Processing of Data Streams with Prediction Functions

Sergio Ilarri*, Ouri Wolfson†, Eduardo Mena*, Arantza Illarramendi‡, Naphtali Rishe§
*IIS Department, Univ. of Zaragoza
María de Luna 3, 50018. Zaragoza, Spain.
Email: {silarri,emena}@unizar.es
†CS Department, Univ. of Illinois
Chicago, IL, 60607
Email: wolfson@cs.uic.edu
‡LSI Department, Univ. of the Basque Country
Apdo. 649, 20080 Donostia, Spain
Email: jipileca@si.ehu.es
§School of Computer Science, Florida International University
University Park, ECS-243, Miami, FL, 33199
Email: rishe@fiu.edu

Abstract

Networks of sensors arise naturally in many different fields, from industrial applications (e.g., monitoring of environmental parameters in a chemical plant) to surveillance applications (e.g., sensors that detect the presence of intruders in a private property). The common feature of these applications is the necessity of a monitoring infrastructure that analyzes continuous supplies of data streams and outputs the values that satisfy certain constraints.

In this paper we present an approach to process monitoring queries in a network of sensors with prediction functions. We consider sensors that communicate their values according to a threshold policy and our query processing leverages prediction functions to compare tuples efficiently and generate answers even in the absence of new incoming tuples. We deal with two types of constraints: window-join constraints and value constraints.

1. Introduction

There is a recent interest in techniques for monitoring networks of sensors in a variety of contexts. Networks of sensors are used in many different fields, from industrial applications (e.g., monitoring of environmental parameters in a chemical plant) to surveillance applications (e.g., sensors that detect the presence of intruders in a private property).

Monitoring queries, that are processed as continuous queries [3], arise naturally in this environment. Thus, monitoring applications (e.g., fleet tracking applications, monitoring of the levels of certain gases in a chemical environment, etc.) are usually of great interest in this context. The common feature of monitoring applications for networks of sensors is the need of handling a continuous supply of data streams. For this purpose, many works propose the use of a *Data Stream Management System* (DSMS) [2], [7] in the *monitoring computer*, which implements suitable non-blocking techniques to process unbounded amounts of data.

A DSMS must process the types of monitoring queries that are of interest in networks of sensors. Typical queries in this context contain constraints about sensor values of a certain type, that we call *value constraints*; e.g., "alert when there is a carbon dioxide sensor that detects a value exceeding a certain, dangerous, threshold". Similarly, constraints that compare the values measured by sensors of are also of great interest. We refer to this last type of constraints as *window-join constraints*, and they can also require a certain temporal relationship for two values to be joined; for example, a query retrieving pairs of nearby sensors that measure a very different value at approximately the same time could reveal the existence of sensors that are malfunctioning.

We believe that in many contexts a sensor can predict the values that it will measure in the near future. For example, a location sensor (e.g., a GPS receiver) in a car could predict future locations by considering the current speed and route. Similarly, the temperature in a room will probably evolve during the day following some predictable patterns. Thus, each sensor value will be associated with a prediction function in a way that the sensor will update its value only when it differs significatively from the predicted value. This strategy allows a reduction of both the communication and query

0-7695-2507-5/06/$20.00 (C) 2006 IEEE

processing efforts. Due to the use of prediction functions together with an update policy where only significant values are transmitted, a pair of sensors could start satisfying a required constraint even when the system does not receive a new tuple from any of the sensors.

The rest of the paper is as follows. In Section 2 we describe our use of prediction functions. In Section 3 we describe the types of constraints we deal with: value constraints and window-join constraints. In Section 4 we describe the architecture of our system. In Section 5 we focus in window-join constraints and explain how tuples are compared in order to generate predicted tuples as a result. Some experiments showing the feasibility and performance of our approach are presented in Section 6. We conclude the paper with some related work in Section 7 and conclusions and future work in Section 8.

2. Use of Prediction Functions

In this section we first advocate the use of prediction functions and then describe the structure of the data streams released by sensors and the proposed update policy.

2..1. Prediction Functions as a Saving Mechanism

Predictions can be reasonably used in a variety of contexts. For example, a GPS in a car could send not only the current location of the car but also its vector of movement or expected trajectory [20]. Similarly, the values measured by a temperature sensor indoors are not expected to change in normal conditions. The level of fuel or the distance traveled by a vehicle, the temperature of an area, the altitude of an airplane, the intensity detected by a light sensor during the day, the number of persons entering a mall over a certain period, etc. are all examples of values that can be estimated with a prediction function. In general, predictions are specially useful when the values are expected to change according to predefined patterns, or when the interest is in the detection of unexpected changes.

Thus, we propose that every value measured by a certain sensor be attached to a prediction function that will be used to predict future values of that sensor. In this way, the sensor will only send significative values to the monitoring computer instead of sending them continually, saving a great amount of wireless communication efforts at the sensors. This is very important as wireless communications are expensive and drain quickly the energy of wireless devices [13]. A reduction in the number of communicated values also leads to a decrease in the processing overhead at the monitoring computer, increasing the scalability of the query processing.

In different environments prediction functions could be obtained in a variety of ways. In this paper we do not make any assumption about the way prediction functions are obtained [16], [8], which is out of the scope of our work. Similarly, we do not delve into the details of how thresholds are specified. They could be specified by the user that issues queries; for example, if he/she is not interested in decimal positions of a value a threshold of one unit can be specified. Alternatively, more sophisticated approaches, as adaptive thresholds [20], are also possible.

2..2. Sensor's Data Streams

In this section we describe first the structure of tuples released by sensors and then we explain when a tuple is "applicable".

