ to "embody the leading role of a certain factor". For example, they map the weekdays from Monday to Saturday to be $0.10, 0.2, ..., 0.6$, while assigning Sunday the value of $2.0$. This kind of a non-linear mapping is also applied to those temperatures which exceed a certain maximum (or a minimum) value. As a distance function, they use the cosine similarity (see Formula 6.8). One can argue whether the latter is a good metric when applied to measuring the similarity of weather conditions: the cosine similarity only takes into account directions of the vectors, ignoring their magnitudes. For example, it will mistakenly report weather conditions on one day (20°C, humidity 100%) being similar to those on another day (10°C, humidity 50%). For this reason, the first approach was chosen to be used in the project covered by this thesis.

$$d(x, y) = \frac{\sum_{k=1}^{m} x_k y_k}{\sqrt{\sum_{k=1}^{m} x_k^2 \sum_{k=1}^{m} y_k^2}} \quad (6.8)$$

Worth mentioning, when applied to measuring similarities between two atmospheric states, as stated by [Matulla et al., 2007], the Euclidean distance proved to give satisfactory results for the most factors, although it is also stated that the choice of a distance function should depend on a particular factor.

Some of the factors which impact travel times are best expressed in terms of true/false, or binary (or generally, nominal), values. These factors include day of week (which is either a weekday or a weekend), morning (and evening) rush (which is either true or false depending on whether a moment in time belongs to a certain interval or is outside of it), ice on the road surface (true or false) etc. It is possible for these factors to have linear values (for instance, one can use current time instead of a boolean value indicating rush hour, but then "15:00" will have greater distance from "8:00" than "12:00" even though travelling conditions at 12:00 will not necessarily differ from those at "15:00" since both time moments are outside of the rush hours time slots.

31
Finally, when we have found trips with similar conditions for every segment along the route, we compute the estimated travel time for this route by calculating a sum of the average travel times over all route segments as illustrated by Algorithm 3.

**Algorithm 3**: Travel time estimation for a given route based on finding trips having similar conditions for each segment along the route.

**Data**: $T = \{t_1, t_2, \cdots, t_n\}$ is a set of all recorded trips.

**Input**: $a$ and $b$ are source and destination points, respectively;

$K$ is a number of similar trip segments to base the estimations on.

**Result**: $\tau$, travel time estimation for a trip from $a$ to $b$.

$\tau \leftarrow 0$;

Find a route from $a$ to $b$;

**foreach** segment $s_i$ along this route do

- fetch current conditions for the first and the last node belonging to this segment;
- search through all recorded trips and apply Algorithm 2 to find $K$ trip segments which coincide with $s_i$ and have conditions similar to the current conditions for this segment;
- $\tau \leftarrow \tau +$ average travel time along this segment over $K$ trip segments;

end
8 Evaluation

8.1 Methodology

To evaluate the design of the system, we took into account real-time performance of the system and its memory consumption. The DB has been populated with test data by using the script presented by Listing [in Appendix C]. The tests were done on Intel Core i5-2400 CPU 3.10 GHz, 8.00 GB RAM, Windows 7 Ultimate 64-bit.

To isolate the performance of the server-side code from the one of the client-side code, the server was run on a separate machine. Clients connected to the server from a web browser (Google Chrome) via HTTP protocol. To simulate multiple simultaneous users, every client was started in its own tab in the Chrome.

To simulate a vehicle travelling along a route and transmitting its GPS coordinates to the server, a client-side script written in JavaScript was used. It sent messages to the server which triggered route finding routine and scheduled a "crawler" function which would periodically transmit GPS coordinates along the route thus imitating a vehicle physically travelling along that route.

8.1.1 Real-time performance

The goal was to build a system capable of serving multiple users, providing the information in real-time within certain time-constraints. We measured both the data retrieval- and data processing performance, which includes:

- routines before the trip:
  - acquiring coordinates of the nodes which correspond to the source and destination provided by the user, which is done by fetching it from the DB;
  - finding a route by querying the pgRouting module in the DB;
  - preparing the route structure by caching source- and destination coordinates for each edge and the coordinates of the corresponding data stations;
  - caching the history for each edge belonging to the current route;

- routines during the trip, which we want to run in real-time:
  - estimating the travel time for the remaining part of the route;

- routines after the trip:
  - finishing the trip, i.e. calculating travel time for each edge and writing it to the DB.
8.1.2 Memory consumption

Because we deal with large amount of data, storing this data, no matter whether we do it permanently or temporary, should be done with respect to efficiency. The following has been measured with respect to the required memory space:

- storing the route structure containing the route itself and the data logged for the current trip;
- storing history for a single edge;
- storing history for all edges belonging to the route.

8.2 Results

All data obtained during the tests is presented in Appendix A.

