Real-Time Route Planning with Stream Processing Systems:
A Case Study for the City of Luzern

by
Aslı Özal

Supervised by
Prof. Dr. Nesime Tatbul
Dr. Anand Ranganathan

December 2010 – June 2011
Abstract

Today, many cities in the world are facing serious transportation challenges. With continuous increase of car usage in urban areas, traffic congestion has become an important problem in daily life. Intelligent Transportation Systems (ITS) aim to ease environmental, social, and economic implications of traffic congestion through the application of modern information technology and communications.

In this thesis, we aim to handle the problem of route planning in the presence of traffic congestion. We present a system architecture that applies stream data management technology on traffic information data for real-time route planning. We explain the techniques to gather traffic data from various data sources, to calculate traffic and speed estimations, and to execute continuous route queries with real-time traffic data.

We make an implementation of our system design as a case study. Lucerne city is chosen as the execution area, where route calculations and road delay estimations are performed. We benefit from the traffic data collected by the induction loop detectors in Lucerne. Using the collected induction loop detector data, average speed and delays of roads are calculated continuously. The system monitors these delays and calculates the fastest route for queries.

Experiments show that our system performs well in small cities like Lucerne, and with a few improvements it has the potential to serve thousands of users in larger cities.
I would like to express my sincere thanks to Prof. Nesime Tatbul for her constant support and guidance during the course of this thesis. I would also like to thank Dr. Anand Ranganathan from IBM Research for the valuable discussions and his useful advices.

I am deeply grateful to Romeo Kienzler for his support throughout my thesis. I thank Marc Angwer and Thomas Karrer from VBL for providing me the induction loop data. I also thank Barış Güç for the implementation of route calculations.

Lastly, I am very thankful to my family and my friends for their support.
# Contents

Abstract i

Acknowledgements ii

List of Figures v

Abbreviations vi

1 Introduction 1
   1.1 Background and Motivation ........................................ 1
   1.2 Problem Statement .................................................. 2
   1.3 Contribution ....................................................... 2
   1.4 Thesis Organization ................................................. 3

2 Related Work 4
   2.1 Overview ............................................................ 4
   2.2 Stream Traffic Data Processing Systems ......................... 4
   2.3 Traffic Time Estimation and Average Speed Calculation Models ........ 5
   2.4 Shortest Path Calculation ........................................... 7

3 Background 8
   3.1 Stream Processing Engine: IBM InfoSphere Streams ............. 8
       3.1.1 IBM InfoSphere Streams .................................... 8
       3.1.2 Spade Language .............................................. 9
   3.2 Induction Loop System ............................................. 11
   3.3 Road Map: OpenStreetMap ....................................... 12
   3.4 Travel Time Estimation: Conventional Model ................... 12

4 Data Model 15
   4.1 OpenStreetMaps Lucerne Map Data ................................ 15
   4.2 ILD Data for Lucerne City ....................................... 17
   4.3 Induction Loop Detector Data for the City of Lucerne ........ 18
   4.4 Average Speed Updates for Links ................................ 19
   4.5 Path Queries ....................................................... 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 System Implementation</td>
<td>21</td>
</tr>
<tr>
<td>5.1 General System Architecture</td>
<td>21</td>
</tr>
<tr>
<td>5.2 Pre-Processing</td>
<td>22</td>
</tr>
<tr>
<td>5.3 Average Speed Calculation</td>
<td>24</td>
</tr>
<tr>
<td>5.4 Shortest Path Calculation</td>
<td>29</td>
</tr>
<tr>
<td>6 Experiments</td>
<td>31</td>
</tr>
<tr>
<td>6.1 Experimental Setup</td>
<td>31</td>
</tr>
<tr>
<td>6.1.1 Data Set and Query Specification</td>
<td>31</td>
</tr>
<tr>
<td>6.1.2 Metrics</td>
<td>31</td>
</tr>
<tr>
<td>6.2 Experiment Results</td>
<td>32</td>
</tr>
<tr>
<td>6.3 Discussion</td>
<td>33</td>
</tr>
<tr>
<td>7 Conclusion and Future Work</td>
<td>35</td>
</tr>
<tr>
<td>7.1 Conclusion</td>
<td>35</td>
</tr>
<tr>
<td>7.2 Future Work</td>
<td>36</td>
</tr>
<tr>
<td>A Appendix A</td>
<td>37</td>
</tr>
<tr>
<td>A.1 Pre-Processing</td>
<td>37</td>
</tr>
<tr>
<td>A.2 Average Speed Calculation</td>
<td>38</td>
</tr>
<tr>
<td>A.3 A* Algorithm</td>
<td>39</td>
</tr>
<tr>
<td>Bibliography</td>
<td>41</td>
</tr>
</tbody>
</table>
# List of Figures

3.1 Induction loop system .................................................. 11  
3.2 G-factors for induction loop detector .............................. 13  
4.1 OpenStreetMaps map extraction for the city of Lucerne .......... 16  
4.2 Lucerne city induction loop detectors’ locations ................. 19  
5.1 System architecture ........................................................ 21  
5.2 Pre-Processing component .............................................. 22  
5.3 Comparison of errors in calculation .................................. 23  
5.4 Join of detector-road mapping and traffic data .................... 25  
5.5 Structure of average speed calculation component ............... 25  
5.6 Sliding window structure ............................................... 26  
5.7 Occupancy and count calculation in the system ................... 27  
6.1 Change of response time with different ILD readings rate ........ 32  
6.2 Change of response time with increasing query rate ............ 33
Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITS</td>
<td>Intelligent Transportation System</td>
</tr>
<tr>
<td>TTE</td>
<td>Traffic Time Estimation</td>
</tr>
<tr>
<td>VBL</td>
<td>Verkehrsbetriebe Luzern (Lucerne Department of Transportation)</td>
</tr>
<tr>
<td>ILD</td>
<td>Induction Loop Detector</td>
</tr>
<tr>
<td>TMS</td>
<td>Time Mean Speed</td>
</tr>
<tr>
<td>SMS</td>
<td>Space Mean Speed</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

With the continuous increase of car usage in urban areas and increasing traffic congestion rate, intelligent transportation management becomes more helpful in daily life. An Intelligent Transportation System application, ‘Traffic-aware Route Planning’, is the focus of this thesis. We present a modular system design to handle route queries with up-to-date traffic information and with low latency.

Section 1.1 gives a brief overview of the thesis motivation. Section 1.2 defines the problem statement, and Section 1.3 presents the thesis contribution. Section 1.4 finalizes this chapter with an outline of subsequent chapters.

1.1 Background and Motivation

Traffic congestion has been increasing worldwide as a result of increased motorization, urbanization, population growth, and changes in population density [1]. Delays and economic costs caused by traffic congestion are a source of significant concern. Congestion reduces efficiency of transportation infrastructure and increases travel time, air pollution, and fuel consumption. Interest in Intelligent Transportation Systems (ITS) arises from the problems caused by traffic congestion. ITS have the potential to improve traffic conditions, to reduce travel delays and to manage the surveillance of the roadways easily [2].

