A Query Processor for Prediction-Based Monitoring of Data Streams

Sergio Ilarri
IIS Department
Univ. of Zaragoza
María de Luna 1
50018 Zaragoza, Spain
silarri@unizar.es

Ouri Wolfson
CS Department
Univ. of Illinois
Chicago, IL, 60607
wolfson@cs.uic.edu

Eduardo Mena
IIS Department
Univ. of Zaragoza
María de Luna 1
50018 Zaragoza, Spain
emena@unizar.es

Arantza Illarramendi
LSI Department
Univ. of the Basque Country
Apdo. 649
20080 San Sebastián, Spain
a.illarramendi@ehu.es

Prasad Sistla
CS Department
Univ. of Illinois
Chicago, IL, 60607
sistla@cs.uic.edu

ABSTRACT

Networks of sensors are used in many different fields, from industrial applications to surveillance applications. A common feature of these applications is the necessity of a monitoring infrastructure that analyzes a large number of data streams and outputs values that satisfy certain constraints.

In this paper, we present a query processor for monitoring queries in a network of sensors with prediction functions. Sensors communicate their values according to a threshold policy, and the proposed query processor leverages prediction functions to compare tuples efficiently and to generate answers even in the absence of new incoming tuples. Two types of constraints are managed by the query processor: window-join constraints and value constraints. Uncertainty issues are considered to assign probabilistic values to the results returned to the user. Moreover, we have developed an appropriate buffer management strategy, that takes into account the contributions of the prediction functions contained in the tuples. We also present some experimental results that show the benefits of the proposal.

1. INTRODUCTION

There has been a great interest in techniques for monitoring networks of sensors in a variety of contexts (e.g., fleet tracking, monitoring the levels of certain gases in a chemical environment, etc.). 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*) [3], in a *monitoring computer*, which implements suitable non-blocking techniques to process unbounded amounts of data.

In this context, sensors send their data as continuous and independent streams of data. As the values they measure can change very frequently (e.g., a GPS receiver in a car measures its location, that is changing constantly), this could imply *many communications of data from each sensor*. On the one hand, this implies that the DSMS has to process a large amount of incoming tuples. On the other hand, there is a great usage of the network, which is specially important in wireless environments where communications are expensive and quickly drain the energy of wireless devices [22].

However, in many situations, 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. Taking into account this situation, this work proposes to associate each sensor value with a prediction function in a way that the sensor will update its value on the monitoring computer only when it differs significantly from the predicted value. This strategy reduces both the communication and the query 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 no new tuple is received from any of the sensors. The use of prediction functions requires the definition of a suitable query processor able to handle them efficiently and effectively. The main contributions of our work are:

- A query processor is defined, that processes queries on data streams with prediction functions.
- Two types of constraints are considered: value constraints and window-join constraints. In particular, window-join constraints have not been studied in other works.

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.

- Uncertainty issues are considered and managed by the query processor.
- An appropriate buffer management is performed to deal with storage limitations.
- An experimental study shows the interest of the proposal in a variety of conditions.

The rest of this paper is as follows. In Section 2, we describe the use of prediction functions and the format of tuples considered. In Section 3, we explain the types of constraints processed by the proposed query processor: value constraints and window-join constraints. In Section 4, we detail the main components of the query processor. In Section 5, we explain how the uncertainty in the values measured by the sensors is managed. We present an experimental evaluation showing the feasibility and performance of the query processor in Section 6. The paper finishes with some related works in Section 7 and conclusions and future work in Section 8.

2. SENSORS AND PREDICTIONS

This paper argues that 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 [35]. 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 examples of values that can be estimated with a prediction function (which can be built using different techniques [15, 27]).

Thus, in our proposal, every value measured by a sensor is attached to a prediction function that will be used to predict future values of that sensor. In this way, the sensor will only send significant values (e.g., values that differ from the predicted value by more than a certain *threshold*) 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 [22]. 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.

Data streams from sensors are composed of *update-tuples*: tp_j=<s_i, $type_i$, ts_j, $f_j(t)$>, where s_i is the identifier of a sensor (assumed unique, as in many other works on query processing in sensor networks, such as [5, 12, 13, 26]), $type_i$ is the type of value it measures, ts_j is the (implicit or explicit [33]) timestamp of the update, and $f_j(t)$ is a prediction function that, given a certain time instant t, retrieves the expected sensor value at that time. The value of the sensor at the update time is given by $f_j(ts_j)$. It should be noted that sometimes several prediction functions can be included in the same tuple (e.g., in the experiments presented in Section 6, the sensors measure two-dimensional values, corresponding to the horizontal and vertical coordinates of moving objects). As an example, <locCar20, location, 10, {15 + 4.5t, 3t}> is a tuple sent by a location sensor (identified by $locCar20$) in $car20$, moving with a horizontal speed of 4.5 m/s and a vertical speed of 3 m/s, whose location a $t = 10$ is $(60, 30)$. This function can be maintained by some dead-reckoning method [34]: each vehicle sends its location and current speed to the monitoring computer, and the speed and current location are sent again when the deviation between the actual location and the location estimated using the predicted function exceeds a threshold.

For simplicity, we consider that the sensors can estimate future values based on past measurements. So, a sensor can obtain and communicate a prediction function to the monitoring computer. Although there may be some processing cost associated to the estimation of prediction functions, its use will save communications by the sensors, which is a key element to reduce energy consumption. Moreover, the proposed query processor does not contradict proposals where a prediction model is computed and assigned to the sensors by a third party, such as [14]. Indeed, this is required when the prediction model is based on data not available at the sensor (e.g., based on values measured by sensors nearby).

In order to predict the value of a sensor at a given time instant, the last prediction function received from the sensor before that time instant must be applied. 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 is called the *Interval of Applicability of the Prediction Function*: $IAPF_{tp_k}$=$[IAPF_{tp_k}.start, IAPF_{tp_k}.end)$.

Sensors can follow a number of *update policies* [34] in order to decide when a value is significant and should be communicated to the monitoring computer. We advocate *a threshold policy with a maximum period between updates*. With this policy, every sensor commits to communicate an update whenever: 1) the difference between its current value and 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 two Celsius degrees with at least one update every three minutes. The *update period* T indicates the maximum 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*). Appropriate threshold values can be specified depending on the Quality of Service (QoS) requirements. Moreover, the threshold update policy could also be adaptive, i.e., a sensor could dynamically adjust its threshold as a result of an assessment of the tradeoff between the communication cost and the cost of the imprecision of predictions [35] (in this case, the format of tuples should be extended with the value of the threshold considered by the sensor at the timestamp of the tuple).

3. TYPES OF CONSTRAINTS

Two types of query constraints, which may appear simultaneously in a query, are considered by the proposed query processor: *value constraints* and *window-join constraints*.

Constraints such as "the selected temperature sensors must measure a temperature under $50F$ degrees" are called *value constraints* and are represented by *vConstr(type, comp, K)*, where *type* is a type of sensor value, *comp* is a comparator among $\leq$, $<$, $>$, $\geq$, $<>$, and $=$, and K is a constant. The sample constraint above would be expressed as *vConstr(temperature, <, 50F)*.