Tables A1 and A2 present time- and memory resources consumed by the system for a single user for trips taken within city of Umeå. The route covered by this test is meant to depict the longest trip (ca 12 km) one can take within the city bounds.

Tables A3 and A4 compare time-wise performance of the system for a single user for trips taken between two regions.

Tables A10-A12 and tables A13-A18 show the behaviour of the system when multiple simultaneous users connect for a short route and a long route, respectively. To capture the pattern in the performance, the system is tested for different number of sensors and different history size. These tests proceeded as follows. The system was started. The first user (User 1) connected to the server and immediately requested a route. As soon the user had got his or her route discovered and has started moving along the route, the second user (User 2) connected. The same procedure was repeated for all users. Depending on the performance, number of users varied between 3 and 10 for different usage scenarios.

Figures 10 and 11 present the time required for recalculating the estimated time of arrival when multiple users are tracked by the system simultaneously. On the x-axis, the left-most measurements depict time values for a single travelling user. Gradually, more users join the system and start travelling along their routes. Eventually, the right-most values show response times for the travel time calculation routines for a single user when 10 users are tracked simultaneously.

For the longer trips, the measurements were produced only for that “realistic” scenario with respect to the corresponding start-up times.
8.3 Conclusions

Both the CPU-time- and the memory requirements of the system are linearly dependent on the distance covered by the route and the number of sensors for which we collect the trip data.

Time required to save trip data to the DB during the trip finalization phase is small and can be neglected.

8.3.1 Tracking short trips

The system behaves well for short trips within city of Umeå. The trip preparation phase takes acceptable amount of time and the during-the-trip routines perform very good allowing for multiple vehicles to be tracked at the same time. The majority of the CPU-time is consumed by route searching and resolving of the source and destination points. Time to cache additional edge info can be neglected. For moderately large history (100 trips per route), the system is expected to serve several dozens of users within seemingly constant time with start-up time less than 30 seconds.

The travel time estimations generated by the system continuously during the trip take an average of <1 sec for 1-10 users load, with peaks reaching 1.5 sec as number of users increases.

Every user consumes just <1MB of memory when history contains 100 trips and up to 4MB for reasonably large history (400 trips). Memory consumption is split evenly between storage used by the route structure and the history for every edge. One can conclude that memory-wise there are no immediate limitations on the user count.

8.3.2 Tracking longer trips

For trips of longer length (e.g. about 260 km) the performance is mainly affected by time consumed by caching, which in turn, as the history size increases, is dominated by history caching. The start-up time varies from 30 seconds to couple of minutes depending on the settings. For moderately large history size the system requires up to 5 minutes to initiate a trip even when hosting just a few users.

The on-track travel time estimations takes an average of <1 seconds and reaches peaks of 2.5 Hz.

Comparing to short trips, the travel time prediction for longer trips require much more memory. Generally, memory consumption grows linearly as trips become longer. Eventually, handling a single user can require from 20 to 60 MB of memory space depending on the history size. Still, given modern hardware, the memory should suffice for >100 users.

8.3.3 Performance adjustment

Halving the number of trips stored in history we can double the number of users tracked by the system. The main CPU time consumer is caching.

By halving the number of sensors we can serve 1.5 times more users.

From the previous conclusions, for the logging frequency to match the response time of the
travel time calculation routines, it cannot exceed 2.5 Hz. Otherwise, the system will not be able to process all the incoming messages from the clients in time.
9 Discussion

During the project, a set of tools and approaches were identified in order to meet the requirements. A working prototype was implemented and its performance was analysed for different usage scenarios.

Generally, the outcome of the project satisfies the initial goals. The system meets the requirements and with additional tuning it should be possible to use it in some settings as described previously. Still, the tests reveal some of the weaknesses of the solution, which does not allow to employ it for large-scale projects without major improvements. Caching which was believed to be the winning strategy didn't give the satisfactory results when applied to larger datasets, at least as by current implementation. One believes that better hardware may mitigate bad performance to a certain extent, since the testing environment was not best suited for the production (i.e. separate high-end machines for different components of the system), but the current results still suggest that some of the decisions used by the project need to be taken with a grain of salt.

Also, the choice to base travel time predictions solely on comparing current conditions with historical data introduces probably the main limitation of the current design: one can only make very short-term assumptions. Given current conditions along the route, we assume these conditions remain constant as we travel along the route. In reality, this is not the case. Because we do not account for the conditions to change in the future, the error of our estimations will increase for road segments which lie further away. Some factors as time of day or day of week can be estimated with better precision since we only need to account for vehicle’s speed along the route to find out what time the time sensor will return when the vehicle is at a certain route node. However, the system is not capable of predicting such factors as weather better than assuming it to remain constant. Fetching the latest sensor data at regular intervals during the trip and updating travel time accordingly will slightly mitigate the mentioned problem, but we are still vulnerable to this kind of errors.

