In-Network Execution of Monitoring Queries in Sensor Networks

Xiaoyan Yang[1] Hock Beng Lim[1] M. Tamer Özsu[2] Kian Lee Tan[1]

[1]School of Computing
National University of Singapore, Singapore
{yangxia2, limhb, tankl}@comp.nus.edu.sg

[2]School of Computer Science
University of Waterloo, Canada
tozsu@cs.uwaterloo.ca

ABSTRACT

Sensor networks are widely used in many applications for collecting information from the physical environment. In these applications, it is usually necessary to track the relationships between sensor data readings within a time window to detect events of interest. However, it is difficult to detect such events by using the common aggregate or selection queries. We address the problem of processing window self-join in order to detect events of interest. Self-joins are useful in tracking correlations between different sensor readings, which can indicate an event of interest. We propose the Two-Phase Self-Join (TPSJ) scheme to efficiently evaluate self-join queries for event detection in sensor networks. Our TPSJ scheme takes advantage of the properties of the events and carries out data filtering during in-network processing. We discuss TPSJ execution with one window and we extend it for continuous event monitoring. Our experimental evaluation results indicate that the TPSJ scheme is effective in reducing the amount of radio transmissions during event detection.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Information filtering

General Terms

Experimentation, Performance

Keywords

Sensor Networks, Self-Join Queries

1. INTRODUCTION

Sensor networks are being increasingly deployed in many important applications from environmental monitoring to military surveillance. Such networks consist of many small sensor nodes with sensing, data processing and wireless communication capabilities. Since sensor nodes are resource-constrained, novel data management techniques are needed to meet the application requirements taking into account the resource limitations.

Sensor-based applications have different data requirements ranging from simple data logging to complicated event detection. Existing approaches like TinyDB [12] and Cougar [16] are query-based data collection systems developed for sensor networks. Users acquire interesting data by issuing SQL-like queries while the underlying data retrieval and collection processes are transparent to them. Many of the early works on query processing in sensor networks have focused on aggregate queries such as COUNT, SUM and MAX, and have developed in-network aggregation techniques that significantly reduce the amount of data transmission. Some approximate query processing techniques [7, 10, 6, 5] were also proposed to reduce the cost of data collection while providing high quality query results.

However, in certain scenarios, questions cannot be answered simply by using the aggregate or selection queries. In many applications, we are interested in tracking the correlations among sensor data within a time window to detect events of interest. For example, scientists often study the correlations between environmental or weather changes, and the habitats of animal species. They may want to know whether the animals tend to gather at a region in the forest when it rains, and leave that place when it is sunny. This question could be answered by the following SQL query.

Q_1:

SELECT $A_1.humidity$, $A_1.num$, $A_1.time$, $A_1.loc$,
$A_2.num$, $A_2.time$, $A_2.loc$
FROM $Animal$ AS A_1, $Animal$ AS A_2
WHERE $A_1.humidity > \delta$
AND $A_2.num - A_1.num > \theta$
AND $near(A_1.loc, A_2.loc, r)$
AND $A_2.time > A_1.time$
AND $A_2.time - A_1.time < h$

Assume we have a table named *Animal* that stores the data collected by sensor nodes deployed in the region of interest. Each sensor periodically reports the humidity reading and the number of animals that pass by. In query Q_1, humidity being larger than δ indicates rainy weather. Attribute *num* records the number of animals detected. The predicate $A_2.time > A_1.time$ suggests that animals are gathering when it begins to rain; while $A_2.time - A_1.time < h$ means that two matching tuples are within a time window of size h. The function $near(loc_1, loc_2, r)$ checks whether

SIGMOD'07, June 11–14, 2007, Beijing, China.

the location of sensor N_2 (loc_2) is within distance r of the location of sensor N_1 (loc_1) using some distance function. Thus, Q_1 queries whether there are sensors near each other that detect a rapid increase in the number of animals within a window of h after it rains.

Another interesting scenario is a volcano monitoring application, where scientists are interested in the pressures detected within a certain region around the volcanic mountain. After noticing that the volcanic activity of the mountain has increased, they may want to know whether the pressures detected have crossed a certain threshold and is continuously increasing within some period of time. This can be captured by query Q_2.

Q_2:

SELECT $P_1.pressure$, $P_1.time$, $P_2.pressure$, $P_2.time$
FROM $Pressure$ AS P_1, $Pressure$ AS P_2
WHERE $P_1.pressure > \delta$
AND $P_2.pressure > P_1.pressure$
AND $P_2.time > P_1.time$
AND $P_2.time - P_1.time < h$

In query Q_2, $Pressure$ is a table that stores pressure readings of sensor nodes. Q_2 answers the question of whether the sensors detect continuously increasing pressure over a given threshold.

Queries like Q_1 and Q_2 involve self-join operations in sensor networks. Although there are well established techniques to process self-join queries in traditional distributed databases, they are not directly applicable to sensor networks. Existing data collection approaches for sensor networks do not provide efficient evaluation for complicated queries, especially those with 'join' operators. Thus, it is a major challenge to implement self-join queries, since the sensor nodes are resource-constrained, with limited battery energy, processing power, storage, and communication bandwidth.

A direct method to evaluate this type of queries is by flooding. Since each sensor node does not know beforehand which nodes may generate data matching its own and when these data will be generated, messages containing local sensor data have to be propagated to neighboring nodes. In the worst case, tuples from one sensor node may have to be flooded in the entire network so as to find all matching tuples under particular join conditions. In such a case, the messages for exchanging information may overwhelm the limited network bandwidth. Furthermore, for further processing, each sensor node needs to store the local historical sensor data and the data from other sensor nodes for at least a window size, which may consume a large amount of local storage. Evaluating local self-join in a sensor node will also incur a lot of computational power and memory, which may become a heavy burden for such a small device.

A more centralized approach to solve this problem is to let all nodes periodically sense and transmit all the data back to the base station, which would be in charge of the data filtering and event detection. Since the base station is much more powerful than the sensor nodes, it is better equipped to handle self-join queries. However, this method suffers from high data transmission cost compared to in-network processing as all the nodes send data back to the base station. It would be very inefficient especially when the join selectivity (event frequency) is low. Furthermore, even though the base station is powerful, it may still become a bottleneck.

By examining queries Q_1 and Q_2 carefully, we find some common properties among them:

1. They involve self-joins, as the join predicates are posed on the same table (sometimes even on the same column).

2. There is a window predicate posed on the correlated tuples. Even if two tuples satisfy the join conditions, they will be filtered out from the final results if they fall out of a window.

3. They include a selection predicate specifying the conditions after which certain correlations among sensor data are of interest. Therefore, this usually indicates the possible start of an interesting event.

Based on the above observations, we propose the Two-Phase Self-Join (TPSJ) technique to efficiently process self-joins for this type of event detection in sensor networks. The key idea of TPSJ is to process a self-join query in two phases. The goal of the first phase is to do some preliminary filtering and find some candidates that might be in the final result. In the second phase, a window join is run and those candidates found in phase one are used to do further filtering within the network. In this manner, TPSJ tries to find matching tuples within a time window. TPSJ coordinates the continuous execution of monitoring queries for event detection over long periods.

We conducted extensive simulations to evaluate TPSJ performance relative to straightforward approaches. Our experimental results indicate that TPSJ significantly reduces the amount of data transmission during event detection.

The rest of this paper is organized as follows. Section 2 outlines the related previous work. In Section 3, we provide the problem definition for our work. Section 4 presents our TPSJ scheme focusing on one execution of such a query including the algorithms for the two phases, while Section 5 extends this discussion to continuous monitoring. In Section 6 we discuss the methodology and results of our experimental evaluation. Finally, we conclude the paper in Section 7.

2. RELATED WORKS

Joins are common in applications for target tracking or detection [16]. There are several works that consider join query processing in sensor networks, starting with [16], which proposes a technique similar to the centralized approach we discussed in the introduction. It has a destination node, called the *leader*, that collects all the tuples from other sensor nodes through a multi-hop routing protocol. The *leader* decides whether to send out all tuples it has received from other nodes to the base station or to compute the join results locally based on whether or not the join operation will increase the resulting data size. This method requires relevant catalog data about selectivities. However, for a root node (*leader* as in [16]) connected to a base station, it could simply send out all the tuples and let the base station do the joins. This work does not consider in-network join processing.