Window-join constraints such as "the selected pairs of gas sensors must measure a similar concentration of carbon dioxide within an interval of 10 seconds" are more complex and are represented by *wJConstr(type$_1$, type$_2$, w, comp, K)*, where *type*$_1$ and *type*$_2$ are two types of sensor values, w is called the *valid-time window* and specifies the maximum time difference allowed between two values to be joined, and the rest of parameters are as explained for value constraints. The sample constraint above would be expressed, using this syntax, as *wJConstr(CO_2, CO_2, 10 seconds, >, 1%)*, considering similar values those that 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 sensors' clocks are not precisely synchronized), and 2) also in some specific queries (e.g., retrieving pairs of buses that arrive at the same stop within 30 minutes of each other). Other scenarios where window-join constraints can be useful include: in network data processing to detect accesses from the same IP at approximately the same time (it may indicate a *DOS* attack) or other traffic patterns that occur simultaneously, in financial applications to find trends when comparing stock quotes, to compare segments of trajectories of objects which move unsynchronizely, to monitor credit card transactions, etc.

4. QUERY PROCESSOR MODULES

In this section, we describe the components of the proposed query processor (shown in Figure 1): the Tuple Evaluator, the Buffer Manager, and the Prediction Validator. At the end of the section, we also explain how they interact.

4.1 Tuple Evaluator

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, it must verify how that tuple affects the active queries (i.e., the continuous queries in execution). 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 type 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.

The comparison of tuples for window-join constraints is performed in two stages: a filter step and an evaluation step. In the *filter step*, possible matches are filtered out based on the idea that two predicted values cannot match if they are not comparable due to the difference in their timestamps. For those pairs of tuples that are not filtered out, the window-join constraint must be evaluated over them to determine when they will match (if ever). In the *evaluation step*, *linear prediction functions* are considered[1], so that the problem of processing window-join constraints translates to comparing prediction functions by *solving a system of linear inequalities*, which can be performed efficiently. The corresponding system of linear inequalities for window-join constraints and a couple of prediction functions $f_k(t_k) = a \cdot t_k + b$ and $f_l(t_l) = c \cdot t_l + d$ is shown in Figure 2, where T is the period of the threshold update policy (see Section 2). A graphical resolution of the system of linear inequalities is possible, by representing each constraint with a line, as shown in the example of Figure 3. As a result of the comparison of two tuples, predicted tuples that will satisfy the constraint are generated. The cost of processing an incoming tuple for a window-join constraint obviously depends on the number of tuples stored for the other sensor class involved in the join and, especially, on the number of potential matches that pass the filter step; for each non-filtered potential match, a system of linear inequalities like the one shown in Figure 2 must be solved. For more details about this process, the interested reader is referred to [18, 19].

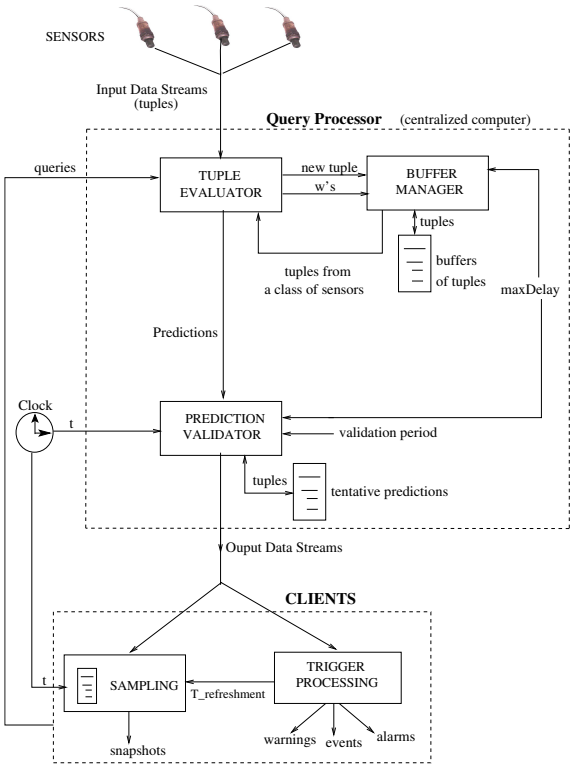

Figure 1: Modules of the query processor

For each active query, the Tuple Evaluator 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 is explained in the following, depends on whether the query includes a window-join constraint or not.

[1]Many well-known estimation techniques, such as the linear extrapolation, the double exponential smoothing, or the Kalman filter, are linear models. Moreover, non-linear functions can be linearized in many cases [23].

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

Figure 2: Linear inequalities for window-joins

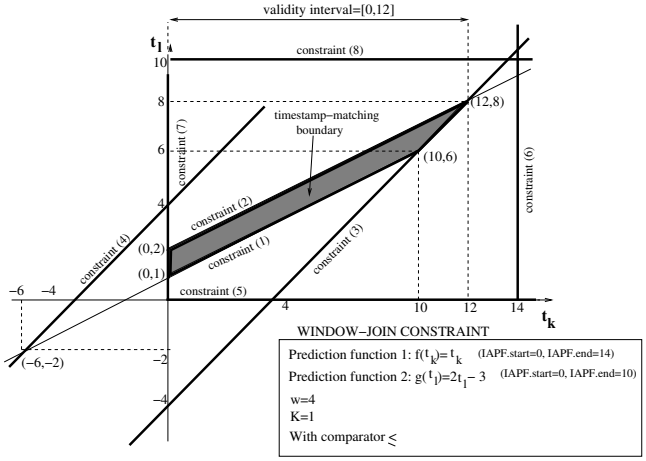

Figure 3: Timestamp-matching boundary

For queries that include a constraint about pairs of sensors, the format of an output predicted tuple is: <tp_i, tp_j, 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 the prediction functions of tp_i and tp_j can be used to estimate the values of the corresponding sensors, values that will satisfy the constraint. The validity mark can be seen as a generalization of the idea of validity period presented in [32], and it has two components:

- A *validity interval* I. It 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. It is a polygon that constraints the admissible combinations of times t_i and t_j for a match (as shown in Figure 3), where t_i is a time instant for the evaluation of the prediction function $f_i(t)$ in tp_i, and t_j is an evaluation time instant for $f_j(t)$ in tp_j. This can be used, for example, to find examples of values that match. More importantly, it is used to validate predicted tuples (see Section 4.3).

If the query consists only of a value constraint, the format of predicted tuples is: <tp_i, VM=I>, where the validity mark is given just by the validity interval I. For example, a tuple <s_1, $type_1$, t_1, 3t+2, [5,15]> indicates that the sensor s_1 satisfies the constraint during the interval [5,15] by considering the prediction function $3t + 2$ (with timestamp $t1$).

4.2 Buffer Manager

The Tuple Evaluator communicates the tuples it receives to a module that is called the *Buffer Manager*. The Buffer Manager decides which input tuples will be stored (because 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 asks 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. In the following, we first explain the basic mechanism to decide whether it is interesting to store a tuple or not. Then, we explain the policy applied when there is insufficient storage space for all the tuples.

4.2.1 Temporal Width of a Buffer

The tuples that should be stored in the buffers are determined by the constraints of the active queries. 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 $IAPFs$ intersecting with a given time interval need to be stored. Thus, storing more tuples would imply both a waste of storage space and a higher join processing cost. The temporal width of the buffer for the type of value S is given by:

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

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 an estimation of the maximum delay of input tuples (i.e., a tuple with timestamp t can arrive in the query processor as late as $t+maxDelay$). A tuple tp_k for the type of value S must be kept in the buffer only if:

$$([currentTime - tWidth(S), currentTime] \bigcap IAPF_{tp_k}) \neq \emptyset$$

since, as mentioned in Section 2, a tuple is not only applicable at its timestamp but at any time instant within its $IAPF$.

