A Distributed Algorithm for Joins in Sensor Networks

Alexandru Coman Mario A. Nascimento
Department of Computing Science
University of Alberta, Canada
{acoman | mn }@cs.ualberta.ca

Abstract

Given their autonomy, flexibility and large range of functionality, wireless sensor networks can be used as an effective and discrete means for monitoring data in many domains. Typical sensor nodes are very constrained, in particular regarding their energy and memory resources. Thus, any query processing solution over these devices should consider their limitations. We investigate the problem of processing join queries within a sensor network. Due to the limited memory at nodes, joins are typically processed in a distributed manner over a set of nodes. Previous approaches have either assumed that the join processing nodes have sufficient memory to buffer the subset of the join relations assigned to them, or that the amount of available memory at nodes is known in advance. These assumptions are not realistic for most scenarios. In this context we propose and investigate `DIJ`*, a distributed algorithm for join processing that considers the memory limitations at nodes and does not make a priori assumptions on the available memory at the processing nodes. At the same time, our algorithm still aims at minimizing the energy cost of query processing.*

1. Introduction

Recent technological advances, decreasing production costs and increasing capabilities have made sensor networks suitable for many applications, including environmental monitoring, warehouse management and battlefield surveillance. Despite the relative novelty and small number of real-life deployments, sensor networks are considered a highly promising technology that will change the way we interact with our environment [13]. Typical sensor networks will be typically be formed by a large number of small, radio-enabled, sensing nodes. Each node is capable of observing the environment, storing the observed values, processing them and exchanging them with other nodes over the wireless network. While these capabilities are expected to rapidly grow in the near future, the energy source, be it either a battery or some sort of energy harvesting [8], is likely to remain the main limitation of these devices. Hence, energy efficient data processing and networking protocols must be developed in order to make the long-term use of such devices practical. Our focus is on energy efficient processing of queries, joins in particular, over sensor networks. We study this problem in an environment where each sensor node is only aware of the existence of the other sensor nodes located within its wireless communication range, and the query can be introduced in the network at any node.

Users query the sensor network to retrieve the collected data on the monitored environment. The most popular form for expressing queries in a sensor network is using an SQL-like declarative language [6]. The data collected in the sensor network can be seen as one relation distributed over the sensor nodes, called the *sensor relation* in the following. The queries typically accept one or more of the following operators [6, 9]: selection, projection, union, grouping and aggregations. We note that the join operation in sensor networks has been mostly neglected in the literature.

A scenario where join queries are important is as follows. National Parks administration is interested in long-term monitoring of the animals in the managed park. A sensor network is deployed over the park, with the task of monitoring the animals (e.g., using RFID sensing). Park rangers patrol the park and, upon observing certain patterns, query the sensor network through mobile devices to find information of interest. For instance, upon finding two animals killed in region A, respectively B, the rangers need to find what animals, possibly ill of rabies, have killed them. The ranger would issue the query *"What animals have been in both region A and B between times T1 and T2?"*. If joins cannot be processed in-network, then two, possibly long, lists of animals IDs appearing in each region will be retrieved and joined at the user's device. On the other hand, if the join is processed in-network, only possibly very few animal IDs are retrieved, substantially reducing the communication cost.

In this paper we focus on the processing of the join operator in sensor networks. Since the energy required for communication is three to four orders of magnitude higher than the energy required by sensing and computation [9], it is important to minimize the energy cost of communication during query processing. Recently, a few works addressed in-network processing of join queries. Bonfils and Bonnet [3] investigate placing a correlation operator at a node in the network. Pandit and Gupta [11] propose two algorithms for processing a range-join operator in the network and Yu at al. [16] propose an algorithm for processing equi-joins. These works study the self-join problem where subsets of the sensor relation are joined. Abadi et al. [1] propose several solutions for the join with an external relation, where the sensor relation is joined with a relation stored at the user's device. Coman et al. [5] study the cost of several join processing solutions with respect to the location of the network region where the join is performed. Most previous solutions either assume that nodes have sufficient memory to buffer the partition of the join relations assigned to them for processing, or that the amount of memory available at each node is known in advance and the assigned data partitions can be set accordingly. These assumptions are unrealistic for most scenarios. It is well known that sensor networks are very constrained on main memory and the energy cost of using their flash storage (for those devices that have it) is rather prohibitive to be used for data buffering during query processing. In addition, in large scale sensor networks, it is not feasible for the sensor nodes or the user station to be aware of up-to-date information on memory availability of all network nodes.

In this paper our contributions are three-fold. First we analyze the requirements of a distributed in-network join processing algorithm. Second, to our knowledge, this is the first work to develop and discuss in details a distributed algorithm for in-network join processing. Third, based on the present algorithm, we develop a cost model that can be used to select the most efficient join plan during the execution of the query. Our join algorithm is general in the sense that it can be used with different types of joins, including semi-joins, with minor modifications to the presented algorithm and cost model. As well, our algorithm can be used within the core of other previously proposed join solutions for relaxing their assumptions on memory availability.

2. Background

In our work we consider a sensor network formed by thousands of fixed nodes. Each node has several sensing units (e.g., temperature, RFID reader), a processor, a few kilobytes of main memory for buffer and data processing, a few megabytes of flash storage for long-term storage of sensor observations, fixed-range wireless radio and it is battery operated. These characteristics encompass a wide range of sensor node hardware, making our work independent of a particular sensor platform. Further on, we consider that each node is aware of its location, which is periodically refreshed through GPS or a localization algorithm [14] to account for any variation in a node's position due to environmental hazards. Each node is aware of the nodes located within its wireless range, which form its *1-hop* neighbourhood. A node communicates with nodes other than its *1-hop* neighbours using multi-hop routing over the wireless network. As sensor nodes are not designed for user interaction, users query the sensor network through personal devices, which introduce the query in the network through one of the nodes in their vicinity.