1) Structure of Tuples: Sensors measure values of a certain type (e.g., the petrol level in a car, the temperature, etc.) continuously or at a certain sampling frequency (e.g., 10 times per second for a GPS receiver), and we assume they have a unique identifier within the system. We focus on a *parallel sensor network topology* as in [1]: each sensor communicates its values to a centralized monitoring computer, which analyzes the incoming data streams *on the fly* to process monitoring queries. Data streams from sensors are logically composed of *update-tuples*:

$$tp_j = <s_i,\ type_i,\ t_j,\ f_j(t)>$$

where s_i is the identifier of a sensor, $type_i$ is the type of value it measures, t_j is the timestamp of the update, and $f_j(t)$ is a prediction function that, given a certain time instant t, retrieves the expected sensor value. The value of the sensor for the update time is given by $f_j(t_j)$. As an example, <speedCar12, speed, 10, 60 + 4.5 * t> is a tuple sent by a speedometer in car12 moving at time instant 10 with a speed of 60 m/s and an acceleration of 4.5 m/s^2.

For simplicity, we consider that sensors estimate themselves their values based on their past measurements and communicate them to a centralized computer together with the prediction functions. However, our work does not contradict proposals where a prediction model is computed and assigned to sensors by a third party[1], such as [6].

2) Interval of Applicability of a Prediction Function: In order to predict the value of a sensor at a given time instant, we must apply the last prediction function received from the sensor before that time instant. In other words, the prediction function of a tuple tp_k is applicable during the time interval between the timestamp of that tuple and that of the next tuple from the same sensor, which we call *Interval of Applicability of the Prediction Function* $IAPF_{tp_k}$=$[IAPF_{tp_k}.start, IAPF_{tp_k}.end)$.

[1]This is required, for example, when the prediction model is based on data not available at the sensor itself, such as values measured by other sensors in its neighborhood.

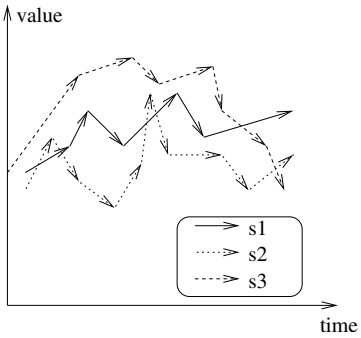

Fig. 1. Communications of Prediction Functions

In Figure 1 we show a conceptual example where we represent the communications of prediction functions. The x-axis is time, and the y-axis represents the value of a sensor. We show the changes in the prediction functions of three sensors (s_1, s_2 and s_3) along time. In the figure, we consider prediction functions that are linear (i.e., they are of the form $f_j(t) = a * t + b$ with constants $a, b \in \Re$) and, consequently, each prediction function is represented with an arrow. The projection of the arrow over the x-axis indicates the time interval during which that prediction function is applicable, and the projection over the y-axis indicates how predicted values change along that interval. The orientation of the arrow is always from left to right, as prediction functions are not used to estimate *past* values.

2..3. Threshold Update Policy with a Maximum Period Between Updates

Sensors can follow a number of *update policies* [21] in order to decide when a value is significative and should be communicated to the monitoring computer. For example, a sensor with a periodic policy would send the values it measures at a certain frequency (e.g., every 10 seconds).

In this paper, we advocate a threshold policy with a maximum period between updates. With this policy, *every sensor commits to update its value whenever: 1) the difference with the value estimated using the last prediction function exceeds a certain threshold* (correction update), *or 2) the time elapsed since the last update has exceeded a certain period* T (heartbeat update). For example, a temperature sensor could communicate its current value whenever the difference between the predicted value at a given time instant and its real value exceeds one Celsius degrees with at least one update every three minutes. The maximum update period indicates the amount of time during which a prediction function can be applied: outside that period, the prediction function is not reliable and it can be assumed that the sensor is unable to communicate new updates (e.g., it is *not alive*).

3. Types of Constraints

Constraints such as "sensors must measure a temperature under $50F$ degrees" are what we call *value constraints*:

VConstr(type, comp, K)
returns $\{$*(s, v, t)*$\}$ *such that:*

(type(v) = type) $\wedge$ *(value(s) = v)* $\wedge$
(timestamp(v) = t) $\wedge$ *(v comp K)*

where *type* is a type of sensor value, *v* is a sensor value of that type, *s* is the sensor that measures the value, *comp* is a comparator among $\leq$, $<$, $>$, $\geq$, $<>$ and $=$, and K is a constant. The sample constraint above would be expressed with this syntax as *VConstr(temperature, <, 50F)*.

More complex constraints, such as "pairs of gas sensors must measure a similar concentration of carbon dioxide within an interval of 10 seconds" are more complex and we term them *window-join constraints*:

wconstraint(type1, type2, w, comp, K)
returns $\{$*(s1, s2, v1, v2, t1, t2)*$\}$ *such that:*

(type(v1) = type1) $\wedge$ *(type(v2) = type2)* $\wedge$
(value(s1) = v1) $\wedge$ *(value(s2) = v2)* $\wedge$
(timestamp(s1) = t1) $\wedge$ *(timestamp(s2) = t2)* $\wedge$
(|t1-t2| $\leq$ *w)* $\wedge$ *(|v1-v2| comp K)* $\wedge$ *(s1* $\neq$ *s2)*

where *type1* and *type2* are two types of sensor values, and *w* is called the *valid-time window* and specifies a condition between the timestamps of two sensor values. The sample constraint above would be expressed with this syntax as *wconstraint(*CO_2*Concentration,* CO_2*Concentration, 10, >, 1)*, consider similar values those which differ less than 1%. The relative-timestamp condition allows comparisons between values as long as they refer to *approximately* the same time instant. Valid-time windows are useful: 1) in cases where timestamps are uncertain such as when the clock sensors are not precisely synchronized, and 2) in some queries such as those retrieving pairs of buses that arrive at the same stop within 30 minutes.

