2005

Multi-way Joins for Sensor-Network Databases

M. H. Ali

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Ibrahim Kamel

Report Number:
05-021

MULTI-WAY JOINS FOR SENSOR-NETWORK DATABASES

M. H. Ali
Walid G. Aref
Ibrahim Kamel

CSD TR #05-021
October 2005

Multi-way Joins for Sensor-Network Databases

M. H. Ali[1] Walid G. Aref[1] Ibrahim Kamel[2]

[1]Department of Computer Science, Purdue University, West Lafayette, IN
{mhali, aref}@cs.purdue.edu
[2]College of Information Systems, Zayed University, U.A.E.
Ibrahim.Kamel@zu.ac.ae

Abstract. The multi-way join operator serves a wide range of data-streaming applications. In this paper, we introduce the *SNJoin* (or Sensor-Network Join) operator that is specially designed for dynamically-configured large-scale sensor networks. *SNJoin* is a multi-way join operator that scales with respect to the number of data-streaming sources without compromising the output rate. Moreover, in a sensor field with hundreds or thousands of sensors, every single sensor does not have to participate in the join. Instead, bundles of sensors that experience the same environmental conditions may join with each other. The *SNJoin* operator joins sensor data partially and updates its result with the arrival of new joining tuples. We introduce the *SNJoin** operator as the distributed variation of the *SNJoin* operator. The ultimate goal of the *SNJoin** operator is to shift the join query processing from the centralized data stream management system (*DSMS*) to the sensor-network level. To avoid unnecessary communication cost, the *SNJoin** operator accepts *relevance feedback* to tune query processing towards sensors that show similar behavior. The relevance feedback is computed based on the join output to guide the join probing sequence to sensors that are more likely to join. Experimental studies illustrate the performance gains of the proposed multi-way join operators.

1 Introduction

With the evolution of data stream processing, the join operation conserves its importance and wide applicability to data-streaming applications, e.g., [7–9, 11, 15]. Multi-way join extends binary join by handling multiple input sources. Sensor networks are major input sources of multiple data streams and, therefore, sensor-network databases experience large popularity of multi-way join queries. In [1, 2], multi-way join queries are used to detect and track the propagation of various phenomena that strike a sensor field. Other applications of multi-way join in sensor networks are object tracking, surveillance, and environmental monitoring [8, 21, 20].

A multi-way join over data streams can be performed using trees of non-blocking binary joins (e.g., *symmetric hash join* [17] or *xjoin* [15]). This technique performs the multi-way join in multiple steps and may incur several delays. Also, the output rate of binary-join trees is sensitive to the join order. For this

reason, binary-join tress are usually equipped with a dynamic scheme for tree reorganization at execution time (e.g., see [4]). To overcome the shortcomings of binary-join trees, the work in [16] introduces the *MJoin* operator, a *single-step* multi-way join operator that is symmetric with respect to all input streams. Hence, *MJoin* produces early output, maximizes the output rate, and avoids reorganization of the query plan at execution time.

Although *MJoin* seems to be satisfactory for a moderate number of data streams, the multi-way join operation over sensor networks has the following four challenges. (1) The *scalability* challenge: Sensor networks are typically deployed in a large scale with hundreds or thousands of sensors. (2) The *dynamic-configuration* challenge: Sensors can be added and removed from the sensor field dynamically based on the network conditions, the sensors' life time, and the availability of additional sensors. (3) The *variable-arity* challenge: It may be prohibitive and meaningless to include all the sensors in the join. Usually, the sensor field spans a wide area such that only subsets of sensors that are exposed to the same environmental conditions are eligible to join with each other. This behavior results in a variable-arity output join tuple. (4) The *distributed-execution* challenge: The join operation should be performed in a distributed fashion where sensor readings join with each other on their route to the destination data stream management system (*DSMS*) taking into consideration the number of transmitted messages. The number of transmitted messages has an impact on the limited power capabilities of the network.

