NILE-PDT: A Phenomenon Detection and Tracking Framework for Data Stream Management Systems

M.H. Ali[1], W.G. Aref[1], R. Bose[3], A.K. Elmagarmid[1], A. Helal[3], I. Kamel[2], M.F. Mokbel[1]

[1]Department of Computer Science, Purdue University, West Lafayette, Indiana {mhali, aref, ake, mokbel}@cs.purdue.edu

[2]College of Information Systems, Zayed University, U.A.E. Ibrahim.Kamel@zu.ac.ae

[3] Computer and Information Science & Engineering, University of Florida {rbose, helal}@cise.ufl.edu

Abstract

In this demo, we present *Nile-PDT*, a *Phenomenon Detection* and *Tracking* framework using the *Nile* data stream management system. A phenomenon is characterized by a group of streams showing similar behavior over a period of time. The functionalities of *Nile-PDT* is split between the Nile server and the *Nile-PDT* application client. At the server side, *Nile* detects *phenomenon candidate members* and tracks their propagation incrementally through specific sensor network operators. *Phenomenon candidate members* are processed at the client side to detect phenomena of interest to a particular application. *Nile-PDT* is scalable in the number of sensors, the sensor data rates, and the number of phenomena. Guided by the detected phenomena, *Nile-PDT* tunes query processing towards sensors that heavily affect the monitoring of phenomenon propagation.

1 Introduction

Many sensor-network applications are interested in detecting and tracking phenomena that appear in their fields of interest. Examples of interesting phenomena include the spatiotemporal propagation of pollutants, e.g., an oil spill region or a gas leakage cloud. Formally, we define a phenomenon to be a group of sensors that join with each other, over similar values, α times in a time-window of size w. This definition is controlled by two parameters, the strength (α) of a phenomenon and its time span (w). The strength parameter (α) qualifies a set of sensors to form a phenomenon if the sensors produce the same value at least (α) times. A value that appears less than α times is considered noise and is not reported as a phenomenon. The time span parameter (w) can be viewed as a time-tolerant parameter. w limits how far a sensor can be lagging in reporting a phenomenon.

Nile [3], a stream query processing engine developed at *Purdue* University, provides a pipelined execution of continuous queries over sensor data streams. In this demo, we introduce *Nile-PDT*, a framework for *Phenomenon Detection* and *Tracking* using Nile.

2 Features of Nile-PDT

The task of phenomena detection and tracking is divided among the Nile server and the Nile-PDT client application. At the server side, we make use of a new query operator, the *SN-join* (or the *Sensor Network join*) operator. *SN-join* is a generalized similarity-based binary join operator that is designed for sensor network applications. The main challenges in realizing SN-join are the following: First, SN-join is a similarity-based join. In other words, it is not necessary that sensor values match exactly. Sensor readings that are close to each other in value will still join and produce an output tuple. Second, and most interesting, is the following feature of SN-join. The input to SN-join is a sensor network SN that is composed of many sensors, each supplying an input data stream. For example, the data stream of sensor i in *SN* is referred to by *SN[i]*. Since there are many sensors in *SN*, it is the responsibility of SN-join to figure out which pair of sensors *SN[i]* and *SN[j]* out of the many sensors in *SN* that have similar joinable values within a time window w of each other. SN-join uses a relevance feedback mechanism that guides SN-join as to which streams to join in order to increase the likelihood of producing binary join output tuples. *Nile-PDT* applies *SN-join*, along with other query plan operators, over the incoming sensor data streams to report the sensors that join with each other over a time-window w as *phenomenon candidate members*. The client tracks the detected *candidate*

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

SELECT i, j, value, ts
FROM SN
WHERE $SN[i].value \equiv SN[j].value$
AND i <> j
AND <*other conditions*>
WINDOW W

Figure 1: Nile-PDT SQL queries

members and aggregates them to form a phenomenon with the desired features, e.g., having the minimum number of occurrences (α), the minimum number of sensors in a phenomenon, the spatial location of sensors, and the connectedness of the members, etc.

The main features of *NILE-PDT* are summarized as follows:

1. **Query processing with relevance feedback.** The query processor aims at maximizing the number of detected phenomena. Based on the detected *phenomenon candidate members*, query processing is tuned towards sensors where phenomena are most likely to develop. In this demo, we developed two query operators that make use of relevance feedback: SN-join and SN-scan.

2. **Load shedding and scalability.** *Nile-PDT* provides feedback to the *Nile* stream manager to control the sampling rate of sensors. Sensors that contribute heavily to the propagation of phenomena are given more attention, while sensors that participate in no phenomena are sampled at a lower rate. As a result, *Nile-PDT* scales with the number of sensors, sensor data rates, and the number of detected phenomena.