Joins between two regions in sensor networks are further discussed in [4]. This work proposes an in-network join processing method that determines, for one join query, a set of sensor nodes in the network to which tuples from two regions are sent and the join executed. This method tries to minimize the overall communication cost based on a cost model. However, it can not be applied to event detection discussed in the first section directly. This is because, for a join query like Q_1, we do not know beforehand which two nodes might participate in the joins and produce tuples that would appear in the join results. Further, it does not take into account the impact of query selectivity; the join results are simply the cartesian product of two base relations. It is expected that join operators may provide opportunities for possible data reduction, especially for queries like Q_1.

The problem of optimal operator placement when performing sliding-window joins [2] in sensor networks has been studied [14]. However, this work is based on the assumption that data are collected at edge nodes and transmitted through a hierarchy of nodes with progressively increasing computing power and network bandwidth. It tries to minimize the overall computation and data transmission along the hierarchy. It also assumes that data streams that participate in the joins and the nodes that collect these data are known before joins. However, in the applications we consider, in the extreme, all nodes may be involved in the join operation. The difficulty in our environment is to efficiently filter unrelated tuples while transmitting only those tuples that will contribute to the join results; the problem studied in [14] does not address this difficulty.

Joins of multiple data streams in sensor networks within a sliding window are studied in [9]. By defining the width of the sliding window, joins are limited within a certain period. However, this paper also does not consider how data are to be collected from multiple data streams that originate from different sensor nodes. REED [1] also applies joins to filtering and event detection in sensor networks. However, it mainly deals with joins between sensor data and static data tables rather than joins between sensor data streams, and does not take temporal relationships into consideration.

Much work has been done on join processing of multiple data streams [13, 3, 8]. Most of these assume that multiple data streams flow into the central system at different rates. Large amount of data must be processed properly to provide quick response to multiple user queries. Limitations such as network bandwidth and power supply are not considered in these systems. Further, these works do not consider in-network query processing.

3. EXECUTION SEMANTICS

Before we describe our proposed approach for in-network execution of self-join queries in sensor networks, we define, in this section, the semantics that we wish to achieve. We will do this by referring to a baseline case where all the sensors report their data to the base station where the query is executed (i.e., what we referred to as the centralized approach earlier). Our task, therefore, is to achieve a distributed, in-network execution of these queries that maintains the semantics defined in this section.

Assume, as indicated above, that each sensor node N_i sends to the base station its readings at each sampling interval as a tuple: $T_j = \langle att_1^i, att_2^i, \ldots, att_n^i, N_i, ts_j \rangle$, where att_i's represent readings for multiple attributes (e.g., temperature, pressure, etc), N_i is the sensor id, and ts_j is the timestamp that identifies when the data were read. In sensor networks, the sensors are calibrated such that each one takes a reading at fixed intervals, which means that tuples that come from different sensors with the same timestamp ts_j represent readings "at the same time". Systems issues related to synchronizing sensors are beyond the scope of this paper and have been studied elsewhere [11].

When the tuples arrive at the base station, they are put in a relation called *Sensor* that has $m(= n + 2)$ attributes. This relation is sorted by the timestamp, thus all of the tuples arriving from all of the sensors with timestamp ts_j are grouped together. Naturally, this is a conceptual table and not one that is physically maintained; it is used merely to define the semantics of what we wish to achieve.

The queries are executed over this (conceptual) *Sensor* table. The queries we are interested in are monitoring queries that address event detection problems similar to the examples given in Section 1 (i.e., queries of the type Q_1 and Q_2). These queries have a common form:

Q^*

SELECT $S_1.AT_1, S_2.AT_2$
FROM *Sensor* AS S_1, *Sensor* AS S_2
WHERE $p_1(S_1.AT_3)$
AND $p_2(S_1.att_j,\ S_2.att_h)$
AND *window* $(S_1.ts, S_2.ts, W)$

where AT_i represents the subset of attributes from *Sensor* table, W is the size of the sliding window, which is specified by the user to decide the temporal relationship among tuples of the target event, predicate p_1 is of the form att_i *opt Constant*, and p_2 is of the form att_i *opt* att_j, where *opt* can be any of $\{<, >, \leq, \geq, \neq, =\}$. Notice that we use window predicate *window* $(S_1.ts, S_2.ts, W)$ to restrict that tuples from S_2 is sampled later than that of S_1 as well as that they are within a time window of size W. In this context p_1 is a selection predicate over *Sensor* and p_2 is the join predicate. S_1 and S_2 both are aliases to the *Sensor* table.

The operational semantics of this query execution is as follows. When the query is run, p_1 is evaluated over the tuples in *Sensor*. If a tuple T_j is found where $p_1(T_j) = true$, then a window is defined of size $ts_j + W$. All the tuples in *Sensor* whose timestamps are within this window are subjected to self-join and the result is output. Note that there may be multiple tuples in *Sensor* that satisfy p_1. As long as the timestamps of these tuples are the same, they are considered within one execution of the query (i.e., only one window is defined). For each tuple that has a different timestamp, a new window is defined, and the queries are executed separately for each window.

4. TWO-PHASE SELF-JOIN APPROACH

In this section, we present our two-phase window self-join (TPSJ) approach for in-network execution of monitoring queries with the semantics as specified in Section 3. We first describe the pre-processing that needs to be done (Section 4.1, and then discuss the algorithm (Section 4.2). To simplify the description, we shall present, in this section, a single execution of a query of type Q^*; the continuous execution of such queries is considered in Section 5.

4.1 Preprocessing: Query Decomposition

After the base station receives a user query of type Q^*, it needs first to do some preprocessing. As mentioned in the above section, we start a window join when we find some tuples that satisfy the selection predicate p_1. These tuples are candidates that are likely to contribute to the final result. They will be used to filter sensor readings when we evaluate the window join in-network. Based on this, we define two main tasks for the window self-join query evaluation. The first is to determine when to start a window join while the second is to actually evaluate the join query in-network. So, in preprocessing, we rewrite the original query into two 'new' queries corresponding to these two tasks. These will then be processed using the two-phase self-join scheme presented in the next section.

A user query Q^* is rewritten into two queries after preprocessing as follows:

Q_1^*

```
SELECT $S.AT_1$ INTO $R_1$
FROM $Sensor$ AS $S$
WHERE $p_1(S.AT_3)$
```

Q_2^*

```
SELECT $S.AT_2$
FROM $R_1, Sensor$ AS $S$
WHERE $p_2(R_1.att_j, S.att_h)$
AND $window(R_1.ts, S.ts, W)$
```

The first query Q_1^* is a selection query that finds tuples that satisfy the selection predicate and stores them in a temporary relation R_1. The self-join in the original query now becomes a join, in Q_2^*, between relation *Sensor* and the intermediate relation R_1.

The two new queries will be evaluated in-network separately in two phases, though the second query depends on the result of the first one. Furthermore, *Sensor* is never materialized at the base station, but is maintained in individual sensor nodes (i.e., it is a conceptual relation in Q_1^* and Q_2^*). Q_1^* represents the detection, by one or more sensors, of the event that is of interest in the monitoring application. Q_2^* finds whether the correlation between readings that are important to the application exists in the readings *after* the event is detected.