Both types of constraints can be combined. For example, *wconstraint(*CO_2*Concentration,* CO_2*Concentration, 10 seconds, >, 1%)* $\wedge$ *VConstr(*CO_2*Concentration, >, 12%)* is satisfied by the pairs of sensors of CO_2 fulfilling the previous window-join constraint and measuring a concentration higher than 12%.

4. Architecture

In this section we explain the query processing architecture that we propose, shown in Figure 2.

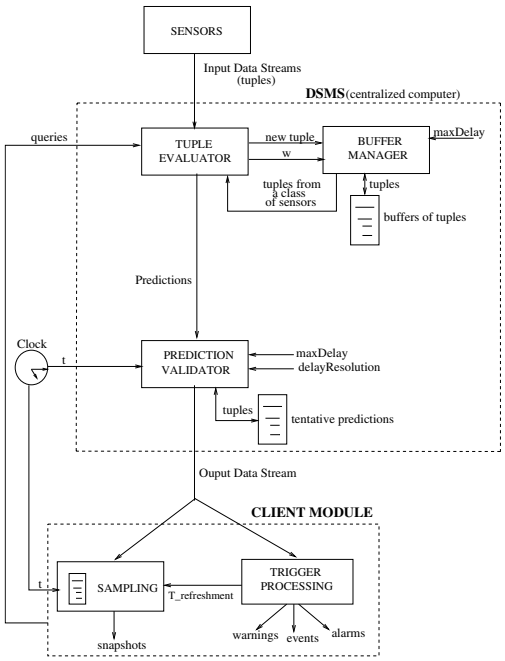

Fig. 2. System architecture

4..1. Tuple Evaluator

The Tuple Evaluator is the core of our architecture. As opposed to what happens in traditional databases, queries must be evaluated in an incremental way in order to cope with a high arrival rate of data sent by an arbitrarily high number of sensor devices. Thus, whenever a new tuple is received by the Tuple Evaluator, the system verifies how that tuple affects the active queries. For queries that consist only of a value constraint, the new tuple is considered alone. For queries that involve a window-join constraint, the new tuple is compared with previous tuples corresponding to the other types of values involved in the window-join constraint (potential matches). This incremental approach will detect all the answers to the constraint, as it is event-driven instead of being based on periodic evaluations at specific times.

For each active query, the Tuple Evaluator (explained in more detail in Section 5) generates an output data stream of tuples that are predicted to satisfy the required constraints in the future. The format of these output tuples, which we explain in the following, depends on whether the query includes a window-join constraint or not.

1) Predicted Tuples for Queries with a Window-Join Constraint: For queries that includes a constraint about pairs of sensors, the format of an output predicted tuple is:

$$<tp_i,\ tp_j,\ \text{VM=(I,P)}>.$$

where tp_i and tp_j are the input tuples that have been joined and VM is the *validity mark* of the output predicted tuple. The validity mark indicates under which conditions the predicted tuple applies; that is, when we can use the prediction functions of tp_i and tp_j to estimate the values of the corresponding sensors, values that will satisfy the query. The validity mark can be seen as a generalization of the idea of validity period presented in [18], and it has two components:

- A *validity interval* I: is the time interval during which the values of the sensors match, according to the specified constraint and the given prediction functions.
- A *timestamp-matching boundary* P: is a polygon that delimits the admissible combinations of times t_i and t_j for a match, where t_i is a time instant for the evaluation of $f_i(t)$, and t_j is an evaluation time instant for $f_i(t)$. This can be used, for example, to find examples of values that match.

2) Predicted Tuples for Queries without a Window-Join Constraint: In this case, the query consists only of a value constraint. These constraints retrieve sensors that individually satisfy certain conditions, so the format of predicted tuples is:

$$<tp_i,\ \text{VM=I}>$$

where the validity mark is given just by the validity interval I. For example, a tuple $<s_1,\ type_1,\ t_1,\ 3*t+2>$, [5,15]$>$ indicates that the sensor s_1 satisfies the constraints during the interval [5,15] by considering the prediction function $3 * t + 2$.

4..2. Buffer Manager

Sensors send the values they measure as data streams that are received by the Tuple Evaluator, which communicates them to a module that we term *Buffer Manager*. The Buffer Manager will decide which input tuples will be stored (i.e., they can be needed to answer monitoring queries) and which ones will be discarded. In this way, when the Tuple Evaluator receives a new tuple tp_i from a sensor s_i of value type S_i, it will ask the Buffer Manager about tuples tp_k of other types of values involved in window-join constraints with S_i in order to find possible matches.

The tuples that should be stored in the buffers are determined by the constraints of the active queries in the system. As different types of values are subjected to different query constraints, a different buffer is used for each type of value. In the buffer of a certain type of value, only tuples with timestamps within a given time interval need to be stored: storing more tuples would imply both a waste of storage space and a higher join processing cost.

In Figure 3 we show the tuples received from two sensors s_1 and s_2 in a sample scenario. Tuple tp_1 from sensor s_1 arrives, due to the wireless communications involved, with a delay D. If the valid-time window that affects the window-joins for the types of values of s_1 and s_2 is w, tuple tp_2 from sensor s_2 should be considered as a possible match for tp_1. This implies that tuple tp_2 should still be stored in the buffer for its type of value. Therefore, the temporal width of the buffers for the types of values of s_1 and s_2 is $w+D$. Notice that if tuples from the same sensor cannot arrive unordered, there will not be any tuples with timestamps greater than that of tp_1 in Figure 3; however, this does not affect the previous explanation.