Much of the data available for use in intelligent transportation management is in the form of data streams, such as induction loop detector data, Automatic Vehicle Location (AVL) systems on buses, and live traffic signal data. Inductive loop detector (ILD) is the most widely deployed type of traffic sensor that provides data for travel time estimation. Further, many cities, including Zurich, Lucerne, are storing these data streams and
creating large transportation data archives. We can benefit from the existing traffic data to fulfill the fast transportation need of people in urban areas. Travelers can plan their journeys more efficiently with the current information about traffic conditions.

1.2 Problem Statement

This thesis focuses on applying stream data management technology to route planning applications. The main goal of the thesis is to develop a traffic-aware route planning system architecture. The system will calculate the fastest route between departure and destination points, depending on current situation of traffic. The route should be correct and the latency for queries should be acceptable for a real-time application.

The designed system will then be tested in a case study for the city of Lucerne in Switzerland. Traffic data used in the case study will be the real induction loop detector data collected in Lucerne city. Using the induction loop detector data, average speed and delay for roads will be calculated continuously. System will monitor delays and calculate the fastest route at the instant that route query is executed. Execution time and scalability experiments will be performed to evaluate system performance.

1.3 Contribution

Main contribution of this thesis is the design of a system architecture for traffic-aware route planning by combining stream-processing technologies and traffic information data. We benefit from the performance and execution capabilities of stream processing engines to develop an intelligent transportation system. We present a methodology and explain the necessary architecture to build a system that can:

- process traffic data stream to get useful information out of it.
- benefit from spatial information (city map, bus stops, etc.)
- calculate real-time traffic congestion information.
- perform fastest route calculations for continuous user queries.

We propose a modular architecture with several components. Components have unique tasks and each component can be changed independently from each other. This makes our system to be able to work with various data sources and to serve different functions if needed.
We present our system design with a case-study for the city of Lucerne. We use IBM InfoSphere System as a stream processing engine to benefit from its continuous query processing capabilities. For traffic information, we benefit from induction loop detector readings collected in Lucerne city. We build experiments for our case-study implementation to evaluate performance of our system design.

We evaluate the performance of our system in two aspects. First, correctness of route calculations are experimented. Second, scalability of the system is tested with different data rates and number of queries. We observe the change in average response time of queries as we increase rate of traffic data fed into system. Possible performance improvements are stated as future work.

1.4 Thesis Organization

The remainder of the thesis is organized as follows:

- **Chapter 2** presents the related work in stream processing and route planning domain.
- **Chapter 3** gives background information for the technologies and the software programs used in the system. Technical details are explained.
- **Chapter 4** describes the data flow and the data model of the system.
- **Chapter 5** explains the system architecture and the implementation details.
- **Chapter 6** presents experiment setups, results and the analysis of the results.
- **Chapter 7** covers the conclusion and future work ideas.
Chapter 2

Related Work

2.1 Overview

Due to the general description of the system explained in Section 1.3, this thesis covers three different research topics which are:

- Stream Traffic Data Processing Systems
- Travel Time Estimation and Average Speed Calculation Models
- Shortest Path Calculation Models

Some of the related works in each area are explained in the following sections.

2.2 Stream Traffic Data Processing Systems

With the rising need to process huge amount of dynamic data, stream data processing systems have been developed for the last few decades. Streaming applications can be used to process large volumes of highly dynamic, time-sensitive, continuous streams of data. With these applications, effects of the changes in the system can be directly reflected to the result.

Stream data processing systems can be used in different areas like stock markets, weather monitoring, and recently in intelligent transportation systems. Improvements in the location tracking technologies like GPS systems and the growing need for traffic performance monitoring in cities have increased the interest in intelligent transportation services. We
will discuss three of the works in this area, GeoInsight, InfoSphere Streams, and Latte system.

GeoInsight [3] is a framework developed for geo-streaming applications. Similar to our work, GeoInsight focuses on processing and analyzing stream data with geographic and spatial information. GeoInsight employs Microsoft StreamInsight as its base for complex event processing, and extends it in two directions. First, online processing support is integrated. Second, a module for historical (archived) stream data querying is implemented. Different than our work, they perform analysis of historical data together with the real-time data to refine the answer of real-time queries and predict the answer in the near future.

Work in [4] introduces a prototype system that generates dynamic transportation information for the city of Stockholm. Like our system, this work uses IBM InfoSphere Streams as its stream processing platform. The introduced system consists of a set of stream processing applications that consecutively process real-time GPS data, generate different kinds of real-time traffic statistics, and perform customized analysis in response to user queries. Different than the ILD data that we use to get traffic information, they benefit from GPS data which gives speed information of vehicles. This work tackles with scalability and complexity issues of large volumes of data processing via pipelining and parallelization with a distributed runtime infrastructure.

We have stated the goal of our project as applying data stream management technology to Intelligent Transportation System (ITS) applications. The Latte project [5], is developed with the same goal by using the NiagaraST stream processing system [6] and the PORTAL transportation data archive. Latte focuses on travel-time estimation queries that combine live data streams with large data archives. It introduces the concept of comparing current data to ‘similar’ historical data for reliable travel time estimation. Contrary to our real-time travel-time calculations, they find similarities between past data and current data to estimate travel-time.

### 2.3 Traffic Time Estimation and Average Speed Calculation Models

Computing real-time transportation metrics is an important piece of intelligent transportation management. Traffic Time Estimation models (TTE) have been studied to measure transportation performance and to plan improvement of areas. After decades of development, TTE are used recently in the area of road network performance, traveler information systems and dynamic route guidance [7].
Traffic time estimation (TTE) models vary according to the input data used, such as inductive loop detector (ILD), probe vehicle technologies, license plate matching, and GPS. ILD is the most widely deployed type of traffic sensor that provides data for TTE models. ILDs provide a number of point-based measurements of traffic variables such as spot speed, flow and occupancy.

Different than technologies like GPS and video surveillance, ILD data does not provide speed information of vehicles. In order to calculate average speed values for vehicles in the traffic, TTE models are investigated. To summarize all existing TTE models, we can refer to research of Krishnan et.al. [8]. They divide traffic speed estimation models using ILD data into three main groups:

- **Conventional Models:** Average vehicle length is assumed constant, and spot speed is calculated from ILD data to estimate travel time [7],[9].

- **Learning-based Models:** Linear regression, artificial neural networks, k-nearest neighbor models are used to relate ILD data with travel time. Reference travel time data and ILD output are collected for a calibration period. The calibrated model is then used to estimate travel time using ILD output for other time periods. The performance of these models depend on the quality of ILD data [10],[11],[12].

- **Relationship between Time-Mean-Speed and Space-Mean-Speed:** SMS is calculated directly from ILD. Linear and complex equalities are then used to estimate TMS, which is required for travel time calculations [13],[14].

In the scope of this project, we focus on the traditional models in detail (Section 3.4). We propose an architecture for real-time route planning with several components, where each component can be changed and improved. For our case study we implemented one of the traditional speed calculation models. To improve average speed calculation and speed estimation of links, other models can be implemented in average speed calculation module.

Here, we present definitions for some of the traffic terms for better understanding:

**TMS** or Time Mean Speed is defined as the arithmetic mean of the speed of vehicles passing a point during a given time interval. Hence, TMS only reflects the traffic condition at one specific point. TMS can be calculated directly from induction loop data.

