IBM InfoSphere Streams for Scalable, Real-Time, Intelligent Transportation Services

Alain Biem, Eric Bouillet, Hanhua Feng, Anand Ranganathan, Anton Riabov, Olivier Verscheure
IBM TJ Watson Research Center, Hawthorne, NY 10532
{biem, ericbou, hanhfeng, arangana, riabov,ov1}@us.ibm.com

Haris Koutsopoulos, Carlos Moran
KTH Royal Institute of Technology, Stockholm
{hnk, carlos}@infra.kth.se

ABSTRACT

With the widespread adoption of location tracking technologies like GPS, the domain of intelligent transportation services has seen growing interest in the last few years. Services in this domain make use of real-time location-based data from a variety of sources, combine this data with static location-based data such as maps and points of interest databases, and provide useful information to end-users. Some of the major challenges in this domain include i) scalability, in terms of processing large volumes of real-time and static data; ii) extensibility, in terms of being able to add new kinds of analyses on the data rapidly, and iii) user interaction, in terms of being able to support different kinds of one-time and continuous queries from the end-user. In this paper, we demonstrate the use of IBM InfoSphere Streams, a scalable stream processing platform, for tackling these challenges. We describe a prototype system that generates dynamic, multi-faceted views of transportation information for the city of Stockholm, using real vehicle GPS and road-network data. The system also continuously derives current traffic statistics, and provides useful value-added information such as shortest-time routes from real-time observed and inferred traffic conditions. Our performance experiments illustrate the scalability of the system. For instance, our system can process over 120000 incoming GPS points per second, combine it with a map containing over 600,000 links, continuously generate different kinds of traffic statistics and answer user queries.

Categories and Subject Descriptors

H.3 [**Information Storage and Retrieval**]: Systems and Software; J.0 [**Computer Applications**]: General

General Terms

Performance, Design

SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.

Keywords

Stream Processing, Transportation, Geostreaming

1. INTRODUCTION

The rapid growth of demand for transportation and high levels of car dependency caused by urban sprawl have caused great stresses to the transportation infrastructure in many areas. This has resulted in severe traffic congestion, and associated productivity loss, environmental degradation, and consumption of scarce resources. In urban areas, adding capacity through construction of new facilities is a very difficult endeavor due to lack of space and prohibitive costs. Intelligence Transportation Systems (ITS) is an umbrella term encompassing sensor, communications and computing technologies to manage existing infrastructure and transportation systems more efficiently, and hence contribute to the reduction of congestion. An important development within ITS is the emergence and installation of different kinds of sensor technologies for collecting data on the state of the transport system [1]. An important example is the use of GPS for traffic data collection. GPS data and other opportunistic sensors (e.g. smart phones) have great potential to provide the large amounts of data that is needed to support real time management of traffic systems.

Intelligent Transportation Systems can make use of these data sources and data from other related sources (e.g. weather, video cameras, etc) to enable real time traffic monitoring and management with a broader scope and sustainability than usually achieved. There are, however, major challenges in building systems that are flexible and powerful enough to handle diverse demands from a large user base.

The first challenge is one of scalability. As various kinds of sensor technologies become available and adopted, the data they produce must be fused and analyzed in real-time. The rate of processing such data in cities can easily exceed hundreds of thousands of points per second. In addition, the GPS data must be integrated in large maps (road networks) that can potentially contain hundreds of thousands of nodes and links.

A second challenge is the development of the computing infrastructure required to support the needed functionality of ITS, especially given the large volumes and variety of data available and the diverse set of parties involved (such as government agencies, commercial enterprises and end-user commuters). Studies have shown that developing and in-

tegrating the various components of an ITS infrastructure constitute a significant portion of the capital cost and complexity of such systems [14]. These systems access different types of data sources, that produce different kinds of content with different levels of quality. They may use different kinds of software components to process the data. Also, the systems developed by the different parties are not necessarily developed for interoperability. These factors add to the complexity of the system.

Furthermore, there are diverse needs of the traffic data, coming from different kinds of end-users. These end-users include commuters, highway patrols, public service vehicles like fire-engines and ambulances, departments of transportation, urban planners, commercial vehicle operators, etc. These users not only pose large numbers of simultaneous analysis requests, but also require analyses of significantly different natures. For example, dynamic traffic management as performed by the department of transportation requires real time processing of detailed traffic data across the urban area. However, the analysis performed by urban planners requires high level aggregation of the data and the updating of historical databases. This further increases the complexity of the system.

In this paper, we demonstrate the use of IBM InfoSphere Streams [1], a scalable stream processing platform, for tackling these challenges. Stream processing applications in InfoSphere Streams take the form of graphs of modular, reusable software components (called operators) interconnected by data streams. Each operator takes data of certain content and format and perform various analysis. These applications can be deployed on a distributed runtime infrastructure to allow scalability via pipelining and parallelization. InfoSphere Streams provides various services to manage the infrastructure and the applications deployed on it so as to support high-throughput, low-latency stream processing.

One of the key features of InfoSphere Streams is it's component-based programming model. This allows composing and reconfiguring individual components to create different applications that answer different kinds of queries or perform different kinds of analysis. In addition, it enables the creation and deployment of new applications without disrupting existing ones. This facilitates the growth and incorporation of new technologies. Furthermore, InfoSphere Streams allows the new applications to reuse intermediate derived streams produced by existing applications in order to minimize duplicate or redundant processing of data. These different features help in tackling the challenges of dealing with diverse data sources and diverse end-user needs.

We describe a case study demonstrating the use of InfoSphere Streams for Intelligent Transportation Systems. This case study consists of a set of stream processing applications that process real-time GPS data, generate different kinds of real-time traffic statistics, and perform customized analyses in response to user queries. Examples of customized analyses include continuously updated speed and traffic flow measurements for all the different streets in a city, traffic volume measurements by region, estimates of travel times between different points of the city, stochastic shortest-path routes based on current traffic conditions and real time prediction of travel times and traffic conditions (prediction horizons may vary from few minutes to an hour, depending on the application and the conditions), etc. Our performance experiments illustrate the scalability of the system. For instance, our system can process over 120000 incoming GPS points per second, combine it with a map containing over 600,000 links, continuously generate different kinds of traffic statistics and answer user queries. This system runs on a cluster of four x86 blade servers.