Let us return to the volcano monitoring application and use it as an example to illustrate the preprocessing. Q_2 is rewritten as two queries:

$Q_{2.1}$:

```
SELECT $P.pressure$, $P.time$ INTO $R_1$
FROM $Pressure$ AS $P$
WHERE $P.pressure > \delta$
```

$Q_{2.2}$:

```
SELECT $P.pressure$, $P.time$
FROM $R_1$, $Pressure$ AS $P$
WHERE $P.pressure > R_1.pressure$
AND $window(R_1.time, S.time, h)$
```

4.2 Two-Phase Self-Join Processing

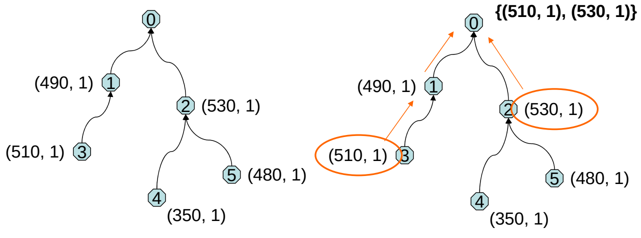
We now focus on the TPSJ algorithm considering only a single execution of query Q^* that has already been decomposed into Q_1^* and Q_2^*.

Phase One:

In phase one, query Q_1^* is executed. The goal of this phase is to find candidates that might contribute to the final result and then properly start the window self-join.

All sensor nodes take periodic readings and insert these readings into a local table that they manage (we refer to the table at node N_i as *Reading*$_i$). When the selection query Q_1^* is injected into the network by the base station[1], each sensor node executes it over its *Reading* table. If a tuple is found to satisfy p_1, it is forwarded to the base station. That particular sensor node stops executing Q_1^* at that point since an event of interest has been detected and the remainder of the tuples within the window will be checked during the execution of Q_2^* in any case. This concludes the first phase of execution.

There are two points that require attention. One is the topology of the sensor network. Our algorithm does not make any assumption about the topology, but in our experiments we use a routing tree topology that is employed in TinyDB [12]. The second issue is the maintenance of the *Reading* table at each sensor node, as this table cannot be allowed to grow too big. The size of the table is a user-defined parameter and entries in it can be managed in a FIFO manner (based on tuple timestamps). It is important, however, that the size is sufficient to hold the readings that can occur in one window; given the window size and the sampling frequency of the sensors, it is possible to set the table size (or set the other two parameters given the restrictions on storage space at the sensor nodes).



(a) Sensor data of epoch 1. (b) Results of *Q2.1*.

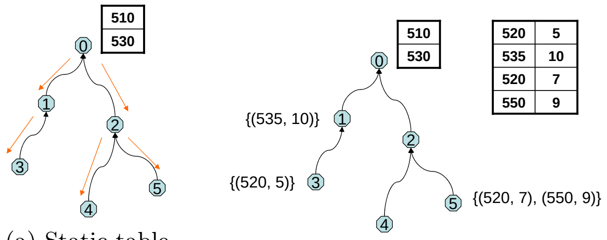
Figure 1: Phase one.

We again use Q_2 as an example to illustrate this process. Assume a routing tree topology for the sensor network as shown in Figure 1(a) where the base station is node 0. Suppose δ is equal to 500 and h is defined as 10 sampling intervals. In phase one, $Q_{2.1}$ is injected into the network and is propagated, along the routing tree, to all the sensor nodes. Sensor data collected at sampling point 1 (i.e., $ts = 1$) is depicted in Figure 1(a) along with their timestamps. Two readings satisfy $p_1 = P.pressure > 500$: readings from nodes N_2 and N_3 (indicated by circles in Figure 1(b)). These two tuples are sent back to the base station along the routing tree as depicted in Figure 1(b)).

Phase Two:

The goal of the second phase is to find matching tuples for the candidates found in phase one and complete event detection within one time window. The base station constructs an intermediate result table R_1 and injects it along with query Q_2^* into the network. Note that all readings in R_1 have the same timestamp since they are sent at the same sampling point when p_1 is detected to be true. Thus R_1 is constructed by projecting out the nodeid and the timestamp attributes from the reading tuples. Notice that the information about the timestamp when R_1 is constructed is included in the window predicate of phase two query. Consequently, R_1 contains only the reading attributes. Once a sensor node receives the table and the new query, it will execute the second query over a window defined over the *Reading* table

[1] Note that in monitoring applications of the type we consider, queries may be pre-defined, which would allow their pre-processing and the pre-deployment of Q_1^* and Q_2^* at the sensors. However, in our experiments we do account for ad hoc queries that are posed by users at any time, with the base station taking over the responsibility of injecting them into the sensor network.



510
530

520	5
535	10
520	7
550	9

(a) Static table propagation.

(b) Results of *Q2.2.*

Figure 2: Phase two.

that it maintains. In other words, it will look at the tuples in *Reading* whose timestamps are within $ts_i + W$ where ts_i is the start time of the window predicate, i.e., the timestamp of the tuples in R_1 (note that all tuples in R_1 have the same timestamp).

Let us again consider example query Q_2 to illustrate phase two. $Q_{2.2}$, together with table R_1, is injected into the network (Figure 2(a)). Each sensor node will execute $Q_{2.2}$ over the window [1, 11] since the timestamp of the tuples in R_1 is 1 and the window size (h) is 10. Results of $Q_{2.2}$ at each sensor node are shown in Figure 2(b) next to each node.

The base station forms the result of the join query as it gets results from individual sensor nodes. For the example we are considering, this is depicted as the second (larger) table in Figure 2(b), which we denote by R_2 in the following discussion. At this point, the base station can easily construct the final result for the original user query, since this is now a simple join operation between R_1 (the result of the selection query) and R_2 (the result of the join query). For example, given tables R_1 and R_2, as shown in Figure 2(b), the final results of Q_2 are {(510, 520), (510, 535), (510, 520), (510, 550), (530, 535), (530, 550)} given that the predicate is $R_2.pressure > R_1.pressure$.

Note that, as indicated at the beginning of this section, we have so far focused on the single execution of the user query; therefore, the results are likely not complete. For example, tuple $\langle 520, 5\rangle$ at N_3 may also join with tuple $\langle 550, 9\rangle$ at N_5, which means that tuple $\langle 520, 5\rangle$ can actually trigger a new window self-join starting at timestamp 5. We discuss how to extend our method to handle continuous event detection (and multiple window join) in Section 5.

```
Algorithm 1: TPSJ($Q$)
    //input: user query $Q$.
    //output: answers to $Q$.
    //$q_1$: selection query for phase one.
    //$q_2$: join query for phase two.
    begin
1       $q_1$, $q_2$ = preprocess($Q$)
2       inject $q_1$ into network and execute
3       if $q_1$ generates a result then
4           form $R_1$, stop $q_1$
5           $R_2$ = windowSelfjoin($q_2, R_1$)
6       calResult($R_1, R_2$)
    end
```

The basic TPSJ algorithm is shown in Algorithm 1. Phase one is captured in lines 2–3, while function *windowSelfjoin* in line 5 mainly addresses the work of phase two. Note that q_1 generating a result (line 3) means that predicate $q_1 = true$ for some tuple(s) at some sensor node(s). Function *calResult* is executed at the base station, which outputs the final results of the user query after receiving the results of two phases. Timely response to the user query can be achieved by incrementally outputting answers as soon as the base station receives one tuple from phase two.

There are some situations where fewer tuples may be sent into the network instead of the whole result table R_1 of phase one. If the join predicate in the user query only involves operators of $\{<, >, \leq, \geq\}$, we can sort R_1 and choose to send the smallest or the largest value to the network. For example, consider Q^*, Q^*_1 and Q^*_2 where predicate p_2 includes "<" operator i.e., $p_2(S_1.att_j, S_2.att_h) = S_1.att_j < S_2.att_h$ in Q^*. Assume the result table R_1 consists of tuples $\{T_1, T_2,...,T_n\}$. We sort table R_1 into $\{T'_1, T'_2,...,T'_n\}$ such that $T'_1.att_j \geq T'_2.att_j \geq ... \geq T'_n.att_j$. The smallest value of T'_n in R_1 is the only tuple that needs to be sent into the network, and is sufficient to get a complete result set. We may further rewrite Q^*_2 into

SELECT $S.AT_2$
FROM *Sensor* AS S
WHERE $p_2(T'_n.att_j, S.att_h)$
AND *window* (W)

which becomes a simple selection query. By doing so, local join operations are avoided at each sensor node. For example, in Figure 1 the result table consists of two tuples $\{\langle 510\rangle, \langle 530\rangle\}$. Instead of sending the whole table into the network, we could just send the tuple with the smaller value which is $\langle 510\rangle$.

