Understanding the Semantics of Sensor Data*

Murali Mani
WPI, Computer Science Dept
mmani@cs.wpi.edu

Abstract

Our system architecture to manage sensor data is described. Our data mining applications require past history of the sensor data. Therefore, unlike most present systems that focus on streaming data, and cache a small window of historic data, we store the entire historic data. Several interesting problems arise in these scenarios. We study two of them: (a) Given that a sensor can send data corresponding to its current configuration at any particular instant, how do we define the data that should be stored in the database? (b) Sensors try to minimize the amount of data transmitted. Also there could be data loss in the network. So the data stored will have lots of "holes". In this case, how can an application make sense of the stored data? In this paper, we describe our approach to solve these problems that enables an application to recreate the environment that generated the data as precisely as possible.

1 Introduction

In recent years, pervasive computing, a scenario where a large number of sensors, often invisible, are distributed in our surrounding environment, is becoming more and more realistic. Such sensor systems find numerous applications in our every day life: network traffic management [2, 5], fraud detection [5], medical applications for monitoring heart beat, blood pressure etc and subscribing medication [3], financial analysis [14], warnings in response to environmental changes, like warnings to a chemical leak, warnings in response to smoke detection [3], building monitoring [9], monitoring wild life habitats [7, 18], and monitoring vehicle traffic [15, 13].

In our lab, we have two projects that use sensor systems. In the bus tracking project [17], we have equipped the buses that run on the UCLA campus with different kinds of sensors: GPS sensors give location, velocity sensors give the speed of the bus, direction sensors give the direction the bus is moving etc. The data is collected from the buses every second and is stored in a database. The user can ask queries on streaming data such as: at this instant, which buses are running and their locations. We can also perform mining on the stored data for predicting the time it will take for the bus to go from point A to point B based on past patterns, and factors such as the time of day, the day of the week, the week in the quarter, and the driver.

In the Smart Kindergarten project [16], we create a "smart" classroom equipped with different kinds of sensors such as seismic sensors, audio and video sensors, and active badges. Information from these sensors are collected continuously. This gives information regarding the activities of the kids, the environmental conditions; it enables us to learn more about behavior of kids as what environmental changes caused what response in the kids, what stories attracted attention of the kids etc. Also the data collected is used by the speech recognition group to build models of speech recognition for the kids.

An important question in this scenario is how do we make use of the sensor data effectively. Different models have been proposed for sensor data: one of the most widely studied models is considering the sensor data as streaming data, and not storing it. This model is used in [1, 8, 20, 19]. It is suited for several applications where the operators are such that we do not need to store the data, and the amount of data generated is enormous and it is not feasible to store the data at the rate at which it is generated. Further, this model provides new and challenging problems in defining operators for streaming data, query processing, optimization, data mining algorithms, computing approximate results etc [10, 5].

However our applications do not fit the streaming model for sensor data: for our bus project, the amount of data generated is "small"; in a day, the

*This material is based upon work supported by the National Science Foundation under Grant Nos. 0086116, 0085773, and 9817773. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

amount of data generated is about 5 MB. Further there are several operators for our applications for which the streaming model does not suffice: to determine patters for how much time it takes to commute from point A to point B, developing models for speech recognition of children, processing the videos of kids etc. We therefore store the sensor data.

However, storing sensor data gives rise to numerous issues, we address two of them in this work. First is what data should be stored? Consider a temperature sensor: should we store data every second, every minute, or every hour? Consider a video sensor that can send videos at multiple resolutions: what is the resolution at which the video should be stored? Second is how do we understand the semantics of stored data. Consider data stored from a temperature sensor, assume we have temperature values stored at 9:00am as 68, 9:30 am as 69 and 10:30 am as 69. What happened at 10:00 am? Was there a data loss? Or is the data stored when temperature is below 70, and at 10:00 am, the temperature was more than 70? Our work tries to answer these questions. A data value to be stored is decided by an application, i.e., current applications in the environment specify what data will be interesting to future applications. We also store the query that generated the data along with the data, this enables a later application to recreate the environment when the data was generated, so it can make effective use of the data.

There are models which try to store the sensor data also. In [3], the system may optionally store the sensor data. However a later application cannot understand the semantics of the stored data. In the above example of the temperature sensor, an application will understand the temperature at 10:00 am as null. Another approach to store the sensor data is provided in [21]. Here the authors assume a fixed amount of storage, so the older data is stored at coarser granularities, and they try to answer aggregate queries over the data. Our work in this paper assumes that we can store all the data, and our focus is on more general queries and not just aggregate queries.