We consider a sensor network deployment where nodes acquire observations periodically and the observations are stored locally for future querying. The data stored at the sensor nodes forms a virtual relation over all nodes, denoted R^*. As nodes store the acquired data locally, each node holds the values of the observations recorded by its sensing units and the time when each recording was performed.

We analyze the self-join processing problem in sensor networks, i.e., the joined relations are spatially and temporally constrained subsets of the sensor relation R^*. We impose no restrictions on the join condition, that is, any tuple from a relation could match any tuple of the other relation. For instance, the query *"What animals have been in both regions R_A and R_B between times T1 and T2?"* (from our example in Section 1) can be expressed in pseudo-SQL as:

```
SELECT   S.animalID
FROM     R* as S, R* as T
WHERE    S.location IN Region R_A
  AND    T.location IN Region R_B
  AND    S.time IN TimeRange [T1, T2]
  AND    T.time IN TimeRange [T1, T2]
  AND    S.animalID = T.animalID
```

Let us denote by A the subset of R^* restricted to Region R_A and by B the subset of R^* restricted to Region R_B. The query may also contain other operators *ops* (selection, projection, etc.) on each tuple of R^* or on the result of the join. As our focus is on join processing, we consider the relations A and B as the resulting relations after the query operators that can be applied individually on each node's relation have been applied. We assume operators that can be processed locally by each sensor node on its stored relation and thus they do not involve any communication. We denote with J the result of the join of relations A and B, including any operators on the join result required by the query: $J = ops_J(A \bowtie B)$. We assume operators on the join result can be processed in a pipelined fashion immediately following the join of two tuples. A general query tree and the notations we use are shown in Figure 1.

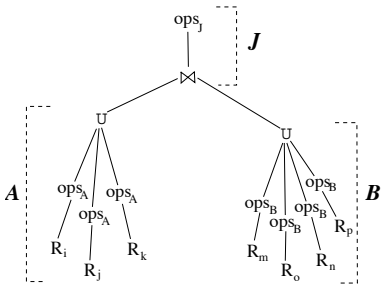

Figure 1. Query tree and notations

3. DIJ: A Distributed Join Processing Algorithm for Sensor Networks

Join processing in sensor networks is a highly complex operation due to the distributed nature of the processing and the limited memory available at nodes. We discuss some of the requirements of an effective and efficient join processing algorithm for sensor networks, namely: distributed processing, memory management and synchronized communication.

- **Distributed processing.** In large scale sensor networks the join operation must be processed in a distributed manner using localized knowledge. For most queries no single node can buffer all the data required for the join. In addition, no node (or user station) has global network knowledge to find the optimal join strategy. As nodes have information only about their neighbourhood, the challenge is to take correct and consistent decisions among nodes with respect to processing the join. For instance, when the join operation is evaluated over a group of nodes, each node in the group must route and buffer tuples such that each pair of join tuples is evaluated exactly once in the join.
- **Memory management.** Each node participating in the join must have sufficient memory to buffer the tuples that it joins and the resulting tuples. For some join queries the join relations are larger than the available memory of a single node. Typically, several nodes must collaborate to process the join operator, pooling their memory and processing resources together. A join processing algorithm should pool these resources together and allocate tasks and data among the participating nodes such that the efficiency of the processing is maximized.
- **Synchronized data flow.** Inter-node communication must be synchronized such that a node does not receive new tuples to process when its memory is full. Otherwise, the node would have to drop some of the buffered or new tuples, which is unacceptable as it may invalidate the result of the join. Thus, each node must fully process the join tuples it holds before receiving any new tuples. A similar problem occurs also for the nodes routing the data. A parent node routing data for multiple children may not be able to buffer all received data before it can forward it. Thus, a join processing algorithm should carefully consider the flow of data during its execution.

In this work we propose a distributed join processing algorithm which considers the above requirements. In our presentation we focus on the join between two restrictions (A and B) of the R^* relation, where the join condition is general (theta-join). Thus, every pair of tuples from relations A and B must be verified against the join condition. Relations A and B are located within regions R_A and R_B and they are joined in-network in a join region R_J. Technique for finding the location of the join region have been presented elsewhere [4, 5, 16] and are orthogonal to our problem. In fact, our algorithm is general with respect to the join relations and their locations and could be used within the core of other previously proposed join solutions (e.g. [5]), including solutions using semi-joins (e.g. [16]). For clarity of presentation we describe our join algorithm in the context of the *Mediated Join* [5] solution.

The *Mediated Join* solution works as follows: relations A and B are sent to the join region (R_J) where they are joined and the resulting relation J is transmitted to the query originator node. (Recall that a query can be posed at any node of the network.) Figure 2 shows in overview the query processing steps and the data flow. The *Mediated Join* seems straightforward based on this description, but there are several issues that must be carefully addressed in the low-level sensor implementation to ensure the correctness of the query result, e.g.:

- How to ensure that both relation A and B are transmitted to the same region R_J?
- How large should region R_J be to have sufficient resources, i.e., memory at nodes, to process the join?
- How should A and B be transmitted such that the join is processed correctly at the nodes in R_J?
- How to process the join in R_J such that the join is processed correctly using minimum resources?