The Buffer Manager will periodically shift the windows of tuples stored in its buffers (removing the unneeded tuples according to the previous considerations) in a process called *purging*. It must also manage problems derived from the lack of space to store new tuples. If there is not enough space to store an incoming tuple, purging is automatically activated. If purging does not solve the problem, the relative importance of the prediction functions of the different tuples is considered by following an appropriate replacement policy. This issue is explained in more detail in the following.

4.2.2 Dealing with Limited Storage

In a real environment, there will probably be a certain limited amount of space available for tuples in the buffers. As

the rate of arrival of new tuples is unbounded, it is possible that some tuples that should be placed in the buffers (according to the previous explanation) cannot be physically stored.

To deal with this problem, we propose a method that trades the need of additional storage space for a greater inaccuracy (which is unavoidable because of the storage limitation that prevents the storage of all the tuples needed). In the event that the maximum allowed space for a buffer is reached, the Buffer Manager decides if a new incoming tuple should replace another tuple in the buffer or be discarded. For that purpose, the *contribution* of each of the tuples stored in the buffer, and also of the new tuple, is considered. In this way, the tuple with the smallest contribution is discarded. Although several methods have been proposed to deal with storage limitations in the context of data streams [3] (e.g., random sampling, histograms and wavelets), none of them considers the peculiarities of data streams with prediction functions.

The proposed buffer replacement policy is based on the idea that each tuple contributes to decrease the error of the sensor values predicted, as it provides an updated prediction function. Thus, each tuple tp_k can be associated to a value that indicates its *contribution* C_{tp_k}. The contribution of a tuple tp_k from sensor s_i is the decrease in the error committed when predicting values of s_i in the interval $IAPF_{tp_k}$ by using the prediction function in tp_k instead of a prediction function from a tuple with a contiguous $IAPF$. Thus, several alternatives to predict values for the time interval $IAPF_{tp_k}$ are possible if tp_k is removed (depending on whether a previous tuple tp_{k-1} and/or a next tuple tp_{k+1} from the same sensor is available or not):

- If there is both a previous tuple tp_{k-1} and a next tuple tp_{k+1} stored in the buffer, then there are two possibilities to choose:

  1. The prediction function $f_{k-1}(t)$ of the previous tuple tp_{k-1} could be applied within $IAPF_{tp_k}$. In that case, the $IAPF_{tp_k-1}$ should be modified as follows:

     $$IAPF_{tp_k-1}=[IAPF_{tp_k-1}.start, IAPF_{tp_k}.end)$$

     By applying the out-of-date prediction function $f_{k-1}(t)$ in tp_{k-1} instead of the updated prediction function $f_k(t)$ in tp_k, a certain error e_k^{k-1} arises, whose accumulated value can be computed as the absolute value of the definite integral of the difference of the prediction functions within $IAPF_{tp_k}$:

     $$e_k^{k-1} = \left| \int_{IAPF_{tp_k}.start}^{IAPF_{tp_k}.end} f_k(t) - f_{k-1}(t)\, dt \right|$$

  2. The prediction function $f_{k+1}(t)$ of the next tuple tp_{k+1} could be applied within $IAPF_{tp_k}$. In that case, the $IAPF_{tp_{k+1}}$ should be modified as follows:

     $$IAPF_{tp_{k+1}}=[IAPF_{tp_k}.start, IAPF_{tp_{k+1}}.end)$$

     As the prediction function $f_{k+1}(t)$ in tp_{k+1} is used to estimate values in $IAPF_{tp_k}$ (a past time interval), instead of using the prediction function $f_k(t)$, an error e_k^{k+1} arises, which can be computed similarly to the previous case:

     $$e_k^{k+1} = \left| \int_{IAPF_{tp_k}.start}^{IAPF_{tp_k}.end} f_k(t) - f_{k+1}(t)\, dt \right|$$

- If there is a next tuple tp_{k+1} but not a previous tuple tp_{k-1}, then the next tuple tp_{k+1} could be applied within the $IAPF_{tp_k}$, as explained above.
- If there is a previous tuple tp_{k-1} but not a next tuple tp_{k+1}, then the previous tuple tp_{k-1} could be applied within the $IAPF_{tp_k}$, as explained above.
- Finally, if neither a previous nor a next tuple is available in the buffer, then it is not possible to predict values within the $IAPF_{tp_k}$ (unless tp_k is stored).

Based on the previous ideas, a *priority queue* is maintained for the tuples in the buffers, where the priority of a tuple is the inverse of its contribution. Thus, the tuple at the head of the queue (one with the smallest contribution) is the candidate to be removed in case of insufficient storage. According to the definition of contribution indicated before and the previous considerations, the contribution of a tuple tp_k is given by:

$$C_{tp_k} = \begin{cases} MIN(e_k^{k-1}, e_k^{k+1}) & if\ (\exists\ tp_{k+1}\ ) \wedge (\exists\ tp_{k-1}) \\ e_k^{k+1} & if\ (\exists\ tp_{k+1}) \wedge (\neg\exists\ tp_{k-1}) \\ \infty & otherwise \end{cases}$$

where an infinite contribution is assigned to a tuple if it is the only tuple stored for the corresponding sensor (intuitively, if that tuple is removed then there is no way to predict the value for its sensor, as no other prediction function for that sensor is available). A tuple with an infinite contribution can only be selected as a victim to be removed from the buffer in case there is no other tuple with a smaller contribution (the selection among several tuples with infinite contribution is random). As shown in the previous formula, the contribution of a tuple tp_k is also set to infinite when it is the last tuple from that sensor (i.e., $\neg\exists\ tp_{k+1}$), even though (as it has been indicated before) the previous tuple could be applied within the $IAPF$ of such a tuple. The reason is that the tuple with the greatest timestamp from each sensor should be kept if possible in the buffer; otherwise, the precision of future estimations will degrade. Thus, it should be noted that if the last tuple from a sensor is removed and the previous tuple is extended to cover its $IAPF$, the prediction function of such a previous tuple will be the one considered to compute the contribution of a new future incoming tuple (this is not precise but there is no other choice). By assigning an infinite contribution to the latest tuple from each sensor, this problem is minimized since such tuples will be stored if possible.

As the contribution of a tuple depends on its previous and/or next tuple, the contribution is not fixed. On the contrary, it changes along time when new tuples are inserted or old tuples are removed:

- Case 1: a new tuple tp_k is inserted into the buffer. In this case, the contribution C_{tp_k} of the new tuple must be computed. In case there is a previous tuple tp_{k-1} (from the same sensor) in the buffer, its contribution $C_{tp_{k-1}}$ must also be updated (since the computation of the contribution of a tuple must consider the prediction function of its next tuple –if any–, as explained before). Finally, if there is a next tuple tp_{k+1} (from the same sensor) its contribution must also be updated (as the contribution of a tuple may be affected by the previous tuple).
- Case 2: a tuple tp_k is removed from the buffer. If there exists a previous tuple tp_{k-1} but not a next tuple tp_{k+1} (for the same sensor) in the buffer, then only the previous tuple can "cover the gap" and therefore the $IAPF_{tp_{k-1}}$ is extended by considering the $IAPF_{tp_k}$. Similarly, if there exists a next tuple tp_{k+1} but not a previous tuple tp_{k-1} in the buffer, then the $IAPF_{tp_{k+1}}$ is extended to cover the $IAPF_{tp_k}$. If there is both a previous tuple and a next tuple in the buffer, the one considered to be applied within the $IAPF_{tp_k}$ is the one that minimizes the accumulated error caused by the difference in the prediction functions (the e_k^{k-1} and e_k^{k+1} errors). After that, the contributions of the previous and/or the next tuple ($C_{tp_{k-1}}$, $C_{tp_{k+1}}$) must be recomputed. It should be noted that this whole process is only applied when a tuple is removed due to insufficient storage space. Thus, if a tuple is removed due to purging (i.e., because it is not needed anymore), then only the next tuple must be checked to update its contribution.