1.1 Outline of this paper

The paper is structured as follows. In Section 2, we describe our overall system architecture: *services* that exist in the environment, and how an application can find these services and use them through our *sylph* middleware. In Section 3, we describe the architecture of our data store, define what data is stored in the database, and how the query is stored with the data, and in Section 4, we describe example applications that can be supported by our system.

2 System Architecture

In our lab [11], we are building a generic system for a sensor environment. The overall system architecture is shown in Figure 1. We will describe the different components of our system: **Services**, **Sylph Middleware**, **Applications** and **Data Store**.

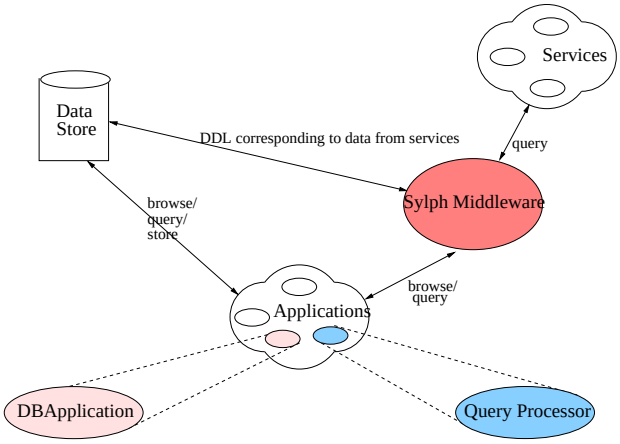

Figure 1: System Architecture illustrating Services, Sylph Middleware, Applications, and Data Store

2.1 Services

A service is any "service" that can provide data to an application. For example, a temperature service provides temperature data, a location service provides location information [4] etc.

There are two kinds of services: **Sensor Service** and **Fusion Service**. *A sensor service provides data corresponding to a particular sensor.* An example is the temperature service that provides data corresponding to a temperature sensor. *A fusion service provides data corresponding to a "higher level service" using data from other sensor and fusion services.* An example is the location service that uses the data (signal strengths) from different sensor services (base stations) to determine the location of a person. The two services can be defined in terms of how they obtain the input data as follows:

```
FusionService := (FusionService | SensorService)*
SensorService := sensor
```

Associated with every service, there is a software component called *ServiceModule*[1]. The ServiceMod-

[1]A ServiceModule could be associated with multiple services also. For example, all the sensors on one bus are associated with one ServiceModule. However, for easier understanding, we define a separate ServiceModule for every service.

ule hides the service implementation details and provides a uniform interface for all services for an application to use. This standard interface allows an application to (a) retrieve the schema for the data produced by the service (b) retrieve attribute values, such as make of sensor, current sampling rate etc (c) set attribute values, such as set the sampling rate, set the resolution for a video service etc, and (d) retrieve values from the service and send it to the application through a *push-based interface*. Here, the application will issue a query in the *Sylph Query Language* discussed in the next section.

In short, the ServiceModule acts as the proxy for the service associated with it. It receives requests for data and to set attributes from different applications, and decides how they should be handled. Our ServiceModule is similar to the Sensor Proxy described in [8], however our services can also be fusion services.

2.2 Sylph Middleware

The sylph middleware is the means of communication between the application and the sensor environment, and between service modules and the data store. It provides three capabilities: (a) It maintains the current list of service modules in the environment. (b) An application can see this list, and issue queries in Sylph Query Language to any service module. (c) Sylph ensures that there is a schema corresponding to any service module in our data store, so that the data corresponding to this service may be stored. If there is no schema corresponding to a service module, it issues appropriate DDL (Data Definition Language) statements to create the schema for this service.

2.3 Applications

Applications make use of the data generated by services. They can be user applications or system applications (those provided by the system). Applications see the list of services available through the sylph middleware, and send queries and obtain data from services. Also, they can store data obtained or ask complicated queries requiring joins of data from multiple services through two system applications provided: *DBApplication*, and *Query Processor*.

- *DBApplication* provides the interface for the application to store data it receives from services. An application can request DBApplication to store data corresponding to a service. DBApplication also provides capabilities for the application to retrieve stored data from the database; it provides synchronized display of multiple stored streams through labView [6].
- *Query Processor* is similar to the query processors provided by [9, 20, 3, 1]. The Sylph query language is limited in its functionality, it can only retrieve data from one service. Some applications might want to join the data from multiple services; an application requests data when temperature and humidity are both greater than 80. We might not have a service that can answer this query directly. Similarly some queries might be too complicated, such as the correlated aggregate query [5]: *select days when the average temperature is more than 10F greater than the average temperature of the previous day.* Such queries are handled by the query processor.