We now describe in details `DIJ`, our join processing algorithm addressing these questions. The steps of `DIJ` are:

1. *Multi-cast the query from originator node O to nodes in R_A and R_B.* Designate the nodes closest to the centres C_A and C_B of the regions R_A, respectively R_B, as regional coordinators. Designate the coordinator location C_J for join region R_J. Disseminate the information about the coordinators along with the query.

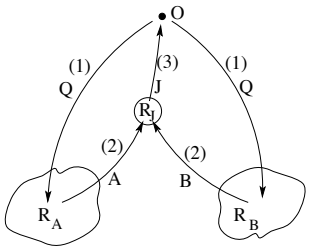

Figure 2. Mediated Join - data flow

2. *Construct routing trees in regions R_A and R_B rooted at their respective coordinators C_A and C_B.*
3. *Collect information on the number of query relevant tuples for each region at the corresponding coordinators*. Each coordinator sends this information to coordinator C_J of the join region R_J.
4. *Construct the join region*. C_J constructs R_J so that it has sufficient memory space at its nodes to buffer A.
5. *Distribute A over R_J.*
   (a) C_J asks C_A to start sending packets with tuples. Once C_J receives A's tuples, it forwards them to a node in R_J with available memory.
   (b) Upon receiving a request for data from C_J, C_A asks for relevant tuples from its children in the routing tree. The process is repeated by all internal tree nodes until all relevant tuples have been forwarded up in the tree.
6. *Broadcast B over R_J*
   (a) Once C_J receives a signal from C_A that it has no more packets (i.e., tuples) to send, C_J asks for one packet with tuples from C_B. When the packet is received, it is broadcast to nodes in R_J.
   (b) Each node in R_J joins the tuples in the packet received from B with its local partition of A, sending the resulting tuples to O. Once the join is complete, each node asks for another packet of B's tuples from C_J.
   (c) Upon receiving a request for tuples from C_J, C_B asks for a *number* of join tuples from its children in the routing tree. The process is repeated by all internal tree nodes if they cannot satisfy the request alone.
   (d) Once C_J receives requests for B's tuples from all nodes in R_J, Step 6 is repeated unless C_B signals that it has no more packets (i.e., tuples) to send.

In the steps above we chose, only for the sake of presentation, that relation A is distributed over the nodes in R_J and relation B is broadcast over the nodes in R_J. The steps above are symmetrical if the roles of A and B are switched, however the actual order *does* matter in terms of query cost. In Section 4 we explore this issue and show how to determine which relation should be distributed and which should be broadcast in order to minimize the cost of the processing the join operator.

Steps 1-3 of *DIJ* are typical to in-network query processing and do not present particular challenges. In Step 4, the join coordinator C_J must request and pool together the memory of other nodes in its vicinity for allocating relation A to these nodes (in Step 5a). This is a non-trivial task as C_J does not have information about the nodes in its vicinity (except its 1-hop neighbours). Steps 5 and 6 also pose a challenge, that is, how to control the flow of tuples efficiently without buffer overflows, ensuring correct execution of the join. We detail these steps in the following.

3.1. Constructing the join region (Step 4)

Once node C_J receives the size of the join relations A and B from C_A and C_B (in Step 1), it must find the nodes in its vicinity where to buffer relation A. *DIJ* uses the following heuristic for this task, called **k-hop-pooling**:

> If C_J alone does not have sufficient memory to buffer relation A, C_J asks its *1-hop* neighbours to report how much memory they have available for processing the query. If relation A is smaller than the total memory available at the *1-hop* neighbours, C_J stops the memory search. Otherwise, C_J asks its *2-hop* neighbours to report their available memory. This process is repeated for *k-hops*, where k represents the number of hops such that the total memory available at the nodes up to k hops away from C_J plus the memory available at C_J is sufficient to buffer relation A.

An interesting question is how much memory should a node allocate for processing a particular query. If the sensor network processes only one join query at a time (e.g., there is a central point that controls the insertion of join queries in the network), then nodes can allocate all the memory they have available for processing the join. However, if nodes allocate all their memory for a query, but several join queries are processed simultaneously in the network, it may happen that a coordinator C_J will not find any nodes with available memory in its immediate vicinity, forcing it to use farther away nodes during processing, and, thus, consuming more energy. For networks where multiple queries may coexist in the network, nodes should allocate only a part of their available memory for a certain query, reserving the rest for other queries. How to actually best allocate the memory of an individual node is orthogonal to our problem. In this work we assume that nodes report as available only the memory

they are willing to use for processing the requested query. Figure 3 shows a possible memory allocation scheme at a node.

3.2 Distributing A over R_J (Step 5)

In this step two tasks are carried out concurrently: C_A requests and gathers relevant tuples (grouped in data packets) from R_A, and C_J distributes the packets received from C_A over R_J.

Once the set of k-hop neighbours that will buffer A has been constructed, C_J asks for relation A from C_A, packet by packet, and distributes each packet of A's tuples in a round-robin fashion to its neighbours, ordered by their hop distance to C_J. When deciding to which node to send a new packet with A's tuples, a straightforward packet allocation strategy would be for C_J to pick a node from its list and send to it all new packets with A's tuples until its allocated memory is full. This strategy has two disadvantages. As all packets use the same route (for most routing algorithms) to get to their destination node, their delivery will be delayed if there is a delay on one of the links in the route. Also, consecutive packets may contain tuples with values such that they all (or many of them) will join with the same tuple in B. In this case, the node holding all these tuples will generate many result tuples that have to be transmitted, delaying the processing of the join. The hop-based round-robin allocation also ensures that all k-hop neighbours have a fair chance of having some free memory at the end of the allocation process, memory that can be used for other queries.