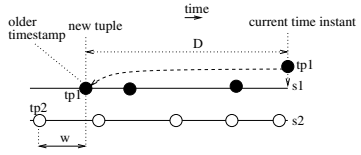

Fig. 3. Determining the Temporal Width of Buffers

Thus, the temporal width of the buffer for the type of value S is given by:

$$tWidth(S) = \begin{cases} \text{Max(w)+maxDelay} & \text{if window-join(S)} \\ \text{maxDelay} & \text{otherwise} \end{cases}$$

where *window-join(S)* returns true iff S is involved in a window-join constraint, $Max(w)$ is the maximum w in these constraints, and $maxDelay$ is a pessimistic[2] estimation of the maximum delay of input tuples. Notice that if implicit timestamps are used, then it is not possible to detect delayed tuples; in this case, $maxDelay$ can be considered to be 0.

The Buffer Manager will periodically shift the windows of tuples stored in its buffers as needed according to the previous considerations. Also, it should manage problems derived from the lack of space to store new tuples, by considering the relative importance of the prediction functions of the different tuples to decide which tuples to store. We will consider the study of this issue as future work.

4..3. Prediction Validator

The Tuple Evaluator releases, for each query, predicted tuples about values that will satisfy the query. The communication of a prediction function by any of the sensors involved in a tuple could invalidate that predicted tuple. Therefore, the predicted tuples output by the Tuple Evaluator are only *tentative* and will be consider *validated* only when there is a guarantee that they cannot be found to be wrong later.

A module called *Prediction Validator* can be plugged into the system between the output of the Tuple Evaluator and the input of a client module. This module will delay the tuples fed into the client module when there is a requirement that they be validated in advance[3].

Prediction functions included in a predicted tuple allow to get *predicted values* that satisfy the constraints for a query during the validity interval of the predicted tuple. Assuming that tuples are not received by the Tuple Evaluator later than $maxDelay$ time units since they were released by sensors, a predicted value is considered committed $maxDelay$ time units after the timestamp of the prediction function that estimated it. Thus, the Prediction Validator stores the tuples received from the Tuple Evaluator and checks them with a certain *validation period* (e.g., one second) to release *valid* tuples that can be inferred from them, by just modifying appropriately the validity mark of the predicted tuple.

Example: For a predicted tuple like $<s_1$, $type_1$, t_1, 3*t+2>, [5,20]>, *maxDelay=5*, and the current time instant equals 15 time units, the tuple that would be released in the output data stream is $<s_1$, $type_1$, t_1, 3*t+2>, [5,10]>. If the validation period is 1 time unit, then the next tuple would be $<s_1$, $type_1$, t_1, 3*t+2>, [5,11]> at time instant 16. If it is 2 time units the next tuple would be $<s_1$, $type_1$, t_1, 3*t+2>, [5,12]> at time instant 17. A new tuple from sensor s_1 could invalidate the predicted tuple and stop the process at any time instant.

4..4. Client Modules

A module client will process the tuples released by the Prediction Validator for further processing as needed. Our approach fits naturally with different types of client modules, such as:

- A *trigger processing module* that requires a certain action when an event is detected. For example, there could be a trigger that fires an alarm if a sensor detects a concentration of a dangerous substance above a certain threshold. It is dependent on the requirements of the application whether a trigger should be activated only once (when the constraints starts satisfying) or several times (while the constraints keep satisfying). For example, an alarm should continue ringing while the problem is not solved.
- A *sampling module* that periodically transforms the data stream into relations that are shown to a user as snapshots at different time instants of the values that satisfy the constraints.

[2]If an input tuple is delayed more than $maxDelay$, it could happen that the Buffer Manager does not keep in storage other tuples that could match the delayed tuple.

[3]Client applications could instead be interested in queries about the future (e.g., to prevent possible collisions between moving robots).

4..5. Interactions between Modules

In Algorithm 1 we show how the different modules of our architecture interact with each other. When the Tuple Evaluator receives a new tuple tp_1 of a type of value $S1$, it first communicates it to the Buffer Manager, which computes the $IAPF_{tp_1}$ for that tuple and updates the $IAPF$ of the previous tuple from the same sensor. For each active query with constraints about type $S1$, the Tuple Evaluator generates predicted tuples for the query, according to the constraints that it contains. Then, an *invalidation tuple* is generated for the previous predicted tuple and the time interval $IAPF_{tp_1}$, indicating that previous prediction functions for tp_1 are not applicable anymore *in that interval*.

Algorithm 1 $receiveTuple(tp_1)$

Require: tp_1 is a new tuple received by the Tuple Evaluator.
Ensure: Generates new predicted and invalidation tuples in the output for each query q, if needed.

```
bufferManager.addTuple(tp1);
IAPF_tp1 ⇐ bufferManager.getIAPF(tp1);
S1 ⇐ tp1.type;
for all q ∈ queries do
  if (((hasJConstr(q) and (q.wJConstr.involves(S1))) then
    /* There are two classes of sensors involved */
    S2 ⇐ q.wJConstr.getJoiningClass(tp1);
    buffer_S2 ⇐ bufferManager.getBuffer(S2);
    for all tp2 ∈ buffer_S2 do
      IAPF_tp2 ⇐ buffer_S2.getIAPF(tp2);
      pred ⇐ predictJ(tp1, tp2, q.getVConstrs());
    end for
    inv ⇐ InvalidationTuple(S1, S2, IAPF_tp1);
    releaseInvalidationTuple(inv, q);
    if (pred ≠ ∅) then
      releasePredictedTuples(pred, q);
    end if
  else
    /* Now window-join. */
    pred ⇐ predictV(tp1, q.getVConstr());
    inv ⇐ createInvalidationTp(tp1.s, IAPF_tp1);
    releaseInvalidationTuple(inv, q);
    releasePredictedTuples(pred, q);
  end if
end for
```

The tuples generated by the Tuple Evaluator are received by the Prediction Validator, as show in Algorithm 2. If the tuple received is an invalidation tuple, it will update the validity marks of any predicted tuple for that query and the involved sensor/s in order not to include the invalidation interval. If it is a predicted tuple, it will store it in the table of tentative predictions and release them as required (see Section 4.3). Notice that the Tuple Evaluator could release just an invalidation tuple or also predicted tuples.