By recomputing the contributions as needed, the Buffer Manager keeps up-to-date the information needed to select, as a victim to be evicted from a buffer, the tuple whose removal is expected to have the smallest impact on the accuracy of the query processor. The benefits of this buffer management policy are evaluated experimentally in Section 6.3, where other alternatives are also considered.

4.3 Prediction Validator

The Tuple Evaluator releases, for each query, predicted tuples about values that will satisfy the query. The communication of another prediction function by any sensor involved in an output predicted tuple could invalidate such predicted tuple. Therefore, the tuples obtained by the Tuple Evaluator are only *tentative* and will be considered *validated* only when there is a guarantee that they cannot be found to be wrong later[2]. A predicted tuple from the Tuple Evaluator can be invalidated due to two reasons: 1) if it will not be valid until a future time instant (according to the validity interval of the predicted tuple), then an updated prediction function from any sensor involved will invalidate the previous prediction function and, therefore, the predicted tuple; and 2) if a tuple that arrives late (i.e., it arrives "now" but it refers to a past time instant) invalidates the prediction.

A module called *Prediction Validator* can be plugged between the output of the Tuple Evaluator and the input of a client, with the goal of releasing only tuples that are guaranteed to be valid[3]. Prediction functions in a predicted tuple allow to get *predicted values* that satisfy the query constraints during the validity interval of the 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/validated $maxDelay$ time units after the timestamp of the prediction function that estimated it. Thus, the Prediction Validator checks the tuples it stores with a certain *validation period* (e.g., every second) to release *valid* tuples that can be inferred from them.

4.3.1 Validating Value Constraints

For predicted tuples with only one sensor identifier (i.e., corresponding to queries with a single value constraint), the validity mark of the predicted tuple is just appropriately modified. As an example, for a predicted tuple $<s_1$, $type_1$, ts, $3t+2$, $[5,20]>$, $maxDelay$=5, and the current time instant equals 15 time units, the tuple that would be released in the output data stream is $<s_1$, $type_1$, ts, $3t+2$, $[5,10]>$. If the validation period is 1 time unit, then the next tuple would be $<s_1$, $type_1$, ts, $3t+2$, $[5,11]>$ at time instant 16. If it is 2 time units, then it would be $<s_1$, $type_1$, ts, $3t+2$, $[5,12]>$ at time instant 17. A new tuple from sensor s_1 could invalidate the predicted tuple at any time before $t=20$ and stop the process.

4.3.2 Validating Window-Join Constraints

Predicted tuples corresponding to queries with a window-join constraint (i.e., tentative tuples with two sensor identifiers) are a little bit more difficult. Thus, the timestamp-matching boundary, which contains points (t_i,t_j) indicating the valid combinations of the timestamps for the sensors matched (as explained in Section 4.1), must be analyzed. In particular, the Prediction Validator must check if the square with vertices $(t'-w,t'-w)$, $(t'-w,t')$, $(t',t'-w)$, (t',t'), where t' is the considered time instant (the current time instant minus $maxDelay$, since the Prediction Validator releases at time $t+maxDelay$ a tuple predicted valid at time t) and w is the valid-time window, intersects the timestamp-matching boundary. If there is an intersection, then one sensor at t' matches the value of the other sensor at a time instant t'', with $t'-w \leq t'' \leq t'$. Notice that it is not enough to check whether t' is within the timestamp-matching boundary, since the value at t' may match only with values of the other sensor at future time instants (therefore, not validated yet). In case of match, the output tuple is released with a validity interval (and timestamp-matching boundary) limited to that considered time instant t'.

As an example, tuples tp_1=$<s_1$, $type_1$, ts, $t>$ and tp_2=$<s_2$, $type_2$, ts', $2t-3>$ match according to the window-join constraint wJConstr($type_1$, $type_2$, 4, $\leq$, 1) at $t=0$; however, the value that matches for tuple tp_2 is one second into the future; if that match is released at $t=0$, a new prediction function from sensor s_1, that invalidates it, could be received in the meanwhile. So, checking the validity interval of a predicted tuple is not enough: the timestamp-matching

[2]Similar to the *potential answers* proposed in [25], which "may turn into *current answers* and be reported to the users".

[3]In some contexts, releasing predicted (non-validated) results could be interesting. Therefore, the Prediction Validator can be turned off.

boundary must be checked too, as we have explained.

4.4 Interactions between Modules

When the Tuple Evaluator receives a new tuple tp_1 from sensor $s1$, of a type of value S_1, 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 S_1, the Tuple Evaluator generates an *invalidation tuple* for sensor $s1$ and the time interval $IAPF_{tp_1}$, indicating that previous prediction functions for that sensor are not applicable anymore in that interval. It also generates new predicted tuples (if any), as explained in Section 4.1. A query could include several window-join constraints; for example, three sensor streams A, B, and C, related with two window-join constraints A-B and B-C can be processed by considering separately the joins A-B and B-C and then integrating the results. Possible optimizations for queries with multiple joins are left as future work.

The tuples generated by the Tuple Evaluator are received by the Prediction Validator. If the tuple received is an invalidation tuple for a sensor, then the Prediction Validator will update the validity mark (validity interval and timestamp-matching boundary) of predicted tuples involving that sensor, in order not to include the invalidation interval for that sensor (this could even invalidate the whole predicted tuple). If the tuple received is a predicted tuple, then the Prediction Validator will store it in the table of tentative tuples and release it as required (see Section 4.3). It should be noted that the Tuple Evaluator could release either just an invalidation tuple or also predicted tuples. The Prediction Validator also validates the tentative tuples according to the required validation period (as explained in Section 4.3).

5. MANAGING UNCERTAINTY

The evaluation step as it is presented in Section 4.1 assumes that there is no uncertainty in the predicted values. However, the values predicted using the prediction functions of tuples are subject to a certain *uncertainty*, given by the threshold update policy (see Section 2). This implies that if the value obtained using the prediction function is x and the threshold used for updates is δ, then the real value can be any value in the interval $[x-\delta, x+\delta]$. In Figure 4, an example of how the uncertainty determines the possible real values corresponding to a certain predicted value is shown. Considering the time interval between seconds three and four, it can be seen that the value of $f_k(t)$ will be in the range $[3.2, 7]$ while $f_l(t)$ will be in the range $[4.2, 7]$. However, with no uncertainty the ranges would be $[4.4, 5.4]$ and $[5, 5.4]$ respectively.