Once node C_A receives a request for tuples from C_J, it has to gather relevant tuples from R_A. If C_A would simply broadcast the tuple request in the routing tree constructed over R_A, nodes in R_A will start sending these tuples toward C_A. As each internal tree node has (likely) several children, it should receive and buffer many packages before being able to send these packages out. Some nodes may not be able to handle such a data flow due to lack of buffer space, possibly dropping some of the packets. To ensure that no packages are lost due to lack of buffer space, we propose a flow synchronization scheme where each node will only buffer one package. In this scheme, the request for A's tuples is transmitted one link at a time. Each node in the routing tree is in one of the following states during the synchronized tuple flow (Figure 4):

- Wait for a tuple request from the parent node (or C_J in the case of C_A) in the routing tree constructed in *Step 2*.
- Send local tuples (from the local storage or receive buffer) to the parent node.
- If buffer space has been freed and there are relevant tuples available at the children nodes in the routing tree, request tuples from a child node that still has tuples to send. Figure 5 shows the routing tree for a region and the information maintained in each node of the tree as tuples are routed from either R_A or R_B to R_J. Note that the number of tuples that each child node will provide has been collected as part of *Step 3*.
- Receive tuples from child, buffer the tuples and update the number of tuples that the child still has available.

Once a node has forwarded to its parent all of A's tuples from its routing sub-tree, it can free all buffers used for processing the query.

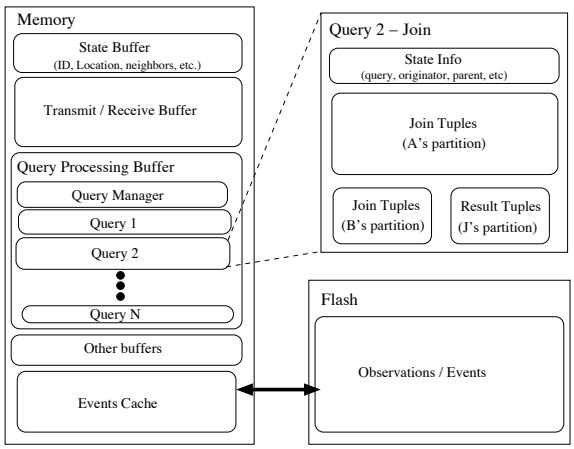

Figure 3. Memory allocation scheme

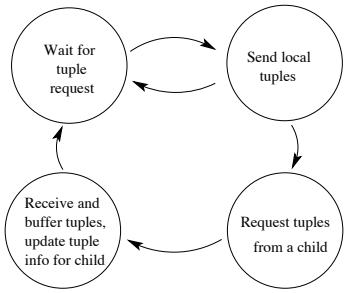

Figure 4. A node's states during tuple routing

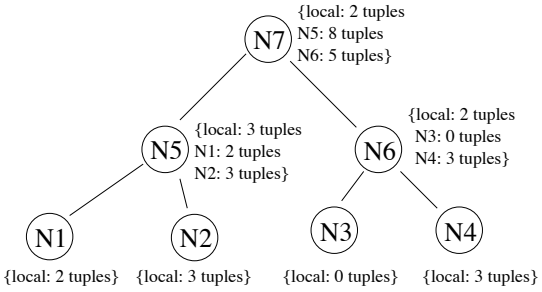

Figure 5. Join tuples information at nodes

3.3. Broadcasting B over R_J (Step 6)

The collection of B's tuples proceeds much like the collection of A's tuples, with one important difference. Whereas C_A gathers and sends *all* of the relevant tuples of A as a a result of a single tuple request from C_J, C_B only sends *one* packet with tuples for each request it receives from C_J. This way, C_J can broadcast such a packet of tuples to all nodes in R_J, wait until all nodes fully process the local joins and send the results, and then request a new packet of tuples from R_B when each node in the join region R_J is ready to receive and join a new set of tuples.

4. Selecting the relation to be distributed

In the previous discussions we have assumed for clarity of presentation that relation A is distributed over the nodes in region R_J and B is broadcast over the nodes in the region. An interesting question is which of the two join relation should be distributed and whether the choice makes a major difference in cost.

Let us focus first on which of the two join relation should be distributed and, subsequently, which should be incrementally broadcast. To decide on this matter, the query optimizer has to estimate the cost of the two options (i.e., distribute A or B) and compare their costs to decide which alternative is more energy efficient. For generality, we derive in the following a cost model for processing the join by distributing relation R_d and broadcasting relation R_b. The actual relations A and B can then be substituted into R_d and R_b (or vice-versa) to estimate the processing costs.

Considering the steps of *DIJ*, the cost of query processing can be decomposed into a sum of components, with one component associated to each step. Several of these components are independent of the choice of the relation that is distributed. Thus, they do not affect the decision of which relation to distribute and do not need to be derived. For instance, we have the cost for disseminating the query in regions A and B (Step 1) and the cost for constructing the routing tree over regions R_A and R_B (Step 2). These costs are identical when processing the join by distributing A or B and do not affect the decision. The steps that have different costs when A or B are the distributed relation R_d are the construction of the join region R_J (Step 4), the distribution of the relation R_d (Step 5a) and the broadcast of the relation R_b (Step 6a). Note that we are only interested in differences in the communication cost between the two alternatives.

4.1. Constructing the join region (Step 4)

As discussed in Section 3.1, we use the **k-hop-pooling** strategy to construct the join region R_J. In each round of memory allocation, C_J broadcasts its request for memory in a hop-wise increasing fashion, until sufficient nodes with the required buffer space are located.

