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

Aslı Özal
Systems Group
ETH Zurich, Switzerland
aoezal@student.ethz.ch

Anand Ranganathan
IBM T.J. Watson Research Center
Hawthorne, NY, USA
arangana@us.ibm.com

Nesime Tatbul
Systems Group
ETH Zurich, Switzerland
tatbul@inf.ethz.ch

ABSTRACT

Traffic-aware real-time route planning has recently been an application of increasing interest for metropolitan cities with busy traffic. This paper approaches the problem from a stream processing point of view and proposes a general architecture to solve it. This work is inspired by a real use case and is implemented on an industry-strength stream processing engine. Experimental results on this implementation demonstrate the scalability of this approach in terms of increasing data and query rates.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*query processing*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

intelligent transportation systems, real-time route planning, travel time estimation, traffic sensors, data stream processing

1. INTRODUCTION

Today, traffic congestion is acceptedly one of the main problems of the everyday life in populous cities. Intelligent Transportation Systems (ITS) aim to ease environmental, social, and economic implications of this problem through the application of modern information and communication technology. A challenging use case for these systems is real-time route planning (RRP) based on current traffic information. More specifically, the goal in RRP is to find the best route (i.e., the route that is currently estimated to have the shortest travel time) between a source and a destination point in a given urban area, taking into account the latest information about the current traffic density on all the roads of that area.

Real-time route planning poses a number of technical challenges. First of all, current traffic information needs to be collected from a monitoring infrastructure that is based on special devices installed either on roadways (e.g., sensors, cameras) or on moving objects themselves (e.g., GPS). Second, the collected information needs to be represented in a proper data model so that it can be easily correlated with a road map. Third, traffic density on each road needs to be estimated. Fourth, given an ad-hoc query with a source-destination pair, the best route needs to be computed based on the road map augmented with density estimations. The last but not the least, there is a need to support continuous streams of traffic updates from potentially a large number of monitoring devices as well as ad-hoc queries from potentially a large number of users in a scalable way such that latest estimates can be provided to the users in real time.

From a data management point of view, the real-time route planning problem requires integrating and reasoning about various forms of data in real time, including dynamic streams of traffic data and user queries as well as static reference data such as the road map and the locations of the monitoring devices. In this paper, we argue that data stream processing technology forms a suitable foundation to solve this problem, since it not only allows us to easily represent the streaming data sources and the continuous push-based processing on them, but also facilitates integrating the other data sources involved as well. More importantly, we can exploit the low latency/high throughput performance guarantees inherently provided by this technology.

The work described in this paper is inspired by a real use case for the city of Lucerne in Switzerland. In Lucerne, an Induction Loop Detector (ILD) system installed at road intersections collects real-time traffic information by reporting the vehicles passing by the detectors. In addition to the ILD datasets provided by the Lucerne Transportation Services [5], we have made use of the Lucerne city map provided by the OpenStreetMap project [2]. In this paper, we first show how to architect a real-time route planner on top of a stream processing system by describing a number of key components. Some of these components also leverage a number of well-known techniques from traffic management and computer algorithms, such as travel time estimation models and shortest path calculation algorithms. Then we present how to realize this generic design using a state-of-the-art stream processing engine, namely, IBM InfoSphere Streams [1]. This proof-of-concept implementation has been tested with the Lucerne datasets and some scalability experiments have been performed, which we also summarize in this paper [15]. To our knowledge, this work is the first that puts together different algorithms and components in this domain

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWGS '11, November 1, 2011, Chicago, IL, USA
Copyright ©2011 ACM 978-1-4503-1036-9/11/11 ...$10.00.

to build a complete, end-to-end system for real-time route planning.

The rest of this paper is outlined as follows. Section 2 describes the real-time route planning problem, both in general terms and in the context of our specific use case for the city of Lucerne. We then propose in Section 3 a stream-based solution approach to this problem, presenting a general design that builds on a streaming infrastructure and a specific implementation of this design on top of the IBM InfoSphere Streams platform. In Section 4, we report preliminary experimental results that show the basic performance of our implementation in terms of its scalability. We briefly summarize related work in Section 5 and conclude the paper in Section 6 by discussing a few interesting directions for future research.