Future work should also include extensive field testing of the system to find out actual weights for different factors used in travel time estimation.

Even though the memory requirements is not the bottleneck for the performance of the system, one could make memory usage more efficient by sharing pre-calculated routes between users travelling along the same road segments. The same could be applied to the history caching.

Since some time-consuming tasks use different resources, distributing server-side functionality between multiple machines could also increase performance. One could run a separate server to host the database containing the routing graph, and store trip history and cached data on another machine.

Another aspect worth further improvements is how we locate data stations to fetch data from. Depending on the factors, the corresponding data stations can be spread very sparsely,
each data station covering a large area. Every time the user needs data for a particular location, we have to query database to give us the nearest data station. More efficient approach would be to anchor the user to a particular data station by means of calculating a polynomial region covered by this station and checking if the position of the user falls within the bounds of this region. This technique is known as Voronoi tessellation.


Xiaoqiang Pan Liangyi Huang Qingqing Mu, Yonggang Wu and Xian Li. Short-term Load Forecasting Using Improved Similar Days Method. 


Jeff Shaner. Smartphones, Tablets and GPS Accuracy. 


Michael Stonebraker. SQL Databases v. NoSQL Databases. 


Ioannis Tsapakis, Tao Cheng, and Adel Bolbol. Impact of weather conditions on macroscopic urban travel times. 


Jan Sipke van der Veen, Bram van der Waaij, and Robert J. Meijer. Sensor Data Storage Performance: SQL or NoSQL, Physical or Virtual. 

Fei Wang and Jimeng Sun. Distance Metric Learning in Data Mining. 

## Appendix A: Test results

### Table A1 Time-wise performance for the route Umeå, Ersboda - Umeå, Umedalen. Length: 12 km. Number of road segments: 40. 3 sensors. Caching additional info for each edge: 0.35 sec.

<table>
<thead>
<tr>
<th>trips</th>
<th>finding a route, sec</th>
<th>travel time est, sec</th>
<th>history caching, sec</th>
<th>finishing trip, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.16 / 7.44</td>
<td>0.015</td>
<td>0.013</td>
<td>0 / 0.141</td>
</tr>
<tr>
<td>1</td>
<td>6.10 / 7.35</td>
<td>0.013</td>
<td>0.03</td>
<td>0 / 0.173</td>
</tr>
<tr>
<td>10</td>
<td>6.15 / 7.45</td>
<td>0.014</td>
<td>0.18</td>
<td>0 / 0.181</td>
</tr>
<tr>
<td>20</td>
<td>6.12 / 7.41</td>
<td>0.015</td>
<td>0.37</td>
<td>0 / 0.272</td>
</tr>
<tr>
<td>50</td>
<td>6.11 / 7.41</td>
<td>0.018</td>
<td>0.87</td>
<td>0 / 0.312</td>
</tr>
<tr>
<td>100</td>
<td>6.13 / 7.42</td>
<td>0.022</td>
<td>1.78</td>
<td>0 / 0.438</td>
</tr>
<tr>
<td>200</td>
<td>6.24 / 7.43</td>
<td>0.030</td>
<td>3.45</td>
<td>0 / 0.590</td>
</tr>
<tr>
<td>400</td>
<td>6.13 / 7.35</td>
<td>0.041</td>
<td>6.876</td>
<td>0 / 0.975</td>
</tr>
</tbody>
</table>

### Table A2 Memory consumption for the route Umeå, Ersboda - Umeå, Umedalen. 3 sensors. Storage for current trip conditions for one edge: 748 B, for all edges: 29 920 B.

<table>
<thead>
<tr>
<th>trips</th>
<th>route structure, B</th>
<th>history per edge, B</th>
<th>history, all edges, B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>38 434</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>42 774</td>
<td>230</td>
<td>9 200</td>
</tr>
<tr>
<td>10</td>
<td>81 710</td>
<td>1 202</td>
<td>48 080</td>
</tr>
<tr>
<td>20</td>
<td>124 910</td>
<td>2 282</td>
<td>91 280</td>
</tr>
<tr>
<td>50</td>
<td>254 510</td>
<td>5 522</td>
<td>220 880</td>
</tr>
<tr>
<td>100</td>
<td>470 462</td>
<td>10 922</td>
<td>436 880</td>
</tr>
<tr>
<td>200</td>
<td>902 510</td>
<td>21 722</td>
<td>868 880</td>
</tr>
<tr>
<td>400</td>
<td>1 766 494</td>
<td>43 322</td>
<td>1 732 880</td>
</tr>
</tbody>
</table>
Table A3 Time-wise performance for the route Umeå - Sundsvall. Length: 258 km. Number of road segments: 834. 3 sensors. Caching additional info for each edge: 7.4 sec.