3. **Incremental processing.** *Nile-PDT* incrementally monitors phenomena in the sensor network and continuously updates the user with the appearance and the disappearance of phenomena. *Nile-PDT* takes advantage of *Nile's* notions of positive and negative tuples [2] to incrementally track the phenomenon propagation.

Phenomena detection and tracking is initiated by a continuous SQL query issued by the client. To support the execution of continuous queries over sensor data, the system is extended with the abstract data type (*ADT*) *SensorNetwork-ADT*. *SensorNetwork-ADT* extends the functionality of relational tables by appending extra information to each tuple. A sensor reading is in the form $SN[ID].(value, ts)$, where *ID* is a sensor identifier and *value* is the reading value of that sensor at timestamp *ts*. Figure 1 introduces the general form of SQL-queries that are issued by the client. Sensor network *SN* is joined with itself, which means any two sensors from *SN* are eligible to join with each other based on a similarity join over *SN.value*. The condition $(i <> j)$ prevents the sensor from being joined with itself. Other conditions can be specified as well in the *where* clause, e.g., timestamp and value predicates. The result is sent to the Nile-PDT client to be grouped and analyzed

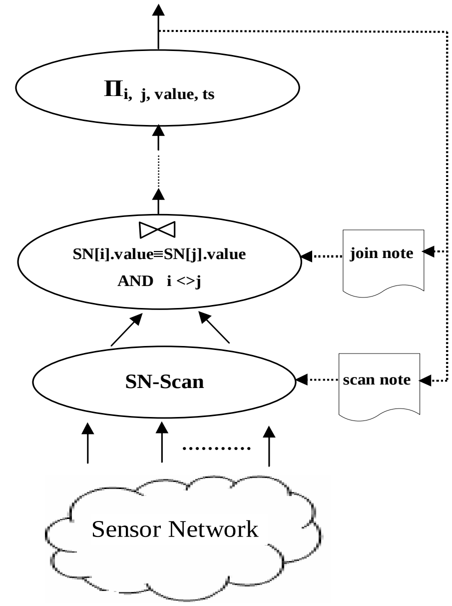


Figure 2: Nile-PDT query plan

then to report sensors that join with each other more than α times within the last time-window w.

3 Query Processing with Relevance Feedback

Figure 2 gives the query plan for the query in Figure 1. The stream tuples are pushed from the sensor network into the system's input buffers through the *SN-scan* operator. Then, the *SN-join* operator is applied over the incoming streams to detect which sensors give the same or similar readings over the specified time-window. *SN-scan* and *SN-join* are special operators that are tuned for sensor-network processing. These operators may accept feedback (or hints) from other query plan operators that express the relevance of the join output tuples to the query result.

The SN-scan Operator. *SN-scan* is responsible for attaching the sensor-network platform to the sensor-network abstract data type (*SensorNetwork-ADT*). *SN-scan* scans the sensors for fresh readings and passes these readings up in the query plan. *SN-scan* is optimizable through its capability to accept *scan notes* from higher operators in the query plan. The *scan notes* update the relative frequencies at which the *SN-scan* operator reads from the sensors. The *scan notes* are extracted by estimating the likelihood of a sensor to contribute to the output. The *scan note* is a well-defined interface through which the *SN-scan* operator can be tuned to increase the scanning rate of a specific sensor.

The SN-join Operator. A traditional join operation does not scale to a sensor network that contains thousands of sensors. Stream join has been discussed in literature, e.g., [1, 5]. In the context of Nile-PDT, each sensor does not have to join with every other single sensor in the sensor network (e.g., a phenomenon spans only a portion of the sensor network). The challenge is to find the join pairs from among the many sensors that join together over the time-window w.

To address this challenge, we developed a new join operator, the *SN-join* operator that is especially designed for large-scale sensor networks. *SN-join* is guided by the output of the query to direct the join operation towards sensor pairs that are more likely to contribute to the join output. In the context of *Nile-PDT*, *SN-join* is guided by the detected *phenomenon candidate members* to perform the join among sensors with similar behavior. *SN-join* maintains a 2-d matrix (P) that records the *probe probability* between each two sensors. A reading from sensor $SN[i]$ probes sensor $SN[j]$ for a join based on the probability P_{ij} (i.e., with probability $1 - P_{ij}$, the probing overhead will be skipped). Higher operators in the query plan provide the *SN-join* with *join notes* that help update the probability matrix (P). Based on the portion of a sensor stream that has been seen so far, *join notes* are extracted by estimating the likelihood of two sensors to contribute to the join output. The *join note* is a well-defined interface through which *SN-join* can be tuned to favor the join operation among certain sensor pairs. Several sensor probing mechanisms are explored in the context of Nile-PDT. The purpose is to track existing phenomena (guided by the join notes), but at the same time detect new phenomena that emerge in new regions in the sensor network. This feature is captured in our demo by measuring the time delay between when a phenomenon actually happens and when it is detected by Nile-PDT.