2.4 Data Store

Data Store is described in detail in the next section. An application can request DBApplication to store some of the data it receives from a service in the data store. The data store consists of a relational database for storing data from the service, queries and some metadata about the queries, and a unix file system for storing multimedia data.

3 Data Store Architecture

In this section, we describe our data store architecture, and the semantics of stored data: when is data stored, and how we store the query with the data. We start with the *Slyph Query Language*.

3.1 Sylph Query Language

Sylph Query Language supports three kinds of queries.

- *Obtain Attribute Values*: get ⟨attr⟩ for ⟨service⟩
  attr is the attribute name, and service is the name of the service.
- *Set Attribute Values*: set ⟨attr⟩ value ⟨val⟩ for ⟨service⟩
  Here val is the value to be set for attr.
- *Obtain Data Values*: read ⟨colList⟩ from ⟨service⟩ every ⟨interval⟩ [for ⟨duration⟩] [start ⟨startTime⟩] [end ⟨endTime⟩] [where ⟨predicateExp⟩]
  Here ⟨colList⟩ is the list of columns whose values the application is interested in, interval gives how periodically the application wants to receive the data, duration is optional and specifies the duration for which values are needed, startTime and endTime are optional and specify the time

when the service needs to start sending data and stop, and predicateExp is optional and specifies the condition that needs to be satisfied when the data should be sent.

Let us consider a few example queries.

1. Obtain temperature data every 30 minutes for 8 hours starting at 9:00 am.
   read temp, time from tempService every 30 min for 8 hrs start 9:00 am

2. Same as 1, but only when > 75.
   read temp, time from tempService every 30 min for 8 hrs start 9:00 am where (temp > 75)

3. Same as 2, but only when humidity is > 80.
   read temp, time from tempService every 30 min for 8 hrs start 9:00 am where (temp > 75 and humidity > 80)

3.2 Stored Data

Let us see what data gets stored in the database. First, every service whose data can be stored registers its schema with the database. When the service registers with sylph, it checks if there is a schema corresponding to its data; if not, new relation/(s) are defined to store data corresponding to that service. For example, the schema corresponding to the temperature service could be one relation with columns $(temp, time, setQueries)$, where *temp* is the actual temperature value, *time* is when that temperature value was obtained and *setQueries*, which we discuss in the next subsection, is a set of queries that generated this data.

An application can request data from a service through sylph. It can also request that this data be stored. There could be data loss and some data from the service might not reach the application, but we assume that there is no data loss from the application to the data store. In short, *the data stored in the database at any instant t is a subset of the data produced by all service modules at t.*

Let us consider the example of temperature service. An application A wants to store the values for the query: read temp, time from tempService every 30 min for 8 hrs start 9:00 am. The tempService can sample temperature at a much faster rate, say once a minute. However the tempService module will send data every half an hour to A, at 9:00 am, 9:30 am etc, but not at 9:15 am. Suppose no other application has requested data from tempService to be stored. The values stored in the database will correspond to the data received by A. There could be data loss, and a temperature value produced at 10:00 am might not reach A; in this case, the temperature at 10:00 am is not stored in the database also. We can see that if all the data produced by all queries is stored in a time interval $[T_1, T_2]$, then a later application will see the same values as all applications during $[T_1, T_2]$.

3.3 Storing Query with Data

We discussed above what data is stored in the database so that a later application B sees the same values as the current applications. However this is insufficient for B to understand the semantics of stored data. For example, suppose there are temperature values stored at 9:00 am, 9:30 am, and 10:30 am. What happened at 10:00 am? B cannot distinguish whether data was lost, or the predicate was not satisfied. On the other hand, consider the application A that issued the query: read temp, time from tempService every 30 min for 8 hrs start 9:00 am. A knows that there was data loss at 10:00 am. We provide this semantics to B by storing the query with the data.