<table>
<thead>
<tr>
<th>trips</th>
<th>finding a route, sec</th>
<th>travel time est, sec</th>
<th>history caching, sec</th>
<th>finishing trip, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.12 / 7.46</td>
<td>0.067</td>
<td>0.154</td>
<td>0.002 / 2.531</td>
</tr>
<tr>
<td>1</td>
<td>6.13 / 7.42</td>
<td>0.074</td>
<td>0.538</td>
<td>0.001 / 2.802</td>
</tr>
<tr>
<td>10</td>
<td>6.13 / 7.49</td>
<td>0.087</td>
<td>3.665</td>
<td>0.001 / 3.326</td>
</tr>
<tr>
<td>20</td>
<td>6.12 / 7.46</td>
<td>0.097</td>
<td>7.214</td>
<td>0.001 / 4.079</td>
</tr>
<tr>
<td>50</td>
<td>6.12 / 7.39</td>
<td>0.126</td>
<td>17.462</td>
<td>0.002 / 6.129</td>
</tr>
<tr>
<td>100</td>
<td>6.14 / 7.45</td>
<td>0.172</td>
<td>35.297</td>
<td>0.001 / 3.461</td>
</tr>
<tr>
<td>200</td>
<td>6.09 / 7.35</td>
<td>0.283</td>
<td>70.199</td>
<td>0.001 / 2.611</td>
</tr>
</tbody>
</table>

Table A4 Memory consumption for the route Umeå - Sundsvall. 3 sensors. Storage for current trip conditions for one edge: 748 B, for all edges: 623 832 B. Values for 200 trips are estimations.

<table>
<thead>
<tr>
<th>trips</th>
<th>route structure, B</th>
<th>history per edge, B</th>
<th>history, all edges, B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>785 856</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>876 020</td>
<td>230</td>
<td>191 820</td>
</tr>
<tr>
<td>10</td>
<td>1 686 684</td>
<td>1 202</td>
<td>1 002 468</td>
</tr>
<tr>
<td>20</td>
<td>2 587 412</td>
<td>2 282</td>
<td>1 903 188</td>
</tr>
<tr>
<td>50</td>
<td>5 289 572</td>
<td>5 522</td>
<td>4 605 348</td>
</tr>
<tr>
<td>100</td>
<td>9 793 252</td>
<td>10 938</td>
<td>9 122 292</td>
</tr>
<tr>
<td>200</td>
<td>≈18 800 585</td>
<td>≈21 812</td>
<td>≈18 191 530</td>
</tr>
</tbody>
</table>

Table A5 Time-wise performance for the route Umeå, Ersboda - Umeå, Umedalen. 10 sensors.

<table>
<thead>
<tr>
<th>trips</th>
<th>finding a route, sec</th>
<th>travel time est, sec</th>
<th>history caching, sec</th>
<th>finishing trip, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>same as above</td>
<td>0.05</td>
<td>0.937</td>
<td>0.001 / 0.308</td>
</tr>
<tr>
<td>100</td>
<td>same as above</td>
<td>0.073</td>
<td>1.855</td>
<td>0.001 / 0.311</td>
</tr>
<tr>
<td>200</td>
<td>same as above</td>
<td>0.117</td>
<td>3.748</td>
<td>0.001 / 0.275</td>
</tr>
</tbody>
</table>

Table A6 Memory consumption for the route Umeå, Ersboda - Umeå, Umedalen. 10 sensors. Storage for current trip conditions for one edge: 1 396 B, for all edges: 55 840 B.

<table>
<thead>
<tr>
<th>trips</th>
<th>route structure, B</th>
<th>history per edge, B</th>
<th>history, all edges, B</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>619 408</td>
<td>14 832</td>
<td>593 280</td>
</tr>
<tr>
<td>100</td>
<td>1 199 408</td>
<td>29 332</td>
<td>1 173 280</td>
</tr>
<tr>
<td>200</td>
<td>2 359 408</td>
<td>58 332</td>
<td>2 333 280</td>
</tr>
</tbody>
</table>

Table A7 Time-wise performance for the route Umeå, Ersboda - Umeå, Umedalen. 20 sensors.

<table>
<thead>
<tr>
<th>trips</th>
<th>finding a route, sec</th>
<th>travel time est, sec</th>
<th>history caching, sec</th>
<th>finishing trip, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>same as above</td>
<td>0.106</td>
<td>1.054</td>
<td>0.001 / 0.607</td>
</tr>
<tr>
<td>100</td>
<td>same as above</td>
<td>0.188</td>
<td>2.069</td>
<td>0.001 / 0.637</td>
</tr>
<tr>
<td>200</td>
<td>same as above</td>
<td>0.310</td>
<td>3.999</td>
<td>0.001 / 0.642</td>
</tr>
</tbody>
</table>

A2
### Table A8
Memory consumption for the route Umeå, Er-skoda - Umeå, Umedalen. 20 sensors. Storage for current trip conditions for one edge: 2 356 B, for all edges: 94 240 B.