In the rest of the paper, we describe the InfoSphere Streams platform and our case study in the Intelligent Transportation Systems space. Section 2 introduces InfoSphere Streams and its features. Section 3 describes the ITS case study and Section 4 goes into details of one of the critical components in our system; viz. the stream visualization framework. Section 5 provides some performance numbers for our system.We then describe related work in Section 7 and finally conclude.

2. INFOSPHERE STREAMS

InfoSphere Streams [7] (or Streams) is a new IBM product that supports high performance stream processing. It has been used in a variety of sense-and-respond application domains, from environmental monitoring to algorithmic trading. It offers both language and runtime support for improving the performance of sense-and-respond applications in processing data from high rate streams.

InfoSphere Streams (or just Streams) supports structured as well as unstructured data stream processing and can be scaled to a large number of compute nodes. The 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 analytics and resides in execution containers called Processing Elements (PEs), which are distributed over the compute nodes. The compute nodes are organized as a shared-nothing cluster of workstations or as a large supercomputer (e.g., Blue Gene). 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 [5] (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. Concretely, SPADE offers:

1. An intermediate language for flexible composition of parallel and distributed data flow graphs. This language sits in-between higher level programming tools and languages such as the Streams IDE or Stream SQL and the lower level Streams programming APIs.

2. A toolkit of type-generic built-in stream processing operators. SPADE supports all basic stream-relational operators with rich windowing semantics.

3. The ability to extend the set of built-in operators with user-defined ones, programmable in either C++ or Java.

4. A broad range of stream adapters. These adapters are used to ingest data from outside sources and publish data to outside destinations, such as network sockets,

[1]InfoSphere Streams is the product name of the System S stream processing research platform

relational and XML databases, filesystems, as well as proprietary platforms such as IBM Websphere Front Office, and IBM DB2 Data Stream Engine, etc.

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. In other words, when going from a SPADE program to the actual deployable distributed program, the logical streams may be implemented as simple function calls (for fused operators) to pointer exchanges (across PEs in the same computational node) to network communication (for PEs sitting on different computational nodes).This code generation approach is extremely powerful because through simple recompilation one can go from a fully fused application to a fully distributed one, adapting to different ratios of processing to I/O provided by different computational architectures (e.g., blade centers versus BlueGene).

SPADE supports a modular, component-based programming model, which allows reuse, extensibility and rapid prototyping. Apart from the built-in operators and stream adapters that it provides, it also allows developers to create new operators in either C++ or Java. It also allows developing applications that offer high-availability through replicated processing and operator checkpointing.

InfoSphere Streams includes a scheduler component that decides how best to partition a data-flow graph across a distributed set of physical nodes [15]. The scheduler uses the the computational profiles of the operators, the loads on the nodes and the priority of the application in making its scheduling decisions.

2.1 Language Support via different kinds of operators

SPADE was conceived around the idea of providing toolkits of operators. These operators can be used to implement any relational query as well as windowing extensions commonly required by streaming applications. Additional convenience stream manipulation operators are also included, providing workload partitioning capabilities and generation of window boundaries, among other functionalities. We commonly refer to these operators as built-in operators (as opposed to user defined operators). The operators currently supported in the stream-relational toolkit are:

1. **Source** A Source operator is used for creating a stream of data flowing 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 System S. 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. In these manipulations, the Functor 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. This operator supports a large number of grouping mechanism and summarization functions.

5. **Join**: A Join operation is used for correlating two streams. System S 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. The ordering algorithm can be tweaked in several ways.

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 punctuations 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 relational algebra toolkit, the language was designed to support extensions. Currently 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 Builtin Operators (UBOPs), which are fully templatized operators. UBOPs' names and associated syntax are recognized by the SPADE language parser. 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 like, for example, signal processing, scientific computing, and stream data mining.

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.

2.2 Compiler and Runtime Support

Given an application specification in SPADE's intermediate language, the SPADE compiler generates optimized code that will run on InfoSphere Streams. SPADE's effective code generation and optimization framework enables it to fully exploit the performance and scalability of InfoSphere Streams. The reliance on code generation provides the means for the creation of highly optimized platform- and application-specific code. In contrast to traditional database query compilers, the SPADE compiler outputs code that is tailored to the application at hand as well as system-specific aspects such as: the underlying network topology, the distributed processing topology for the application (i.e., where each piece will run), and the computational environment. In most cases, applications created with SPADE are long-running queries. Hence the long running times amortize the build costs. Nevertheless, the SPADE compiler has numerous features to support incremental builds, reducing the build costs as well.

3. THE INTELLIGENT TRANSPORTATION SYSTEMS PILOT

In the last few years, there has been an explosion in the number and variety of data sources that are available for monitoring the transportation system. These include GPS devices on different kinds of vehicles, GPS-enabled cellphones, video-cameras monitoring roads, loop-detectors, toll booths, congestion pricing portals, etc. As a result, transport agencies and organizations have moved from a state of no data, to a state of data overload, and are ill prepared to deal with the challenges of the new environment and take full advantage of the opportunities.

The goal of the Intelligent Transportation Systems Pilot was to show city transport agencies the possibilities of processing the large amounts of streaming data in real time. In this pilot, we focused on the city of Stockholm. We obtained historical vehicle GPS data from Trafik Stockholm, as well as information on the road network. We identified a set of useful, real-time services for both individual drivers and for the transport agency, and developed stream processing applications on Infosphere Streams to support these services. These services included monitoring the current traffic conditions in different parts of the city (on a link or over a region), comparing these conditions with historical statistics, estimating travel times between different points in the city, and computing stochastic shortest paths between different points in the city. For demonstration purposes, we replayed and processed the historical data. In the future, though, we will be able to directly access the real-time feeds from the data sources.