**SMS** or Space Mean Speed is the harmonic mean of the speeds of vehicles passing on a point on highway during interval of time. SMS gives the average speed information for a given distance. It can be calculated directly only with GPS or video data.
Density is the number of vehicles per unit area of roadway. Inverse of density is spacing, which is the distance between two vehicles.

Flow is the number of vehicles passing a reference point per unit of time.

G-factor is the effective vehicle length in induction loop systems.

2.4 Shortest Path Calculation

An important part of traffic-dependent route planning is to find a path from a source to a destination on a road network that incurs the minimum travel time and considers current and future traffic conditions on all road links. Therefore, dynamic time-dependent shortest path computations are required to update the travel time of each link as new traffic information is obtained. We discuss two works on dynamic shortest path calculation in this section.

[15] presents a new approach for executing continuous route planning queries over a road network, when there are updates to the delay estimates of links. The proposed technique involves a mix of pre-computation and on-the-fly route calculation. For registered queries, likely-good paths are pre-calculated with the historical delay data. System monitors the delay updates, and new routes are sent to users when there is a significant delay change.

Another work, [16] introduces a stream-processing infrastructure that performs scalable, real-time, time-dependent shortest path computations with high throughput and low latency on large road networks. The application uses time-dependent average speed information for all the links to compute time-dependent shortest paths with. The work uses time-dependent A* algorithm for shortest path calculations by extending the algorithm to handle dynamic updates to link travel times.

In our system, we benefit from the work in [16]. We adapted the implementation of the A* algorithm with dynamic updates for traffic-dependent shortest path calculation (Section 5.4).
Chapter 3

Background

This chapter gives background information and presents the technologies used in development of our system.

3.1 Stream Processing Engine: IBM InfoSphere Streams

Intelligent Transportation Systems aim to benefit from several data sources for real-time traffic monitoring and management. Two main challenges of ITS are handling large amount of real-time data and providing service for huge user base. A computing infrastructure is required to support the needed functionality of ITS.

In our work, we use IBM InfoSphere Streams [17] as stream processing engine to deal with the large amount of data. Component-based model of InfoSphere Streams allows us to divide the whole system into individual components, each of which can be replaced or improved to handle different queries or requests. Component-based programming makes our system easy-to-build, apprehensible and adjustable for further needs. Also, InfoSphere Streams allows applications to run in a distributed environment which empowers scalability via pipelining and parallelization. Detailed information for InfoSphere Streams and its features are presented in this section.

3.1.1 IBM InfoSphere Streams

InfoSphere Streams (or Streams) is an IBM product that supports large-scale, high-performance stream processing. It offers both language and runtime support for improving the performance of sense-and-respond applications in processing data from high rate streams. Streams runtime can execute a large number of long-running jobs (queries)
that take the form of data-flow graphs. A data-flow graph consists of a set of operators connected by streams, where each stream carries a series of Stream Data Objects (SDOs).

Each operator implements data stream analytic and resides in execution containers called Processing Elements (PEs), which are distributed over the compute nodes. The operators communicate with each other via their input and output ports, connected by streams. The operator ports as well as streams connecting them are typed.

Spade (Stream Processing Application Declarative Engine) is the declarative stream processing engine of Streams. It is also the name of the declarative language used to program Spade applications. Spade provides a rapid application development (RAD) front-end for Streams. Spade uses code generation to fuse operators into PEs.

The PE code generator produces code that (1) fetches tuples from the PE input buffers and relays them to the operators within, (2) receives tuples from operators within and inserts them into the PE output buffers, and (3) for all the intra-PE connections between the operators, it fuses the outputs of operators with the inputs of downstream ones using function calls.

InfoSphere Streams includes a scheduler component that partitions a data-flow graph across a distributed set of physical nodes as good as possible.

### 3.1.2 Spade Language

Spade [18] provides several built-in operators and stream adapters that are commonly required by streaming applications. The operators currently supported in the stream-relational toolkit are:

1. **Source**: A Source operator is used for creating a stream of data coming from an external source. This operator is capable of performing parsing and tuple creation as well as interacting with external devices.

2. **Sink**: A Sink operator is used for converting a stream into a flow of tuples that can be used by components that are not part of InfoSphere Streams. Its main task consists of converting tuples into objects accessible externally through the file system or network.

3. **Functor**: A Functor operator is used for performing tuple-level manipulations such as filtering, projection, mapping, attribute creation and transformation. Funct operator can access tuples that have appeared earlier in the input stream.
4. Aggregate: An Aggregate operator is used for grouping and summarization of incoming tuples.

5. Join: A Join operation is used for correlating two streams. InfoSphere Streams can be paired up in several ways and the join predicate, i.e., the expression determining when tuples from the two streams are joined, can be arbitrarily complex.

6. Sort: A Sort operator is used for imposing an order on incoming tuples in a stream.

7. Barrier: A Barrier operator is used as a synchronization point. It consumes tuples from multiple streams, outputting a tuple only when a tuple from each of the input streams has arrived.

8. Punctor: A Punctor operator is used for performing tuple-level manipulations, with the exception of filtering. Unlike a Functor, a Punctor can insert punctuation into the output stream based on a user supplied punctuation condition.

9. Split: A Split operator is used for splitting a stream into multiple output streams, based on a split condition that is used to determine which of the output streams a tuple is to be forwarded to.

10. Delay: A Delay operator is used for delaying a stream based on a specified amount of delay, allowing time-shifting of streams.

In addition to the existing operators, Spade allows developers to extend the language. There are three ways of extending the language:

1. The developer can create User-Defined Operators (UDOPs), which are operators that can be used to wrap legacy libraries and provide customized processing. UDOPs are specialized for application-specific stream schemas.

2. The developer can create User Built-in Operators (UBOPs), which are fully templated operators. The templatization makes these operators usable in a type- and stream schema-generic fashion, similarly to regular built-in operators. The UBOP support is the fundamental aspect that allows the language to provide support for the creation of toolkits geared towards other application domains.

3. The developer also can create user-defined functions that can be used in expressions within operators anywhere in the program. For example, the functions can be used in the filtering predicates of a Functor, or they may be used for parsing or formatting in Source and Sink operators, etc.
In our system, we benefit from standard Streams operators. However, similar to many ITS, our system has complex analytic requirements. We deal with different types of structured and unstructured data. We need to analyze the incoming data stream and extract relevant information. Therefore, we enhance the standard Streams Source operator such that only relevant data is used as a data stream within the rest of the application. For further processing of the streams data, several algorithms are implemented. We run our algorithm implementations in C++ as user-defined Operators (UDOPs) inside InfoSphere Streams.

### 3.2 Induction Loop System

A data source is required to be able to collect traffic information and to estimate real-time traffic condition. We benefit from the induction loop technology to get real-time traffic information. We collect data from induction loop detectors and use this data to estimate traffic flow, traffic congestion and average travel time in a road network.

Inductive loop technology is the most common and most reliable system used to detect presence of vehicles. An inductive loop vehicle detector system consists of simply three components: a loop, a loop extension cable and a detector [19].