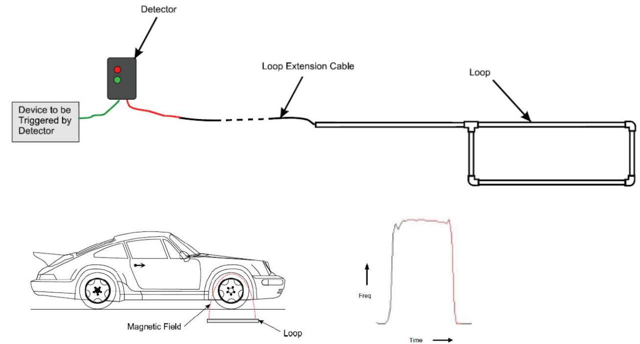


Figure 1: Induction Loop Detector system [4]

2. REAL-TIME ROUTE PLANNING (RRP)

In this section, we will briefly introduce the real-time route planning problem, both in general terms and in the context of a specific real-world use case.

2.1 Problem Overview

The real-time route planning (RRP) problem can be defined as follows: Given the latest information about the traffic density on all roads in a geographical area and a route query in the form of a (source, destination) pair, find the route from the given source point to the given destination point that is estimated to have the shortest travel time at that time point.

RRP comes with a number of technical challenges. First of all, real-time traffic information needs to be collected from a monitoring infrastructure installed on various traffic units, such as roads or vehicles. Depending on the format in which this raw monitoring data is presented, it may be necessary to pre-process or augment this data with more detailed information regarding geographical location. Then the traffic density on each road needs to be estimated by aggregating the sensor readings in some way. Finally, route queries should be answered as they are received from the users. An especially challenging aspect of this problem is that data and query rates can get very high depending on the size and population of the area that is being monitored.

For a better understanding of the RRP problem, we will next describe a real use case for the city of Lucerne in Switzerland.

2.2 RRP for the City of Lucerne

Being the most populous city in central Switzerland and a world-renowned popular tourism destination, Lucerne is one of the major transportation hubs in its region. Lucerne Transportation Services (VBL AG) [5] has installed an Induction Loop Detector (ILD) system in Lucerne in order to collect traffic information. VBL currently uses the collected data for historical analysis of the general traffic patterns in the city. However, it is of interest to VBL to be able to use this data in real-time applications as well, such as providing an RRP service.

Before describing the Lucerne use case further, let us first give some technical background on the ILD technology. ILD technology is essentially a form of sensor technology that presents a reliable way to detect presence of moving objects at a given location. An inductive loop vehicle detector system consists of three main components: a loop, a loop extension cable, and a detector (see Figure 1). The loop is a continuous run of wire that enters and exits from the same point and is buried under the traffic lane. The 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. 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 then trigger a number of other devices such as an audio intercom system, a gate, a traffic light, etc.

For the case of Lucerne, we examined a dataset from 121 induction loop detectors installed in the center region of Lucerne with relatively busy traffic. Separate detectors are installed on different lanes of a given road so that multiple vehicles simultaneously passing through the road in reverse directions can be more accurately detected. Our dataset covers the traffic observed on Tuesdays for an 11-week period, from 22.04.2008 to 01.07.2008. Data for each day consists of 24 files of 1-hour period each.

Induction loop detectors in the area are listed at the beginning of a file and properties for each detector are given within 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, whereas *index* of a detector is used as a reference for following the timestamp readings and *name* of a detector is used to identify its location on the city map. Following these detector properties, induction loop readings themselves are recorded. At the beginning of the measurements section, initial values for each detector is given with the starting timestamp of the file. Reading value can either be 1, which indicates that there is a car on the corresponding induction loop detector, or a 0, which represents the absence of a car. At the subsequent timestamps, only value changes for the detectors are recorded. In other words, 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 recorded for a detector during that time, which might happen, for example, if there is no car passing through that period, or if the traffic has stopped at the cor-

responding lane due to a traffic light, accident, or extreme congestion.

Given continuous streams of ILD readings from 121 detectors in the format described above, our goal in this use case is to be able to answer a continuous stream of ad-hoc route planning queries from a multitude of users *in real time*. Each ad-hoc query comes with the following metadata: *query-id*, *submission-time*, *departure-point*, and *destination-point*. Furthermore, the answer to each such query needs to contain the following information: *query-id*, *submission-time*, *termination-time*, *departure-point*, *destination-point*, *best-route*, *estimated-travel-time*. The answer can simply be presented in textual format, or by means of a GUI that consists of a map displaying the suggested route along with the estimated travel time.

3. A STREAM-BASED APPROACH

In this paper, we present a stream-based solution approach to the real-time route planning problem that is outlined in the previous section. Data stream processing technology not only allows us to easily represent the streaming data sources and the continuous push-based processing on them, but also facilitates integrating the other data sources involved as well. More importantly, we can exploit the low latency/high throughput performance guarantees inherently provided by this technology. This section describes our approach both in terms of general design and a specific implementation of this design, and we evaluate the basic performance of this implementation in the next section.