<table>
<thead>
<tr>
<th>trips</th>
<th>route structure, B</th>
<th>history per edge, B</th>
<th>history, all edges, B</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>1 140 716</td>
<td>28 132</td>
<td>1 125 280</td>
</tr>
<tr>
<td>100</td>
<td>2 240 724</td>
<td>55 640</td>
<td>2 225 600</td>
</tr>
<tr>
<td>200</td>
<td>4 440 724</td>
<td>110 640</td>
<td>4 425 600</td>
</tr>
</tbody>
</table>

### Table A9
Time-wise performance for the route Umeå, Er-skoda - Umeå, Umedalen. 50 sensors.

<table>
<thead>
<tr>
<th>trips</th>
<th>finding a route, sec</th>
<th>travel time est, sec</th>
<th>history caching, sec</th>
<th>finishing trip, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>same as above</td>
<td>0.475</td>
<td>1.38</td>
<td>0.001 / 1.341</td>
</tr>
<tr>
<td>100</td>
<td>same as above</td>
<td>1.130</td>
<td>2.734</td>
<td>0.001 / 1.209</td>
</tr>
<tr>
<td>200</td>
<td>same as above</td>
<td>1.902</td>
<td>5.196</td>
<td>0.001 / 18.216</td>
</tr>
</tbody>
</table>
### Table A10

Time-wise performance for the route Umeå, Ersboda - Umeå, Umedalen. Pre-trip routines. Multiple simultaneous users. 10 sensors, 100 trips.

<table>
<thead>
<tr>
<th>user ID</th>
<th>resolve src and dest, sec</th>
<th>get routing table, sec</th>
<th>cache additional info, sec</th>
<th>cache history, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>6.19</td>
<td>7.36</td>
<td>0.35</td>
<td>1.96</td>
</tr>
<tr>
<td>User 2</td>
<td>6.20</td>
<td>7.56</td>
<td>0.32</td>
<td>1.96</td>
</tr>
<tr>
<td>User 3</td>
<td>6.19</td>
<td>7.55</td>
<td>0.31</td>
<td>1.99</td>
</tr>
<tr>
<td>User 4</td>
<td>6.20</td>
<td>7.68</td>
<td>0.32</td>
<td>2.03</td>
</tr>
<tr>
<td>User 5</td>
<td>6.13</td>
<td>7.54</td>
<td>0.32</td>
<td>2.12</td>
</tr>
<tr>
<td>User 6</td>
<td>6.22</td>
<td>7.57</td>
<td>0.32</td>
<td>2.12</td>
</tr>
<tr>
<td>User 7</td>
<td>6.24</td>
<td>7.67</td>
<td>0.32</td>
<td>2.18</td>
</tr>
<tr>
<td>User 8</td>
<td>6.23</td>
<td>7.57</td>
<td>0.34</td>
<td>2.14</td>
</tr>
<tr>
<td>User 9</td>
<td>6.24</td>
<td>7.58</td>
<td>0.36</td>
<td>2.12</td>
</tr>
<tr>
<td>User 10</td>
<td>6.19</td>
<td>7.62</td>
<td>0.33</td>
<td>2.21</td>
</tr>
</tbody>
</table>

### Table A11


<table>
<thead>
<tr>
<th>user ID</th>
<th>resolve src and dest, sec</th>
<th>get routing table, sec</th>
<th>cache additional info, sec</th>
<th>cache history, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>6.16</td>
<td>7.39</td>
<td>0.35</td>
<td>3.87</td>
</tr>
<tr>
<td>User 2</td>
<td>6.14</td>
<td>7.40</td>
<td>0.32</td>
<td>4.17</td>
</tr>
<tr>
<td>User 3</td>
<td>6.19</td>
<td>7.42</td>
<td>0.42</td>
<td>3.94</td>
</tr>
<tr>
<td>User 4</td>
<td>6.19</td>
<td>7.45</td>
<td>0.39</td>
<td>4.13</td>
</tr>
<tr>
<td>User 5</td>
<td>6.20</td>
<td>7.59</td>
<td>0.48</td>
<td>4.17</td>
</tr>
<tr>
<td>User 6</td>
<td>6.23</td>
<td>7.57</td>
<td>0.72</td>
<td>4.24</td>
</tr>
<tr>
<td>User 7</td>
<td>6.26</td>
<td>7.66</td>
<td>0.32</td>
<td>4.91</td>
</tr>
<tr>
<td>User 8</td>
<td>6.42</td>
<td>7.77</td>
<td>0.32</td>
<td>5.02</td>
</tr>
<tr>
<td>User 9</td>
<td>6.23</td>
<td>7.78</td>
<td>0.31</td>
<td>5.22</td>
</tr>
<tr>
<td>User 10</td>
<td>6.46</td>
<td>7.71</td>
<td>0.42</td>
<td>5.20</td>
</tr>
</tbody>
</table>
Figure 1: Time-wise performance for the route Umeå, Ersboda - Umeå, Umedalen. Pre-trip routines. Multiple simultaneous users. 10 sensors, 100 trips.