5. Processing Incoming Tuples

In this section we explain how the Tuple Evaluator deals with new tuples received from the sensors. The

Algorithm 2 processPredictedTuple(tp, q)

Require: tp a tuple released by the Tuple Evaluator regarding the query q.
Ensure: updates conveniently the table of Tentative Predictions tt for query q.

```
if (isInvalidationTp(tp) then
  tt.q.trimPredicted(tp.s1, tp.s2, tp.I);
else
  tt.q.addPredicted(tp);
end if
```

comparison of tuples is performed in two stages: a filter step and an evaluation step.

5..1. Step 1: Filter Step

We will first filter out possible matches based on the idea that two predicted values cannot match if they are not comparable due to the difference in their timestamps. Notice first that two predicted values from two sensors can be compared even if they do not refer to the same time instant, as long as their timestamps are within valid-time w from each other. In Figure 4 we show a graphical example with two prediction functions, $f_k(t)$ and $f_l(t)$, and a window-join constraint with a $\leq$ comparator. Although values $v_1(t1)$, corresponding to $f_k(t)$, and $v_2(t2)$, corresponding to $f_l(t)$, refer to a different time instant (t_1 and t_2 respectively), they match. This is due to the fact that they are within the same valid-time window w and the difference between their values do not exceed K. However, if we compare values with the same timestamp t_1 or t_2 they do not match, as $|v_1(t1) - v_2(t1)| > K$ and also $|v_1(t2) - v_2(t2)| > K$.

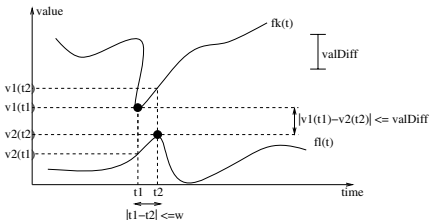

Fig. 4. Values for different time instants can match

The fact that two predicted values can be comparable even if they have different timestamps implies that two prediction functions can be comparable even if their $IAPFs$ are disjoint. However, we still can dismiss a potential match by comparing the $IAPFs$ of two tuples tp_k and tp_l to match. If after extending one of the intervals by w on its right and left sides they do not intersect, the tuples do not match:

match(tp_k,tp_l)= false if
I' $\cap$ $IAPF_{tp_l}$ = $\emptyset$
where *I'*=[$IAPF_{tp_k}$.*start - w,* $IAPF_{tp_k}$.*end + w*].

5..2. Step 2: Evaluation Step

For those pairs of tuples that are not filtered out in the previous step, we must evaluate the window-join constraint over them to determine when they will match (if ever). As a result of the comparison of two tuples, predicted tuples that will satisfy the constraint are generated.

In this step we limit our study to linear prediction functions, so our problem translates thus to comparing prediction functions by *solving a system of linear inequalities*, which can be performed efficiently. We would like to stress that many well-known estimation techniques such as the linear extrapolation, double exponential smoothing or the Kalman filter are linear models, and that non-linear prediction functions can be approximated in many cases by linear functions using a variety of techniques [14]. Also, for clarity of exposition we will focus only on a window-join constraint with a $\leq$ comparator.

First we change the name of the temporal variables t in the prediction functions to capture the fact that they can refer to different time instants t_k and t_l (otherwise, we would be implicitly requesting that $t_k = t_l$, i.e., that both functions must be evaluated at the same time instant):

$$f_k(t_k) = a * t_k + b$$
$$f_l(t_l) = c * t_l + d$$

The corresponding system of linear inequalities for the window-join constraint becomes:

$$a * t_k - c * t_l + b - d \leq K \quad (1)$$
$$c * t_l - a * t_k + d - b \leq K \quad (2)$$
$$t_k - t_l \leq w \quad (3)$$
$$t_l - t_k \leq w \quad (4)$$
$$t_k \geq IAPF_{tp_k}.start \quad (5)$$
$$t_k \leq MIN(IAPF_{tp_k}.end, IAPF_{tp_k}.start + T) \quad (6)$$
$$t_l \geq IAPF_{tp_l}.start \quad (7)$$
$$t_l \leq MIN(IAPF_{tp_l}.end, IAPF_{tp_l}.start + T) \quad (8)$$

All of these inequalities must be satisfied at the same time. Inequalities (1) and (2) derive from the requirement $|f_k(t_k) - f_l(t_l)| \leq$ K and the definition of absolute value. Inequalities (3) and (4) result from the condition $|t_k - t_l| \leq w$ of the window join constraint and the definition of absolute value. Finally, (5), (6), (7) and (8) are conditions about the IAPFs of tuples tp_k and tp_l (a prediction function cannot be applied at a time instant out of its $IAPF$ or after the maximum update period).