3.1 Architectural Overview

In this section, we present the general architectural design of a stream-based solution to the real-time route planning problem described in the previous section.

Based on the problem requirements, we identified a number of different data sources and key functional components to process data from these sources. Figure 2 shows these and how they can be architected on top of a stream processing system.

First of all, an RRP application needs to integrate information from various types of external data sources. Each source may present its data in different syntax and semantics. A pre-processing step is required to interpret and convert these into a format that can be easily consumed by the rest of the system. Thus, the first key component in our architecture is the *Pre-Processor*. This component is essentially responsible for receiving external input data, parsing it, and generating the relevant data streams that will further be used in the system. Next, the pre-processed data along with real-time traffic updates should be further processed to infer the current traffic density in the given city. Thus, the second key component, *Average Speed Calculator*, calculates the average speed and delay estimations for each link on a given map. Finally, a *Shortest Path Finder* component takes in the route queries and calculates the shortest paths for them based on the most recent traffic density information from the Average Speed Calculator. Both the Average Speed Calculator and the Shortest Path Finder runs continuously as streams of traffic updates and queries arrive, while the Pre-Processor executes once in the beginning to derive the necessary information from the static reference data. Let us now look at each of these components more closely.

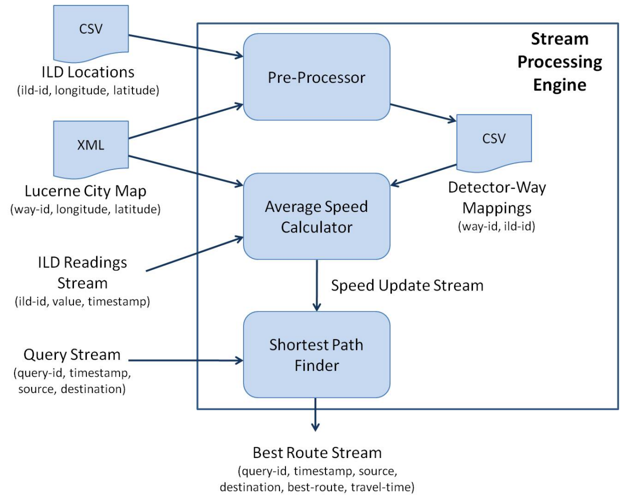


Figure 2: A stream-based system architecture for real-time route planning

3.1.1 *Pre-Processing*

As mentioned above, the Pre-Processor component runs in advance of the other streaming components to prepare the data from static data sources for continuous consumption of these other components. For example, in the use case that we studied, we observed the need for pre-processing of two types of data: (i) ILD locations and (ii) the city map. The former is needed, as readings from ILD sensors do not provide any location information, and in order to be able to compute location-specific traffic estimates, we need this information. The Pre-Processor reads the ILD locations from a CSV file and correlates them with the city map information (represented in XML) to compute mappings between ILD devices and the roads that they are installed on. To find the roads where each detector is installed, we calculate the distance of each detector to all the roads in the city map. The detector is then mapped to the road that has the minimum distance to it.

Depending on how detector locations and map data are represented, distances between detectors and roads may need to be calculated in special ways. For example, in our datasets, both the detector locations and the road segment (a.k.a., node) locations are represented with earth coordinates. First, a given detector's (longitude, latitude) values are converted into Cartesian coordinates. Each road may consist of several nodes, each of which has a (longitude, latitude) value that represents a point in the real world. We also convert the earth coordinates of the nodes of a way into Cartesian coordinates. Then we create segments between two consecutive nodes in every road. Cartesian distances between a detector and all road segments are calculated. Each detector is mapped to the road that contains the shortest-distance segment, i.e., the segment that has the minimum Cartesian distance to the detector. Figure 3 illustrates this mapping.

3.1.2 *Average Speed Calculation*

As real-time traffic updates from detector devices are received, we need to update the current traffic density on the roadways so that when a route query is received, we can formulate an answer based on the most recent information.