The proposed query processor supports uncertainty management. On the one hand, queries can be processed with different semantics to take uncertainty into account. On the other hand, a tuple in an answer can be tagged with the probability that the tuple satisfies the query constraints, according to the existing uncertainty. Moreover, the minimum probability required for a tuple to be shown in the answer can be also specified as a query condition. Although some works, such as [9], have studied probabilistic queries, we study uncertainty issues in the context of data streams with prediction functions and focusing on the new window-join

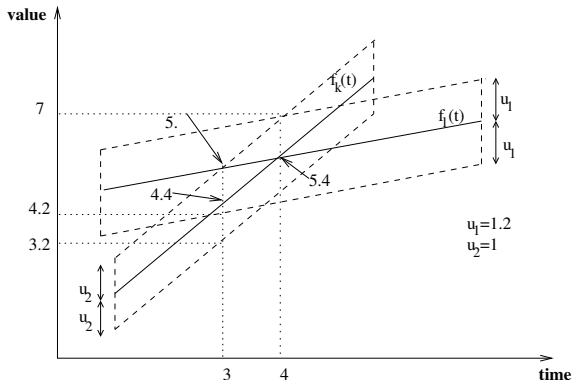

Figure 4: Uncertainty in the prediction functions

constraints. In the following, we explain how the uncertainty is managed by the query processor.

5.1 Query Semantics for Uncertainty

To take uncertainty into consideration, two different semantics are defined –*may* and *must*– for satisfaction of a query. For a given constraint, any of these two semantics could be required. Such semantics were first proposed in [31] in the context of moving objects and were also used in [34]. This semantic distinction can be of great interest in sensor networks. For example, we could need to know if the temperature increases above a certain level with logging purposes (*must*) or to detect the possibility of values of a dangerous chemical substance above a certain level (*may*).

In the following, we formally define the semantics of *may* and *must* satisfaction for window-join queries, and we show how the evaluation step (given in Section 4.1) can be modified to process the queries according to these two different semantics. The queries given by value constraints can be handled similarly under the two uncertainty semantics. Now, consider the following window-join constraint on two different sensors s_1, s_2: $|v_1 - v_2| \leq K$ and $|t_1 - t_2| \leq w$, where v_1, v_2 are the values of sensors s_1 and s_2 at times t_1 and t_2 respectively. Let $[l_1, u_1]$ and $[l_2, u_2]$ be the uncertainty intervals for the values of sensors v_1 and v_2 respectively (l_1 and l_2 are the lower uncertainty bounds and u_1 and u_2 are the upper uncertainty bounds). The values of sensor s_1 at time t_1 and the value of sensor s_2 at time t_2 *may* satisfy the above window-join constraint if $|t_1 - t_2| \leq w$ and $\exists\, x \in [l_1, u_1]$ and $\exists\, y \in [l_2, u_2]$ such that $|x - y| \leq K$. Note that this condition is equivalent to requiring a non-empty intersection between the intervals $[l_1 - K, u_1 + K]$ and $[l_2, u_2]$. The following lemma is easily seen from this observation.

Lemma 1: Assume that the uncertainty intervals for the values v_1 and v_2 of the sensors s_1 and s_2 at times t_1 and t_2 are given by $[l_1, u_1]$ and $[l_2, u_2]$, respectively. Then, they *may* satisfy the window-join constraint $|v_1 - v_2| \leq K$ and $|t_1 - t_2| \leq w$ iff $l_1 - u_2 \leq K$ and $l_2 - u_1 \leq K$ and $|t_1 - t_2| \leq w$.

In view of the above lemma, the evaluation step given in 4.1 is modified for checking *may* satisfaction as follows. Let δ_k and δ_l respectively be the thresholds used for the updates of the two sensors. Then, the intervals of uncertainties on the

predicted values of these two sensors are $[f_k(t_k)-\delta_k, f_k(t_k)+\delta_k]$ and $[f_l(t_l)-\delta_l, f_l(t_l)+\delta_l]$, respectively. Inequalities (1) and (2) in Section 4.1 (see Figure 2) are simply replaced by the following inequalities:

$$a \cdot t_k - c \cdot t_l + b - d - \delta_k - \delta_l \leq K$$
$$c \cdot t_l - a \cdot t_k + d - b - \delta_l - \delta_k \leq K$$

Now, the *must* satisfaction is defined. The value of sensor s_1 at time t_1 and the value of sensor s_2 at time t_2 *must* satisfy (or definitely satisfy) the previous window-join constraint if $|t_1 - t_2| \leq w$ and $\forall\, x \in [l_1, u_1]$ and $\forall\, y \in [l_2, u_2]$ it is the case that $|x - y| \leq K$. In order for the above condition to be satisfied it is necessary and sufficient that $u_2 \leq l_1 + K$ and $l_2 \geq u_1 - K$. Note that this condition implies that the lengths of the two intervals are bounded by $2K$. From these observations, the following lemma is obtained.

Lemma 2: Assume that the uncertainty intervals for the values v_1 and v_2 of the sensors s_1 and s_2 at times t_1 and t_2 are given by $[l_1, u_1]$ and $[l_2, u_2]$, respectively. Then, they *must* satisfy the window-join constraint $|v_1 - v_2| \leq K$ and $|t_1 - t_2| \leq w$ iff $|t_1 - t_2| \leq w$ and $u_2 - l_1 \leq K$ and $u_1 - l_2 \leq K$.

Inequalities (1) and (2) in Section 4.1 (see Figure 2) are simply replaced by the following inequalities for checking the *must* satisfaction:

$$c \cdot t_l - a \cdot t_k + d - b + \delta_l + \delta_k \leq K$$
$$a \cdot t_k - c \cdot t_l + b - d + \delta_k + \delta_l \leq K$$

5.2 Determining the Probability of a Match

Assuming a certain probability distribution of the real value $v_1 \in [l_1, u_1]$ of a sensor s_1 at time t_1, the probability of a match can also be computed. Let V_1 denote the random variable corresponding to the value of s_1 at time t_1. We assume that each value within the interval $[l_1, u_1]$ is equally possible for V_1. So, the real value of sensor s_1 is given by the random variable V_1 with the following density function:

$$P_{V_1}(v_1) = \begin{cases} \frac{1}{u_1 - l_1} & if\ l_1 \leq v_1 \leq u_1 \\ 0 & otherwise \end{cases}$$

In order to compute the probability of satisfaction of a window-join constraint for the values v_1 and v_2 of sensors s_1 and s_2 at time instants $t1$ and $t2$, v_2 is considered to be a random variable V_2 whose density function has value $\frac{1}{u_2 - l_2}$ within the range $[l_2, u_2]$ and zero otherwise. Now, a random variable $Z = V_1 - V_2$ is defined. Assuming that the variables are independent, the density function for this new variable is given by the convolution:

$$P_Z(z) = \int_{-\infty}^{+\infty} P_{V_2}(z - \tau) \cdot P_{V_1}(\tau)\ d\tau$$

Since $P_{V_1}(\tau)$ is zero if τ is outside the interval $[l_1, u_1]$, and $\frac{1}{u_1 - l_1}$ otherwise, the following holds good:

$$P_Z(z) = \int_{l_1}^{u_1} P_{V_2}(z - \tau) \cdot \frac{1}{u_1 - l_1}\ d\tau$$

Let us assume that $u_2 - l_2 \leq u_1 - l_1$; if this is not satisfied v_1 and v_2 can be interchanged to satisfy this. It is not difficult to see that $(l_1 - u_2) \leq (l_1 - l_2) \leq (u_1 - u_2) \leq (u_1 - l_2)$. It can now be shown that the probability density function $P_Z(z)$ has the shape of a trapezoid and is given as follows, where $D = \frac{1}{(u_1 - l_1) \cdot (u_2 - l_2)}$:

$$P_Z(z) = \begin{cases} 0 & if\ z \leq l_1 - u_2 \\ (z - l_1 + u_2) \cdot D & if\ l_1 - u_2 \leq z \leq l_1 - l_2 \\ \frac{1}{u_1 - l_1} & if\ l_1 - l_2 \leq z \leq u_1 - u_2 \\ (u_1 - l_2 - z) \cdot D & if\ u_1 - u_2 \leq z \leq u_1 - l_2 \\ 0 & if\ z \geq u_1 - l_2 \end{cases}$$