During a round h, each node within h-hops from C_J broadcast the memory request and its 1-hop neighbours receive the request message. Thus, the total energy cost is:

$$E_4^{memreq} = \sum_{h=0}^{k-1}(E_t N_n^h M_r + E_r N_n^h N_n^1 M_r),$$

where N_n^h represents the average number of nodes within h hops from a node, E_t and E_r represents the energy required to transmit, respectively receive, one bit of information and M_r represents the size of the memory request message (in bits). N_n^h is a network-dependent value independent of our technique and it is derived in the Appendix.

When a node receives a memory request message for the first time, it allocates buffer space in its memory and sends the memory information to C_J. The nodes located h-hops away from C_J perform two tasks: they send their own memory information to the nodes located $h-1$ hops away; and they forward the information they have received from the nodes located between $h+1$ and k hops away from C_J. If we denote by M_i the size of the memory information for one node, the total energy cost of collecting the information on available memory is:

$$\begin{aligned} E_4^{meminfo} = & \sum_{h=1}^{k}((E_t+E_r)(N_n^h - N_n^{h-1})M_i \\ & +(E_t+E_r)(N_n^k - N_n^h)M_i) \\ = & (E_t+E_r)(kN_n^k - \sum_{h=1}^{k-1} N_n^h)M_i \end{aligned}$$

Note that $(N_n^h - N_n^{h-1})$ represents the number of nodes h-hops away and $(N_n^k - N_n^{h-1})$ represents the number of nodes located more than h and up to k hops away from C_J. The total energy cost of the fourth step of *DIJ* is:

$$E_4 = E_4^{memreq} + E_4^{meminfo}.$$

Note that the costs of Step 4 do not depend on the join relations directly, but through k which determines the size of the join region R_J and it is determined by the size of the join relation R_d.

Let B_s be the average size (in bits) of the buffer space that each node in R_J can allocate for processing the query. The minimum number of nodes that must be used to store relation R_d in region R_J is $\frac{||R_d||}{B_s}$, where $||R||$ denotes the size (in bits) of relation R. Since nodes are added to R_J in groups based on their hop distance, k is the lowest number of hops such that the nodes within k hops from C_J have sufficient buffer space to buffer R_d:

$$k = \{\texttt{min}\ h \mid N_n^h B_s \geq ||R_d||\}.$$

4.2. Distributing R_d over R_J (Step 5a)

In Step 5a of *DIJ*, C_J receives and distributes relation R_d at the nodes in R_J. Nodes located h hops away from C_J receive from the nodes located $h-1$ hops away from C_J partitions of R_d of size B_s for buffering. They also route toward their destination the partitions B_s allocated to the nodes between $h+1$ and $k-1$ hops away from C_J, as well as the partitions allocated to the nodes k hops away. Note that nodes located k hops away will only buffer whatever is left of R_d instead of B_s as the other nodes do. Therefore, the energy cost for distributing R_d at the nodes in R_J is:

$$\begin{aligned} E_{5a} = & \sum_{h=1}^{k-1} ((E_t + E_r)(N_n^h - N_n^{h-1})B_s \\ & + (E_t + E_r)(N_n^{k-1} - N_n^h)B_s \\ & + (E_t + E_r)(||R_d|| - N_n^{k-1}B_s)) \\ = & (E_t + E_r)((k-1)\,||R_d|| - B_s \sum_{h=0}^{k-2} N_n^h). \end{aligned}$$

4.3. Broadcasting R_b over R_J (Step 6a)

In Step 6a of *DIJ*, C_J broadcasts relation R_b (packet by packet) over the nodes in R_J, where it is joined with the buffered partitions of R_d. Note that only the nodes in R_J up to $k-1$ hops away from C_J need to broadcast R_b so that all nodes participating in the join receive it. The total energy cost of the broadcast is:

$$E_{6a} = E_t N_n^{k-1} ||R_b|| + E_r N_n^{k-1} N_n^1 ||R_b|| \qquad (1)$$

4.4. Discussion

Using the cost models for Steps 4, 5a and 6a of *DIJ*, C_J can determine which of the two join relation should be R_d and which should be R_b. To calculate the energy costs, C_J need to know the value of the parameters used in the models. C_J learns the size of the join relation A and B in the Step 3 of *DIJ*. C_J can estimate B_s based on the size of the available memory at itself and its 1-hop neighbours. We show in the Appendix how N_n^h can be estimated. The other parameters used in the cost model are network or algorithm constants.

5. Cost Model Evaluation

The cost model developed in Section 4 allows C_J to choose which of the join relation should be distributed (R_d) in the the join region R_J and which should be broadcast (R_b) to minimize the cost of processing the join operator. In this section we further investigate the behaviour of *DIJ* based on the cost model. In our evaluation we consider a sensor network with nodes uniformly distributed over a two dimensional region. We are interested in evaluating the relative performance of two alternatives: distributing relation A in R_J and broadcasting relation B over R_J (denoted by $R_d = A$); and distributing relation B in R_J and broadcasting A over R_J (denoted by $R_d = B$). Our measure of efficiency is the energy used for communication while processing the join operator. Thus, we compare the energy cost $E(R_d = A)$ of *DIJ* when A is the distributed relation with the cost $E(R_d = B)$ when B is distributed. We only consider the energy costs of the Steps 4, 5a and 6a as they are the ones that determine the difference between the two options of *DIJ* for which relation to distribute and which to broadcast. Figures 6, 8 and 9 evaluate the relative cost of *DIJ* when $R_d = B$ compared to the cost when $R_d = A$: $E(R_d = B)/E(R_d = A)$. When the cost ratio is equal to 1, both alternatives for which relation to distribute have the same cost. When the cost ratio is lower than 1 it is more efficient to distribute relation B ($R_d = B$) and broadcast relation A, while for cost ratios higher than 1 relation A should be the distributed relation ($R_d = A$). The cost model parameters and their default values used in our evaluation are presented in Table 1.