Figure 2: Time-wise performance for the route Umeå, Ersboda - Umeå, Umedalen. Pre-trip routines. Multiple simultaneous users. 10 sensors, 200 trips.
Figure 3: Time-wise performance for the route Umeå, Ersboda - Umeå, Umedalen. Pre-trip routines. Multiple simultaneous users. 10 sensors, 400 trips.

Figure 4: Time-wise performance for the route Umeå, Ersboda - Sundsvall. Pre-trip routines. Multiple simultaneous users. 10 sensors, 20 trips.
Figure 5: Time-wise performance for the route Umeå, Ersboda - Sundsvall. Pre-trip routines. Multiple simultaneous users. 10 sensors, 50 trips.

Figure 6: Time-wise performance for the route Umeå, Ersboda - Sundsvall. Pre-trip routines. Multiple simultaneous users. 10 sensors, 100 trips.
Figure 7: Time-wise performance for the route Umeå, Ersboda - Sundsvall. Pre-trip routines. Multiple simultaneous users. 5 sensors, 20 trips.

Figure 8: Time-wise performance for the route Umeå, Ersboda - Sundsvall. Pre-trip routines. Multiple simultaneous users. 5 sensors, 50 trips.
### Table A12
Time-wise performance for the route Umeå, Ersboda - Umeå, Umedalen. Pre-trip routines. Multiple simultaneous users. 10 sensors, 400 trips.

<table>
<thead>
<tr>
<th>user ID</th>
<th>resolve src and dest, sec</th>
<th>get routing table, sec</th>
<th>cache additional info, sec</th>
<th>cache history, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>6.16</td>
<td>7.33</td>
<td>0.35</td>
<td>7.64</td>
</tr>
<tr>
<td>User 2</td>
<td>6.15</td>
<td>7.39</td>
<td>0.32</td>
<td>8.14</td>
</tr>
<tr>
<td>User 3</td>
<td>6.12</td>
<td>7.49</td>
<td>0.32</td>
<td>8.65</td>
</tr>
<tr>
<td>User 4</td>
<td>6.10</td>
<td>7.51</td>
<td>0.31</td>
<td>9.36</td>
</tr>
<tr>
<td>User 5</td>
<td>6.15</td>
<td>7.68</td>
<td>0.31</td>
<td>10.07</td>
</tr>
<tr>
<td>User 6</td>
<td>6.38</td>
<td>7.91</td>
<td>0.72</td>
<td>10.78</td>
</tr>
<tr>
<td>User 7</td>
<td>6.23</td>
<td>7.83</td>
<td>0.31</td>
<td>10.97</td>
</tr>
<tr>
<td>User 8</td>
<td>6.17</td>
<td>7.75</td>
<td>0.32</td>
<td>11.11</td>
</tr>
<tr>
<td>User 9</td>
<td>6.32</td>
<td>7.85</td>
<td>0.51</td>
<td>11.63</td>
</tr>
<tr>
<td>User 10</td>
<td>6.21</td>
<td>8.17</td>
<td>0.75</td>
<td>12.89</td>
</tr>
</tbody>
</table>

### Table A13

<table>
<thead>
<tr>
<th>user ID</th>
<th>resolve src and dest, sec</th>
<th>get routing table, sec</th>
<th>cache additional info, sec</th>
<th>cache history, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>6.2</td>
<td>7.4</td>
<td>6.5</td>
<td>8.3</td>
</tr>
<tr>
<td>User 2</td>
<td>6.2</td>
<td>7.5</td>
<td>7.1</td>
<td>9.1</td>
</tr>
<tr>
<td>User 3</td>
<td>6.2</td>
<td>7.7</td>
<td>7.9</td>
<td>10.2</td>
</tr>
<tr>
<td>User 4</td>
<td>6.2</td>
<td>7.8</td>
<td>8.6</td>
<td>11.3</td>
</tr>
<tr>
<td>User 5</td>
<td>6.4</td>
<td>8.3</td>
<td>9.9</td>
<td>12.4</td>
</tr>
<tr>
<td>User 6</td>
<td>6.5</td>
<td>8.3</td>
<td>11.3</td>
<td>14.6</td>
</tr>
<tr>
<td>User 7</td>
<td>6.8</td>
<td>8.6</td>
<td>13.0</td>
<td>17.0</td>
</tr>
<tr>
<td>User 8</td>
<td>7.0</td>
<td>8.8</td>
<td>16.5</td>
<td>20.6</td>
</tr>
<tr>
<td>User 9</td>
<td>7.4</td>
<td>9.2</td>
<td>21.8</td>
<td>25.6</td>
</tr>
<tr>
<td>User 10</td>
<td>7.4</td>
<td>9.7</td>
<td>31.8</td>
<td>35.7</td>
</tr>
</tbody>
</table>