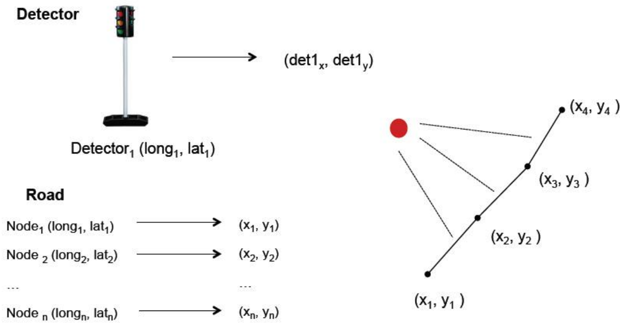


Figure 3: Locating an ILD device on the road map

The Average Speed Calculator provides this update functionality based on a travel time estimation model.

This component takes the city map, real-time traffic readings stream, and the detector-way mappings from the Pre-Processor component as input. It first performs a join operation between the readings stream and the detector-way mappings so that current traffic readings can be associated with the roadways. Next, for each roadway, current average traffic speed as well as the time delay need to be estimated.

The Average Speed Calculator follows a three-layer data representation (see Figure 4). At the highest layer are the roads of the city map, for which we need to compute the travel time estimates. Each road has a number of detector devices installed on it. Finally, at the lowest layer, each detector receives a stream of updates which needs to be chopped into *windows* for continuous processing. Windows represent finite excerpts of a stream that have a certain relationship (e.g., within a certain time interval) [9].

For our use case, a variation of the time-based sliding window model is used. Like a traditional time-based sliding window, each window keeps readings of a detector for a certain period of time, that we call the *reporting period*. When a window is filled with its first subgroup of data, it performs an operation and then continues sliding, as new data is received. A *slide period* determines when to open the next windows relative to the opening of the previous window. Unlike the traditional time-based sliding windows however, in our case, windows are not necessarily of the same size, but the actual size of a window is determined by the incoming data. This is a special situation that arises from the semantics of the ILD readings.

In the ILD stream, presence of a vehicle is represented with value '1', and the absence of a vehicle is represented with value '0'. We would like to maintain statistics about vehicle counts and total occupancy (i.e., the total time for which the road was occupied by a vehicle) for the time period determined by the window. This requires that a window should not be closed before a reading value of 0 arrives after a reading value of 1. As a result, the windows are not exactly of the same size (both time-wise as well as in terms of actual number of tuples contained in them).

Figure 5 illustrates how we compute traffic density statistics (count and occupancy) based on a sliding window model. The input stream contains 0/1 values as well as timestamps. For a given window, we keep internal state for the current count and the current occupancy since the opening of that window. This incremental way of processing both optimizes

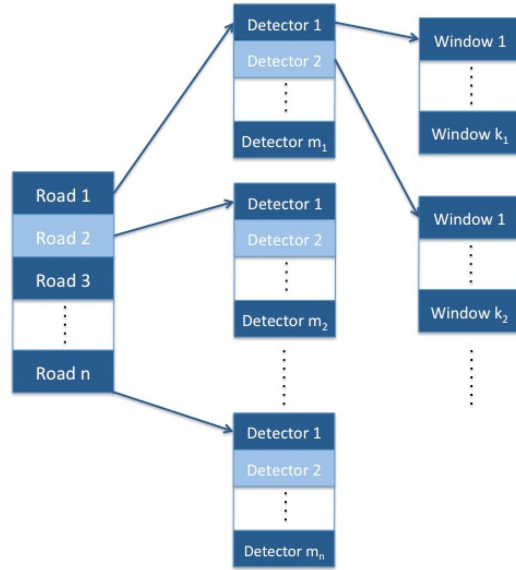


Figure 4: Three-layer data representation

memory consumption and processing time. When a 0 reading arrives indicating the absence of a vehicle, its timestamp is added to the total occupancy duration in the window. On the other hand, when a 1 reading arrives indicating vehicle presence, its timestamp is subtracted from the total occupancy duration in that window. When we ensure that each 1 value in a window is matched with a corresponding 0 value in that window and the window has exceeded its reporting period, that window can be closed and a final average speed value can be emitted.

3.1.3 Shortest Path Finding

The Shortest Path Finder component receives the real-time route planning queries from the users as a stream and answers them based on the latest traffic density estimates from the Average Speed Calculator component.

To find the best route between a given pair of departure and destination points, we need to find the shortest path with the most recent travel times of the roads in the map. To achieve this, we utilize the modified A* algorithm presented in related work [10].