Our measure of efficiency is the energy used for communication while processing a query. According to [12], the energy used to transmit and receive one bit of information in wireless communication is given by $E_t = \alpha + \gamma \times d^n$ and

Table 1. Cost model parameters

Parameter	Value (default)
Network area	1000x1000
Wireless range (W)	50
Average number of neighbours (N_n^1)	12 ($N = 1655$)
Number of tuples in A	500
Number of tuples in B	500
Number of tuples per node in R_J (B_s/T_s)	25
Size of a tuple (T_s)	192 bits
Size of a memory request message (M_r)	8 bits
Size of a memory information record (M_i)	80 bits

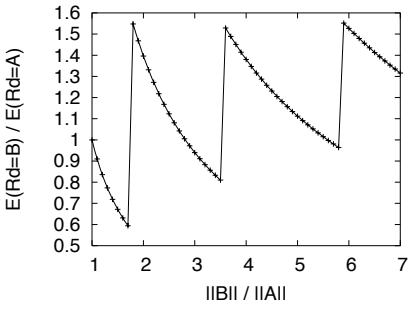

Figure 6. Energy cost ratio for variations in the join relation sizes

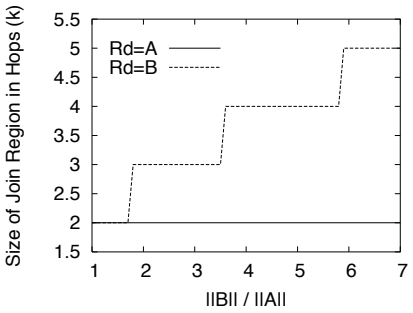

Figure 7. Size of R_J in number of hops for variations in the relative size of A and B

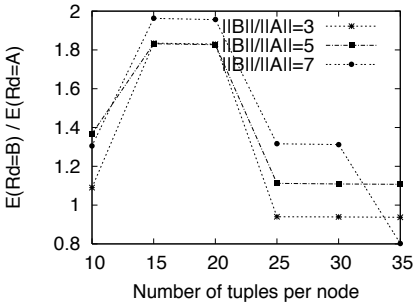

Figure 8. Energy cost ratio for variations in the size of available buffer space

$E_r = \beta$, where d is the distance to which a bit is being transmitted, n is the path loss index, α and β capture the energy dissipated by the communication electronics and γ represents the energy radiated by the power-amp. In our evaluation, we use the following values for these parameters [7]: $\alpha = \beta = 50nJ/bit$, $n = 2$, and $\gamma = 10\ pJ/bit/m^2$.

In the first experiment, we evaluate the relative costs of the Steps 4, 5a and 6a for different ratios between the sizes of the join relations. We keep relation A fixed and we vary B such that it is between 1 to 7 times larger than A. Figure 6 shows the relative performance of the two alternatives. When both relations have the same size ($||B||/||A|| = 1$), the cost of the processing the join operator is the same for both alternatives, as one would expect. For rations $||B||/||A||$ up to 6, the best relation to distribute changes with variations in the ratio. For ratios higher than 6 (we only show ratios up to 7 in the graph), the smaller relation (A) is always the relation that should be distributed for lower join processing costs. The sharp changes in the cost ratio are caused by the increase in the number of hops (k) that are required so that R_J is sufficiently large to store R_d. When k increases, the cost of broadcasting relation R_b in Step 6a increases substantially as another set of nodes are added to R_J. Note that k does not vary for $R_d = A$ as the size of A does not change, but it does vary between 2 and 5 for $R_d = B$ as shown in Figure 7. For instance, when B is between 1.8 to 3.5 times larger than A, nodes up to $k = 3$ hops away from C_J are required to buffer relation B when distributed over R_J. As k stays constant for these ratios, the cost of broadcasting relation A over R_J stays constant as well. At the same time, as the size of B increases, the cost of distributing A over R_J and broadcasting B increases. Thus, the cost ratio $E(R_d = B)/E(R_d = A)$ decreases and it becomes more efficient to distribute B when $||B||/||A||$ is between 2.9 and 3.5. When $||B||/||A|$ reaches 3.6, C_J must contact another "hop" of nodes ($k = 4$) so that R_J is sufficiently large to buffer B. Not only that the cost of distributing B increases with the addition of new nodes, but the cost of broadcasting A over the 4-hop neighbourhood is substantially higher than the cost of broadcasting it over the 3-hop neighbourhood. When this happens, it becomes more efficient to distribute A over the smaller R_J ($k = 2$) required to buffer it and broadcast the larger relation B over the smaller region. In general, the cost of Step 6a (broadcasting) dominates by a large margin the costs for Steps 4 and 5a. Thus, distributing the smaller relation and broadcasting the larger one over the join region R_J performs better for most ratios than broadcasting the smaller relation. The reason is that the cost of broadcasting increases quadratically as the size of R_J increases (see Equation 1). It is only when the size of R_J must be increased to accommodate a larger relation that the processing cost increases drastically, causing a sharp change in the cost ratio.