Let us first consider how A can distinguish between data loss, and that the predicate was not satisfied. Consider the query: read temp, time from tempService every 30 min for 8 hrs start 9:00 am where (temp > 75). Suppose there are values at 9:00, 9:30 and 10:30 am. We want A to determine whether there was data loss or whether temperature ≤ 75 at 10:00. The data values from the service module are of the form: ⟨dataColumns, time, prevTime⟩. Here dataColumns is the columns corresponding to the data, time is the current time stamp and prevTime is the previous time the predicates were satisfied and data sent to the application. prevTime helps A to determine all data losses in the network; unless there are two consecutive data losses. Suppose the data A receives at 10:30 am specifies that the previous time the data was sent was at 10:00 am, then A knows that the temperature at 10:00 am was > 75, and there was data loss. If the data A receives at 10:30 am specifies that the previous time the data was sent was 9:30 am, then it knows that the temperature at 10:00 am was ≤ 75. Our solution for B to get the same semantics as A is as follows:

- Every query is appended with an implicit connectivity predicate. For example, the query read temp, time from tempService every 30 min for 8 hrs start 9:00 am where (temp > 75) is translated to read temp, time from tempService every 30 min for 8 hrs start 9:00 am where (temp > 75 and connPredicate).

- We store a "parsed form" of the query in XML

as described below. Every query is stored in a relation $Query(queryID, queryText)$, where $queryID$ is a unique id for each query, and $queryText$ is the actual query (with the connectivity predicate). We use XML as it gives a "convenient parsed form". A query is inserted into this relation by the DBApplication, when an application requests that data corresponding to this query be stored.

- We have a separate table for storing values for query predicates. These are stored in a relation $QueryPredicates$ ($queryID$, $time$, $Node_1$, $Node_2$, ..., $Node_n$). This can store queries with atmost n predicates, n is a suitable number chosen. Tuples are inserted into this relation by DBApplication as data is stored. The value corresponding to a $Node_i$ for a query Q can take one of four values: *true*, *false*, *unknown* if the value of predicate with id i in Q is unknown, or *null*, if there is no predicate with id i in Q.

- We have a separate relation for storing the values from the services. The relation for service S is $SRelation$ ($colList$, $time$, $setQueries$), where $colList$ is the list of columns corresponding to this service as described in the schema for this service, $time$ is the timestamp when a data tuple was generated, and $setQueries$ is the set of query ids that generated this tuple.

Let us examine our solution in detail. The XML document corresponding to a query conforms to the schema given as a regular tree grammar [12] in Table 1. The XML document for the Sylph query $Q1$: read temp, time from tempService every 30 min for 8 hrs start 9:00 am where (temp > 75) is shown in Table 2. When the application requests that data from $Q1$ be stored, DBApplication stores $Q1$ in the relation $Query$ as shown in Table 3 (a). Corresponding to $queryText$, we store the XML document in Table 2.

```
<query id='Q1'>
 <attrs><attr>temp< /attr>
  <attr>time< /attr>< /attrs>
 <service>tempService< /service>
 <interval>30 min< /interval>
 <duration>8 hrs< /duration>
 <startTime>9:00 am< /startTime>
 <predicates @id='1' @logOp='AND'>
  <predicate @id='2'>temp > 75< /predicate>
  <connPredicate @id='3'/ >
< /query>
```

Table 2: Example Sylph Query in XML

Let us examine how data is stored in the other two relations $QueryPredicates$ and $SRelation$. DBApplication receives a data tuple to be stored corresponding to a query, say Q. Let the data tuple received be $(val, t, previoust) = (val_1, t_1, t_0)$. The DBApplication also maintains the previous time stamp when a data tuple was stored in the database corresponding to query Q, let this be $prev$.

- **Case 1.** $t_0 = t_1 - interval$, *and* $prev = t_0$. In this case, there is no data loss in the previous time stamp; also the predicates were satisfied. DBApplication inserts $(val_1, t_1, \{Q\})$ into $SRelation$, and $(Q, t_1, predicateValues)$ into $QueryPredicates$, where $predicateValues$ are appropriate values for the different predicates.

- **Case 2.** $t_0 \neq t_1 - interval$, *and* $prev = t_0$. Here, there is no data loss, but the predicates were not satisfied between $(prev, t_1)$. DBApplication inserts $(val_1, t_1, \{Q\})$ into $SRelation$. It also inserts appropriate values into $QueryPredicates$ for time stamps t_1 and when the predicates were not satisfied.

- **Case 3.** $prev \neq t_0$. Here, there is data loss at t_0. DBApplication inserts $(val_1, t_1, \{Q\})$ into $SRelation$. It also inserts into $QueryPredicates$ values that indicate the data loss at t_0, and that predicates are satisfied for t_1.