![Induction loop system](image)

**Figure 3.1: Induction loop system**
Chapter 3. Background

Structure of an induction loop system is shown in Figure 3.1(a). The pre-formed loop is buried in the traffic lane. The loop is a continuous run of wire that enters and exits from the same point. Two ends of the loop wire are connected to the loop extension cable, which in turn connects to the vehicle detector. The detector powers the loop causing a magnetic field in the loop area. The loop resonates at a constant frequency that the detector monitors. A base frequency is established when there is no vehicle over the loop. When a large metal object, such as a vehicle, moves over the loop, the resonate frequency increases (Figure 3.1(b)). This increase in frequency is sensed and, depending on the design of the detector, forces a normally open relay to close. The relay will remain closed until the vehicle leaves the loop and the frequency returns to the base level. The relay can trigger any number of devices such as an audio intercom system, a gate, a traffic light, etc. The actual structure of induction loop data used in our case-study can be found in Section 4.2.

3.3 Road Map: OpenStreetMap

Geographical data is used to describe objects and relations in space. For route-planning applications, geographical data for the area of interest is critical. In our case study, we benefit from OpenStreetMap [20] to get earth coordinates and properties of the roads, streets, and links in the city of Lucerne.

OpenStreetMap is an editable map of the whole world, which is being built largely from scratch, and released with an open content license. The OpenStreetMap License allows free access to the map images and all of the underlying map data. The project aims to promote new and interesting uses of the map data.

OpenStreetMap (OSM) follows a similar concept as Wikipedia does, but for maps and other geographic facts. People gather location data from a variety of sources such as recordings from GPS devices, from free satellite imagery or simply from knowing an area, and upload this data to OpenStreetMap. The data can be further modified, corrected and enriched by anyone who notices missing facts or errors. OpenStreetMap data is available for download in a variety of formats and for different geographical areas.

3.4 Travel Time Estimation: Conventional Model

As discussed in Section 2.3, induction loop detectors provide information about the number of vehicles at a point for a period, but not the speed of vehicles over the loop or the time-mean-speed. Route planning applications require real-time traffic information, like
average speed of the traffic flow in the link or the travel time for the link to calculate the best route possible. We need a model in order to calculate the average speed estimation of the links from ILD data. We use a simple conventional travel time estimation model for this purpose.

Conventional models estimate travel time based on the assumption of a constant average effective vehicle length. The formula to estimate the speed at a single-loop detector data is given as:

\[
V(t) = g(t) \times \frac{c(t)}{o(t) \times T}
\]  

(3.1)

In the formula, \( T \) is the duration of the reporting period. The count \( c(t) \) is the number of vehicles that crossed the detector during period \( t \), and the occupancy \( o(t) \) is the fraction of time during this period that the detector sensed a vehicle above it. The ‘g-factor’ \( g(t) \) is the effective vehicle length in this period. \( g(t) \) cannot be directly measured at single loops, and its value must be assumed or estimated. The formula gives \( v(t) \) as the calculated average speed.

As seen in the formula, reliability of the method depends on the estimated value for \( g \) and the validity of \( g \) to be constant over traffic conditions. If the true \( g(t) \) is different than assumed value, error in speed estimation will be in same proportion. Therefore, there has been a lot of work on estimation of \( g(t) \) value for more precise and correct average speed calculation [21], [22].

In [9], the empirical evidence of variability in the g-factor is provided.

\[
\text{occupation} = \frac{g_{\text{vehicle}}}{v} + \frac{g_{\text{detector}}}{v}
\]  

(3.2)

where \( v \) is the speed of vehicle, \( g_{\text{vehicle}} \) is the length of the vehicle and \( g_{\text{detector}} \) is a function of threshold value and the slope of the detector signal. Assuming during period
Chapter 3. Background

$t$, $c(t)$ vehicles cross detector at speed $v(t)$ then the detector occupancy is written as:

$$o(t) \times T = c(t) \times (g_{traffic}(t) + g_{detector}) \times v(t) \quad (3.3)$$

Here, $g_{traffic}(t)$ is the average value of vehicle length in the traffic. Comparing this equation with the basic speed formula in 3.1, then $g(t)$ is:

$$g(t) = g_{traffic}(t) + g_{detector} \quad (3.4)$$

As seen above, $g(t)$ consists of two components. The first component, $g_{traffic}(t)$ depends on the mix of vehicle types (autos, trucks) crossing the detector during period $t$. So, facts like the traffic situation at the report time $t$, the time of the day and the lane number change the value of this component significantly. The second component $g_{detector}(t)$, depends on the characteristics of the detector itself. The detectors in an area are deployed over a period of years. Therefore, $g_{detector}(t)$ may not be uniform. As a result, $g$-factor can show significant changes depending on different factors, and the accuracy of average speed calculation could vary accordingly.
Chapter 4

Data Model

Intelligent Transportation Systems benefit from real-time, dynamic data to improve modern transportation. Data provided from different parties play a critical role for an ITS application. For our case-study, we use Lucerne city traffic data as representative data to simulate the execution of our route planning system. Although structure and properties of input and output data have an important impact on the system, data is not specific to our case. We present the necessary components and methodologies for a route-planning system in a generic way which can be easily implemented.

In our case study, we have 4 types of data sources. First, we use city map data for the coordinates of streets and the connections between streets. Second, location of induction loop detectors are used to match detectors with the roads they are installed on. Third, readings from ILD are collected to calculate average speed for the roads. Lastly, query stream is used for the requests of shortest path calculation of users. All data sources in our case-study are explained in the following sections.

4.1 OpenStreetMaps Lucerne Map Data

As explained in Section 3.3, OpenStreetMaps is used for the map of Lucerne city. A selected area is exported in XML format (Figure 4.1) from the main site of OpenStreetMaps [20].

The exported XML file consists of raw data Tags and OpenStreetMap data primitives. OpenStreetMaps data primitives are Nodes, Ways, and Relations. A node is the basic element, building block, of the map scheme. Nodes consist of latitude and longitude (a single geospatial point). Optionally, the third dimension, altitude, can be recorded.
Chapter 4. Data Model

Figure 4.1: OpenStreetMaps map extraction for the city of Lucerne

An example for a node in the map XML file is:

```
<node id='254965883' lat='51.517' lon='-0.140' timestamp='2007-01-28T11:40:26Z'>
  <tag k='highway' v='traffic_signals'/>
</node>
```

A **way** is an ordered interconnection of at least two and at most 2,000 nodes that describe a linear feature such as a street, footpath, railway line, river, area or building outline. Each way is characterized with uniform properties like way type (motorway, federal highway, tertiary...), surface quality, speed, etc. Ways are split into shorter sections if different properties exist. For example, if a street has a one-way section, that section is defined as a different way from the two-way section, even though they share the same name. An example XML structure for a way is:

```
<way id='5090250' visible='true' timestamp='2009-01-19T19:07:25Z'>
  <nd ref='822403'/>
  <nd ref='21535912'/>
  <nd ref='333725784'/>
  <tag k='highway' v='motorway'/>
  <tag k='name' v='Bahnhofstrasse'/>
</way>
```

A **relation** is a group of zero or more primitives with associated roles. It is used for specifying relationships between objects, and may also model an abstract object.
For Lucerne city, a map with 653 nodes and 186 ways is extracted. The extracted map is pre-processed (Section 5.2) and properties of roads are exported to a CSV file. It should be mentioned that we assume heights of all roads as zero, such that there is no slope in the city due to lack of height information in input map.