The previous system can be easily solved graphically, as only two variables are involved. As a result, we obtain the timestamp-matching boundary and the validity interval (see Section 4.1.1). The timestamp-matching boundary is the feasible region obtained by graphical resolution (see Figure 5 for an example). The validity interval is [MIN(t_k.start, t_l.start), MAX(t_k.end, t_l.end)], where [t_k.start, t_k.end] and [t_l.start, t_l.end] are the ranges of admissible values for t_k and t_l respectively.

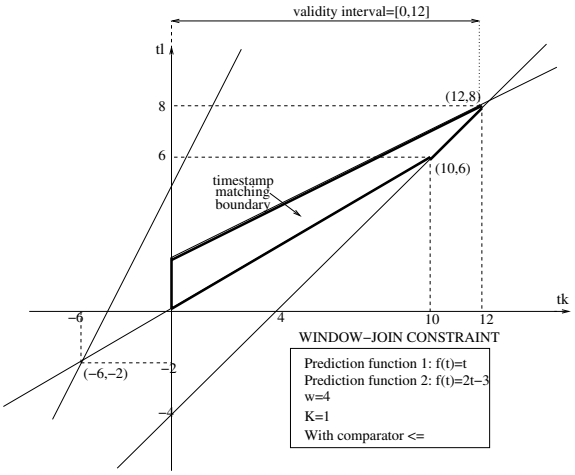

Fig. 5. Timestamp-matching boundary

While we have focused on the $\leq$ comparator, our strategy applies similarly for the *rest of comparators* by considering the linear constraints that derive from them. For example, with the $\geq$ comparator the constraint about the values in a window-join constraint $|f_k(t_k) - f_l(t_l)| \geq K$ transforms into two alternative constraints: $f_k(t_k) - f_l(t_l) \geq K$ or $f_l(t_l) - f_k(t_k) \geq K$. The original constraint satisfies if and only if *any* of these constraints do. Therefore, we have to solve two systems of linear inequalities instead of one, and there could be two validity intervals for the resulting predicted tuple[4]. For the "strict" comparators $<$ and $>$, edges of the timestamp-matching boundary that are defined by strict linear constraints must be appropriately flagged, as they contain points that are not valid combinations of t_k and t_l. The comparator $=$ is the conjunction of $>=$ and $<=$, and $\neq$ is the disjunction of $<$ and $>$.

Finally, we would like to mention that values predicted using the prediction functions of tuples are subject to a certain *uncertainty*, given by the threshold policy update (see Section 2.3). This implies that the real value associated to a certain predicted value will be within a segment of length twice the uncertainty and centered in the predicted value. Adapting the proposal to process queries with *must and may semantics* [21] is omitted here due to the lack of space. This semantic differentiation can be of great interest in sensor networks. For example, we may want to know if the temperature increases above a certain level with logging purposes (must) or detect the possibility of values of a dangerous chemical substance above a certain level (may).

[4]For simplicity, our prototype generates two predicted tuples, each one with a single validity interval.

6. EXPERIMENTS

In this section we show some preliminary results obtained using our prototype, in order to test the feasibility and performance of our approach.

6..1. Experimental Settings

We consider a scenario where the sensors are location devices attached to objects moving at 60 mph (about 26.8 m/s). These objects update their data to a centralized computer by considering a five-meter threshold (unless specified otherwise) and a three-minute maximum period between updates from each moving object. We run monitoring tests for three minutes and sample the output data stream every second. The query evaluated retrieves pairs of objects within 80 meters of each other. To make such query suitable to the window-join constraints described in this paper, we consider the Manhattan distance instead of the euclidean distance.

The experiments were performed on an Intel Pentium 4, CPU 1.70GHz, and RAM 256 Mb, with Red Hat Linux 7.3 (kernel 2.4.18). For simulation purposes, we have defined a set of trajectories that are processed using the IBM *Location Transponder* [17], which assigns threads as needed to meet the deadlines of each location update. We defined a set of nine non-straight trajectories to assign to the moving objects; for experiments with a higher number of objects, we offset the defined trajectories both vertically and horizontally by amounts of 100 meters as needed, such that there are no two objects with the same trajectory. Lastly, we would like to mention that our current prototype does not use any specialized solver for linear equations; therefore, the processing time per comparison could possibly be reduced by using an optimized implementation.

6..2. Scalability of the Number of Moving Objects

In Figure 6 we show the impact of the total number of moving objects in the total processing time of all the input tuples both in the case of using or not an indexing mechanism during the test. In our prototype, we use an R-tree[5] with fill factor 0.4 and a capacity of five branches per node. We have decided to index only the x and y coordinates of the moving objects (we leave aside the time), which is the best choice in our scenario, according to the results that we will present in Section 6.3.

The contribution to the processing time due to maintenance tasks of the R-tree (insertions and removals into/from the index) is also shown in the figure. We can see how the use of an index structure clearly outperforms a strategy where no index is used, and therefore we will use an index in the rest of our experiments. Only for a small number of moving objects (under 30) the overhead of maintaining the R-tree does not pay off. A total processing time above three minutes is indicated by values that do not fit in the figure[6].

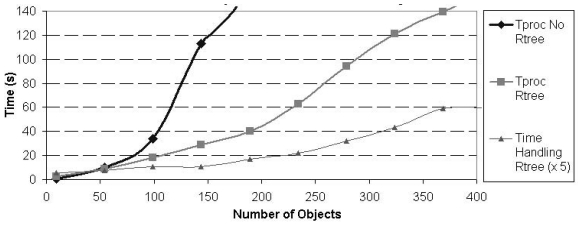

Fig. 6. Effect of the number of objects in the total processing time

As we can see in the figure, the system exhibits an acceptable scalability when the number of moving objects increases, although unfortunately the increase in the processing time is not linear. Notice, however, that we are considering a worst-case situation in the sense that all the input tuples are of the type of value in which the query is interested (horizontal or vertical location). This implies that any new tuple received can potentially match with any other tuple already present in the buffer. The use of the R-tree however helps us to reduce the great number of comparisons needed. Also, as tested in Section 6.4, the threshold of the update policy has a significant impact on the processing overload.