3.1 Input Data

We obtained historical GPS data traces from Trafik Stockholm [13] for the year of 2008. This data included traces from about 1500 taxis and 400 trucks that plied the streets of Stockholm. In total, there was about 170 million GPS probe points for the whole year. Each taxi produces a GPS probe reading once every 60 seconds that includes taxi identification and location information. Also, for privacy reasons, taxis produce the readings only when they are not carrying any passengers. Trucks use more recent and more accurate GPS devices, that produce readings once every 30 seconds and include identification location, speed and heading information. The peak data rate for the whole city was over 1000 GPS readings per minute. In our pilot,though, we often replayed the data at several times the actual rate, just to demonstrate the scalability of the system.

Another information we obtained was information about the city road network. The road network contained about 628,095 directed links spread over a 80km x 80km area around Stockholm. Each link represented a uni-directional road segment, hence two-ways roads are represented with two parallel links in opposite directions.

3.2 Overall Application Description

Having large numbers of vehicles sending real-time GPS data for the city allows us to create a picture of the traffic condition in time and space [9]. We now describe the stream processing applications that allow us to come up with the traffic information, as well as provide various value-added services on top of the basic information.

The application processes the data in three distinct phases. The first phase consists of real-time processing of the data. This includes obtaining, cleaning, de-noising, and matching the GPS data to the underlying road network or specified regions. In the second phase, the data is aggregated to produce traffic statistics per link and per time interval. The traffic statistics are in the form of medians and quartiles of vehicle speeds and vehicle counts on the link or region for the time interval. In the final phase, we make use of the statistics to compute different kinds of derived information such as the estimated travel times and shortest paths between different parts of the city. The final outputs can be sent to the user on different kinds of visualization platforms such as Google Earth, or may be stored in a database for additional offline analysis. A high level flow graph of the application is depicted in Figure 1.

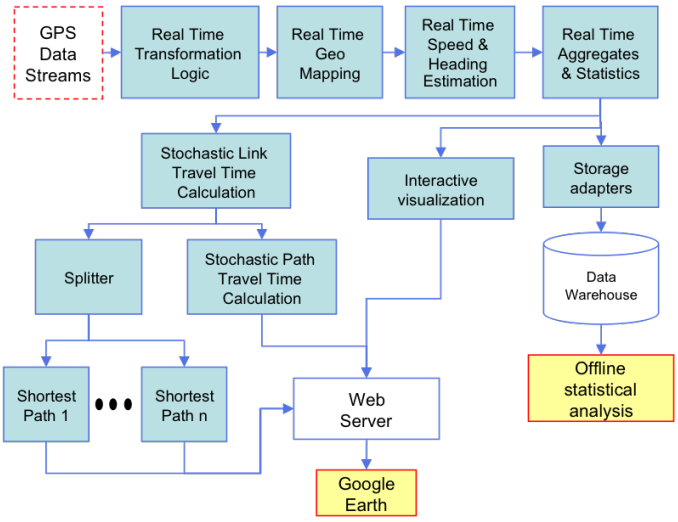

Figure 1: Application Flowgraph

3.3 Real-time GPS data processing

The GPS data is first received by the source operator and converted to tuples with time stamps, GPS device identification, latitude, longitude, heading (i.e. direction), and instantaneous speed attributes. The heading and speed at-

tributes are null when not available. Following the source, we have a preprocessing operator that extracts the time of day and day of week, in tandem with an operator that filters entries whose time stamps, latitudes or longitudes are out of range.

For the GPS data to be useful for measuring traffic conditions on roads, it must be matched with the underlying road network. An important issue with GPS data is its accuracy. There are two kinds of errors that we encounter with GPS data: measurement error caused by the limited GPS accuracy, and the sampling error caused by the sampling rate. The measurement error of GPS devices is typically in the order of a few meters. While this may seem small, this causes a problem when the GPS reading is located close to different links, such as at an intersection. The sampling error causes uncertainty in the vehicle's movement. To illustrate the impact of the sampling rate on the imprecision of the interpolated trajectory data, consider sampling the position of a vehicle every 30 seconds, which is the typical sampling rate for the trucks in our data. At a speed of 100km/h, the traveled distance between position samples is over 833 meters. Since we do not know the positions in-between two consecutive position samples, the best we can do is to limit the possibilities of where the moving vehicle could have traveled. In order to correct the effect of both kinds of errors the application processes the GPS data in two steps, called geo-matching and geo-tracking.

Geo-matching (also called Map-matching) consists of finding a set of links that are within a distance d meters from the GPS probe. The parameter d denotes the maximum expected measurement error, and is set to $d = 10$ meters. For the case where the GPS heading is available, the search is further restricted to links that are within 45 degrees of the heading. The algorithm uses the Harversine formula [12] to calculate geodetic distances in the WGS84 coordinate systems. Our implementation of this formula is accurate to within one meter, but is also computationally intensive and calculating the distance to all the links in order to determine the nearest one is not a tractable solution. Instead the algorithm employs a divide and conquer approach, illustrated in Figure 2, in which it divides the map into a grid of equal areas, and focus its search to the nearest links within the area that contains the GPS coordinates. Included in the search are links that are up to a distance d from the area's perimeter to account for eventual errors in the GPS measurements. The grid is sized to cap the number of links per area, and the list of links contained in each area can be computed offline. As a result it takes a constant time complexity for the method to find the nearest links to a given location, independently of the size of the road network.

For our particular purpose, a single geo-matching operator is sufficient to handle the Stockholm road network size. But if needed, the algorithm can easily be distributed into a hierarchy of operators, whereby the upstream operator distributes the GPS data by means of the grid-based method into downstream operators, each of which applies the geo-matching algorithm on a sub-region of the road-network. Since the operators can be deployed on separate machines, such decomposition makes it possible to handle arbitrarily large road networks, and take advantage of parallel processing to handle higher data rates. In other work [4], we describe how we can scale the geo-matching operation, so as to be able to match GPS data arriving at a rate of 1 million points per second onto a map with 1 billion links.