4.2 ILD Data for Lucerne City

Induction loop detector data used in our case study is provided by the official traffic management agency of Lucerne city in Switzerland [23]. The data is collected on Tuesdays for 11-week-period, starting on 22.04.2008 and ending on 01.07.2008. Data for each day consists of 24 XML files of one-hour period.

In the code snippet above, contents of one ILD reading XML file is shown:

```xml
<Data>
  <Elements>
    <Element Index="0">
      <Code>S</Code>
      <Channel>32</Channel>
      <Name>100.SG01</Name>
      <ShortName>100.SG01</ShortName>
    </Element>
    . . .
    <Element Index="121">
      <Code>D</Code>
      <Channel>4319</Channel>
      <Name>202.BB21</Name>
      <ShortName>202.BB21</ShortName>
    </Element>
  </Elements>
  <Measurements>
    <Timestamp>2008-04-22T00:00:00.046Z</Timestamp>
    <Value Index="0">0</Value>
    <Value Index="1">0</Value>
    <Value Index="2">1</Value>
    . . .
    <Timestamp>2008-04-22T00:00:00.484Z</Timestamp>
    <Value Index="51">1</Value>
    . . .
  </Measurements>
</Data>
```
As seen in the code, all elements in the system (in our case, induction loop detectors in the city) are listed at the beginning of file, within <Element> and </Element> tags. Properties for each detector are given between these two tags. There are five properties for each detector element: index, code, channel, name, and short name. Code, channel and short name are recorded for the internal use of VBL. Index of a detector is used as a reference for following timestamp readings, and name of the detector is used to identify its location on the city map.

Following the detector properties, induction loop readings are recorded between <Measurements> and </Measurements> tags. At the beginning of the measurements section, initial values for each detector is given with the starting timestamp of the file. At the following timestamps, only value changes for the detectors are recorded. Reading value is given between <Value> and </Value> tags, and the index of the detector is indicated with index attribute of the timestamp element.

Here, it should be highlighted that timestamps are recorded only when the value of a detector is changed. There might be no input data for a certain period if there is no value change for the detector, eg. if there is no car passing through that period, or if the way is totally occupied during that period.

### 4.3 Induction Loop Detector Data for the City of Lucerne

It can be observed from Section 4.2 that readings from induction loop detectors do not have detectors’ location information. In order to benefit from loop readings, it is necessary to locate the detector on the map and to find the road that it is installed on.

VBL provided us a manual with figures showing the locations of detectors on the map. One of these figures is shown in Figure 4.2.

We generated a file of detector coordinates by using the given manual and Google maps [24] with the following format:

*<Name of Detector>, <Longitute>, <Latitude>, <Direction of Detector>*

Sample values of the created file for the detectors in the above figure are:

202.D018, 47.054303, 8.310958, FORWARD
202.D028, 47.054303, 8.310958, FORWARD

These coordinates are then used to find the closest way to a detector. Implementation details are explained in Section 5.2. We wanted to make the map of the selected area
flexible and the street names changeable. Therefore, we mapped the detectors with their earth coordinates instead of mapping them directly to the roads.

### 4.4 Average Speed Updates for Links

As discussed, ILD readings show the presence or non-presence of a vehicle at the given timestamp. Using this information, we calculate the average speed and travel time (delay) of the link that ILD is installed on. Calculated delays are sent to shortest path operator as an update to previously constructed road map. Structure of an update for a link is:

\[
\text{<Road Number>, <Start Time>, <End Time>, <New Speed Estimate>}
\]

Delay calculation method and the implementation details can be found in the Section 5.3.
4.5 Path Queries

Our system solves the problem of traffic-aware route planning. It executes continuous route planning queries coming from users over a road network, considering the updates to the delay (average travel time) estimates of links. System then presents the shortest path route at the time query is executed. Structure of a route query created by the user is:

<Query Number>, <Departure Point>, <Destination Point>, <Start Time>

For such a query, shortest route between departure and destination points is calculated and the result is presented in the following form to the user:

<Query Number>, <Departure Point>, <Destination Point>, <Calculated Route>, <Start Time>, <End Time>, <Route Duration>

Later, a map layer may be added to system that displays the route on the map. This may help users to easily benefit from the application.
Chapter 5

System Implementation

In this chapter we present the architecture and the components of our traffic-aware route planning system. We explain the roles of components and communication between these components.

5.1 General System Architecture

The system consists of three main components according to function they have. We have a modular, component-based system where every component can be further improved and extended. The general architecture of the system can be seen in Figure 5.1.
First component of our system is called **Pre-Processor**. This component is responsible for gathering data from outside of the system, parsing it, and generating the data streams that can be further used in the system. ITS applications gather data from various types of sources. Each data source has a different structure. Pre-processing is required to convert these data sources into the desired format that can be easily consumed by the system. Second component **Average Speed Calculator** performs the logical processing of the input data, such as calculating the average speed and delay estimations for each link on the map. It aims to benefit from the data sources and generate useful information out of it. The last component **Shortest Path Finder** has the task to find the shortest route for the route queries by using the most recent delay estimations. This component runs continuously executing the stream of user queries.

### 5.2 Pre-Processing

Before performing any calculations pre-processing is required to parse the raw input data sources. We need location-specific traffic data for real-time average speed calculation of the roads. We get the traffic information from induction loop detectors. As discussed in Section 3.2, induction loop readings do not provide any location information. We define a component, Pre-Processor, that maps the real-time traffic data with geographic data. The structure of pre-processing component is shown in Figure 5.2.

![Pre-Processing component](image)

**Figure 5.2: Pre-Processing component**

Pre-processing requires two types of source data to map the traffic information with real-world roads and links; geographic data of the area and the geographic location of induction loop detectors. To find the road where detector is installed, we calculate the shortest distance of detectors to the roads in the area. For each detector, distances to all roads in the area are calculated. The detector is mapped with a road if its distance to that road is minimum of all distance values.
Distance between a detector and a road is calculated in several steps. First, detector’s (longitude, latitude) values are converted to cartesian coordinates. Recall that we define a road by several nodes. Each of these nodes has longitude and latitude that represent a point in real world. We convert the earth coordinates of the nodes of a way to cartesian coordinates. Then, we create segments between two consecutive nodes in every road. Cartesian distances between a detector and all these segments are calculated. For the minimum cartesian distance between a segment and a detector, we define a mapping. The detector is mapped to the road that owns the shortest-distance segment.

At this point, it may be questioned if the shortest distance on cartesian plane is actually the shortest distance on spherical/earth coordinate system. Remember that in Section 4.1, we mentioned our assumption of the height for all roads as zero. We have done some calculations to compare the error of converting earth coordinates to cartesian coordinates with the error in estimating the height of all roads as zero.