Now, the probability that $|Z| \leq K$ must be computed. This is equal to the probability that $-K \leq Z \leq K$, which is:

$$\int_{-K}^{K} P_Z(z)\ dv_z$$

This value depends upon how the values $-K$ and K are related to the values $l_1 - u_2$, $l_1 - l_2$, $u_1 - u_2$, and $u_1 - l_2$. Note that these last four values are in increasing order. A careful analysis shows that there are only fifteen different possible ranges of values for K. One extreme is when $K \geq u_1 - l_2$ and $-K \leq l_1 - u_2$; in this case, the above probability is 1 and this corresponds to the *must* satisfiability. At the other extreme, $K \leq l_1 - u_2$ or $-K \geq u_1 - l_2$; in this case, the above probability (i.e., the probability of satisfaction) is zero. It is not difficult to see that the complement of this condition corresponds to the condition of the *may* satisfaction and in this case the probability of satisfaction is non-zero. Some other possible ranges are the following. If $u_1 - u_2 \leq K \leq u_1 - l_2$ and $-K \leq l_1 - u_2$, then the above integral and hence the probability comes out to be $1 - \frac{D \cdot (u_1 - l_2 - K)^2}{2}$. If $l_1 - l_2 \leq K \leq u_1 - u_2$ and $-K \leq l_1 - u_2$, then the probability is $\frac{2K + u_2 + l_2 - 2l_1}{2(u_1 - l_1)}$.

For each of the above cases, a set of linear equations can be set up as given in Section 4.1 after replacing the inequalities (1) and (2) by those given in Lemma 1, and for those t_k and t_l satisfying the system of inequalities the probability of satisfaction is computed using the expression given above.

In this analysis, we have assumed that the random variables V_1 and V_2 are uniformly distributed in their respective intervals, which implies that we have considered uncertainty in the worst case. Another reasonable assumption is that they have a bounded normal distribution within their respective intervals; analysis of this case may be more difficult and could be a subject for future work.

6. EXPERIMENTAL EVALUATION

In this section, we show some results obtained using the implemented prototype. The experimental settings are summarized in Table 1. In the experiments, the sensors are

location devices (GPS receivers) attached to objects moving at 60 mph (about 26.8 m/s) within an area of 10 squared kilometers. Every object is initialized with a random location and an orientation that changes randomly, with a 50% probability, every 10 seconds. Thus, a random mobility model is considered to generate the datasets. The objects' trajectories are precomputed and stored in files to be readily available in the different experiments. Every moving object measures its location every three seconds, and communicates input tuples (with two values and two prediction functions based on simple linear extrapolation, for the x and y coordinates) to the query processor according to the specified threshold update policy: a threshold of 25 meters with a minimum of one update every 30 seconds is considered. The trajectory files are unified into a single trace file that is processed using the IBM *Location Transponder* [28], which assigns threads as needed to meet the deadlines of the location updates.

Table 1: Parameters for the experiments

Objects' speed	60 mph ($\sim$ 26.8 m/s)
GPS sampling rate	3 seconds
Period of angle change	10 seconds
Prob. of angle change	50%
Scenario size	10 squared kilometers
Threshold update policy	25 meters
Max. period update policy (T)	30 seconds
K of the window-join constraint	400 meters
w of the window-join constraint	3 seconds
$comp$ of the window-join constraint	"$\leq$"
Validation period	1 second
Duration of the test	3 minutes
Indexing mechanism	R-Tree

In the described scenario, each monitoring test is run for three minutes and new tuples are released into the output data stream every second (i.e., the validation period, defined in Section 4.3, is one second). The query that is evaluated retrieves pairs of objects that are within 400 meters of each other both in the horizontal and vertical dimensions. For this, a window-join constraint with $K = 400$ meters, a comparator $comp =$ " $\leq$ ", and a valid-time window w of three seconds, is considered. The experiments were carried out on a computer with a $2x$Dual-Core AMD Opteron processor running at 2.6 GHz, with 8 GB of RAM, and SunOS 5.10. The Buffer Manager stores tuples in main memory. The current prototype, implemented in Java, does not use any specialized solver for systems of linear inequalities. An R-Tree [16] is used to filter potential matches of an input tuple, previous to the filter step described in Section 4.1 (for more details, please see [19]).

In the rest of this section, we present experiments that analyze: 1) the scalability of the query processor, 2) the benefits of using prediction functions, and 3) the suitability of the proposed buffer management policy. The accuracy is measured in terms of precision, recall, and F-measure, adopting these definitions from the field of Information Retrieval [4]. In order to compute the ideal answer for a given time instant, the last GPS samples of the objects are considered. We focus on window-join constraints because processing this type of constraint is more challenging.

6.1 Scalability of the Query Processor

Figure 5 shows the impact of the total number of moving objects/sensors on the accuracy of the query processor. As can be seen in the figure, the accuracy is not perfect even in scenarios with a small number of moving objects, as there is always some imprecision regarding the values that can be estimated using the prediction functions (imprecision allowed by the threshold update policy). However, it should be emphasized that the query processor exhibits a good scalability when the number of moving objects increases. The slight performance degradation affects especially the recall, as the query processor strives to process a large number of input tuples so as to release new output tuples quickly. However, a large number of moving objects is needed to challenge a single monitoring computer. It should also be noted that a worst-case situation is considered in this experiment, in the sense that all the input tuples are of the type of value in which the query is interested (GPS locations). This implies that any new tuple received could, in principle, match with any other tuple, as all the input tuples are stored in the same buffer. Besides, as the size of the scenario is fixed, the object density increases with the number of moving objects, reducing the efficiency of the R-Tree. Although it is not shown in the figure, the total query processing time (accumulated during the whole experiment) increases linearly with the number of objects (ranging up to about 70 seconds in a scenario with 1000 objects); considering also a value constraint decreases this cost dramatically (20 seconds with 1000 objects), as the value constraint is verified before performing the join, saving many tuple comparisons.

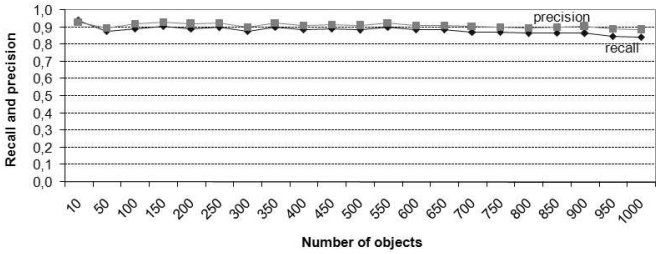

Figure 5: Effect of the number of objects: accuracy

6.2 Benefits of Prediction Functions

In this experiment, the accuracy of the approach presented in this paper is compared with an alternative approach where no prediction functions are used. In this latter case, the threshold update policy described in Section 2 is modified accordingly so as to communicate a new update when the difference between the current value and the last communicated value exceeds the threshold (estimating values is no longer possible). The general query processor architecture is maintained in both cases.