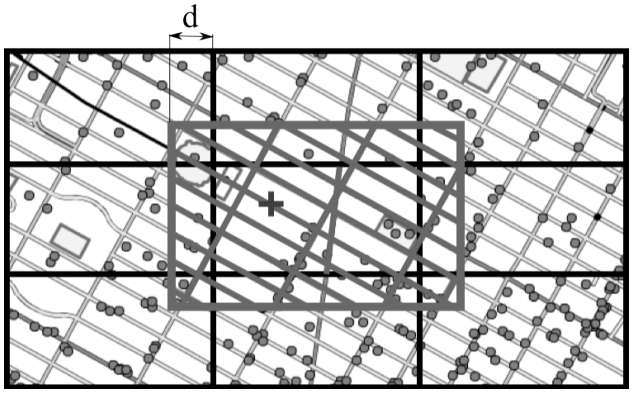

Figure 2: Illustration of a grid-based geo-matching algorithm. The algorithm first identifies the area that contains the GPS coordinates (+) out of a $W \times H$ grid, it then searches for the nearest links up to a distance d (to take into account GPS measurement errors) around the area.

After matching each GPS reading to the map, the next operation consists of estimating the trajectory of each vehicle, and removing the bad readings. This is achieved by means of a geo-tracking operator that looks for a sequence of GPS probes and compares all possible path combinations connecting the matching links. Our current implementation considers the last two GPS probe readings and finds the possible paths connecting them. In order to minimize sampling errors, the operator only considers probes that are less than 75 seconds apart, which corresponds to the maximum average sampling interval of the GPS devices plus a 25% margin to account for possible fluctuations in the transmission latency. Given the paths, the operator derives the average speed from the path lengths and time stamp information, and selects the path that results in the most realistic speed. The operation is illustrated in Figure 3.

The SPADE code snippet below shows the implementation of this operation. It uses a User-Defined Built-in Operator (or UBOP) called `GpsTrack`, which takes as input a stream of map-matched GPS points, and generates the `heading`, `estspeed` and other attributes. The UBOP is parameterized by the `maxinterval`, the map information, etc. There is also a `node` annotation, which instructs the SPADE compiler to place the operator in the first node in a defined pool of nodes.

```
stream geoTrackStream(schemaFor(geoRichStream),
  avgheading : Long,
  estspeed : Float,
  confidence : Float)
    := GpsTrack(geoRichStream)[
        maxinterval : 75 ;
        map         : "/data/stockholm.lnk"
    ]{} -> node(mainpool,1)
```

Another use of the GPS data is to match it to regions instead of links. Such information can be used, for instance, to count the number of vehicles in a parking area or other enclosures that cannot be represented with line segments. The method is reminiscent of the geo-matching algorithm, with the exception that matching is done by detecting if a

Figure 3: Illustration of the geo tracking process. The same vehicle appears at different points in time joined by the estimated trajectory.

point is inside a polygon delineating the region instead of computing distances to the links.

3.4 Traffic Statistics Generation

Once the GPS data is matched with a road network link, we want to use this information to generate statistics such as average vehicle speeds and unique vehicle counts per link at various times of the day. For this purpose, the application uses an Aggregator operator to slice the data every five minutes. In this time frame, this operator computes, for every link, the average estimated speeds of vehicles and also, the number of vehicles (each vehicle being counted once during the time frame). This operation is easily implemented in InfoSphere Streams using built-in Aggregators with per-link tumbling windows which are shifted (i.e., tumbled) each time the time stamp attribute enters a new five minute interval since midnight. In order to work properly, this approach requires that the time-stamp attribute of the tuples is accessed in increasing order. This requirement is satisfied with a Sort operator, which holds onto the tuples for a brief period of time and reorders them within that time window before sending them to the aggregator.

The code snippet below shows an implementation of the aggregation operator. The window slides whenever the `absTimeIndex` field is incremented (i.e., when the difference of the current `absTimeIndex` value compared with the value in the previous tuple is greater than 0). The `absTimeIndex` is incremented by 1 every 5 minutes. The output of the operator is a stream called `roadAttributeStream` with the schema as shown, containing 8 fields. The input stream is a `sortedGeoTrackStream` that contains the mapping of each GPS point mapped to a link, and the instantaneous estimated speed of the vehicle. The Aggregate operator groups by `timeIndex` and `shapeid` (which is the link id). For each unique pair of `timeIndex` and `shapeid`, it generates the `avgSpeed`, `minSpeed` and `maxSpeed` over a 5 minute period. Finally, there is a `partition` annotation, which tells the SPADE compiler to fuse this operator with any other operator that has the same annotation value.

```
stream roadAttributeStream(
    timestamp       : Long,
    shapeid         : Long,
    timeIndex       : Long,
    weekDay         : Integer,
    monthId         : Integer,
    avgSpeed        : Float,
    minSpeed        : Float,
    maxSpeed        : Float,
  )
   := Aggregate(sortedGeoTrackStream
            <attrib(absTimeIndex, 01), perGroup>)
            [ timeIndex . shapeid ]
  {
       timestamp      := Any(timestamp),
       shapeid        := Any(shapeid),
       timeIndex      := Any(timeIndex),
       weekDay        := Any(weekDay),
       monthId        := Any(monthId),
       avgSpeed       := Avg(estspeed),
       minSpeed       := Min(estspeed),
       maxSpeed       := Max(estspeed),
       estRoute       := Any(estroute)
  } -> partition["GpsAggregation"]
```

The speed and the vehicle count estimates are then fed to another type of operator, called Inter-Quartiles Range or IQR operator. The role of this operator is to gather primary first-order statistics of the speeds and vehicle counts. Every five minutes during a day for every link, the IQR operator computes the speed and vehicle count summaries (smallest non-outlier, lower, median, upper quartiles and largest non-outlier) from historical observations collected over a several months period. Due to obvious differences in the traffic patterns between working days and non-working days we keep separate set of statistics for week days and week-ends. This is easily implemented in InfoSphere Streams with the help of a generic Split operator, which directs the data flow to one of several IQR operators depending on an expression of the day of week tuple attribute. If desired, this expression can be improved to include holidays or other scheduled events that are known to impact the traffic conditions.

The IQR operator, in combination with the output of the per-link and time interval aggregation, allows us to generate views for each link in which we can compare the current road conditions on the link with statistical measurements collected over longer time period on the same link. This allows us to detect, for instance, whether the road conditions are normal for that time of the day or not. An example of such a view is illustrated in Figure 4.