6..3. Impact of the Dimensionality of the R-tree

We show in Figure 7 the impact of the dimensionality of the index structure used for the experiments. We consider a choice between a two dimensional index structure (indexing the x and y coordinates) and a three dimensional one (indexing the x and y coordinates, and also the time). The figure shows that a two dimensional index achieves better results in our test scenario, and therefore we have used this index structure in our experiments.

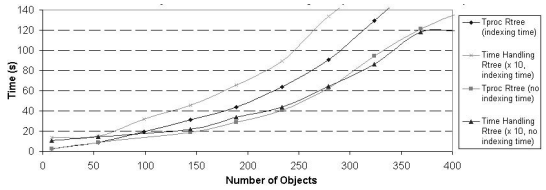

Fig. 7. Impact of the R-tree dimensionality

6..4. Impact of the Threshold of the Update Policy

In Figure 8 we show the impact of the threshold considered by the update policy. Thus, we can see how

[5]Specifically, we use the the Java library 0.44.2b developed by Marios Hadjieleftheriou (http://www.cs.ucr.edu/ marioh/spatialindex/).

[6]For these cases, there was not enough time to process all the input tuples during the monitoring interval!

increasing the threshold decreases the total processing time of the input tuples. This is because a smaller number of updates are sent by the moving objects, and this in turn reduces the number of comparisons of prediction functions needed to determine if two tuples match. In other words, by increasing the threshold of the update policy we trade processing cost for precision. Notice also that the reduction in the processing time is not linear with the increase in the update threshold; for example, it makes a big difference if the moving objects allow a location error of 545 meters instead of just 45, but between 2045 and 2545 the number of updates is similar.

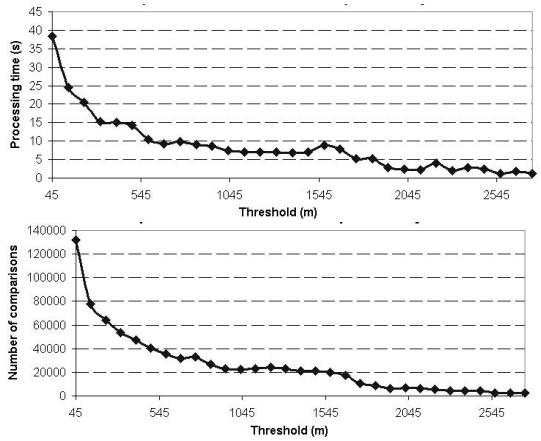

Fig. 8. Impact of the threshold of the update policy

6..5. Impact of the Scenario Size

In Figure 9 we show the impact of the scenario size when there are 180 moving objects in the scenario. We vary the size of the scenario by modifying the offset among the trajectories of moving objects; for example, the size of the scenario is 75 squared kilometers when the offsets among trajectories is 30 meters, and it is 240 squared kilometers for an offset of 70 meters. The figure shows how an increase in the scenario size (with the given distribution of trajectories) leads to a smaller processing time. This is because the R-tree filters out a greater number of tuples as possible matching candidates. Similar to what happened in the experiment of Section 6.4, however, the decrease is not linear.

7. Related Work

It has been proposed in the literature to add some semantic information to data streams in order to be able to process certain queries. Thus, a punctuation [19] is a pattern that describes a substream in a data stream (e.g., "values smaller than 5"), and it allows the evaluation of blocking and unbounded stateful operators (e.g., join and sort) over data streams. On the contrary, we use

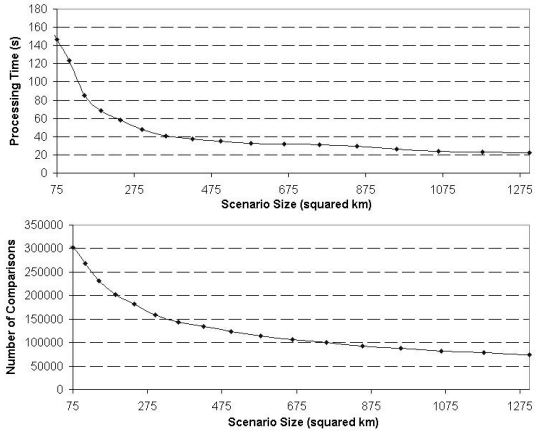

Fig. 9. Impact of the scenario size

prediction functions for performance purposes and focus on window-joins.

There are a wide variety of works on data stream processing (see, for example [2]). In the following, we will just describe some works which, as ours, propose to use predictions in the context of data streams.

The most relevant work is [15], as they also propose to use predictors to reduce the communication costs in a sensor network. They study several prediction techniques, and all of them are based on linear functions. A similar idea is proposed in [12], where they specifically choose a Kalman filter among the linear estimation methods. However, these works do not focus on query processing issues. By considering linear prediction functions, we allow an efficient join processing of values within a certain time window.

In [6] their basic assumption is that in many contexts the readings of nearby sensors are correlated, and propose to analyze the spatio-temporal correlation to compute prediction models in networks with static sensors. However, they do not study how to process efficiently the data received from sensors to answer different types of queries. Therefore, their work is complementary to ours: we could use their ideas to obtain prediction functions in suitable contexts, and they could use our work to process queries over the data obtained by using their prediction models. A similar idea to [6] is presented in [9] but they focus on groups of sensors. Finally, [5] also exploits the spatio-temporal correlations among readings of sensors: some sensors do not communicate their values but they are estimated from those of the rest of the sensors by using a linear model.