When the Shortest Path Finding component is first initialized, it constructs a graph that corresponds to the road network of the given city. Beginning and end points of roads are treated as vertices and the roads themselves are treated as the edges of the graph. Delays (i.e., travel times) for the roads in the map are used as the weights for edges in the graph. These weights are updated as new traffic information comes in. This weighted graph is given as input to the modified 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, the approach proposed by Guc and Ranganathan generates discrete intervals in shortest path calculation [10]. 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

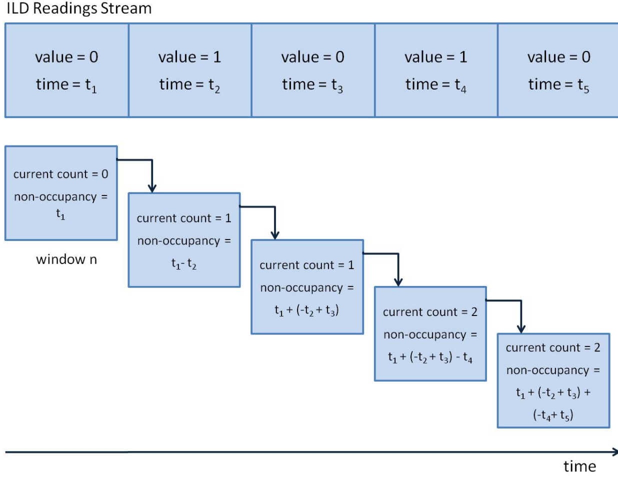


Figure 5: Sliding window-based occupancy and count calculation

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.

A* algorithm requires a heuristic function to estimate the travel time to the destination 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.

After applying the algorithm described above, the Shortest Path Finder component presents the best route to the query as part of the output stream produced by our architecture.

3.2 Implementation

In this section, we present how we implemented the architecture proposed in the previous section within the context the IBM InfoSphere Streams system as a state-of-the-art stream processing engine underneath. We give more details in terms of our Lucerne use case and several other technologies used such as the specific travel time estimation model that we used. It is important to note that while we have chosen the IBM engine for a proof-of-concept implementation of our design, we believe that our design is general enough to be realized on any stream processing engine.

3.2.1 IBM InfoSphere Streams System

In this work, we have used the IBM InfoSphere Streams system (version 1.2) as the underlying stream processing engine [1]. InfoSphere Streams (or Streams for short) has been designed to support large-scale stream processing. Streams runtime can execute a large number of queries that are in the form of data-flow graphs. A data-flow graph consists of a set of operators connected by streams. Each operator implements data stream analytics and resides in execution containers called Processing Elements (PEs), which are distributed over a given set of compute nodes. The operators communicate with each other via their input and output ports, connected by streams.

SPADE is the declarative stream processing engine of Streams [8]. It is also the name of the declarative language used to program SPADE applications. SPADE provides several built-in operators and stream adapters that are commonly required by streaming applications. The built-in operators include source, join, sink, and aggregate. In addition to the existing operators, SPADE allows developers to extend the language with User-Defined Operators (UDOPs), User Built-in Operators (UBOPs), and User-Defined Functions (UDFs). We have made use of these extensibility features of Streams in our work.

In our implementation, we benefit from several standard Streams operators as well as implementing new custom operators. As in many other applications, our system has complex analytic requirements that may not exactly fit in with the standard operator set. First of all, as we deal with different types of structured and unstructured data sources, we need to analyze them to extract only the relevant information. Therefore, we have enhanced the standard Streams source operator as user-defined sources such that only relevant data is used as a data stream within the rest of the application. Moreover, for further processing of the stream data, several special algorithms needed to be implemented. We implemented these algorithms in C++ as User-Defined Operators (UDOPs) inside InfoSphere Streams. More specifically, we implemented three UDOPs, one for each of the three main components in our proposed architecture.

3.2.2 Travel Time Estimation Model

For real-time route planning, we need to calculate current traffic density on roads, which can be represented by average traffic speeds for these roads. In the Lucerne use case, ILD readings stream provide information about the number of vehicles at a road segment for a given time period, but not the speeds or time-mean-speeds of vehicles over the loop. We need a model in order to calculate the average speed estimation of the links in our graph representing the city road map from the available ILD data. We have used a 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 [18]:

$$V(t) = g(t) \times \frac{c(t)}{o(t) \times T} \quad (1)$$

In this 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; the occupancy $o(t)$ is the fraction of time during this period that the detector sensed a vehicle above it; and 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 $g(t)$ and the validity of $g(t)$ to be constant over traffic conditions. If the actual $g(t)$ is different than the assumed value, there would be a proportional error in speed estimation. Therefore, there has been a lot of work on improving the estimation of $g(t)$ value for more precise

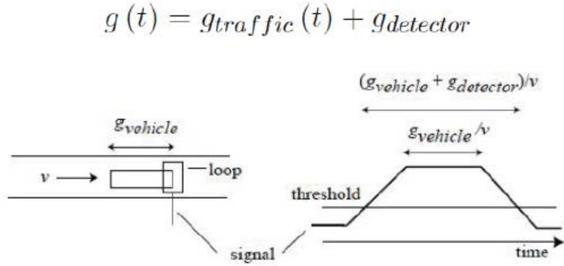


Figure 6: Estimating the effective vehicle length [18]

and correct average speed calculation [7, 16].