Figure 8 shows the relative performance of the two alternatives for distributing the join relations when the size of allocated buffer space at nodes (B_s) varies. The variation of B_s affects the size of the join region R_J (through the number of hops k required to reach sufficient nodes) and thus the performance of the two processing alternatives. We show the relative costs for three ratios between the sizes of the join relations. Note that when the two relations are equal in size ($||B||/||A|| = 1$), the processing costs of the two alternatives are equal as well. The relative performance of the alternatives has a similar trend for the three ratios of the relation sizes and, thus, we discuss in detail the behaviour for $||B||/||A|| = 3$. Consistent to the results shown in Figure 6, distributing the smaller relation A and broadcasting the larger relation B is most efficient for more buffer sizes (B_s) due to the large weight of the cost of broadcasting in the total cost. The exception is, again, when the size of R_J is modified (through k). When the number of tuples that can be stored at a node (B_s/T_s) increases from 20 to 25, the number of hops k required for distributing relation B over the nodes in R_J decreases from 4 to 3, while it stays constant for distributing relation A ($k = 2$). Thus, the cost of the alternative that distributes B decreases substantially due to the much reduced cost of broadcasting A over the smaller number of nodes. At the same time, the cost of the alterna-

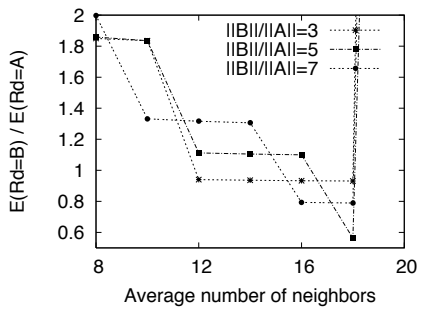

Figure 9. Energy cost ratio for variations in the average number of neighbors (N_n^1)

tive distributing A over R_J varies only slightly as more of A's tuples can be stored closer to C_J (but still up to $k = 2$ hops away from C_J), causing a sharp change in the relative performance of the two solutions. When B_s/T_s varies from 15 to 20 and 25 to 35, the number of hops required for distributing R_d (be it A or B) does not change, and, thus, there is only a small variation in the relative performance of the two alternatives caused by the slightly lower cost of distributing R_d. Similar trends can be observed for $||B||/||A|| = 5$ and $||B||/||A|| = 7$.

Another parameter affecting the relative performance of the two alternatives is the network density. The variation of network density directly affects the size of the join area R_J as nodes farther (in terms of hop-count), or closer, to C_J are required for storing the relation that is distributed when the density decreases, respectively increases. Figure 9 shows the effect of the network density on the relative performance of the two solutions. We evaluate again the relative cost for three ratios of join relation sizes. As the relative performance of the two alternatives shows similar trends for all three ratios, we focus our discussion on $||B||/||A|| = 3$. When the average number of neighbours (N_n^1) is between 8 and 18, only nodes up to $k = 2$ hops are required to buffer relation A in R_J. This effectively means that the cost of the alternative distributing A in R_J varies only slightly, while the alternative that distributes B decreases its cost at a faster rate as k decreases from 4 to 3 when the N_n^1 increases from 10 to 12. When the number of neighbours is 20, the network is sufficiently dense so that C_J uses only its 1-hop neighbours ($k = 1$) to store A. As such, the cost of broadcasting B decreases sharply, causing a similar decrease in the overall processing cost. Note that for $k = 1$, only one node (C_J) need to broadcast relation R_b. This is a sharp decrease in the number of broadcasts since for $N_n^1 = 18$ and $k = 2$ the number of nodes broadcasting R_b is 19 (C_J plus 18 1-hop neighbours). On the other hand, the cost of the alternative distributing B varies only slightly when the N_n^1 increases from 18 to 20. Therefore, there is a sharp change in the relative performance of the two solutions, and it becomes substantially more efficient to distribute A than to distribute B when $N_n^1 = 20$.

6. Related Work

Research on query processing in sensor networks has mostly focused on processing of selection, unions, grouping and aggregation operators [6]. Recently, a few works addressed the processing of join queries.

Adaptive placement of a correlation operator is studied in [3]. Initially, the operator is randomly placed at a network node. The position is progressively refined by moving the operator to the nodes with lower processing cost during the lifetime of a continuous query. The refinement allows the operator placements to adapt to changes in data during the query lifetime. For historical and short continuous queries the solution would perform much worse than the optimal cost due to the initial random placement. The authors focus on the operator's placement problem, assuming that each node that will hold the operator is able to handle the flow and processing of data alone.

Chowdhary and Gupta [4] propose an algorithm for performing joins in-network over a processing region. The algorithm is a form of distributed block-nested loop join and it is similar in spirit with the algorithm proposed in this paper. Differently from us, they do not investigate the allocation of memory at the nodes in the join region and the synchronized data flow. Pandit and Gupta [11] propose two algorithms for in-network processing of the range-join operator. Both works [4, 11] consider that the optimal join location is the weighted centroid of the triangle *ABO*. The centroid has the property that it minimizes the weighted sum of the *squared* distances, and thus it is not optimal.

Yu et al. [16] investigate the processing of self-join queries with equi-joins over historical data in sensor networks. In their solution they constructs synopsis (e.g., histograms) of both join relations, which are then used for filtering out the tuples that will not join. The solution performs best when the join selectivity is high and it is similar to a semi-join. The join of the synopsis is performed in a square join region whose size is determined based on the size of the synopsis, the network density and the average memory available at the join nodes, which is similar in spirit to our approach. When allocating the synopsis to the join partition, they fail to consider the memory available at the individual nodes in their hash-based allocation scheme, which would cause buffer overflows and invalidate the join result. They also assume that nodes have sufficient memory when performing the final join of the filtered tuples.