As shown in the Drawing 5.3(a), the error caused by converting earth coordinates to cartesian coordinates is:

\[ l = 2R \cdot \sin \frac{\alpha}{2} E_{Estimation} = \left| \alpha - 2 \cdot \sin \frac{\alpha}{2} \right| \]  

(5.1)

whereas the error caused by estimating height of a road as 0 given in Drawing 5.3(b) is:

\[ E_{Altitude} = \alpha \left| 1 - \frac{1}{\cos \beta} \right| \]  

(5.2)

If we compare these two errors, it is clearly seen that the conversion error can be neglected:

\[ \frac{E_{Estimation}}{E_{Altitude}} = \frac{\alpha^2}{\beta^2 \cdot 48} \rightarrow E_{Estimation} \ll E_{Altitude} \]  

(5.3)

Up to now, we presented the general functionality and explained how the calculations are performed by pre-processor component. For our case-study, we implemented a Streams User-Defined Operator (UDOP) called ‘Detector Location Finder’ in the pre-processor.
component. Apart from calculating shortest distances and mapping detectors with the roads, our UDOP also performs parsing of the input data.

InfoSphere Streams provide Source operator (3.1.2) that reads data from file and generates vstream (name given for streams inside InfoSphere Streams). However, in our case both the detectors' geographic data and the map data (OpenStreetMaps) are in XML format, which is not supported by the default Source operator. Therefore, parsing of input files are performed inside ‘Detector Location Finder’ UDOP.

The map of the city of Lucerne is parsed in the UDOP. As explained in 4.1, a road is defined by several nodes in an OpenStreetMap file. Accordingly, ‘Node’ objects are created with longitude, latitude, and index values. ‘Road’ objects are created with the list of indices of the nodes they are made up. Then, the second input file with detector locations is parsed. ‘Detector’ objects are created with longitude and latitude values. As explained in the general pre-processor structure above, the coordinates are converted to cartesian and the segments are generated for every road. We then calculate cartesian shortest distance and generate the detector-road mappings as vstream in InfoSphere Streams. This output stream is exported to a file, which is then consumed by the InfoSphere Streams Source operator inside the main application. Actual code snippets for ‘Detector Location Finder’ UDOP can be found in Appendix A.

To sum up, pre-processing component runs before the main application. It handles static, persistent data which does not affect the execution of the main application and provides easiness and compatibility for various data sources.

5.3 Average Speed Calculation

The second component in the system is Average Speed Calculator. This component has the task to process real-time traffic data, to calculate average speed for roads and to perform continuous updates on road delays.

Pre-processor generates the mappings of detectors with the roads. In order to perform further average speed calculations on the traffic data collected from induction loop detectors, Average Speed Calculator component first performs a JOIN operation (Figure 5.4). After this join, traffic data collected from detectors are also mapped with the roads. As a result, joined data can directly be used for delay estimations and statistics of the roads.
Main function of this component in the system is to calculate average speed estimations and delay durations for the roads of the selected area. This component can be implemented with a three-layer logical structure (Figure 5.5).

First layer is a collection (in our case, hash-map) of ‘Road’ objects, each corresponding to one road in the map. Several properties like road name, road type, delay, etc. for each road are stored in these objects. Type of the road is used to set the maximum limit
for that road. Delay attribute of the road corresponds to the actual (latest calculated) average travel time for the road. Each ‘Road’ object has a collection of ‘Detector’ objects representing the induction loop detectors installed on that road. All information for a detector like detector id, name, earth coordinates are stored as ‘Detector’s attributes.

Third layer symbolizes the window structure in stream processing, which is used to group sections of the data stream with similar time intervals. The ‘Window’ objects in our design have very dynamic nature and they play an important role to create logical groupings of incoming readings. In our system architecture, a variation of time-based ‘Sliding Windows’ is used. This means that the ‘Window’ objects collect incoming data for a certain period of time, called REPORT_PERIOD by the system. If the window exceeds the time-constraint, newly coming tuples can cause eviction of previous tuples. When a sliding window is filled with its first subgroup of data, it performs an operation and then continues with data sliding, as new data is received. A new window is created with an opening time SLIDE_PERIOD later than the last window’s opening time. In sliding window structure, same tuples can be present for multiple processing steps, as shown in Figure 5.6.

![Figure 5.6: Sliding window structure](image)

Definition of sliding-window in our system architecture is a little different than the conventional sliding-window definition. The ‘Window’ objects are not same-sized, and the total size of windows in terms of time duration, depend on the incoming data. The induction loop readings stream, coming from the JOIN operation, contains the real-time traffic data. Recall that the presence of vehicle is represented with value ‘1’, and the absence of vehicle is represented with value ‘0’ in the incoming reading stream. We should consider total occupancy of a vehicle in the same window. This requires that each window can
not be closed before a reading value of 0 arrives after a reading value of 1. As a result, the windows are not exactly same time-sized and they can contain different number of tuples depending on the incoming data.

The average speed calculation starts after JOIN operation that generates readings stream with detector id, timestamp and value of reading. When a new reading comes, the corresponding ‘Detector’ object is located. If the new reading belongs to a detector with no previous windows, a new window is opened. The timestamp of the new reading is recorded as the last window open time. If there are open windows for the ‘Detector’ object, all windows are checked for the window duration (that is if window is opened for more than report time). All windows that will be closed after update are marked. Then for all open windows, statistics are updated.

From the conventional average speed calculation formula (Section 3.4);

\[ v(t) = g(t) \times \frac{c(t)}{a(t) \times T} \]  \hspace{1cm} (5.4)

only the count of vehicles and the occupancy percentage during report period are required. In order to optimize storage and to simplify our calculations, each ‘Window’ object stores only the count of vehicles and the total amount of milliseconds during which the detector detects the presence of a vehicle. When a new reading comes, statistics of ‘Window’ are updated accordingly.
When a reading with value 0 comes, it indicates the absence of the vehicle, and that timestamp is added to the total occupancy duration in the window. If the incoming value is 1, it shows that a vehicle is detected, and the timestamp of the reading is subtracted from total occupancy in that window (Figure 5.7). When we ensure that incoming 1 values are always matched with the incoming 0 values with our window structure, the actual incoming tuple is not stored in the system, and we can calculate the total occupancy of detector in that period very easily.

There might be several windows open at a time, so every open window is updated with the new coming reading. When a new record arrives and if one or more windows have exceeded their reporting period, they are closed immediately and the average speed for that window is calculated. The pseudocode for operation of window updates is given in the code of Algorithm 5.3.1.

**Algorithm 5.3.1: WINDOWS UPDATE**

```java
if No Windows Exist
    lastWindowOpenTime ← timestamp
    Open New Window
    if timeDifference(timstamp, lastWindowOpenTime) >= SLIDE_PERIOD
        Open New Window
    for i ← 0 to windowNumber
        do
            timeDifference ← timeDifference(timestamp, window(i).openTime)
            if timeDifference < REPORT_PERIOD
                window[i].updateCalculation(readingValue, timestamp)
            else if timeDifference >= REPORT_PERIOD and
                window[i].lastValueInWindow is not 1
                then
                    Close Window[i]
                    Send Result Tuple
            else if timeDifference >= REPORT_PERIOD
                window[i].updateCalculation(readingValue, timestamp)
                then
                    Close Window[i]
                    Send Result Tuple
```

When a window is closed and the result tuple is emitted, the output stream of the average speed calculation is in the form of:

```
stream updateStream( wayId: String, speed: Double, delay: Double)
```
Here, delay represents the travel time for the way. It is calculated by dividing the length of way to the estimated flow speed in that way.