A more specific situation is that, when selectivity is low, it is possible that the result of phase one consists of few tuples. When table R_1 consists of one tuple $\{T\}$, similar rewriting techniques as in the above situation can be applied to Q^*_2 regardless of the type of predicate p_2. This is especially useful when we want to track some rare events where the frequency of results of phase one is low.

5. CONTINUOUS TPSJ

In continuous TPSJ, we aim to detect target events over a long period of time. In this section, we discuss how to extend TPSJ from one window execution, as discussed in the previous section, to continuous event detection.

As we have mentioned in Section 4, it is possible that there are some tuples that may trigger a new window self-join within the current processing window. For example, we noted that while $Q_{2.2}$ (a self-join) was executing within the window [1,11], another reading at $ts = 5$ may satisfy the selection query. It will be necessary, in continuous monitoring, to start a new window [5,15] and execute the self-join within that window as well. However, this has to be done carefully in order to ensure that unnecessary work is not done within overlapping windows.

Therefore the first task is to detect tuples that satisfy the selection predicate (i.e., execute query Q^*_1) continuously. This requires minor changes in phase one: phase one does not stop after the detection of the first tuple that satisfies p_1; instead, Q^*_1 runs continuously from its insertion into the sensor network until it is explicitly terminated. Tuples satisfying p_1 are continuously sent back to the base station.

The second task is to modify the processing that is performed at the base station. As tuples that satisfy selection predicate p_1 continuously flow back, the base station must react properly to start new window self-joins for them. As indicated above, the naive way is to compile relation R_1 as before and start a new window for each R_1 (recall that each R_1 contains readings with the same timestamp). However, this is very inefficient in several aspects. First, in the case of overlapping windows, one query may already be (partially or fully) answered by another. Second, frequent triggering of queries may incur significant transmissions in-network, as new queries have to be propagated from the base station to all the sensor nodes. Finally, result tables of phase one among consecutive window queries may also overlap. We aim to find an efficient way of triggering window self-joins of phase two at the base station.

Based on the above observations, we have developed several rules to guide the triggering of window self-joins (i.e., query Q_2^*).

Rule A - One window self-join per sampling interval

This rule requires that at most one window self-join is triggered for one time clock, i.e., for all the tuples that have identical timestamps, phase two is triggered only once. This is not so much a rule as it is a statement of the semantics for completeness.

Rule B - Delay Query Triggering As Much As Possible

This rule addresses the relationship between two consecutive query executions during overlapping windows and aims to reduce the amount of work that is done. Assume Q is the phase two (self-join) query currently executing and Q' is the subsequent execution of the same query triggered by a tuple that has a timestamp that falls within the execution of window Q (Figure 3). It is possible that the two queries will generate some join results that are identical. This is because, if R_1 and R_1' refer to the phase one result tables that will be used in Q and Q', respectively, these tables may have overlapping tuples (it is possible that one may even be contained in the other). To deal with this case, we follow a set of rules. These rules are based on two factors: (1) the relationship between the result tables generated in phase one, and (2) the join predicate.

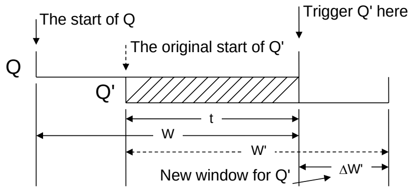


Figure 3: Delay Triggering Query

1. If $R_1 \subset R_1'$,
   (a) If the join predicate is $att_j = att_h$, then the remaining answer of Q is completely included in Q''s answer. Therefore, Q' is injected into the sensor network with R_1' and the sensor nodes are instructed to stop executing Q and start Q' instead.

   (b) If the join predicate is $att_j \neq att_h$, then part of Q''s answer is included in Q's answer. Therefore, instead of initiating the execution of Q' immediately, it is delayed until the end of Q's execution window. At this point, Q' is injected into the network with R_1'[2]. The window of Q' is adjusted, because we only need to run it in the remainder of its original window. For example, window W is described by $[t_s, t_e]$, where t_s is the start time and t_e is the end time. If query Q' with window $W' = [t_s', t_e']$ is delayed by query Q with window $W = [t_s, t_e]$, then W' is modified to $[t_e + 1, t_e']$.

2. If $R_1' \subset R_1$,
   (a) If the join predicate is $att_j = att_h$, then this is identical to *rule* B1(b).

   (b) If the join predicate is $att_j \neq att_h$, then this is identical to *rule* B1(a).

3. If $R_1 = R_1'$, then Q and Q' have exactly the same join results during overlapping window. We consider Q''s answer to be partially included by Q and delay Q' till the end of Q's execution window. Note that another option is to inform each sensor node to extend the window W of Q to include W' of Q'. However, doing so incurs similar cost in radio transmissions as injecting Q' in a later time. The merit of doing so will be further discussed in the following rule.

4. If none of 1, 2 or 3 holds for join predicate $att_j \ \theta \ att_h$ where $\theta \in \{=, \neq\}$, Q' together with R_1' is injected into the network for execution.

5. If the join predicate is $att_j \ \theta \ att_h$ where $\theta \in \{<, >, \leq, \geq\}$, result table R_1 (or R_1') is of size 1, i.e., it contains only one tuple as discussed in Section 4. Assume $R_1 = \{a\}$ and $R_1' = \{b\}$. The join predicate in Q may be rewritten as $a \ \theta \ att_h$.
   (a) If $b < a$,
      i. If $\theta \in \{<, \leq\}$, then the remaining answer of Q is completely included in Q''s answer. Therefore, Q' is injected into the sensor network with the join predicate rewritten to $b \ \theta \ att_h$ and the sensor nodes are instructed to stop executing Q and start Q' instead.

      ii. If $\theta \in \{>, \geq\}$, then part of Q''s answer is included in Q's answer. Therefore, instead of initiating the execution of Q' immediately, it is delayed until the end of Q's execution window. At this point, Q' is injected into the network with the join predicate rewritten as $b \ \theta \ att_h$ and the window adjusted as $\triangle W' = W' - W$.

   (b) If $b > a$,
      i. If $\theta \in \{<, \leq\}$, then this is identical to *rule* B5(a)ii.

      ii. If $\theta \in \{>, \geq\}$, then this is identical to *rule* B5(a)i.

[2] It is possible to reduce the amount of data injected into the sensor network by adjusting R_1' as $\triangle R_1' = R_1' - R_1$ and executing over window $\triangle W'$ as shown in Figure 3. However, this introduces additional complexity at the sensor nodes since they have to reconstruct R_1' as $R_1' = \triangle R_1' \cup R_1$. Therefore, we have not implemented this optimization.

(c) If $b = a$, Q and Q' have exactly the same join results for overlapping window. Similar to *rule* B3, we delay Q' accordingly.

Consider the following example. Assume $R_1 = \{5, 9\}$ is the result table used by the active query Q, the self-join predicate p_2 requires that the sensor reading be not equal to any element of R_1, and Q' has result table $R'_1 = \{2, 5, 9\}$. In this case, $R_1 \subset R'_1$ and thus the results of Q' are at least partially contained by Q, i.e., answer to Q' during the overlapping period of length t are provided by that of Q. By applying *rule* B1(b), we delay sending Q' and R'_1 into the network until the end of the Q's window as illustrated in Figure 3. Notice that the window W' of Q' is modified accordingly (as shown by the dashed line to the solid line in Figure 3)

Rule C - Hidden Query

This rule covers the case that may occur when a query Q' is delayed until the currently active query Q is completed, as discussed in *Rule B*. It is possible that, while Q' is waiting to be issued by the base station, another tuple may trigger a third query Q'' that may make it unnecessary to issue Q' at all. This would happen in several situations. Let us consider query Q, Q', Q'' with windows W, W', W'' respectively (Figure 4) and R_1, R'_1 and R''_1 are the phase one result relations used by these queries.