The second challenge in realizing SN-join is that of similarity matching. Due to sensor calibration and/or measurement errors, sensor readings can be similar in value but are not necessarily the same. As a result, SN-join is a similarity-based join. The Nile-PDT demo reflects two similarity-based techniques. The first technique uses a pre-clustering operator that is below SN-join in the query pipeline. This pre-clustering operator dynamically clusters the sensor readings and feeds SN-join with cluster-ids. In this case, SN-join performs equi-join based on the cluster-ids. Alternatively, the second technique is to push a similarity distance function inside SN-join, so that sensor readings join with each other if the distance between the readings is less than a threshold. Both techniques are reflected in the Nile-PDT demo and their performance is contrasted.

4 Load Shedding and Scalability

Data streams may arrive with high rates at the system's input buffers and they can be bursty in nature. Such behavior overloads the system and deteriorates the query performance. *Load shedding* avoids heavy-load periods by dropping some of the input tuples. In contrast to dropping the tuples randomly, the tuple dropping policy favors a certain performance measure. Load shedding that is sensitive to phenomena detection tries not to lose phenomena while dropping some of the input data. Load shedding is achieved through the *SN-scan* operator where sensors that contribute to phenomena are processed more frequently than sensors that do not contribute to any phenomenon.

Scalability in Nile-PDT is achieved through the *SN-scan* and *SN-join* operators. Both operators avoid wasting the processing time in sensors that do not help in detecting and tracking phenomena. For example, using the relevance feedback mechanism, an incoming sensor reading may end up probing a few tens of sensors looking for a match instead of probing thousands of sensors in the sensor network. As illustrated in our demo, during simulations that include a sensor network of thousand sensors with each sensor stream having an average inter-arrival time of one second, *Nile-PDT* is able to capture more than 90% of the outstanding phenomena.

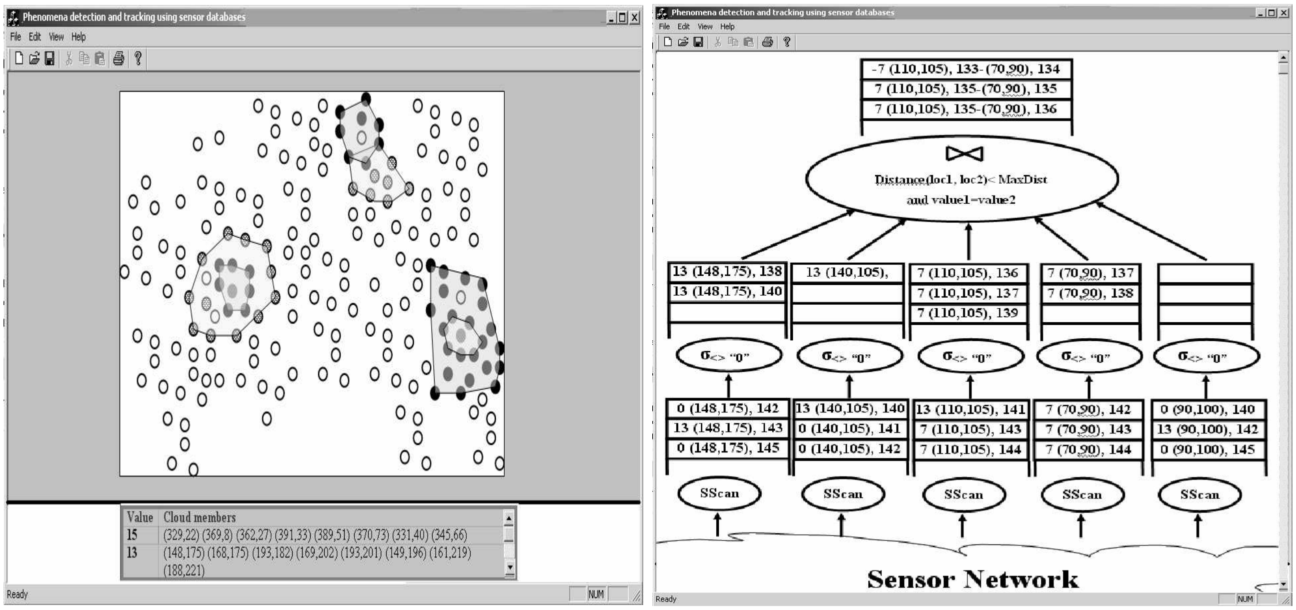
5 Incremental Processing

Once a phenomenon is detected, the tracking process is conducted incrementally at both the server and the client sides. At the server side, incremental processing is achieved through the notions of positive and negative tuples [2]. A positive tuple is reported when a join occurs to denote the appearance of phenomenon candidate members. A negative tuple is reported when one of the previously-reported join components expires, i.e., becomes old enough to get outside of the most recent time-window w. Negative tuples are important to invalidate phenomenon candidate members if sensors stop showing the same behavior over the most recent time-window w.