In related work [18], $g(t)$ is divided to two components, as illustrated in Figure 6. The first component, $g_{traffic}(t)$ depends on the mix of vehicle types (autos, trucks) crossing the detector during period t. Therefore, factors like the traffic situation at 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.

For our case study, considering Lucerne is a small city and it is not very common to see big trucks in the city traffic, we have assumed an effective vehicle length of 5 meters.

3.2.3 OpenStreetMaps

To implement our architecture, we have also needed geographical data to locate various objects on the earth surface. In our implementation, we have made use of the OpenStreetMap project to get earth coordinates and various other properties of the roads, streets, and links in the city of Lucerne [2].

OpenStreetMap is an editable map of the whole world. It is released with an open content license. The OpenStreetMap allows free access to the map images and all of the underlying map data. OpenStreetMap follows a similar concept as Wikipedia does. 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, or 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.

We have selected a region of interest in Lucerne city for our case study, and exported all the streets and links in XML format. The exported XML file consists of nodes and ways. A node is the basic element, building block, of the map scheme. Nodes consist of latitude and longitude (a single geospatial point). A way is an ordered interconnection of at least two nodes that describe a linear feature such as a street. For Lucerne city, a map with 653 nodes and 186 ways has been extracted.

4. EXPERIMENTS

In this section, we present a few results from an experimental study that we conducted on our RRP implementation for Lucerne. The main goal is to see to what extend the system scales with increasing stream rates (ILD readings processed per time unit as well as ad-hoc user queries handled per time unit).

4.1 Experimental Setup

All our experiments were executed on a laptop with Intel Core2 Duo 2.53 GHz processor and 4 GB memory running the Windows 7 operating system.

As our continuous traffic stream, we replayed the ILD dataset for Lucerne that was described in detail in Section 2. This dataset has a total size of 1,600,000 records. In our experiments, we replayed the data 5 times.

As our continuous query stream, we generated random pairs of departure and destination points.

In the first experiment, we test the performance of our system with an increasing amount of traffic data by changing the number of ILD readings per second. We measure the average response time of a total of 5,000,000 queries, feeding them at two different rates: first at 50,000 queries/second and then at 500,000 queries/second .

In the second experiment, we test the system performance by increasing the number of queries per second. We again measure the average response time of queries. Note that this experiment has been run for two different data rates: first 50,000 ILD readings/second and then 500,000 ILD readings/second.

For both experiments, query response time is measured as the time difference between the start timestamp of the query (i.e., its *submission-time*) and the end timestamp of the query (i.e., its *termination-time*). All experiments were repeated 3 times.

4.2 Experimental Results

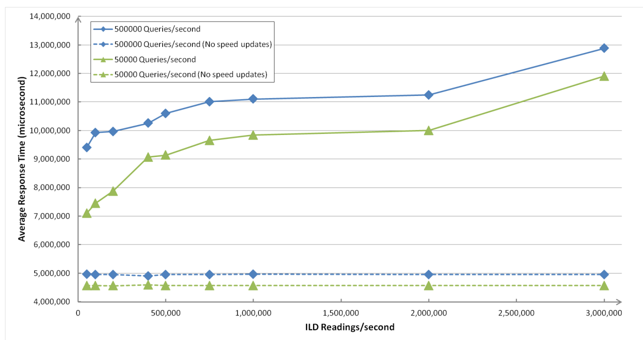
Figure 7(a) shows the average response time for route planning queries on the y-axis, while the x-axis represents the number of induction loop readings per second, varying from 50,000 to 3,000,000. The experiment was run for two fixed query rates – 50,000 and 500,000 queries/second, respectively.

The two dotted lines in the graph show the average response times, when there are no delay updates in the system (i.e., when the route calculations are done with constant travel times assigned to roads where the shortest path graph stays fixed over time). We plot these to serve as conservative baselines for the two respective query rates.