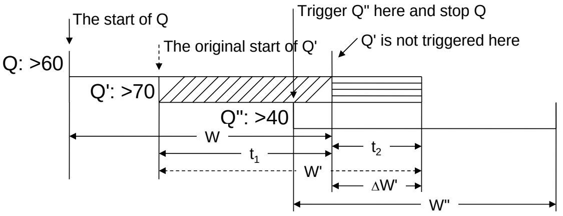


Figure 4: Hidden Query

Algorithm 2: Continous TPSJ

```
//Input: query $Q$ of type $Q_2^*$ and relation $R_1$ as result of phase one
begin
    for each cycle tCurrent do
        for each Q in ActiveQueue do
            if $Q.t_e < tCurrent$ then
                ActiveQueue.dequeue($Q$, $R_1$)
        for each Q in DelayQueue do
            if $Q.t_s = tCurrent$ then
                DelayQueue.dequeue($Q$, $R_1$)
                NewqueryQueue.enqueue($Q$, $R_1$)
        if there is a new query Q generated by phase one then
            NewqueryQueue.enqueue($Q$, $R_1$)
        for each Q in NewqueryQueue do
            for each Q' in ActiveQueue do
                if Q can be (partially) answered by Q' then
                    modify $Q.Window$,
                    DelayQueue.enqueue($Q$, $R_1$)
                else
                    if Q' can be answered by Q then
                        ActiveQueue.dequeue($Q'$, $R'_1$)
                        InjectQueue.enqueue($Q$, $R_1$)
            if Q is not in DelayQueue or InjectQueue then
                InjectQueue.enqueue($Q$, $R_1$)
        for each Q in InjectQueue do
            for each $Q' \neq Q$ in InjectQueue do
                if Q is a hidden query caused by Q' then
                    InjectQueue.dequeue($Q$, $R_1$)
                if Q can be (partially) answered by Q' then
                    modify $Q.Window$
        for each Q in InjectQueue do
            delete all hidden queries in DelayQueue caused by Q
            if $Q.t_s > tCurrent$ then
                DelayQueue.enqueue($Q$, $R_1$)
            else
                Inject $Q$ and $R_1$ into the network
                ActiveQueue.enqueue($Q$, $R_1$)
            InjectQueue.dequeue($Q$, $R_1$)
        for each Q in DelayQueue do
            delete all hidden queries in DelayQueue caused by Q
            modify $Q.Window$ if it is delayed by some query(s) in DelayQueue
end
```

1. If $R'_1 \subseteq R''_1$

    (a) If the join predicate is $att_j = att_h$, then Q''s result during $\triangle W'(= W' - W)$ is completely covered by the result of Q'' (note that this is analogous to *rule* B1(a)). Consequently , the entire result of Q' is covered by the result of Q union result of Q''.

2. If $R''_1 \subseteq R'_1$

    (a) If the join predicate is $att_j \neq att_h$, then similar to *rule* B2(b) and the above, result of Q' during $\triangle W'(= W' - W)$ is completely covered by the result of Q''; its entire result is covered by the union of the result of Q and Q''.

3. If the join predicate is $att_j\ \theta\ att_h$ where $\theta \in \{<, >, \leq, \geq\}$, result table R_1 (or R'_1 or R''_1) is of size 1. Assume $R_1 = \{a\}$, $R'_1 = \{b\}$ and $R''_1 = \{c\}$.

    (a) If $\theta \in \{<, \leq\}$ and $c \leq b$, then Q''s answer (within the new window $\triangle W'$) is completely included in Q'''s answer. Note that the relationship between Q and Q'' does not matter. No matter whether Q'' is delayed by Q if $a \leq c$ or Q'' is injected into the network at once if $c < a$, this would not affect the fact that Q''s answer is fully included in Q''.

    (b) If $\theta \in \{>, \geq\}$ and $c \geq b$, then Q''s answer (within the new window $\triangle W'$) is completely included in Q'''s answer. Similarly, the relationship between Q and Q'' does not affect either.

In all the above 3 cases, we call Q' a *hidden query.* Once we detect a hidden query, it is removed from the delayed query list and need not to be injected into the network; part of the Q''s results will be handled by Q and the rest will be handled by Q''. For cases where two queries have exactly the same join results for overlapping window (covered by *rule* B3 and B5(c)), we delay the newly generated query since it may become a hidden query resulting from later queries and we can save one query dissemination.

Let us use another example to illustrate *Rule C*3. Assume that $R_1 = \{60\}$ and the self-join predicate is "$att_j < att_h$", i.e., Q finds sensor readings larger than 60. Let $R'_1 = \{70\}$,

ts	Active	Delayed	Remarks
1-4	$\varnothing$	$\varnothing$	
5-6	q_1	$\varnothing$	start q_1 at $ts = 5$
7	q_1	q_2	delay q_2, q_2 = (60, 16, 17)
8-9	q_3	$\varnothing$	stop q_1, start q_3 at $ts = 8$ remove hidden query q_2
10-11	q_3	q_4	delay q_4, $q_4 = (70, 19, 20)$
12-13	q_3	q_4, q_5	delay q_5, $q_5 = (80, 21, 22)$
14-18	q_3	q_6	delay q_6, $q_6 = (60, 19, 24)$, remove hidden query q_4, q_5
19-24	q_6	$\varnothing$	start q_6 at $ts = 19$

Table 1: Example of window query triggering.

i.e., query Q' asks for data larger than 70. In this case, due to *Rule B*, Q' is delayed to trigger till when Q ends. Assume now that phase one creates $R''_1 = \{40\}$, triggering query Q'' that asks for readings larger than 40 as shown in Figure 4. According to *rule C*, Q' is a hidden query and need not to be injected into the sensor network.

Based on above rules, we develop Algorithm 2 for the base station to handle continuous monitoring in sensor networks(we show only phase two). The input of the algorithm is a stream of new queries continuously generated from result tuples of phase one by applying *rule A*. The output of the algorithm is a stream of window queries (along with the result tables to be used) that will be injected into the network for execution. The basic idea of the algorithm is that we try to delay issuing a new query as far as possible. In Algorithm 2, *ActiveQueue* stores queries that are currently running while queries in *DelayQueue* are those delayed by others. *NewqueryQueue* stores queries that will be checked in the current time clock to decide whether to inject into the network or not. Queries in *InjectQueue* are candidates that may be injected into the network.

Let us use an example to illustrate how the algorithm works. We consider join predicate $att_j < att_h$ and the window size is defined as 10 sampling intervals. Each new query q is described by (v, t_s, t_e) where t_s (t_e) is the start (end) timestamp of q's window and q's phase one result table is $R_1 = \{v\}$. The join predicate of q may be rewritten as $v < att_h$. Let's consider a series of new queries $\{q_1, q_2, q_3, q_4, q_5, q_6\}$, which $q_1 = (50, 5, 15)$, $q_2 = (60, 7, 17)$, $q_3 = (40, 8, 18)$, $q_4 = (70, 10, 20)$, $q_5 = (80, 12, 22)$ and $q_6 = (60, 14, 24)$. The process of handling these 6 queries are illustrated in Table 1, where ts is timestamp, *Active* indicates queries that are executed currently and *Delayed* shows delayed queries. At $ts = 12$ when q_5 is formed, it is delayed not only by the active query q_3 but also delayed by q_4, and therefore the start time of its window becomes 21. At $ts = 14$ when q_6 is formed, although it is delayed by q_3, it forces q_4 and q_5 to become hidden queries. In all, only three queries out of six are propagated into the network, which illustrates that our algorithm helps saving up to 50% in query transmissions in this example.

When the self-joins whose join predicate is of the type $att_j\ \theta\ att_h$ where $\theta \in \{<, >, \leq, \geq\}$, we can get the following interesting results for Algorithm 2:

Lemma 1. *Consider a self-join query with join predicate involving $\{<, >, \leq, \geq\}$. For any two phase two queries Q and Q' with overlapping windows, either Q is (partially) answered by Q' or Q' is (partially) answered by Q.*

Lemma 2. *Consider a self-join query with join predicate involving $\{<, >, \leq, \geq\}$. For each time clock, there is at most one active phase two query running in the network, which cannot be (partially) answered by any other query with a overlapping window.*

Theorem 1. *Consider a self-join query with join join predicate involving $\{<, >, \leq, \geq\}$. The number of phase two queries injected by Algorithm 2 is minimal.*

Due to lack of space, we omit the proof of Theorem 1. Since the two lemmas hold, Algorithm 2 will not issue any unnecessary queries into the network.