### Table A14
Time-wise performance for the route Umeå, Ersboda - Sundsvall. Pre-trip routines. Multiple simultaneous users. 10 sensors, 50 trips.

<table>
<thead>
<tr>
<th>user ID</th>
<th>resolve src and dest, sec</th>
<th>get routing table, sec</th>
<th>cache additional info, sec</th>
<th>cache history, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>7.9</td>
<td>8.1</td>
<td>6.5</td>
<td>20.4</td>
</tr>
<tr>
<td>User 2</td>
<td>6.3</td>
<td>7.8</td>
<td>8.0</td>
<td>25.9</td>
</tr>
<tr>
<td>User 3</td>
<td>6.3</td>
<td>8.1</td>
<td>10.7</td>
<td>37.9</td>
</tr>
<tr>
<td>User 4</td>
<td>7.1</td>
<td>8.8</td>
<td>20.0</td>
<td>59.9</td>
</tr>
<tr>
<td>User 5</td>
<td>7.1</td>
<td>9.4</td>
<td>38.5</td>
<td>114.7</td>
</tr>
</tbody>
</table>
### Table A15
Time-wise performance for the route Umeå, Erskboda - Sundsvall. Pre-trip routines. Multiple simultaneous users. 10 sensors, 100 trips.

<table>
<thead>
<tr>
<th>user ID</th>
<th>resolve src and dest, sec</th>
<th>get routing table, sec</th>
<th>cache additional info, sec</th>
<th>cache history, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>6.2</td>
<td>7.3</td>
<td>6.5</td>
<td>41.9</td>
</tr>
<tr>
<td>User 2</td>
<td>6.3</td>
<td>8.0</td>
<td>11.7</td>
<td>74.1</td>
</tr>
<tr>
<td>User 3</td>
<td>7.9</td>
<td>9.1</td>
<td>42.4</td>
<td>199.4</td>
</tr>
</tbody>
</table>

### Table A16
Time-wise performance for the route Umeå, Erskboda - Sundsvall. Pre-trip routines. Multiple simultaneous users. 5 sensors, 20 trips.

<table>
<thead>
<tr>
<th>user ID</th>
<th>resolve src and dest, sec</th>
<th>get routing table, sec</th>
<th>cache additional info, sec</th>
<th>cache history, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>6.2</td>
<td>7.4</td>
<td>6.6</td>
<td>8.1</td>
</tr>
<tr>
<td>User 2</td>
<td>6.2</td>
<td>7.5</td>
<td>6.9</td>
<td>8.6</td>
</tr>
<tr>
<td>User 3</td>
<td>6.4</td>
<td>7.8</td>
<td>7.4</td>
<td>8.9</td>
</tr>
<tr>
<td>User 4</td>
<td>6.3</td>
<td>8.0</td>
<td>8.0</td>
<td>9.9</td>
</tr>
<tr>
<td>User 5</td>
<td>6.5</td>
<td>8.0</td>
<td>8.7</td>
<td>10.7</td>
</tr>
<tr>
<td>User 6</td>
<td>6.4</td>
<td>8.3</td>
<td>9.9</td>
<td>11.4</td>
</tr>
<tr>
<td>User 7</td>
<td>6.6</td>
<td>8.4</td>
<td>10.8</td>
<td>12.7</td>
</tr>
<tr>
<td>User 8</td>
<td>6.6</td>
<td>8.7</td>
<td>11.9</td>
<td>14.5</td>
</tr>
<tr>
<td>User 9</td>
<td>6.9</td>
<td>8.8</td>
<td>13.2</td>
<td>15.8</td>
</tr>
<tr>
<td>User 10</td>
<td>7.6</td>
<td>9.4</td>
<td>16.1</td>
<td>18.3</td>
</tr>
</tbody>
</table>

### Table A17
Time-wise performance for the route Umeå, Erskboda - Sundsvall. Pre-trip routines. Multiple simultaneous users. 5 sensors, 50 trips.