Abadi et al. [1] study the processing of joins with an external relation. If the external relation is small, it is flooded in the network and the join occurs locally at each node. When the external relation is too large to be stored in the

network, bloom filters and partial joins are used for filtering the sensor tuples. Non-filtered tuples are then joined at the base-station. When the external relation fits into a group of nodes, the join between the external relation and every new generated tuple is performed over the group.

Omotayo et al. [10] study the problem of using the memory of the nodes as a shared resource. Similar to us, they consider the problem of using the memory of some nodes to store or buffer the observations of other nodes. Differently from us, their goal is maximizing the size of the history that is stored in the network. The join operation for streaming sensor data is studied in [2, 15]. Ali at al. [2] study the use of a multi-way join operator for detecting and tracking phenomena. Schmidt at al. [15] focus on re-sampling the sensor streams to allow meaningful temporal joins.

7. Conclusions

In this paper we have discussed in details a technique (DIJ) for processing the theta-join operator in a sensor network. The strength of the technique is that we take into account the memory available at the sensors nodes and the synchronization of the data flow. Both issues have been overlooked or simplified in the existing literature. We have also developed a cost model that allows our technique to be optimized with respect to the size of the join relations and the amount of available memory at the nodes processing the join. Another important aspect is that our technique is general in the sense that it can be re-used in the core of other previously proposed join solutions for relaxing their assumptions on memory availability at nodes. Finally, we studied the technique's behaviour through the cost models under several combinations of query and network parameters. We have shown that the size of the region over which the join is processed (represented by k) has a strong impact on the cost of the processing.

References

[1] D. Abadi, S. Madden, and W. Lindner. REED: robust, efficient filtering and event detection in sensor networks. In *Proc. of VLDB*, pages 769–780, 2005.

[2] M. Ali, W. Aref, and I. Kamel. Scalability management in sensor-network phenomenabases. In *Proc. of SSDBM*, pages 91–100, 2006.

[3] B. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *Proc. of IPSN*, pages 47–62, 2003.

[4] V. Chowdhary and H. Gupta. Communication-efficient implementation of join in sensor networks. In *Proc. of DASFAA*, pages 447–460, 2005.

[5] A. Coman, M. Nascimento, and J. Sander. On join location in sensor networks. In *Proc. of MDM*, 2007.

[6] J. Gehrke and S. Madden. Query processing in sensor networks. *Pervasive Computing*, Jan. 2004.

[7] W. Heinzelman. *Application-Specific Protocol Architectures for Wireless Networks*. PhD thesis, MIT, 2000.

[8] X. Jiang, J. Polastre, and D. Culler. Perpetual environmentally powered sensor networks. In *Proc. of IPSN*, 2005.

[9] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *Proc. of SIGMOD*, pages 491–502, 2003.

[10] A. Omotayo, M. Hammad, and K. Barker. Efficient data harvesting for tracing phenomena in sensor networks. In *Proc. of SSDBM*, pages 59–70, 2006.

[11] A. Pandit and H. Gupta. Communication-efficient implementation of range-join in sensor networks. In *Proc. of DASFAA*, pages 859–869, 2006.

[12] T. Rappaport. *Wireless Communications: Principles and Practice*. Prentice-Hall Inc., 1996.

[13] A. Ricadela. Sensors everywhere. *Information Week*, Jan. 24, 2005.

[14] A. Savvides, M. Srivastava, L. Girod, and D. Estrin. Localization in sensor networks. *Wireless sensor networks*, pages 327–349, 2004.

[15] S. Schmidt, M. Fiedler, and W. Lehner. Source-aware join strategies of sensor data streams. In *Proc. of SSDBM*, pages 123–132, 2005.

[16] H. Yu, E. Lim, and J. Zhang. On in-network synopsis join processing for sensor networks. In *Proc. of MDM*, pages 32–39, 2006.

Appendix – Estimating the number of nodes within *h-hops* from a node

Let A_N be the area of the network, N the number of sensor nodes uniformly distributed over the network area and S the node whose h hop neighbours we try to determine. The number of nodes located up to h-hops away from S is equal to the number of sensor nodes located in an area equal in size to the area where these nodes are located. Let us denote this area with A_h. We have that $N_n^h = N\frac{A_h}{A_N}$ for $h \geq 1$. For $h = 0$ we have $N_n^0 = 1$ to account for S.

We need to find the size of the area A_h. For the 1-hop neighbours, A_1 is equal to the circle of wireless range W and we have $N_n^1 = N\frac{\pi W^2}{A_N}$. The average distance from S to its 1-hop neighbours is $d_{1hop} = \int\int_{A_1} d_{SN_i} dA_1 = \frac{2}{3}W$, where d_{SN_i} represents the distance between S and a 1-hop neighbour N_i. Since the 1-hop neighbours are located in average at distance d_{1hop} away from S, and the neighbours of the 1-hop neighbours could be located as far as W, we have that the 2-hop neighbours of S are located in average within a circle of radius $d_{1hop} + W = \frac{5}{3}W$. Generalizing this result for h-hop neighbours, we have that they are located within a circle of $(h-1)d_{1hop} + W = \frac{2h+1}{3}W$ radius from S. Therefore, we have:

$$N_n^h = \frac{\pi(2h+1)^2W^2}{9A_N}N \quad \text{for} \quad h \geq 1.$$