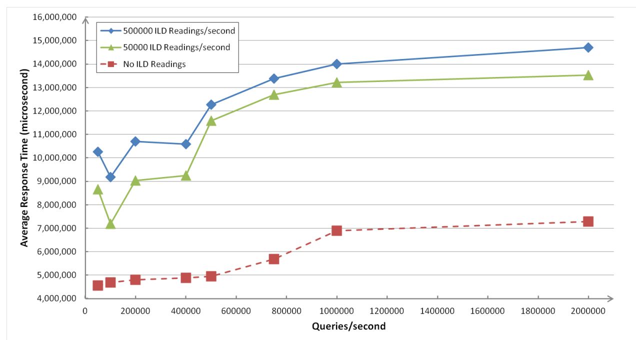
The two solid lines show the increase in response time with the increasing data rate, following a similar trend. In both cases, we observe that the performance of the system starts significantly degrading after 2,000,000 ILD readings per second. This is because, as we increase the input date rate, there will be more average speed calculations per time unit and consequently, the number of road delay updates will also increase.

In our second experiment, we wanted to see the potential of our system for supporting a large number of user queries. We stressed the system with increasing query rate and measured the average response time. Figure 7(b) shows the results for this experiment, repeated for two different input data rates.

The dotted line in the graph again shows the average response times, when there are no new delay updates in the system (i.e., when the route calculations are done with con-



(a) Scalability with increasing data rate



(b) Scalability with increasing query rate

Figure 7: Experimental results

stant travel times assigned to roads where the shortest path graph stays fixed over time). We plot this to serve as a conservative baseline for the two input data rates. Note that unlike in the previous experiment, in this case, the baseline is common for the two workloads, since rate of the ILD readings which distinguishes these two workloads is irrelevant in the baseline case as the delay updates are ignored in this case. Furthermore, the baseline also shows a slight increase in average response time as the query rate increases (even more noticeably after 500,000 queries/second), since executing more queries in the system is expected to cause higher latencies, and possibly to overload at some point.

The two solid lines show the increase in response time with the increasing query rate, following a similar trend. In both cases, the system shows an initial dip in average response time. More specifically, below 100,000 queries/second, a lot of the processing time is taken up by other tasks like reading files, building up data structures for shortest path calculations, and so on. At around 100,000 queries/second, total execution is occupied evenly by shortest patch calculations and the other tasks. Beyond this point, the shortest path calculations themselves start to dominate the processing time, and as a result, we see that the average response time starts increasing as the query rate increases. After around 500,000 queries/second, we observe that system can not handle the queries with reasonable response times any more. This is the point where the system starts being overloaded, as is also suggested by the uprising behavior of the baseline beyond this point.

All in all, the two results we have shown above demonstrate that our RRP implementation can support up to an order of million sensor readings per second and user queries per second, which would be sufficient limits for many metropolitan cities (and certainly more than enough for Lucerne). For larger-scale scenarios, several alternatives exist to boost up the system performance including exploiting distributed / parallel stream processing, multi-query optimization, and approximate processing, which we briefly discuss as part of future directions in the last section of this paper.

5. RELATED WORK

Data stream processing systems have a wide range of application domains such as stock market data analysis, weather monitoring, and recently, also intelligent transportation systems (ITS). Advances in location tracking technologies like GPS and sensor systems, and the growing need for traffic performance monitoring in large cities have increased the interest in ITS applications. We discuss a representative set of recent research work that are closely related to our RRP work.

Microsoft's GeoInsight is a framework developed for geostreaming applications [11]. 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 from our work, they perform analysis of historical data together with the real-time data to refine the answer of real-time queries and to predict the answer in the near future.

Our work involves applying data stream management technology to ITS applications. The latte project [17] is developed with the same goal by using the NiagaraST stream processing system [12] and the PORTAL transportation data archive [3]. 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. Different from our real-time travel time calculations, they find similarities between past data and current data to estimate travel time.

More recently, Malviya et al. have proposed a dynamic route planning technique for a given set of route queries pre-defined in the system [13]. Different from our work, this work uses historical delay data for pre-computation of candidate routes for the pre-registered queries. Based on this, the system checks delays and sends query results as updates to users if the fastest route for any of these route queries changes.

Lastly, the IBM InfoSphere Streams system was also used for a similar application in previous work, where a prototype system that generates dynamic transportation information for the city of Stockholm was developed [6]. That implementation 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 from the ILD data that we use to get traffic information, they utilize GPS data which gives speed information of vehicles

directly. This work also explores how to improve scalability using distributed operation techniques, which is complementary to our work.