<table>
<thead>
<tr>
<th>user ID</th>
<th>resolve src and dest, sec</th>
<th>get routing table, sec</th>
<th>cache additional info, sec</th>
<th>cache history, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>6.2</td>
<td>7.4</td>
<td>6.5</td>
<td>18.9</td>
</tr>
<tr>
<td>User 2</td>
<td>6.2</td>
<td>7.8</td>
<td>7.5</td>
<td>23.3</td>
</tr>
<tr>
<td>User 3</td>
<td>6.4</td>
<td>7.8</td>
<td>8.6</td>
<td>27.7</td>
</tr>
<tr>
<td>User 4</td>
<td>7.0</td>
<td>8.1</td>
<td>11.7</td>
<td>36.6</td>
</tr>
<tr>
<td>User 5</td>
<td>7.2</td>
<td>8.9</td>
<td>18.2</td>
<td>50.0</td>
</tr>
</tbody>
</table>

### Table A18
Time-wise performance for the route Umeå, Erskboda - Sundsvall. Pre-trip routines. Multiple simultaneous users. 5 sensors, 100 trips.

<table>
<thead>
<tr>
<th>user ID</th>
<th>resolve src and dest, sec</th>
<th>get routing table, sec</th>
<th>cache additional info, sec</th>
<th>cache history, sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>6.2</td>
<td>7.3</td>
<td>6.5</td>
<td>38.5</td>
</tr>
<tr>
<td>User 2</td>
<td>7.0</td>
<td>8.1</td>
<td>8.6</td>
<td>57.0</td>
</tr>
<tr>
<td>User 3</td>
<td>6.8</td>
<td>9.1</td>
<td>18.5</td>
<td>116.9</td>
</tr>
</tbody>
</table>
Figure 9: Time-wise performance for the route Umeå, Ersboda - Sundsvall. Pre-trip routines. Multiple simultaneous users. 5 sensors, 100 trips.
Figure 10: Response time for the travel time calculation routine when tracking 10 users simultaneously. The number of users gradually increases from 1 to 10. Umeå, Ersboda - Umeå, Umedalen. 10 sensors, 400 trips.

Figure 11: Response time for the travel time calculation routine when tracking 10 users simultaneously. The number of users gradually increases from 1 to 10. Umeå, Ersboda - Sundsvall. 5 sensors, 20 trips.
Appendix B: GUI of the system

Figure 1: The system has found a route. The only buttons the user will be able to see are Set destination and Start trip, the rest of the buttons are only used for the purpose of demonstration and administration. When the user has logged in to the application, his or her geographical location will be automatically discovered by the system. Then the user is expected to choose a destination by tapping the corresponding button followed by tapping on the map. The system will discover the route, if any, and provide the user with travel time estimation. The curve depicting the route will contain blue segments for those parts of the route for which the system had data to produce the travel time estimation from. If a segment is coloured grey, the system did not have any data collected for it. The ”+” symbol next to the travel time estimation indicates the presence of these “grey” segments with no collected data. When the preparations are complete and the user has started the trip, the user is expected to notify the system to start tracking the trip by tapping the Start trip button. Map data ©2015 Google.
Figure 2: The user is moving along a route. The vehicle’s current position is marked on the map. As the vehicle proceeds along its route, the remaining travel time decreases and the curve indicating the route is updated accordingly. Map data ©2015 Google.

Figure 3: The user is about to finish the route soon. The system has the complete data to produce the travel time estimation for the remaining part of the route. Thus, there is no “+” symbol next to the travel time estimation. Map data ©2015 Google.
Figure 4: Serving multiple users. Map data ©2015 Google.

Figure 5: The debug mode. One can see the actual positions of the sensor data stations which provide the data for the selected route. Map data ©2015 Google.
Figure 6: A vehicle is following a route in the debug mode. The black points indicate the locations where the logging of the sensor data succeeded. Map data ©2015 Google.
Appendix C: Populating the DB with test data

Listing 1: PL/pgSQL script for populating the DB with test data

```sql
/*
* Creates the given number of copies of the trip with the highest tripid and appends them to the sensor_data table, and copies the corresponding entries in the times table.
* @param n number of copies to create
* @return tripid of the last copy
*/
CREATE OR REPLACE FUNCTION duplicate_sensor_data(
  n integer)
REturns integer AS $$
DECLARE
  max_tripid integer;
BEGIN
  FOR i IN 1..n LOOP
    SELECT INTO max_tripid MAX(tripid)
    FROM sensor_data;
    INSERT INTO sensor_data(tripid, edgeid, sensorid, value)
    SELECT max_tripid+1, edgeid, sensorid, value
    FROM sensor_data
    WHERE tripid=max_tripid;
    INSERT INTO times(tripid, edgeid, ttime, reversed)
    SELECT max_tripid+1, edgeid, ttime, reversed
    FROM times
    WHERE tripid=max_tripid;
  END LOOP;
  RETURN max_tripid+1;
END;
$$ LANGUAGE plpgsql;

select duplicate_sensor_data(100)
```

C1