At the client side, the client receives phenomenon candidate members on the form of a tuple that consists of the IDs of the two joining sensors and the join value. Each tuple can be positive or negative to denote the appearance or disappearance of the candidate members. The client acts based on each tuple. Upon receiving a positive tuple, the client may perform one of the following actions: (1) create a new phenomenon, (2) add one more sensor to an existing phenomenon, or (3) merge two phenomena into one bigger phenomenon if the two phenomena get connected. Upon receiving a negative tuple, the client may perform one of the following actions: (1) delete an existing phenomenon, (2) remove a sensor from an existing phenomenon, or (3) split one phenomenon into two smaller phenomena if they get disconnected.

6 Demo Description

A graphical user interface (*GUI*) is developed for both the Nile-PDT client and the Nile server to visualize the phenomenon detection and tracking processes. Figure 3 gives snapshots of the *GUI* of both the client and the server. Our demo has two setups: one where the sensor network is simulated (as described in Section 4) and the other is using real sensors, as described below. Our demo hardware consists of a grid of heat sensors (Figure 4) that are connected via a wireless sensor platform [4]. Each platform interfaces with the sensors that are connected to it as OSGi service bundles [6]. The sensor platform used in Nile-PDT has a flexible modular architecture that consists of a processing



(a) (b)

Figure 3: Snapshots of the Nile-PDT visualization tools: (a) client visualization (b) server visualization

module, a communication module, and a testing module. Each sensor platform has a limited processing capability. Details about the sensor platform and the modules can be found in [4].

When we run the demo using a simulated sensor network, the client *GUI* (Figure 3a) represents each sensor by a circle that reflects its location in space. Sensors are spread all over the space arbitrarily. The client can keep track of both the original phenomena that are computed offline given infinite resources (depicted as gray circles) and the phenomena that are detected by the system (depicted as black circles). The client *GUI* shows the efficiency of the system in two aspects: (1) the number of detected phenomena relative to the original number of existing phenomena, and (2) the response time (delay) of the system. The response time is identified by how far the detected phenomena propagation lags after the original phenomena propagation.

The server *GUI* (Figure 3b) demonstrates the system's internals and shows how the query plan is executed. The query plan is displayed graphically and the incoming tuples keep moving up the query plan from one operator to the next. When we run the demo using a simulated sensor network, the server can be executed in slow-motion (via inserted delays) and the number of sensors is reduced for the sake of demo clarity.

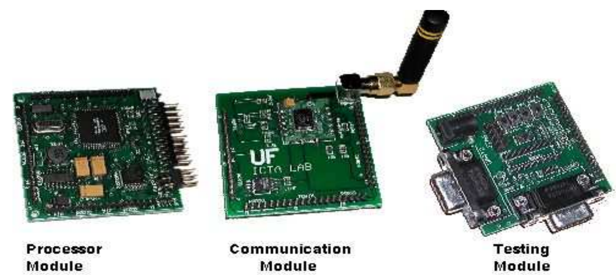


Figure 4: The various sensor modules in a sensor platform.

References

[1] M. Hammad, W. Aref, and A. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *Proceedings of the SSDBM Conference*, pages 75–84, July 2003.

[2] M. A. Hammad, T. M. Ghanem, W. G. Aref, A. K. Elmagarmid, and M. F. Mokbel. Efficient execution of sliding-window queries over data streams. Technical Report CSD-03-035, Department of Computer Science, Purdue University, June 2004.

[3] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, and X. Xiong. Nile: A query processing engine for data streams. In *Proceedings of the Intl. Conf. on Data Engineering*, page 851, April 2004.

[4] A. Helal, H. Zabadani, J. King, Y. Kaddoura, and E. Jansen. The gator tech smart house: A programmable pervasive space. *Cover Feature, IEEE Computer*, 38(3):64–74, March 2005.

[5] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of the Intl. Conf. on Data Engineering*, pages 341–352, March 2003.

[6] D. Marples and P. Kriens. The open services gateway initiative: An introductory overview. *IEEE Comm. Magazine*, 39(12):110–114, 2001.