3.5 Additional User-Specific Computations - Travel Times and Shortest Paths

The traffic statistics generated are used to compute different kinds of derived information such as the estimated travel times and traffic-dependent shortest paths between different parts of the city. These computations are performed continuously for each individual end-user query that contains information on the source and destination points. In the next section, we describe how the queries are actually obtained from the end-user and used to direct these computations.

In order to compute the travel times and the shortest

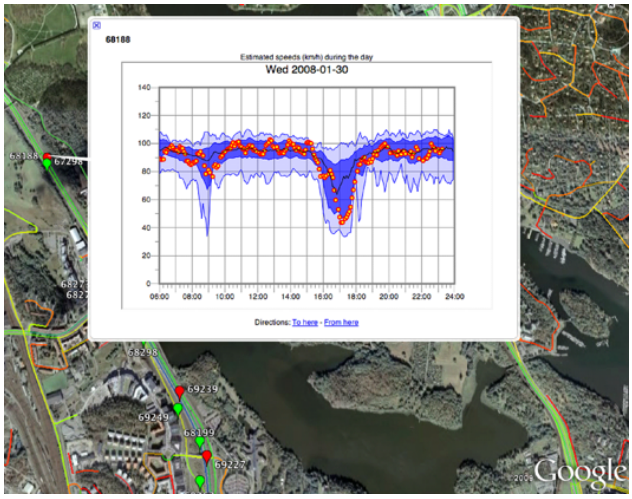

Figure 4: Illustration of the traffic statistics generation that shows a time series of the average speed (dotted line) measured at five minutes intervals during the day on a given link. The average speed is compared to historical measurement (area plot).

paths, we first compute the stochastic travel times for each link. This involves computing a time-varying probability distribution of the time taken to traverse each link. This distribution information for each link can then be used to compute the distribution information for a whole path using a convolution operation.

The stochastic travel times for each link are also used to calculate the shortest path (in terms of travel time) between any two points. This calculation uses the A algorithm. Since the shortest path calculation is computationally expensive, we split the computation across different operators that can be placed on different hosts. SPADE offers declarative ways of doing the splitting that makes it easy for developers to parallelize various kinds of computations. For example, the code snippet below shows the use of for loops to first split the stream of shortest path requests by `destID` and then calculate shortest paths using `sp_num` different operators:

```
# Split
for_begin @part 1 to sp_num
  stream gpsSPQueryStream_P@part
              (schemaFor(gpsJoinStream))
for_end
      := Split(gpsSPQueryStream)
       [hashCode(mod(destId,sp_numl) )]{}

# Shortest path operators. Each operator gets a
# part of gpsSPQueryStream
for_begin @part 1 to sp_num
   stream shortestPathStream_P@part
    (gpsId : Long, fromArc : Long, toArc : Long ,
     path : LongList, starttime : Long,
     endtime : Long, traveltime : Long )
        := ShortestPath(gpsSPQueryStream_P@part;
            roadUpdateStream)[]{}
for_end
```

4. END-USER INTERACTIONS WITH STREAM PROCESSING APPLICATIONS

End-users can interact with the ITS application in different ways. They can visualize traffic statistics for the whole city of Stockholm on Google Earth. They can also submit specific queries, such as to view the average speed on different links, the estimated travel time between two different points, and the shortest path between two different points that takes into account the current traffic conditions.

In this pilot, we support a web-based visualization and interaction mechanism. For this, we make use of the web browser integrated within Google Earth. The overall interaction framework includes a Visualization Server, which is a standalone web-server using the Java Servlet framework, running outside of InfoSphere Streams. In addition to the HTTP port, this Visualization Server listens to a data port for incoming connections from various stream processing application. The Visualization Server maintains a buffer for each user-requested output of the application, e.g. the traffic statistics for a single link. Each buffer stores a time-series of some kind (e.g. average vehicle speeds on a link). Each buffer is also associated with a query ID. So, a browser can query the time series from any buffer, and it is possible to share the same server to show subsequences of one or more streams in one or more browsers.

The visualization component comprises several types of visualizers, which can show data in different ways. All these visualizers are AJAX-based, using XMLHttpRequest to periodically get data from the same data servlet, which responds with the data from one of the buffers. Users have to specify the data identifier in URLs to select the buffer they want to see. In order to save bandwidth, we use CSV-formatted text for these XMLHttpRequest connections. On the browser side, this format can be handled by a very short piece of Javascript code. The CSV format consumes much less bandwidth than other formats such as JSON or XML.

In visualizer URLs, users can specify the window size, the columns they want to see (so that other columns can be removed by the server to save bandwidth), and refresh rate of data (i.e., the frequency of XMLHttpRequest), among others. Almost all configurable parameters can be specified in a URL, as long as there are no security issues, and most of them have default values; the requirements for server-side customization are then minimized.

Queries from the end-users are handled via a servlet in the visualization component that allows web clients to send a tuple to interact with InfoSphere Streams. The tuple is sent to a remote TCP or UDP port that the stream processing application listens to. This allows us to establish a feed back control mechanism that allows the web-client to specify the desired output to the application. This mechanism is illustrated in Figure 5 and works as follows: the client queries a time series from the visualization server. In the query the client describes the desired output in a form that the data streams processing application understands. This can be the unique numerical identifier of given road network link. The client also specifies the identifier of a circular buffer in which the results will be stored in the visualization server. This URL is decoded and both identifiers are included in a tuple that is sent to the data streams processing application. The application uses the description of the output to control the

flow of data and ensure that at the sink operator connecting to the visualization component, only the output matching this description is sent. As shown in Figure 5, this can be done with the help of a Join operator that selects only tuples that match the output description (e.g. matching link identifiers). If historical data is required, such as time series since the start of the current day, the window Join operator can be used instead. The Join operator inserts the specified buffer identifier in the tuple, so that the visualization components stores the time series in the appropriate buffer, from where the client can later retrieve it.