For self-joins with predicate involving $\{=, \neq\}$, the above two lemmas do not hold. Assume the operator is $=$ and there are three queries Q, Q' and Q'' with corresponding phase one result tables $R_1 = \{2, 3\}$, $R'_1 = \{4, 5\}$ and $R''_1 = \{2, 3, 4\}$. Both Q and Q' are active queries and Q'' is the subsequent execution of the same self-join. Because $R''_1 \nsubseteq R_1$ and $R''_1 \nsubseteq R'_1$, according to Algorithm 2, Q'' will be injected into the network. However, $R''_1 \subset R_1 \cup R'_1$ and Q'''s answer during overlapping window will be included in the union of the results of Q and Q'. Therefore, Q'' does not need to be injected into the network. A direct improvement to Algorithm 2 is to calculate the union of result tables of all active queries for each overlapping window. At each time cycle, when a new query comes, the result table is compared with the union. Similar operations as in rules B and C could be applied.

Processing on Phase One Result Table

When the window self-join of phase two involves join predicates with $\{=, \neq\}$, the entire phase one result table needs to be disseminated into the network. This may incur high transmission cost especially when the size of the result table is not trivial. A possible solution is to express the result table in a compressed way while maintain all the important information. When the range of the values of the result table is known, we may use a bitmap to represent a result table, i.e., if a belongs to the result table, the a's bit is set to 1. Using bitmap also eases the comparison among result tables. Whether one result table is a subset of the other can now be easily known by XORing their bitmaps and comparing it to 0.

It is expected that fewer phase two queries are activated in the network by continuous TPSJ, therefore saving query transmission cost as well as query execution cost at the sensor nodes, resulting in power savings. The algorithm also stores fewer queries locally, saving storage at the sensor nodes, which is a limited source for such a small device. Moreover, our algorithm also simplifies the work at sensor nodes. Each sensor maintains a table storing active queries. When a new query arrives, it is evaluated over historical data. Tuples that satisfy the join predicate and window predicate are transmitted to the base station. New tuples will be evaluated over all active queries. A tuple is sent out once though it may satisfy several active queries.

6. EXPERIMENTS

We have implemented and evaluated our TPSJ scheme using our own simulator. The size and shape of the network topology are varied in our experiments. We simulate four configurations of sensor network sizes, with the sensor nodes organized in a 2 × 2, 4 × 4, 6 × 6 and 8 × 8 grid[3]. The

[3] Experiments on larger grids with different density have shown consistent results as discussed in this section.

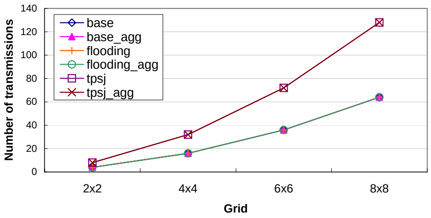


Figure 5: Query transmissions

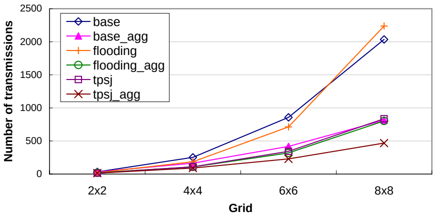


Figure 6: Result transmissions

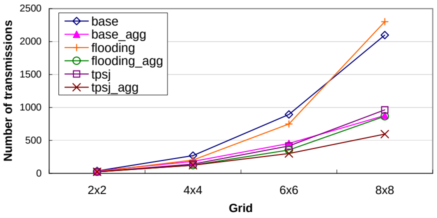


Figure 7: Total transmissions

spacing between each row and column of nodes is set at 20 feet. The transmission radius is set at 50 feet. We adopt similar routing tree construction method as in TinyDB. In the routing tree, each node directly points to its parent, to which its sensor data are sent. The user queries we consider are of the following form:

Sample User Query:

```
Select $S_1$.nodeid, $S_1$.A, $S_1$.B, $S_2$.nodeid, $S_2$.B
From Sensor As $S_1$, Sensor As $S_2$
Where $S_1.A \ < \ \delta$
And $S_2.B \ opt \ S_1.B$
And $Window(S_1.ts, S_2.ts, D)$
```

where $opt \in \{<, >, \leq, \geq, =, \neq\}$. Queries are propagated into the network from the root node along the routing tree by broadcasting to its child nodes. Each node will receive the query from its parent node and rebroadcast it to all its children.

Besides TPSJ, we have implemented two other methods. One is the centralized method we discussed in Section 1, where all sensor readings are transmitted back to the base station for query evaluation (referred to as BASE in the remainder). The other method is *flooding*. In this method, each sensor node broadcasts its readings satisfying the selection predicate throughout the network. Self-joins are performed locally at each node by evaluating local readings over those from other nodes. This method requires each node to store readings received from other nodes. *Flooding* works similarly as TPSJ as both methods use the results of the selection query to filter future readings. The only difference lies in that the results of the selection query are not transmitted back to the base station for possible processing in *flooding*. For each method, we also implement two versions. One is with aggregation, in which multiple data transmissions arriving at the same node may be aggregated before sending. The other is without aggregation. Note that the packet size for one radio transmission is fixed. Therefore, data larger than one packet cannot be aggregated and will be transmitted in multiple transmissions.

Our simulator also handles node failures. In our implementation, each node periodically broadcasts one message to all of its neighbors to inform them that it is active. Each node maintains a neighbor table, recording the number of messages that it has received from that node and the number of hops that node is away from the root. If one node fails to hear from its parent for some time, it will choose a new parent node from its neighbor table based on the historical information about transmission quality and distance from the root.

In the following experiments, we examine the performance of both TPSJ and Continuous TPSJ.

6.1 Experiments for TPSJ

The first set of experiments examines TPSJ for one window. The key performance metric we use is power savings. We use the number of radio transmissions as an approximation of power savings [1]. We first consider user queries with *opt* equal to '<'[4].

6.1.1 Network Topology

This experiment tests the effect of network topology and the total number of sensors on the number of transmissions. We measure the number of transmissions in-network and identify two cost categories. One is the cost for query propagation, the other is cost of result transmissions. We set the window size as 10 sampling intervals and the selection and self-join selectivity as 0.4 and vary the underlying network topology among 2 × 2, 4 × 4, 6 × 6 and 8 × 8 grids. Note that the selectivity of the selection predicate is to limit the amount of tuples that need to be flooded in the network by *flooding*.

First, we compare the number of query transmissions incurred by our TPSJ scheme with that of BASE and *flooding*. Figure 5 shows that the number of query transmissions increases as the size of the grid increases. This is expected as more transmissions are needed to make the query reach more nodes at deeper depth as the grid size increases. Because TPSJ incurs two-phase query processing, more information need to be sent into the network than the other two methods (as shown in Figure 5). Since each time only one query is propagated into the network along the routing tree, no aggregation is applicable here.

Figure 6 shows the number of result transmissions for each method. All methods perform better after aggregation is applied, as results are aggregated as much as possible when their routes to the same destination nodes converge in the routing tree. When the grid size is small, *flooding* performs better than BASE. However, *flooding* performs worse than BASE without aggregation in the 8x8 grid. This is because, in *flooding*, although self-joins are performed locally and less results are transmitted back to the base station as TPSJ, the cost for flooding local readings to others dominates under such a small window. Among the three methods, TPSJ performs best both with aggregation and without aggregation. The reason is twofold. TPSJ filters out many irrelevant tuples especially in phase two. Although *flooding* could achieve similar filtering results, the cost of flooding the results satisfying the selection predicate in-network is too high.