6. CONCLUSIONS

In this paper, we investigated the real-time route planning problem and proposed a stream-based approach to solve it. Our work has been inspired by a real use case for the city of Lucerne, Switzerland. We first proposed a general architecture of a stream-based route planner and then described how we realized it within the context of the IBM InfoSphere Streams engine for the Lucerne use case. Experimental results on this implementation show that our solution has an acceptable scale-up behavior for increasing data and query rates.

This work can be improved along several directions. First, different travel time estimation models can be analyzed in depth to improve the accuracy of the system. Second, the usability of the system can be significantly improved by integrating a map-based GUI for the users. In addition to these immediate improvements, we also envision a few research ideas for elevating the performance of our approach. More specifically, scalability of the system can be further increased by applying a few common techniques that would fit very well with the RRP application semantics. For example, most streaming engines provide facilities for distributed and parallel processing. We can distribute user queries or ILD readings for different regions of the map to make use of multiple processing resources. As another example, multiple user queries that are similar can be answered in less time by exploiting their commonalities. Thus, the shortest path finding algorithm can be improved in this direction. The last but not the least, as the experiments have shown, updating the time estimates on the links of the graph that represents the road network significantly affects the query response time. Thus, we can use approximate processing techniques by down-scaling the refresh rates when the system has more load that it can handle (e.g., in similar ways to the work proposed in [14]).

7. ACKNOWLEDGMENTS

We would like to thank Romeo Kienzler and Bugra Uytun from IBM Switzerland for their help with the IBM InfoSphere Streams system, Marc Amgwer from VBL AG and Thomas Karrer from Stadt Luzern for providing information on the Luzern use case, and Baris Guc from ETH Zurich for his contributions to the implementation of the shortest path algorithm. This work has been supported in part by an IBM faculty award.

8. REFERENCES

[1] IBM InfoSphere Streams. `http://www-01.ibm.com/software/data/infosphere/streams/`.

[2] OpenStreetMap. `http://www.openstreetmap.org/`.

[3] PORTAL Transportation Data Archive. `http://portal.its.pdx.edu/home/`.

[4] The Basics of Loop Vehicle Detection. `http://www.marshproducts.com/pdf/InductiveLoopWriteup.pdf`.

[5] Verkehrsbetriebe Luzern AG. `http://www.vbl.ch/`.

[6] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services. In *ACM SIGMOD Conference*, Indianapolis, IN, USA, June 2010.

[7] D.J. Dailey. A Statistical Algorithm for Estimating Speed from Single Loop Volume and Occupancy Measurements. *Elsevier Transportation Research Part B: Methodological*, 33(5), June 1999.

[8] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S Declarative Stream Processing Engine. In *ACM SIGMOD Conference*, Vancouver, BC, Canada, June 2008.

[9] L. Golab and M. T. Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2), June 2003.

[10] B. Guc and A. Ranganathan. Real-time, Scalable Route Planning using a Stream-Processing Infrastructure. In *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, Madeira Island, Portugal, September 2010.

[11] S. J. Kazemitabar, U. Demiryurek, M. Ali, A. Akdogan, and C. Shahabi. Geospatial Stream Query Processing using Microsoft SQL Server StreamInsight. *PVLDB*, 3(2), 2010.

[12] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier. Out-of-Order Processing: A New Architecture for High-Performance Stream Systems. *PVLDB*, 1(1), August 2008.

[13] N. Malviya, S. Madden, and A. Bhattacharya. A Continuous Query System for Dynamic Route Planning. In *IEEE ICDE Conference*, Hannover, Germany, April 2011.

[14] A. Moga, I. Botan, and N. Tatbul. UpStream: Storage-centric Load Management for Streaming Applications with Update Semantics. *VLDB Journal*, 2011 (to appear).

[15] A. Özal. Real-time Route Planning with Stream Processing Systems: A Case Study for the City of Lucerne. Master's thesis, ETH Zurich, Switzerland, 2011.

[16] S.Turner et al. Travel Time Data Collection Handbook. Technical report, Federal Highway Administration Research Report, 1998.

[17] K. Tufte, J. Li, D. Maier, V. Papadimos, and J. R. R. L. Bertini. Travel Time Estimation Using NiagaraST and latte. In *ACM SIGMOD Conference*, Beijing, China, June 2007.

[18] J. Zhanfeng, C. Chao, B. Coifman, and P. Varaiya. The PeMS Algorithms for Accurate, Real-time Estimates of G-factors and Speeds from Single-loop Detectors. In *IEEE International Conference on Intelligent Transportation Systems (ITSC)*, Oakland, CA, USA, August 2001.