Let us consider the sample query $Q1$; the data from the service is stored in the relation $TempRelation$ ($temp$, $time$, $setQueries$). Let us assume that the temperature is > 75 at 9:00 am, 9:30 am, 10:30 am, 11:00 am, 11:30 am, ... However, there is data loss at 11:00 am, and the temperature is $\leq$ 75 at 10:00 am. The data tuple at 10:30 am will indicate that the predicates were not satisfied at 10:00 am. Similarly the data tuple at 11:30 am will indicate that there was a data loss at 11:00 am. The relations corresponding to $QueryPredicates$ and $TemperatureRelation$ are shown in Tables 3 (b) and (c) respectively.

4 Example Applications

Let us examine how a later application can make use of the stored data. The model used is based on simple linear interpolation to find out values when data is not stored. To be more precise, suppose we have data values corresponding to time instants t_1 and t_2, and t is the earliest time instant between t_1 and t_2 for which we have values in the $QueryPredicates$ relation, and not in the $SRelation$ (that is the relation corresponding to the service data has no value at t). Then

$Query \rightarrow query(@id, Attrs, Service, Interval, Duration^?, StartTime^?, EndTime^?, Predicates)$,
$Attrs \rightarrow attrs(Attr^*)$, $Attr \rightarrow attr(PCDATA)$,
$Service \rightarrow service(PCDATA)$, $Interval \rightarrow interval(PCDATA)$, $Duration \rightarrow duration(PCDATA)$,
$StartTime \rightarrow startTime(PCDATA)$, $EndTime \rightarrow endTime(PCDATA)$,
$Predicates \rightarrow predicates(@id, @logOp, Predicate^*, ConnPredicate)$,
$Predicate \rightarrow predicate(@id, (@logOp, Predicate^*) \ + \ PCDATA)$, $ConnPredicate \rightarrow connPredicate(@id)$

Table 1: XML Schema corresponding to a Sylph Query

Query	
queryID	queryText
Q1	Table 2

(a) Query Relation

QueryPredicates				
queryID	time	Node1	Node2	Node3
Q1	9:00 am	true	true	true
Q1	9:30 am	true	true	true
Q1	10:00 am	false	false	unknown
Q1	10:30 am	true	true	true
Q1	11:00 am	false	true	false
Q1	11:30 am	true	true	true

(b) QueryPredicates Relation

TempRelation		
temp	time	setQueries
76	9:00 am	{Q1}
77	9:30 am	{Q1}
76	10:30 am	{Q1}
78	11:30 am	{Q1}

(c) Temperature Relation

Table 3: The different relations and values

the value corresponding to t is obtained by linearly interpolating the values for t_1 and t_2 and finding the "closest" value that satisfies the predicate values at t as stored in the *QueryPredicates* relation[2]. Values at other instants when predicates are known but data values are not known are given by the above technique. When query predicates are also not recorded, then values are given by simple linear interpolation.

For example, consider the data as explained in the previous section. The value at 10:00 am is obtained as 75, and the value at 11:00 am is obtained as 77. If a user asks for temperature values between 9:00 am and 11:30 am to be displayed, he gets the view in

[2]In our actual implementation, the value is not guaranteed to be the theoretically closest value. We try a few values that are close to this value, and when we obtain a value that satisfies the predicates, we stop, and consider this as the value at t.

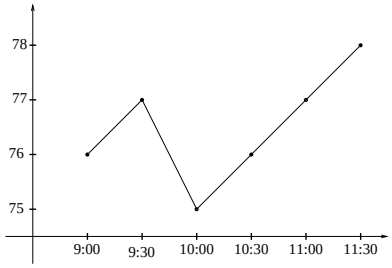

Figure 2: Displaying the temperature values. The interpolated values are displayed with a white circle, and the stored values are displayed with a dark circle

Figure 2. If a user asks queries such as when was the temperature $\leq$ 75, he gets the answer as 10:00 am. Similarly if the user asks when was there more than 2 F change in temperature in 30 minutes, he gets the answer that the temperatures at 9:30 and 10:00 differ by 2F. Similarly the temperature at 9:45 is 76 F.

Let us consider one more usage: where we can answer queries on an environment parameter whose values are not stored. For example, consider temperature values stored from the query: read temp, time from tempService every 30 min for 8 hrs start 9:00 am where (humidity $>$ 75). Suppose TempRelation and QueryPredicates relation have values shown in Table 3. *Node*2 in the query corresponds to the predicate (humidity $>$ 75). The data stored can be used to answer queries on humidity values to a limited extent: we know that at 10:00 am, humidity was $\leq$ 75.