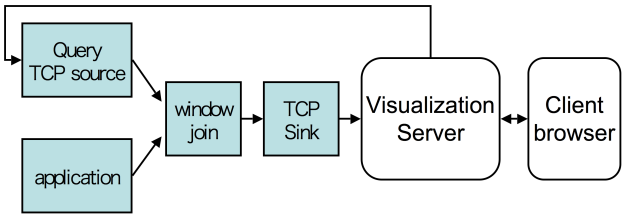

Figure 5: Architecture of visualization component.

This query servlet can be either used through AJAX or with a page, or frame, or iframe URL. For the latter case, the query servlet automatically redirect the returned page to a visualization URL. After the results from InfoSphere Streams arrive, they are sent to the browser in the same page or frame for visualization.

5. PERFORMANCE EVALUATION

For the performance measurements we deployed the ITS application on a cluster of 64bit Intel Xeon® quad-core machines running at 3Ghz with 16G bytes of memory each. The tested application consists of a total of 46 operators, 10 of which are UBOPs (user-defined built-in) operator, and others are built-in operators. Figure 6 shows a screenshot of the deployed application. In the figure, boxes represent operators, interconnections represent data streams, and the resulting topology represents the entire application flow-graph. It also shows how the operators are fused together to form a PE (Processing Element), represented as large dark background rectangles that contain one or more individual operators.

Figure 7 shows how those PEs are distributed across various hosts (nodes). In this example, the distribution is based on instructions in the SPADE program assigning different operators to various nodes, but it can also be done automatically by the InfoSphere Streams scheduler.

Finally, Figure 8 shows the performance of the ITS application with changing hardware infrastructure. The metric is the throughput, measured as the number of tuples per second processed by the source operators as we increase the total number of hosts used for the entire application. Here each tuple corresponds to one GPS point. As we increase the number of hosts, the InfoSphere Streams runtime can distribute the different operators across different hosts. For example, Figure 7 shows how the runtime distributes the operators in the ITS application on 4 different hosts.

Figure 8 shows that the throughput scales well as we increase the number of nodes. The data-pipelining and parallelization features of InfoSphere Streams play a key role in achieving this scaling property. In this particular instantiation of the application, we used up to 4 different source operators to read the GPS data from files in parallel. When run on one single node, the system performance is the worst as all the software components share that single resource, even though that single resources is made of various cores. Increasing the number of nodes addresses the bottleneck due to resource limitation. Hence, we see a steady increase in performance as the number of nodes increases. The throughput achieved by our application is more than enough for our current needs. We are also confident that the throughput can be increased even more with additional tuning of the application, including tuning the placement of the different operators on the available nodes, tuning the structure of the application to take further advantage of the parallelism offered by the system, and tuning the operators themselves. Note that InfoSphere Streams can support a distributed cluster containing over a hundred nodes, which offers plenty of scope for improving the performance of the application. In other work [4], we describe a different example where we used up to 10 nodes to map-match GPS data and in that application, we were able to achieve a throughput of 1 million GPS points per second, mapped onto a road network containing 1 billion links.

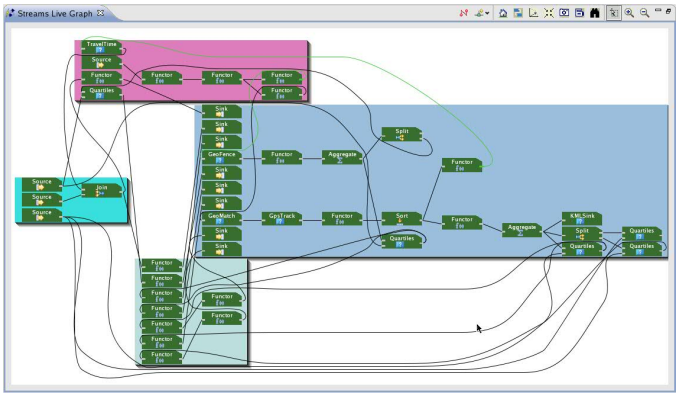

Figure 7: Application flow graph showing PE within hosts.

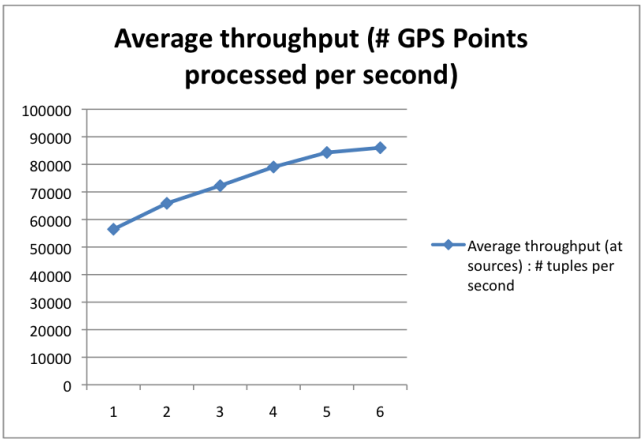

Figure 8: Performance of the ITS Application with increasing number of nodes. The y-axis shows the throughput and the x-axis shows the total number of nodes used for the entire application.