For average speed calculation formula, we discussed that $g$-factor in the formula has two components, $g_{\text{traffic}}$ and $g_{\text{detector}}$ corresponding to average vehicle length and the area of detection of a vehicle’s presence. For average speed calculation in our case-study, we used a constant $g_{\text{traffic}}$ for the average value of vehicle length in the traffic. We perform route planning for city Lucerne with a population around 60,000 people and the area of interest is the city-center where most of the roads are single-lane, and the vehicles are small-size. Considering these factors, it would be appropriate to assume a constant $g$-factor as 5 meters.

### 5.4 Shortest Path Calculation

Third component in the system is the Shortest Path Calculator. Main function of this component is to monitor user queries (route requests for departure, destination points) and to respond these queries with the most recent travel values. This component also monitors road delay updates (sent by Average Speed Calculator component) and reflects the updated values to the route calculation.

To find a route between departure and destination points given by users, we need to find the shortest path with the most recent travel times of the links. We benefit from the modified A* algorithm in [16] to calculate the shortest path. (Detailed explanation for original A* algorithm can be found in Appendix A.)

In the initialization of this component, a graph is constructed. This graph corresponds to the road network in the cities. Beginning and ending points of roads are treated as vertices and the roads themselves are treated as the edges of the graph. Delays (travel times) for the roads in the map are used as the weights for edges in the graph. As the queries arrive to operator, shortest route is calculated with the adapted version of A* algorithm.

Real-time route planning requires a time-dependent shortest path calculation. To make the A* algorithm time-dependent, the delay on each road link should be time-dependent. To achieve time-dependency, [16] generates discrete intervals in shortest path calculation. A day is divided into several time intervals and each link has one delay value for each interval. Travel time on a link is assumed to be constant during an interval. As paths are extended with links during the execution of A*, time is advanced and therefore future delay values of links are used as link costs. If the time interval changes while traveling on a link, the delay value of the new interval should be used for the rest of the link. A link
can be traveled during many time intervals, therefore it is checked for interval changes while extending a path with a link. From definition of A* algorithm, a heuristic function is required to estimate the travel time to the goal as well. As a heuristic function, the lower bound for the travel time to the destination node from the current node is used. This lower bound is calculated by dividing the Euclidean distance between these two nodes by the maximum speed on the network.

With the shortest path calculation algorithm, route between departure and destination points are calculated and it is presented to user as a stream. For example, if we have (249,851) as an input query corresponding to (departure wayID, destination wayID) pair, the route found by the system is presented in the form:

(249, 851, [249, 347, 23, 12, 168, 851], 372, 1046, 674)

corresponding to

(departure way id, destination way id, path, start time, end time, total travel time).

In our case-study, we implemented Shortest Path Calculator component as a InfoSphere Streams user-defined operator (UDOP). Our operator have 3 input streams; Lucerne city map, delay updates for roads (coming from Average Speed Calculator), and the query stream.

```plaintext
stream shortestPathStream(fromArc: Long, toArc: Long, path: LongList,
startime: Long, endtime: Long, traveltime: Long)
:= Udp(mapStream; updateStream; queryStream)
["ShortestPath"]
\O
```
Chapter 6

Experiments

We aimed to show the performance and the scalability of our system with experiments.

6.1 Experimental Setup

All the experiments are executed on a laptop with Intel Pentium Core2 Duo 2.0 Ghz processors and 3.0 GB memory with Red Hat Enterprise Linux 5.5 operating system.

6.1.1 Data Set and Query Specification

As explained in Section 4.2 we used the ILD dataset from induction loop detectors in Lucerne. The readings are collected between 22.04.2008 and 01.07.2008 for average speed calculations. Therefore, all records can be regarded as randomly distributed. The dataset has a size of 1,600,000 records. For scalability experiments, data is re-played 5 times.

Queries for experiments are generated with random departure and destination points with the structure shown in Section 4.5.

6.1.2 Metrics

In the first experiment, we test the performance of our system with an increasing amount of traffic data. In a larger network area, there will be more cars passing through detectors, so more induction loop readings will be collected. We simulate this situation by increasing the number of induction loop detector readings per second. We observe the change in average response time of 60,000 queries.
In the second experiment, we test the system by increasing the number of queries per second to simulate the large user base of bigger cities. We calculate the average response time of the system with increasing query/second rate.

For both experiments, response time is measured as the time difference between the start timestamp of the query, given before route calculation and the ending timestamp of the query, given after the route is calculated. We use the timestamps generated by ‘timeMicrosecond()’ function of IBM InfoSphere Streams.

### 6.2 Experiment Results

Figure 6.1 shows the average response time for 60,000 route planning queries on the y-axis where x-axis represents the number of induction loop readings per second, varying from 50,000 to 4,000,000. The dotted line in the graph shows the average response time when there are no delay updates in the system (When the route calculations are made with constant delays assigned to roads).

![Figure 6.1: Change of response time with different ILD readings rate](image)

As we increase the *ILD Readings/sec* rate, there will be more average speed calculations and the number of *road delay updates/sec* will increase. The straight line shows the increase in response time with the increased readings rate. We observe that the performance of the decreases significantly after 1,000,000 *ILD Readings/sec*. 
In the second experiment, we wanted to see the potential of our system for larger user base. We stressed the system with increasing query/sec rate and observed the average response time. Figure 6.2 shows the results for this experiment. At the beginning, system has a high response time. Up to 10,000 queries/sec, system stays idle and its response time is high. At some point, stream processing engine is saturated with queries and the response time starts to decrease (In our experiment, system performs better with the rate of 100,000 queries/sec). After 1,000,000 queries/sec, we observe that system cannot handle the queries, and response time increases. At this point, the system is full and delays for query executions start.

6.3 Discussion

With these experiments, we aimed to see how our system design performs with a case-study. For the first experiment, we expected that the average response time increases when we increase induction loop readings rate. As shown in the results, the system slows down after the rate of 1,000,000 readings per second. Still, it should be observed that the system responses within an acceptable time of 1.3 seconds. As we increased the rate up to 3 million readings per second, the response time continues to increase. We could not test the system with further stress due to the current configuration.
of experiment environment. However, it is highly expected that the response time will increase considering the trend in the graph.

With the second experiment, we wanted to see how our system will respond to increasing query requests. We observe that system shows good performance up to the point where it is saturated. After that point, system slows down and the response time increases significantly. As seen in the results, system performs well for the rates below 1,000,000 queries/sec. This performance would be enough for small cities like Lucerne. For larger cities, we can benefit from the distributed nature of stream processing engines, and distribute the queries over several machines.
Chapter 7

Conclusion and Future Work

7.1 Conclusion

As stated in the motivation, the main goal of this thesis is to develop a traffic-aware route planning system architecture by using stream data management technology. A short introduction of relevant systems and speed estimation models are provided in Chapter 2. An overview of key points behind the system design and introduction to technologies used in the thesis are presented in Chapter 3.

We designed a system to calculate the fastest route between departure and destination points, depending on current situation of traffic. We benefit from several data sources to estimate real-time traffic information and average delay times for the roads. As a case-study, we try our system with the sample data collected in the city of Lucerne. We give data flow in the system architecture in Chapter 4.