5 Conclusions and Future Research Directions

In this paper, we studied the problem of sensor data management. We argued that there are several applications for which we need to use a non-streaming model, and store the sensor data. In this scenario, we studied how an application can make effective use of the stored sensor data. There are two difficult problems in this scenario: we cannot store all possible values of a sensor, in stead we store it periodically, also from one configuration of a sensor at any

instant. Also sensors try to minimize the data sent out, considering its bandwidth and power limitations. So what is the data that is stored? We defined the data stored as a subset of data that are sent out by the sensors, and correspond to the data received by applications. The second difficult question is the sensor data stored will have lots of holes corresponding to predicates, or network loss. How can an application understand the semantics of sensor data? We proposed a solution for this where the query that generated the data is stored along with the data. This enables a later application to get the identical view as an application that requested the data to be stored. Also more queries on the sensor data can be answered than would be possible by not storing the queries.

There are lot of opportunities for future work. One important future work is given a set of queries that later applications will ask, to determine the data and queries that should be stored. We assumed for the purposes of this paper that the data and queries to be stored is already known. Another interesting future work is index structures for sensor data, and to manage the data assuming a fixed storage model for data. In other words, we have a fixed amount of storage, when we have more sensor data than can be stored in the given storage, we have to erase some data or store older data at coarser granularities. Some preliminary work assuming temporal aggregate queries has been done in [21]. We would like to study this problem with reference to our applications, where we are not limited to purely temporal aggregate queries.

Acknowledgements: I would like to thank Yun Chi and Prof. Richard. R. Muntz for several ideas and discussions throughtout. Also all members of MMSL, UCLA for the system implementation.

References

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford Stream Data Manager (Demo). In *ACM SIGMOD*, San Deigo, CA, Jun. 2003.

[2] S. Babu, L. Subramanian, and J. Widom. A Data Stream Management System for Network Traffic Management. In *Workshop on Network Related Data Management*, Santa Barbara, CA, May. 2001.

[3] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *VLDB*, Hong Kong, China, Aug. 2002.

[4] P. Castro, P. Chiu, T. Kremenek, and R. R. Muntz. A Probabilistic Room Location Service for Wireless Networked Environments. In *Ubiquitous Computing*, Atlanta, GA, Sep. 2001.

[5] J. Gehrke, F. Korn, and D. Srivastava. On Computing correlated aggregates over Continual Data Streams. In *ACM SIGMOD*, Santa Barbara, CA, May. 2001.

[6] National Instruments. Labview: The software that powers virtual instrumentation. http://www.ni.com/labview/.

[7] Berkeley Intel Research Laboratory. Great duck island project. http://www.greatduckisland.net/index.html.

[8] S. Madden and M. J. Franklin. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *IEEE Int'l Conf. on Data Engineering*, San Jose, CA, Feb. 2002.

[9] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously Adaptive Continuous Queries over Streams. In *ACM SIGMOD*, Madison, Wisonsin, Jun. 2002.

[10] G. S. Manku and R. Motwani. Approximate Frequency Counts over Streaming Data. In *VLDB*, Hong Kong, China, Aug. 2002.

[11] MMSL. Multi media systems laboratory, ucla. http://mmsl.cs.ucla.edu.

[12] M. Murata, D. Lee, and M. Mani. "Taxonomy of XML Schema Languages using Formal Language Theory". In *Extereme Markup Languages*, Montreal, Canada, Aug. 2001.

[13] OSU. Traffic monitoring laboratory, osu. http://www.ceegs.ohio-state.edu/ coifman/TML/.

[14] D. S. Parker, R. R. Muntz, and H. L. Chau. The Tangram Stream Query Processing System. In *IEEE Int'l Conf. on Data Engineering*, Los Angeles, CA, Feb. 1989.

[15] Berkeley Path. Path for advanced transit and highways (path), berkeley. http://www-path.eecs.berkeley.edu/.

[16] UCLA. Smart kindergarten. http://nesl.ee.ucla.edu/projects/smartkg/default.htm.

[17] UCLA. Ucla transit services - realtime bus tracking. http://pantheon.cs.ucla.edu/RTV/.

[18] Princeton University. The zebranet wildlife tracker. http://www.ee.princeton.edu/ mrm/zebranet.html.

[19] S. D. Viglas and J. F. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *ACM SIGMOD*, Madison, Wisonsin, Jun. 2002.

[20] Y. Yao and J. E. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. *ACM SIGMOD Record*, 31(3), Sep. 2002.

[21] D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger. Aggregations over Data Streams using Multiple Time Granularities. *Information Systems Journal*, 28(1), Mar. 2003.