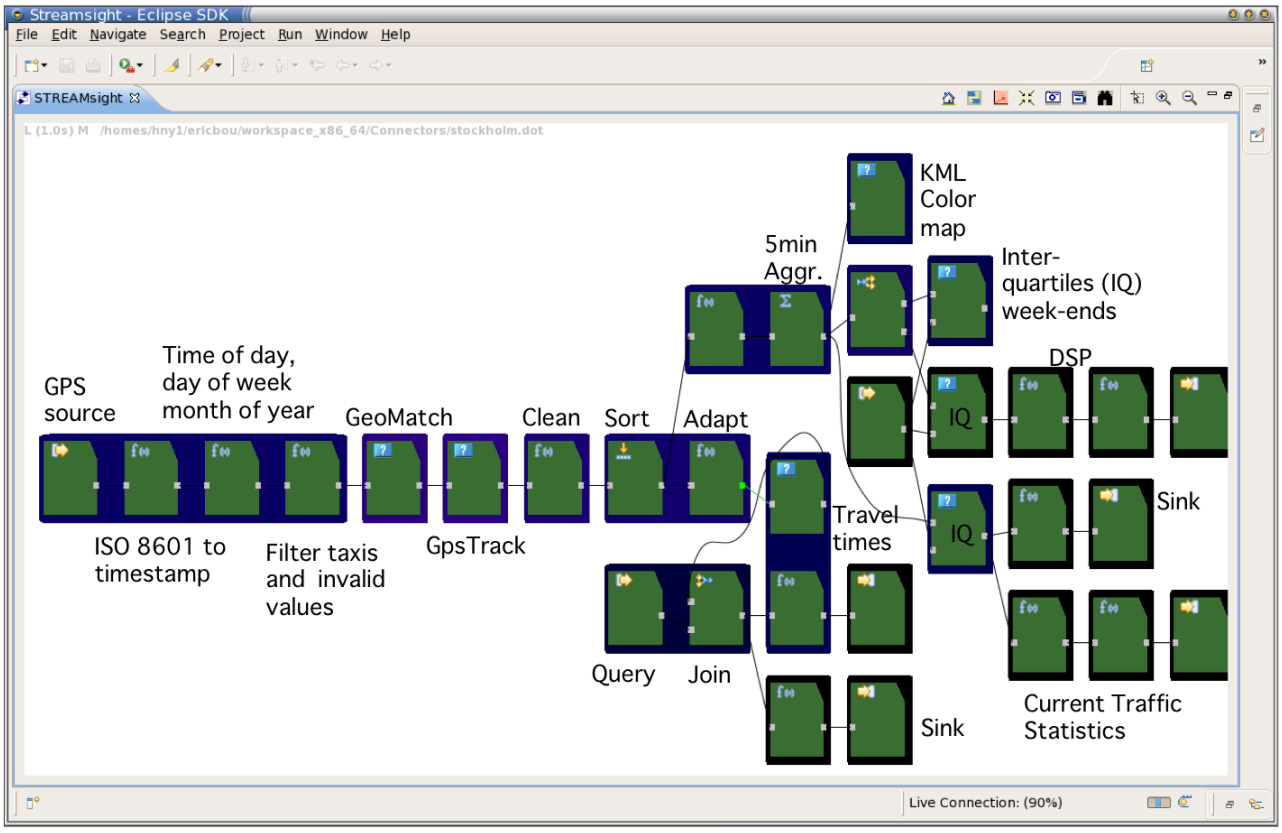

Figure 6: Application flow graph of operators, with labels describing the operations performed by different groups of operators. The figure also shows the fusion of some sets of operators into Processing Elements (PEs). PEs are shown by the blue background boxes containing one or more individual operators.

6. EXPERIENCES

We have already seen how we can scale the ITS application using various InfoSphere Streams features, such as distributing the computation over different nodes and fusing operators into PEs to reduce communication cost. In this section, we describe some of our experiences in using the SPADE language and compiler in developing the application. We describe how various features of SPADE and InfoSphere Streams facilitate the development of stream processing applications. As an example, our team, consisting of 5 developers, was able to develop the ITS application within a month. These features are particularly important when we develop large ITS applications that tap onto diverse sources of data and meet diverse user needs.

6.1 Component-based model

InfoSphere Streams allows applications to be developed as a graph of reusable components (or operators). This makes it easy to develop new applications by reusing existing components. It also encourages developers to develop their component in as general a manner as possible. InfoSphere Streams supports "User-defined Built-in Operators" (or UBOPs), which are operators that can operate on input data of different formats and be used in different contexts and implemented in either C++ or Java.

In our prototype, for example, we were able to build a number of reusable map-matching operators that used different algorithms. We were easily able to plug these operators into the overall ITS application. More generally, we have a toolkit of various geo-spatial operators like map-matching, vehicle tracking, road traffic statistics generation, shortest-path route generation, stochastic travel time computation, etc. that we have been able to reuse in different applications.

6.2 Modular application development

InfoSphere Streams supports two different models of stream interconnections between operators. The application data flow-graph in Spade can specify the interconnections explicitly, or it can use a publish-subscribe mechanism supported by Streams for the interconnections to be established. This publish- subscribe mechanism can be based on either names of streams or based on the properties of the streams.

In addition, InfoSphere Streams allows separate applications to communicate with one another via streams. In this mechanism, one application can export streams, and another application can then import these streams, either explicitly by name, or implicitly by specifying properties of the desired input streams. These features of Streams are extremely useful in decomposing potentially large and complex applications into a number of smaller pieces that can communicate with one another. For example, we actually organized the ITS application as a set of multiple independent applications. One application does the basic GPS data processing. There are then one or more statistics computing applica-

tions that take the processed GPS data streams as input. There are also many user-specific applications that take the streams of statistics as input and perform different computations, such as calculating the shortest route and travel time. These different applications can be composed together dynamically using exported and imported streams.

The component based model and modular application development features of InfoSphere Streams allows a large group of developers to collaborate in developing the applications. Once the types of the input / output streams of components and the imported and exported streams of applications are agreed upon, the developers can program independently and later integrate fairly easily. Also, stream processing by its very nature, limits interactions between components to the streams interconnecting them. This means that there is no need to maintain any global state, which often complicates software integration in many other kinds of systems.

One outcome of these two features is that SPADE allows having developers of two different roles work together : operator developers and application developers. The operator developers write operators to perform a specific task in C++ or Java, and typically supply samples and test-cases showing how their operators can be invoked. The application developers compose these developers' operators along with SPADE built-in operators to come up with an application that meets some end-user's need.

6.3 Declarative Application Configuration

The SPADE language includes constructs to specify how the application is to be deployed on the runtime.

1. SPADE allows specifying the degree of parallelization of an operation.This is achieved by a SPADE pre-processor which allows specifying for-loops in the SPADE program. The pre-processor unrolls the for-loop and allows creates an expanded SPADE graph that then gets compiled. Also, SPADE allows grouping streams into "bundles" so that they can all be referred as a single entity. This is useful, e.g. in specifying all the inputs to a barrier operator as a single bundle.

2. SPADE allows specifying operator fusion constraints. For example, it allows specifying that a certain set of operators must be fused into the same partition, or must be located in different partitions. Note that this is optional; the SPADE compiler can automatically compute the best operator fusion sets by using profiles of each operator's computational characteristics [8].