The total number of transmissions is depicted in Figure 7. Although TPSJ incurs some extra cost in sending more information into the network for filtering, its overall total num-

[4] Similar results could be achieved for *opt* $\in \{>, \leq, \geq\}$.

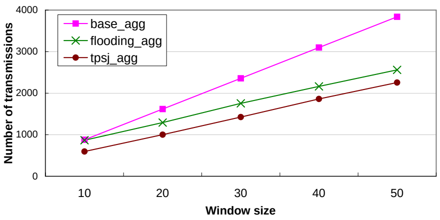


Figure 8: Total transmissions with different window size

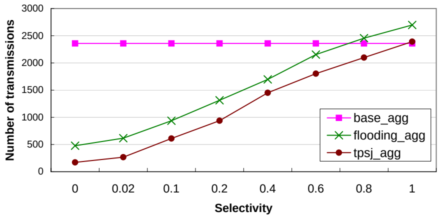


Figure 9: Total transmissions with different self-join selectivity

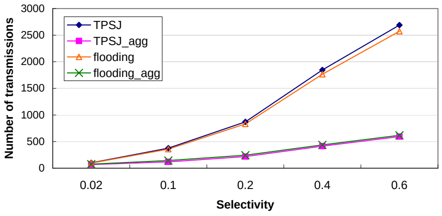


Figure 10: Transmissions for result table dissemination

ber of transmissions is much less than that of the other two methods. Even though the time window of 10 sampling intervals is quite small, the decrease in the total number of transmissions is obvious. The total number of transmissions in TPSJ is more than 50% less than that of BASE and *flooding* without aggregation and 30% less after aggregation is applied in the 8x8 grid. As shown in this experiment, the three methods without aggregation perform worse than their corresponding counterpart with aggregation. So, for the following two experiments, we shall restrict our discussion to the aggregation-based methods.

6.1.2 Window Size

Next, we examine the effect of window size on the total number of transmissions used in the three methods. In this experiment, the network topology is set as an 8×8 grid. The selection and join selectivity is set to 0.4. We vary the window size from 10 to 50 sampling intervals. The results are shown in Figure 8. As the window size increases, the total number of transmissions also increases. The difference between the number of transmissions sent in TPSJ and in BASE also increases as the window size becomes larger. This is because, as the window size increases, more result tuples will be filtered out by TPSJ, and thus reducing the amount of transmissions in the network. However, the difference between TPSJ and *flooding* does not increase with larger window size and remains almost the same. This is because in the one window case, the cost of flooding results satisfying the selection predicate is counted only once.

6.1.3 Self-Join Selectivity

In the above experiments, the join selectivity is fixed. In this experiment, we examine how well TPSJ works under different join selectivity. We use a window size of 30 sampling intervals, because a larger window size will guarantee that there are results even if the selectivity is very low. The network topology is again an 8×8 grid. The selection selectivity is 0.4, while the self-join selectivity is varied from 0 to 1. We measure the total number of transmissions of the three methods. As Figure 9 shows as expected, the total number of transmissions incurred by TPSJ and *flooding* increases with higher selectivity. On the other hand, the number of transmissions in BASE could be viewed as constant as it simply transmits all tuples back to the base station, and thus is not affected by the selectivity. Even with high selectivity, TPSJ generally outperforms BASE by incur less transmissions. When the selectivity reaches 1, TPSJ works just like BASE by sending back all the tuples during phase two. However, this only tests TPSJ at the limit as selectivity as high as 1 are not practical and of less use for real applications.

6.1.4 Result Table Dissemination

In the above experiments, the user query involved '$<$' operator in the self-join predicate. As such, only one tuple has to be disseminated into the network during phase two. The tuple could be contained in the query of phase two and will thus incur no extra transmissions. However, when the operator *opt* $\in \{=, \neq\}$, the whole result table of phase one has to be disseminated into the network. In this section, we examine the cost of result table dissemination for one window TPSJ in the 8x8 grid. We vary the selectivity of the selection query in phase one, which decides the approximate size of the result table, namely, how many nodes of the total 64 nodes will contribute to the result table. We compare TPSJ with *flooding*, which uses flooding to disseminate the information of the results of phase one. For TPSJ, we measure the number of transmissions used to collect results back in phase one plus that for result table dissemination in phase two. Aggregation is applied to both methods. For result table dissemination in phase two of TPSJ, aggregation simply defines at most how many result tuples could be sent in one radio transmission.

The results are shown in Figure 10. When no aggregation is applied, the number of transmissions used in TPSJ is slightly more than that of *flooding*. This is expected as in TPSJ, results of phase one have to be sent to the base station before being disseminated into the network. When aggregation is applied, fewer transmissions are used in TPSJ than in *flooding* with higher selectivity. This is because aggregation are less well utilized in *flooding* as two tuples may be aggregated only when they meet each other at some intermediate node while in TPSJ, the result table is disseminated from the root, which will send the table in the most economical way. As selectivity increases, aggregation achieves much more savings for both methods. From this experiment, we note that if no encoding technique is applied to reduce the cost of result table dissemination, TPSJ will incur similar high data transmission as that of *flooding*.

6.2 Experiments for Continuous TPSJ

In this set of experiments, we examine the performance of continuous TPSJ, especially the performance of Algorithm 2 for triggering window self-joins. We compare our triggering algorithm with the naive method with only *rule A* applied. The key performance metric is savings in query transmissions. Assume n_1 is the number of queries sent into the network by the naive method and n_2 is the number of queries sent into the network by Continuous TPSJ. Savings is defined as $1 - n_2/n_1$.

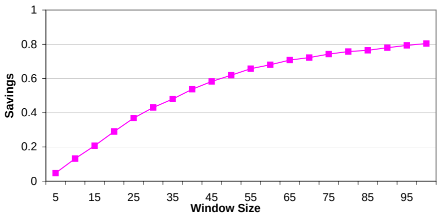


Figure 11: Savings with different window size

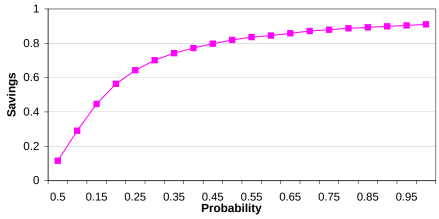


Figure 12: Savings with different probability

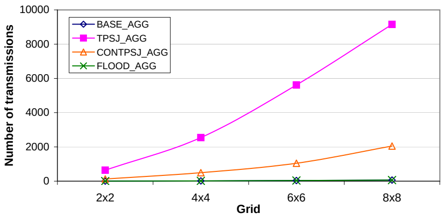


Figure 13: Query transmissions in continuous monitoring

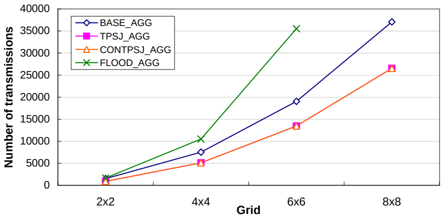


Figure 14: Result transmissions in continuous monitoring

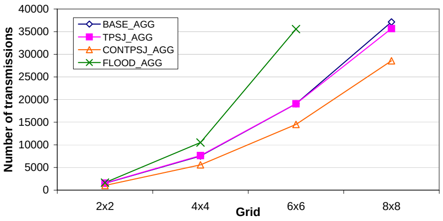


Figure 15: Total transmissions in continuous monitoring

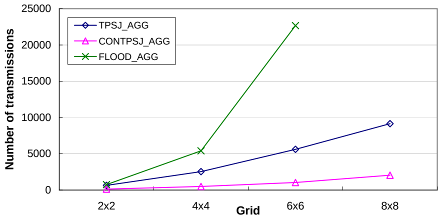


Figure 16: Result table dissemination

We first consider user queries where *opt* $\in \{<, >, \leq, \geq\}$. Therefore the result table to be disseminated into the network consists of only one tuple. The information of this particular tuple is used to rewrite the window self-join query of phase two. In the following experiments, we simulate the generation of new window self-join queries using uniform distribution, i.e., at each time cycle a new query arrives with equal probability. For each new window self-join described as (v, t_s, t_e), the value of v is randomly generated from range [1, 100].