We designed the route-planning system with three main components according to function they have. First component Pre-Processor, is responsible for gathering the data from outside of the system, parsing it, and generating the data streams that can be further used in the system. Second component Average Speed Calculator, performs the processing of the input data. The last component Shortest Path Finder, has the task to find the shortest route. We presented the structure of the components in Chapter 5.

Finally, we tested the performance of our system with our implementation for Lucerne-city in Chapter 6. We showed the case of large number of vehicles in the large cities by increasing the number of induction loop detector readings per second. We observe that system performs correct and fast in small cities like Lucerne. From the experiments we can conclude that the system has the potential to serve thousands of users in larger cities with a few improvements.
To conclude, we can say that traffic information data collected in cities can be used in intelligent transportation applications. These applications will decrease the problems like congestion, gas emission problems caused in urban areas. We can benefit from stream data processing engines for developing such transportation applications. With the strong processing power of stream engines, real-time traffic information can be processed easily and dynamic route planning can be performed continuously. For large application areas, there is also possibility to improve system performance by deploying the system in a distributed environment.

7.2 Future Work

Our route-planning system shows good performance and high throughput rate for our case-study of Lucerne city. However, there are limitations in the system. We have used conventional model with a constant g-factor for the speed calculation. As shown in related works section, there are several models that investigate the nature of g-factor with different traffic scenarios and different installation schemas of loop detectors. Also, the shortest path is calculated every time when a new query comes. Such calculation is not problematic for hundreds of users, but it may be unbearable for thousands of users. Instead of calculating shortest path every time from scratch, some optimization techniques can be implemented. For example, similarities between consecutive queries can be detected and the route calculated for first query can be used in route calculation of the second query.

Besides limitations, there are some possible improvements that will improve the performance of the system. The query engine in the system, IBM InfoSphere Streams, provides parallelization and distribution of operators between several machines. Our system can be executed as on a distributed environment, to simulate millions of users in the real-world. Also, an intelligent update algorithm may be integrated for route calculation, e.g. updating the delay for ways only if a certain difference in travel time is observed. Lastly, a visualization tool may be integrated to the system for better understanding of calculated route, and easy usability of the system in general.
Appendix A

A.1 Pre-Processing

InfoSphere Streams application code for Pre-Processing is as follows:

```java
vstream wayStream(
    wayId: Long,
    wayName: String,
    wayType: String,
    wayStartNodeId: Long,
    wayEndNodeId: Long,
    wayDuration: Long
)

vstream nodeStream(
    nodeId: Long,
    longitude: Double,
    latitude: Double
)

vstream detectorStream(
    detectorIndex: Integer,
    code: String,
    channel: String,
    name: String,
    shortName: String,
    longitude: Double,
    latitude: Double,
    wayId: String
)

stream WayData(schemaFor(wayStream))
stream DetectorData (schemaFor(detectorStream))
stream NodeData(schemaFor(nodeStream))
  := Udp()
  ["Detector_Location_Finder"]
  {map_file="luzern_map.xml", det_coord_file ="detector_coordinates.csv"}
```
As explained in Section 5, ShortestPath UDOP is implemented to locate the induction loop detector on the map data. Here is the C++ code for shortest distance calculation part of the implemented UDOP.

```cpp
float minDist, t;
for(unsigned int i = 0; i < detectorVector.size(); i++){
    if(detectorVector[i]->validCoordinates){
        minDist = 0;
        for(unsigned int j = 0; j < wayVector.size(); j++){
            for (unsigned int k = 0; k < wayVector[j]->segments.size(); k++){
                Point* detP = new Point();
                convertSphericalToCartesianPoint(detectorVector[i]->latitude, detectorVector[i]->longitude, detP);
                t = dist_Point_to_Segment(*detP, *(wayVector[j]->segments[k]));
                if (j == 0)
                    minDist = t;
                if (t < minDist){
                    detectorVector[i]->wayIndex = j;
                    minDist = t;
                }
            }
        }
    }
}
```

A.2 Average Speed Calculation

The function that maintains the windows of the ILD readings inside AverageSpeedCalculator operator is given below.

```cpp
bool updateWindows(std::string time, int value){

    bool returnVal = false;
    if(!isWindowOpen && value != 1){
        /* if no windows exist */
        lastWindowOpenTime = time;
        Window * w = new Window(time, value);
        w->updateCalculations(value, time);
        windows.push_back(w);
        isWindowOpen = true;
    } else {
        if (calculateTimeDifference(time, lastWindowOpenTime) >= SLIDE_PERIOD){
            /* open new window when slide limit is reached */
            lastWindowOpenTime = time;
            Window * w = new Window(time, value);
            w->updateCalculations(value, time);
            windows.push_back(w);
        }
        Window * temp;
        //update existing windows
    }
}
```
App endix A. Implementation Code

```c
for(unsigned int i = 0; i < windows.size(); i++){
    temp = windows[i];
    int timeDiff = calculateTimeDifference(time, temp->openTime);
    if (timeDiff != 0 ) {
        // do not use newly created window again
        if (timeDiff < REPORT_PERIOD ) {
            //update window calculations
            temp->updateCalculations(value, time);
        }
        else if(timeDiff >= REPORT_PERIOD && temp->lastValueInWindow == 0){
            closedWindowIndices.push_back(i);
            returnVal = true;
        }
        else if(timeDiff >= REPORT_PERIOD){
            //close window and send result tuple
            temp->updateCalculations(value, time);
            temp->closeTime = time; // added later
            closedWindowIndices.push_back(i);
            returnVal = true;
        }
    }
} //update all windows
return returnVal;
}
```

A.3 A* Algorithm

A* [25] uses a best-first search and finds the least-cost path from a given initial node to one goal node (out of one or more possible goals). It uses a distance-plus-cost heuristic function (usually denoted $f(x)$) to determine the order in which the search visits nodes in the tree. The distance-plus-cost heuristic is a sum of two functions: the path-cost function, which is the cost from the starting node to the current node (usually denoted $g(x)$) and an admissible "heuristic estimate" of the distance to the goal (usually denoted $h(x)$).

The $h(x)$ part of the $f(x)$ function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, $h(x)$ might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.

If the heuristic $h$ satisfies the additional condition for every edge $x,y$ of the graph (where $d$ denotes the length of that edge), then $h$ is called monotone, or consistent. In such a case, $A^*$ can be implemented more efficiently. No node needs to be processed more than once and $A^*$ is equivalent to running Dijkstra’s algorithm with the reduced cost.
**Search Process:** A* algorithm first searches the routes that appear to be most likely to lead towards the goal. What sets A* apart from a greedy best-first search is that it also takes the distance already traveled into account; the g(x) part of the heuristic is the cost from the start, not simply the local cost from the previously expanded node.

Starting with the initial node, it maintains a priority queue of nodes to be traversed, known as the open set. The lower f(x) for a given node x, the higher its priority. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and h values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). (Goal nodes may be passed over multiple times if there remain other nodes with lower f values, as they may lead to a shorter path to a goal.) The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic. If the actual shortest path is desired, the algorithm may also update each neighbor with its immediate predecessor in the best path found so far; this information can then be used to reconstruct the path by working backwards from the goal node. Additionally, if the heuristic is monotonic (or consistent, see below), a closed set of nodes already traversed may be used to make the search more efficient.
Bibliography