Figure 6 shows how the accuracy of the query processing is considerably better when prediction functions are used, except for scenarios with a small number of moving objects/sensors (< 150). Thus, without prediction functions the values measured by the sensors cannot be estimated, so only the values at update-time can be considered accurate. The decrease in the accuracy of the approach that does not use prediction functions is explained by a high query processing overload, due to the huge increase in the number of tuples communicated by the moving objects when no prediction functions are used (about 300 tuples per second in a

scenario with 1000 objects): the Tuple Evaluator is not able to cope with a very high update rate.

While the use of predictions and the accuracy of the query processing may be affected by factors such as the speed of the objects, the threshold of the update policy, the size of the valid-time window, or the existence of bursts and concept drifts, we have obtained a good accuracy in a variety of settings [19].

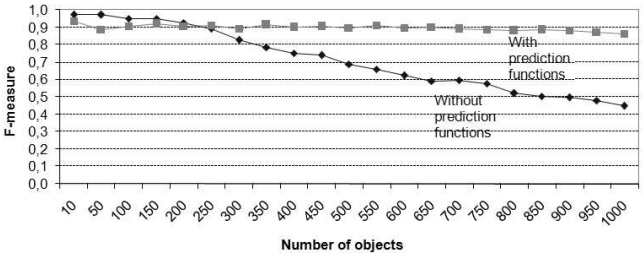

Figure 6: Benefits of prediction functions

6.3 Buffer Management Policy

In this section, the buffer management policy presented in Section 4.2.2 is evaluated. For this, a scenario with 300 objects/sensors and a buffer limitation of 325 tuples is considered. The GPS sampling rate is of one sampling every second. The period of angle change by a moving object is set to five seconds with a probability of 75%; in this way, the rate of input tuples increases considerably, which leads to a higher competition for the use of the limited space available in the buffer for the GPS sensors of the moving objects. For evaluation purposes, the proposed policy (that will be called in this section *discard-min-C*, as it tries to maximize the contribution of the tuples stored) is compared with three other possible policies: 1) *discard-if-full*, where an incoming tuple is just discarded if there is no space to store it; 2) *discard-oldest*, where the oldest tuple stored is discarded to leave space for the incoming tuple; and 3) *discard-randomly*, where a victim is selected randomly. Like in the proposed policy (see Section 4.2.2), existing tuples are used "to cover the gaps" (time intervals for which an accurate prediction function is not available due to removals), except with the *discard-if-full* policy. Doing this with the *discard-if-full* policy would mean that purging would never have an opportunity to discard tuples, as their $IAPFs$ would never fall outside the buffer window, which would have a very negative effect on the precision of the query processor when using that policy.

Figure 7 shows the recall achieved by the different policies for different values of the valid-time window. The policy proposed in this paper (*discard-min-C*) clearly achieves the best recall, the second best policy is to discard the oldest tuple, the third best policy is to discard tuples randomly, and finally the worst policy is to discard incoming tuples while the buffer is full (i.e., until some stored tuples are purged by the Buffer Manager). Similar conclusions are obtained by analyzing the precision (for the sake of clarity, not shown in the figure), although in this case the precision of *discard-min-C* and *discard-oldest* is very similar. Thus, the advantage of *discard-min-C* over *discard-oldest* is a significant increase in the recall (at the expense of a slightly higher processing cost due to the need of computing the contributions of tuples and keeping them up-to-date). It is also remarkable that, in this experiment, the accuracy (in terms of both precision and recall) of the proposed policy is in every case less than 1% worse than in a situation where there is no buffer size limitation (this ideal situation increases only the F-measure in about 0.85% on average). We have performed other experiments simulating delayed incoming tuples, and similar conclusions can be drawn.

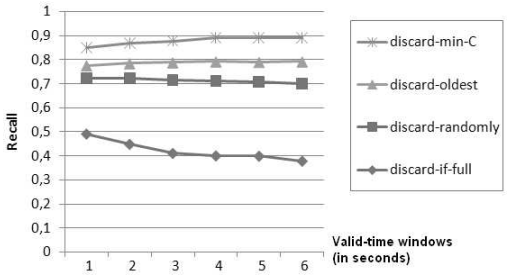

Figure 7: Comparison of different buffer policies

7. RELATED WORK

Due to its importance, minimizing the size and number of transmissions in sensor networks has been the focus of many works (e.g., see [29, 30]). In relation to this problem, there are several works that (as we do) propose to use predictions in the context of data streams:

- The most relevant work is [24], which also advocates the use of *predictors* to minimize the communication costs in a sensor network. The authors of that work study several prediction techniques, all of them based on linear functions. A similar idea is proposed in [21], where a Kalman filter is specifically chosen among the linear estimation methods. However, these works do not focus on query processing issues.

- In [14], the basic assumption is that in many contexts the readings of nearby sensors are correlated. Therefore, the authors propose to analyze the existing spatio-temporal correlations to compute prediction models in networks with static sensors. However, how to efficiently process the data received from sensors to answer different types of queries is out of the scope of their work. Therefore, this work is complementary to ours. There are other works that also exploit correlations among sensors' values (e.g., [10, 30]).

- Another interesting work is [20], 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 is that unexpected events between samplings cannot be detected, which may be a problem if the sampling interval is large. This is also a complementary work to ours: it could be used with our query processor to dynamically adjust the sampling rate of the sensors.

- Probabilistic models are used in [11] to capture correlations among sensors so as to provide more robust interpretations of sensor readings and optimize their

acquisition. Its ultimate goal is similar to ours: to reduce energy consumption and processing overload. However, the approach is different, as it focuses on data acquisition in a pull-based query model. Thus, for example, continuous queries or window-join constraints are not considered.

Regarding the query constraints handled by the proposed query processor, some relevant works are worth mentioning:

- In [17], *time window constraints* are used to limit the sets of sensor tuples that can be matched for a query, and a mechanism to process multi-way joins is proposed. However, the authors 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.
- Also in the context of moving objects, CAMEL (Continuous Active Monitor Engine for Location-based Services) [7, 8] is a location stream database with a location management engine to support intelligent location aware services. It supports unary and binary moving objects triggers, which can be seen as a special case of the value and window-join constraints that we consider. However, it does not deal with prediction functions or temporal conditions. Moreover, as in [17], the objects of interest must be referenced in the query.
- In [6], the *Dynamic Time Warping distance* is considered, such that a point in a time series can be aligned with multiple points of another time series in different (previous or later) temporal positions. So, although the context of that work is different (the goal there is to compare time series), the importance of taking into account temporal shifts is also highlighted (as with the possibility to specify a valid-time window in the window-join constraints proposed in this paper).
- Finally, it is also interesting to mention that different types of sliding windows have been proposed in the literature (e.g., time-based or tuple-based windows [2]), which can be considered as complementary to the proposal presented in this paper: such windows should be considered when extracting the tuples from the buffers for query processing and also as an additional criteria to purge old tuples from the buffers.

To conclude, [1] is a recent relevant work that also advocates implementing query operators by solving equation systems derived from piecewise polynomial models. The main focus of such work is on query validation. A technique called *query inversion* is proposed to translate an accuracy requirement on the output tuples into an accuracy requirement on the input tuples. An accuracy validation drives the proposed online predictive query processing: only input tuples that may cause a change in the result are processed. An important difference between our proposal and this and other works described in this section is that we define a complete query processor architecture for handling data streams with prediction functions, studying all the issues affecting the query processing, from the sensors to the clients.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have described a query processor for handling data streams with prediction functions in a network of sensors. The use of prediction functions allows 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.