6.2.1 *Window Size*

First, we examine the effect of window size for self-join upon the savings. We vary the window size from 5 to 100 sampling intervals. The arrival probability of new queries is 0.1. We run the experiment for 10000 time cycles (1 time cycle = 1 sampling interval). The results are shown in Figure 11. As we can see, the savings increases with larger windows. When the window size reaches 40 sampling intervals, more than half of the new queries are saved from being injected and run in the network. This is easy to understand as window size increases, more overlaps occur among consecutive queries and thus the probability increases that one query may be partially or fully answered by another.

6.2.2 *Arrival Probability*

Next, we vary the arrival probability of new queries from 0.05 to 1 and plot the savings by continuous TPSJ for an execution of 10000 time cycles. The window size is set as 20 sampling intervals. The results are shown in Figure 12. We can see that the savings increases with higher probability. When the arrival rate of new queries is 0.2, i.e., every 5 time cycles a new query comes, more than half of them become hidden queries. When the arrival rate is high, e.g., 1, which may become a very hard problem for naive method as it has to issue new query into the network almost every epoch, only 10% out of all are injected by continuous TPSJ. An interesting conclusion we may come to is that although more new queries are generated with higher arrival rate, the actual number of queries that are disseminated into the network does not increase much. This is because when arrival probability gets higher, average number of queries over time increases and thus the probability of query delay and hidden queries also increases.

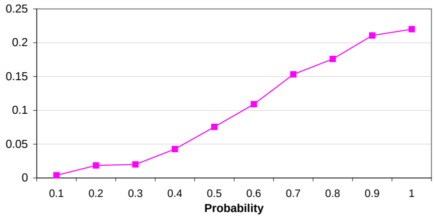


Figure 17: Savings with different probability

6.2.3 *In-network Transmissions*

In this set of experiments, we simulate the running and evaluating of user queries by continuous TPSJ over a long period of time, covering multiple windows. We measure the number of transmissions used to disseminate queries and results respectively over different grids. The probability that a new window self-join will be generated is 0.3. The selectivity for selection query is 0.4; while the self-join selectivity is varied from 0.3 to 0.6 so that different window self-joins are generated. The window size is set as 30 sampling intervals. We run the experiments for 500 time cycles and compare continuous TPSJ (CONTPSJ here after) with BASE and *flooding*. We assume that aggregation is applied to each method[5]. Here we use TPSJ to represent the naive two-phase self-join for continuous queries.

As expected, cost for query dissemination of BASE and

[5] If no aggregation is applied in continuous monitoring, *flooding* will trigger too many transmissions for disseminating results of selection query. It is expected that many collisions will happen in the network. Therefore here we do not consider each method without aggregation.

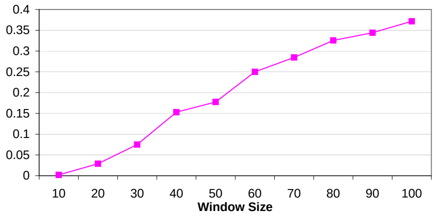


Figure 18: Savings with different window size

flooding could be neglected over such a long period; while TPSJ incurs much more query transmissions than CONTPSJ (Figure 13). This is because for a period of 500 time cycles, around 150 new queries are generated. TPSJ will inject all these queries into the network while only 30 on average are injected by CONTPSJ. On the other hand, much fewer transmissions are used by TPSJ and CONTPSJ to collect results back to the base station (Figure 14). Although the self-join selectivity is varied from 0.3 to 0.6, queries with lower selectivity is more likely to be delayed by queries with higher selectivity during overlapping windows and even become hidden ones. Therefore, it is expected that for a large amount of time over the whole period, nearly 60% nodes are reporting back to the base station. CONTPSJ outperforms under such high selectivity (Figure 15). Note that *flooding* is not applicable in large grid.

We also compare the number of transmissions for disseminating phase one results by *flooding* with that for disseminating new window queries by naive TPSJ and CONTPSJ. From Figure 16 we could see that similar number of transmissions are used in small grid while much more transmissions are used by *flooding* in larger grid. Therefore, although *flooding* could also achieve in-network filtering by broadcasting results of phase one, the cost is too high to be practical.

6.2.4 Result Table Processing in Continuous TPSJ

In this set of experiments, we study the performance of continuous TPSJ for triggering window self-joins with result tables. We consider user queries involving join operators of $\{=, \neq\}$. In section 5, we have discuss how to improve Algorithm 2 for join operator of $\{=, \neq\}$. We use CONTPSJ to denote the original Algorithm 2 while CONTPSJ* with the improvement applied and compare these two methods. The range of the values in the result table is 30 and each result table contains 1 to 10 values randomly chosen. Similarly, we vary the arrival rate and the window size. The experiments are run for 10000 time cycles. Assume CONTPSJ issues n_1 window self-joins into the network while CONTPSJ* n_2. We calculate $1 - n_2/n_1$ to show how much extent CONTPSJ* performs better than CONTPSJ.

First, we fix the window size at 30 sampling intervals and vary the arrival probability of new queries. The results are shown in Figure 17. When the arrival rate is low, both CONTPSJ* and CONTPSJ performs poorly. However, with increasing arrival probability, CONTPSJ* performs better than CONTPSJ and injects 20% less queries into the network when the arrival rate is 0.9. Next, we fix the arrival rate of new queries as 0.5 and vary the window size. From Figure 18, we see that with increasing window size, the difference between CONTPSJ and CONTPSJ* also increases. The reason why CONTPSJ* performs better is as follows. For user queries involving $=$ or $\neq$ operators, the probability for query delay or hidden queries decreases as the relationship of containment between two result sets usually does not hold. However, when the union of result sets of running queries covers most of the interesting values, new coming queries with overlapping windows are more likely to be delayed or become hidden.

7. CONCLUSIONS

In this paper, we have studied the problem of in-network execution of monitoring queries for continuous event detection in sensor networks. We proposed a Two-Phase Self-Join (TPSJ) technique that efficiently processes self-joins in-network. Our experimental results show that TPSJ can significantly reduce the data transmission cost in most cases. We further extend TPSJ to handle continuous monitoring and propose an algorithm for efficient window query triggering. Our method can also be applied to ordinary joins besides self-join, which requires minor modifications to TPSJ.

In the future, we plan to extend this work in the following directions. First, we want to extend our scheme to handle longer series of tuples and more complicated predicates using a similar multi-phase query process, and at the same time seek further performance enhancement. Furthermore, we will integrate our method with multiple query optimization techniques [15] to handle multiple join query processing in sensor networks. These works will be carried out with the goal of producing energy efficient algorithms or implementations.

8. REFERENCES

[1] D. J. Abadi, S. Madden, and W. Lindner. Reed: Robust, efficient filtering and event detection in sensor networks. In *VLDB*, 2005.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[3] S. Babu, J. W. Kamesh Munagala, and R. Motwani. Adaptive caching for continuous queries. In *ICDE*, 2005.

[4] V. Chowdhary and H. Gupta. Communication-efficient implementation of join in sensor networks. In *DASFAA*, 2005.

[5] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, 2004.

[6] A. Deligiannakis, Y. Kotidis, and N. Roussopoulos. Hierarchical in-network data aggregation with quality guarantees. In *EDBT*, 2004.

[7] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, 2004.

[8] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, 2003.

[9] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Stream join: Tracking moving objects in sensor-network databases. In *SSDBM*, 2003.

[10] I. Lazaridis and S. Mehrotra. Capturing sensor-generated time series with quality guarantees. In *ICDE*, 2003.

[11] Q. Li and D. Rus. Global clock synchronization in sensor networks. In *INFOCOM*, 2004.

[12] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. In *TODS*, 2005.

[13] S. R. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[14] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *PODS*, 2005.

[15] S. Xiang, H. Lim, K. L. Tan, and Y. Zhou. Two-tier multiple query optimization for sensor networks. In *ICDCS*, 2007.

[16] Y. Yao and J. Gehrke. Query processing for sensor networks. In *CIDR*, 2003.