Another complementary work is [11], where Kalman filters are used to adjust dynamically the sampling rate of sensors: sensors for which the prediction error is higher will sample at a higher rate. The main disadvantage of

this approach is that it cannot detect unexpected events between samplings.

In [10] they use time window constraints to limit sets of tuples of different sensors that can be matched for a query and propose a mechanism to process multi-way joins. However, they focus on a different problem (tracking the motion of a moving object) and their technique requires the specification of the names of the individual sensors involved in the join, so they must be known in advance.

Finally, some works such as [1], [4] deal with the event/outlier detection problem. As opposed to these works, we propose an architecture for the monitoring of different types of queries that can be useful for a variety of applications. Thus, the output of our system could be used, for example, for offline data mining with the goal of detecting common patterns.

8. Conclusions and Future Work

In this paper we have described a framework for the efficient processing of data streams with prediction functions in a network of moving sensors. The use of prediction functions allows us to minimize communications from the sensors, which is an important concern due to energy and bandwidth limitations, and it also allows an efficient query processing.

While the use of predictions has been already proposed in the literature of data streams, and some works such as [15], [12] compare different prediction techniques, as far as we know no other paper focuses on query processing aspects as we do. Our incremental processing approach detects all the answers, adapts to different types of clients, and allows the processing of predictive queries. We consider two types of interesting constraints, and we focus mainly in window-join constraints that, up to our knowledge, has not been considered so far in other works. We have implemented our architecture and show some experimental results showing the performance and scalability of our approach.

As future work, we plan to conduct a more extensive experimentation and analyze the precision/threshold relation and the update cost in more depth. Adapting the Buffer Manager to deal with memory space constraints is an open issue. Another particularly interesting area is to analyze how to extend our ideas to a distributed context where sensors communicate their values to different computers.

Acknowledgements

Research supported by grant B132/2002 of the Aragón Government and the European Social Fund, NSF grants 0326284, 0330342, ITR-0086144, 0513736, 0209190, EIA-0320956, EIA-0220562, HRD-0317692, CICYT project TIC2001-0660, DGA project P084/2001. We thank A. Prasad for his useful comments.

References

[1] Swaroop Appadwedula, Venugopal V. Veeravalli, and Douglas L. Jones. Energy-efficient detection in sensor networks. *IEEE Journal in Selected Areas in Communications*, 23(4):693–702, April 2005.

[2] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 1–16. ACM Press, 2002.

[3] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. In *ACM SIGMOD*, volume 30, pages 109 – 120. ACM Press New York, NY, USA, September 2001.

[4] Mark Dilman and Danny Raz. Efficient reactive monitoring. In *INFOCOM*, pages 1012–1019, 2001.

[5] F. Emekci, S.E. Tuna, D.Agrawal, and E.Abbadi. Binocular: A system monitoring framework. In *International Workshop on Data Management for Sensor Networks*, August 2004.

[6] Samir Goel and Tomasz Imielinski. Prediction-based monitoring in sensor networks: Taking lessons from MPEG. *ACM Computer Communication Review*, 31(5), October 2001.

[7] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. In *ACM SIGMOD*, volume 32, pages 5–14, June 2003.

[8] Mohinder S. Grewal and Angus P. Andrews. *Kalman Filtering: Theory and Practice*. Prentice Hall, February 1993.

[9] Carlos Guestrin, Peter Bodik, Romain Thibaux, Mark Paskin, and Samuel Madden. Distributed regression: an efficient framework for modeling sensor network data. In *Third International Symposium on Information Processing in Sensor Networks (ISPN)*, May 2004.

[10] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. *15th International Conference on Scientific and Statistical Database Management (SSDBM 2003)*., 2003.

[11] A. Jain and E. Chang. Adaptive data sampling for sensor networks. In *International Workshop on Data Management for Sensor Networks*, August 2004.

[12] A. Jain, E. Y. Change, and Y. Wang. Adaptive stream resource management using Kalman filters. In *ACM SIGMOD International Conference on Management of Data*, June 2004.

[13] Christine E. Jones, Krishna M. Sivalingam, Prathima Agrawal, and Jyh Cheng Chen. A survey of energy efficient network protocols for wireless networks. *Wirel. Netw.*, 7(4):343–358, 2001.

[14] K. Kowalski and W-H. Steeb. *Nonlinear Dynamical Systems and Carleman Linearization*. World Scientific Publishing Company, Incorporated, March 1991.

[15] V. Kumar, B.F. Cooper, and S.B. Navathe. Predictive filtering: A learning-based approach to data stream filtering. In *International Workshop on Data Management for Sensor Networks*, August 2004.

[16] Jeremy Miles and Mark Shevlin. *Applying Regression and Correlation : A Guide for Students and Researchers*. SAGE Publications, March 2001.

[17] Jussi Myllymaki and James H. Kaufman. Location transponder, April 2002. http://www.alphaworks.ibm.com/tech/transponder, [Accessed September 9, 2005].

[18] Yufei Tao and Dimitris Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD Conference*, 2002.

[19] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Punctuating continuous data streams. Technical report, Oregon University, 2001.

[20] O. Wolfson, A. Prasad Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *invited paper, special issue of the Distributed and Parallel Databases Journal on Mobile Data Management and Applications*, 7(3):257–287, 1999.

[21] Ouri Wolfson, Sam Chamberlain, Son Dao, Liqin Jiang, and Gisela Mendez. Cost and imprecision in modeling the position of moving objects. In *ICDE*, pages 588–596, 1998.