3. SPADE allows specifying the assignment of operators to physical nodes. Again this is optional; InfoSphere Streams includes a scheduler component that can make such placement decisions based on the load on each physical node [15].

Thus, we can easily change the application deployment configuration by simply changing a few lines of the SPADE program, and then recompiling and redeploying it. This facilitates rapid prototyping, which is immensely useful in fine-tuning the application to meet various performance objectives. This proved particularly useful when we tuned our map-matching, source and aggregation components to improve their performance.

6.4 Rich set of built-in operators

SPADE supports a very rich set of built-in operators, including different kinds of filters, joins, aggregations, etc. It also comes with a number of built-in operators for accessing external sources and sending data to external sinks. It also supports different ways of creating and updating windows that can be used in some of the built-in operators. The different variations include tumbling and sliding windows, count-based and time-based windows, windows based on differences in attribute values, punctuation based (where punctuations can be inserted by preceding operators), etc. In fact, most of the operators we use in the ITS application were built-in operators, which significantly saved development time.

7. RELATED WORK

There has been plenty of work in the areas of acquiring GPS and other traffic data, mapping it to road-networks, using the data for traffic prediction, and determining shortest paths. Most works however tackle these problems independently, while we have developed an end-to-end application that takes raw GPS data to provide useful real-time services to end-users. Also, many of these works focus more on accuracy rather than throughput. For example, most publications in map matching focus on the accuracy of map-matching, rather than the on improving the throughput of map-matching as the size of the underlying map increases and as the rate of incoming GPS data increases.

A number of approaches have been proposed in the area of matching GPS positions of a moving vehicle to a map. Brakatsoulas et al [2] proposed incremental and global algorithms that consider the trajectory of the moving vehicle and try to minimize the Frechet distance between the trajectory and the sequence of mapped links. Greenfeld [6] introduced a map-matching strategy based on distance and orientation that does not assert any further knowledge about the movement besides the position samples. Civilis et al. [3] in their work in location-based services introduce a map-matching algorithm that is based on edge distance and direction similar to [6]. Yin and Wolfson [16] propose an algorithm based on a weighted graph representation of the road network in which the weights of each edge represent the distance of the edge to the trajectory. Marchal et al [11] describe an approach that maintains a set of candidate paths as GPS data are fed in and to update, constantly, their matching scores. Lou et al [10] propose a global map-matching algorithm called ST-Matching for GPS trajectories with low sampling rate. Our approach is most similar to the incremental algorithm proposed by Brakatsoulas, where we use shortest-path trajectories between successive GPS points to estimate the path taken by the vehicle.

Various approaches have also been proposed for calculating travel times and shortest paths. Our key contribution in this paper is not in the individual algorithms (like map matching or shortest paths), but in demonstrating how these different algorithms can be deployed on a distributed stream processing infrastructure for purposes of scalability. A number of map-matching and route planning systems have been deployed in the real world (for example, the systems behind services like Google Maps and Google Earth, various navigation guidance systems, etc.). However, we have not been able to get a hold of the descriptions of these systems.

8. CONCLUSION

In this paper, we described the use of IBM InfoSphere Streams, a component-based distributed stream processing platform, for tackling the challenges of scalability, extensibility and user interaction in the domain of Intelligent Transportation Services. We described a prototype system that generates dynamic, multi-faceted views of transportation information for the city of Stockholm, using real vehicle GPS and road-network data. We also described some of our experiences in using InfoSphere Streams for this application.

9. ACKNOWLEDGMENTS

We would like to thank IngaMaj Eriksson from the Swedish Road Administration and Tomas Julner from Trafik Stockholm for their support, feedback, and provision of the data for this study and Erling Weibust of IBM for facilitating the IBM/KTH collaboration.

10. REFERENCES

[1] C. Antoniou, R. Balakrishna, and H. Koutsopoulos. Emerging data collection technologies and their impact on traffic management applications. *ASCE Journal of Transportation Engineering*, 2009. To be submitted.

[2] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk. On map-matching vehicle tracking data. In *VLDB*, pages 853–864, 2005.

[3] A. Civilis, C. S. Jensen, J. Nenortaite, and S. Pakalnis. Efficient tracking of moving objects with precision guarantees. In *Mobiquitous*, 2004.

[4] E.Bouillet and A.Ranganathan. Scalable, Real-time Map-Matching using IBM's System S. In *Proceedings of Mobile Data Management Conference (MDM 2010)*, 2010.

[5] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: the System S declarative stream processing engine. In *SIGMOD 2008*, pages 1123–1134, 2008.

[6] J. Greenfield. Matching GPS observations to locations on a digital map. In *81th Annual Meeting of the Transportation Research Board*, 2002.

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

[8] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. L. Wolf, K.-L. Wu, H. Andrade, and B. Gedik. COLA: Optimizing stream processing applications via graph partitioning. In *Middleware*, pages 308–327, 2009.

[9] R. Kuehne, R.-P. Schaefer, J. Mikat, K. Thiessenhusen, U. Boettger, and S. Lorkowski. New approaches for traffic management in metropolitan areas. In *Proceedings of IFAC CTS Symposium*, 2003.

[10] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang. Map-matching for low-sampling-rate GPS trajectories. In *GIS '09: Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2009.

[11] F. Marchal, J. Hackney, and K. Axhausen. Efficient map matching of large global positioning system data sets: tests on speed-monitoring experiment in Zurich. *Transportation Research Record*, 1935:93–100, 2005.

[12] R. W. Sinnott. Virtues of the haversine. *Sky and Telescope*, 68(2):159, 1984.

[13] Trafik Stockholm. http://www.trafikstockholm.com.

[14] US Department of Transportation. Intelligent transport services benefits, costs and lessons learned databases, http://www.itscosts.its.dot.gov/its/benecost.nsf, 2005.

[15] J. Wolf et al. SODA: an optimizing scheduler for large-scale stream-based distributed computer systems. In *Middleware*, pages 306–325, 2008.

[16] H. Yin and O. Wolfson. A weight-based map matching method in moving objects databases. In *16'th SSDBM conference*, 2004.