Although the use of predictions has been already proposed in the literature of data streams, and some works such as [21, 24] compare different prediction techniques, no other work focuses on query processing aspects. The proposed incremental query processing approach detects all the answers, adapts to different types of clients (e.g., a trigger processing module that reacts to certain events, or a sampling module that presents periodically a snapshot of the answer to a user), and allows the processing of predicted future results. Two types of interesting constraints are considered, with an emphasis on window-join constraints, that have not been considered so far in other works. An appropriate buffer management policy, that takes into account the contribution of the prediction functions contained in the tuples, has also been proposed. Moreover, uncertainty issues are managed by supporting *may* and *must* queries and tagging the output tuples with a probability of satisfaction. The experiments performed show the interest of our proposal.

As future work, some aspects of the proposed query processor could be optimized. There is a wide variety of proposals in the literature focusing on optimization issues for query processing (join re-ordering for queries with more than two inputs, sharing computation across similar queries, etc.) and some of them could probably be adopted in our approach.

9. ACKNOWLEDGEMENTS

Research supported by NSF grants 0326284, 0330342, ITR-0086144, 0513736, 0209190, EIA-0320956, EIA-0220562, HRD-0317692, and CICYT project TIN2007-68091-C02-02. We thank Raquel Trillo for her useful comments.

10. REFERENCES

[1] Y. Ahmad, O. Papaemmanouil, U. Cetintemel, and J. Rogers. Simultaneous equation systems for query processing on continuous-time data streams. In *24th International Conference on Data Engineering, ICDE'08*, pages 666–675, Washington, 2008. IEEE Computer Society.

[2] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, 2006.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'02*, pages 1–16, New York, 2002. ACM.

[4] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM / Addison-Wesley, 1999.

[5] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *2nd International Conference on Mobile Data Management, MDM'01*, volume 1987 of *Lecture Notes in Computer Science*, pages 3–14, London, 2001. Springer.

[6] P. Capitani and P. Ciaccia. Warping the time on data streams. *Data & Knowledge Engineering*, 62(3):438–458, 2007.

[7] Y. Chen, F. Rao, X. Yu, and D. Liu. CAMEL: a moving object database approach for intelligent location aware services. In *4th International Conference on Mobile Data Management, MDM'03*, volume 2574 of *Lecture Notes in Computer Science*, pages 331–334, London, 2003. Springer.

[8] Y. Chen, F. Rao, X. Yu, D. Liu, and L. Zhang. Managing location stream using moving object database. In *14th International Workshop on Database and Expert Systems Applications, DEXA'03*, pages 916–920, Washington, 2003. IEEE Computer Society.

[9] R. Cheng, D. Kalashnikov, and S. Prabhakar. Evaluation of probabilistic queries over imprecise data in constantly-evolving environments. *Information Systems*, 32(1):104–130, 2007.

[10] D. Chu, A. Deshpande, J. Hellerstein, and W. Hong. Approximate data collection in sensor networks using probabilistic models. In *22nd International Conference on Data Engineering, ICDE'06*, page 48, Washington, 2006. IEEE Computer Society.

[11] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-based approximate querying in sensor networks. *VLDB Journal*, 14(4):417–443, 2005.

[12] T. Ghanem. Supporting predicate-window queries in data stream management systems. In *22nd International Conference on Data Engineering Workshops, ICDE'06 Workshops*, page 139, Washington, 2006. IEEE Computer Society.

[13] T. Ghanem, W. Aref, and A. Elmagarmid. Exploiting predicate-window semantics over data streams. *SIGMOD Record*, 35(1):3–8, 2006.

[14] S. Goel and T. Imielinski. Prediction-based monitoring in sensor networks: Taking lessons from MPEG. *ACM Computer Communication Review*, 31(5):82–98, 2001.

[15] J. D. Gooijer and R. Hyndman. 25 years of IIF time series forecasting: a selective review. Department of Econometrics and Business Statistics, Monash University, 2005.

[16] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *ACM SIGMOD International Conference on Management of Data, SIGMOD'84*, pages 47–57, New York, 1984. ACM.

[17] M. Hammad, W. Aref, and A. Elmagarmid. Stream window join: tracking moving objects in sensor-network databases. In *15th International Conference on Scientific and Statistical Database Management, SSDBM'03*, pages 75–84, Washington, 2003. IEEE Computer Society.

[18] S. Ilarri, O. Wolfson, E. Mena, A. Illarramendi, and N. Rishe. Processing of data streams with prediction functions. In *39th Hawaii International Conference on System Sciences, HICSS-39*, page 237a, Washington, 2006. IEEE Computer Society.

[19] S. Ilarri, O. Wolfson, E. Mena, A. Illarramendi, and P. Sistla. An architecture for prediction-based monitoring of data streams. Technical Report RR-08-08, University of Zaragoza, September 2008.

[20] A. Jain and E. Chang. Adaptive data sampling for sensor networks. In *1st International Workshop on Data Management for Sensor Networks, DMSN'04*, pages 10–16, New York, 2004. ACM.

[21] A. Jain, E. Chang, and Y.-F. Wang. Adaptive stream resource management using Kalman filters. In *ACM SIGMOD International Conference on Management of Data, SIGMOD'04*, pages 11–22, New York, 2004. ACM.

[22] C. Jones, K. Sivalingam, P. Agrawal, and J. Chen. A survey of energy efficient network protocols for wireless networks. *Wireleless Networks*, 7(4):343–358, 2001.

[23] K. Kowalski and W.-H. Steeb. *Nonlinear Dynamical Systems and Carleman Linearization*. World Scientific, Singapore, 1991.

[24] V. Kumar, B. Cooper, and S. Navathe. Predictive filtering: a learning-based approach to data stream filtering. In *1st International Workshop on Data Management for Sensor Networks, DMSN'04*, pages 88–97, New York, 2004. ACM.

[25] D. Lin, B. Cui, and D. Yang. Optimizing moving queries over moving object data streams. In *12th International Conference on Database Systems for Advanced Applications, DAASFA'07*, volume 4443 of *Lecture Notes in Computer Science*, pages 563–575, Berlin, 2007. Springer.

[26] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.

[27] J. Miles and M. Shevlin. *Applying Regression and Correlation: A Guide for Students and Researchers*. SAGE Publications, London, 2001.

[28] J. Myllymaki and J. Kaufman. IBM Location Transponder. IBM alphaworks, `http://www.alphaworks.ibm.com/tech/transponder`, 2002.

[29] M. Sharaf, J. Beaver, A. Labrinidis, and P. Chrysanthis. Balancing energy efficiency and quality of aggregate data in sensor networks. *The VLDB Journal*, 13(4):384–403, 2004.

[30] A. Silberstein, R. Braynard, and J. Yang. Constraint chaining: on energy-efficient continuous monitoring in sensor networks. In *ACM SIGMOD International Conference on Management of Data, SIGMOD'06*, pages 157–168, New York, 2006. ACM.

[31] A. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Querying the uncertain position of moving objects. In *Temporal Databases: Research and Practice*, volume 1399 of *Lecture Notes in Computer Science*, pages 310–337, Berlin, 1998. Springer.

[32] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *ACM SIGMOD International Conference on Management of Data, SIGMOD'02*, pages 334–345, New York, 2002. ACM.

[33] K. Torp, C. Jensen, and R. Snodgrass. Effective timestamping in databases. *The VLDB Journal*, 8(3-4):267–288, 2000.

[34] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *14th International Conference on Data Engineering, ICDE'98*, pages 588–596, Washington, 1998. IEEE Computer Society.

[35] O. Wolfson, A. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–287, 1999.