In this paper, we address the distributed execution of the *continuous multi-way window join* query over *dynamically-configured large-scale* sensor networks. As we proceed throughout the sections of this paper, we enhance the join algorithm to address the four previously reported challenges, one at a time. The contributions of this paper can be summarized as follows:

1. In Section 2, we handle the *scalability* challenge by introducing the *SNJoin* operator that scales with respect to the number of input sources.
2. In Section 3, we tune *SNJoin* to deal with the *dynamic-configuration* challenge of sensor networks. Then, we formalize the concept of *variable-arity* multi-way join in large-scale sensor networks and adopt this formalization in the context of the *SNJoin* operator (the *variable-arity* challenge).
3. In Section 4, we introduce the notion of stream query processing with *relevance feedback* to focus the join operation among sensors that show similar behavior. The notion of stream query processing with *relevance feedback* addresses the *variable-arity* challenge.
4. In Section 5, we address the *distributed-execution* challenge. We shift the join query processing from the centralized *DSMS* to the sensor-network level through the *SNJoin** distributed operator.
5. In Section 6, we give a mathematical analysis of the proposed SNJoin and SNJoin* operators. In Section 7, we evaluate the proposed join operators, implemented inside Nile [14]. In terms of output rate, performance results show that the *SNJoin* operator is better than binary join trees by up to 150% and is better than MJoin by up to 60%. Also, query processing with

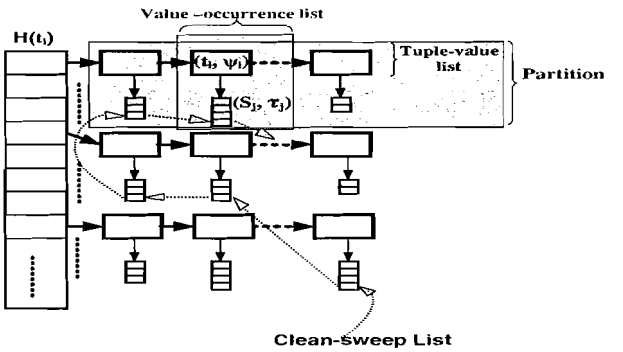


Fig. 1. The structure of the SNJoin hash table.

relevance feedback increases the output rate of SNJoin* by up to 90% and reduces the communication by up to 75%..

2 SNJoin

In this section, we introduce the *SNJoin* operator that performs a *complete* multi-way join among a set of streams over a sliding window (as given by Definition 1). Upon the arrival of a new tuple, say $\hat{t}$, from stream $\hat{S}$, $\hat{t}$ joins with tuples that have the same value from all streams (except its own stream $\hat{S}$) provided that they are within an ω time-window from each other. The multi-way join is *complete* in the sense that every streaming source participates in each output tuple. If a join value is missing in one stream, the output tuple is discarded. This is in contrast to variable-arity multi-way join described in Section 3. Definition 1 is given in terms of a sliding *time-window*. However, the generalization to a sliding *tuple-window* is straightforward. Also, we define a single window ω for all streams, then, we generalize to multiple windows in Section 2.4.

Definition 1. *Given m input streams, $S_1, S_2, \cdots, S_m$, each stream S_i generates tuples of the form $(t_i, [S_i, \tau_i])$ (where t_i is the tuple value that is produced by stream S_i at time τ_i). For a newly arriving tuple $(\hat{t}, [\hat{S}, \hat{\tau}])$, a* complete *multi-way join over window ω produces an output O=$\{(\hat{t}, [\tau_1, \tau_2, \cdots, \hat{\tau}, \cdots, \tau_m])$ such that t_i=$\hat{t}$ and $|\hat{\tau} - \tau_i| \leq \omega$ $\forall\ i = 1 \cdots m,\ S_i \neq \hat{S}\ \}$.*

Hash-based join techniques maintain a hash table per stream. A new input tuple is inserted, based on a hash function, into its own stream's hash table and probes other streams' hash tables looking for matches. With the increase in the number of streaming sources, managing a large number of hash tables becomes costly. To avoid a lengthy join probing sequence, *SNJoin* proposes a single global hash table where all incoming tuples are hashed and are inserted regardless of their streaming sources. Grouping tuples of the same value from various streams in the same partition of a hash table prepares output candidates for the join operation. The details of the *SNJoin* operator are presented in this section.

```
PROCEDURE Insert-Probe
INPUT: (1) a new input tuple (î,[Ŝ, τ̂]) and (2) an SNJoin hash table
OUTPUT: (1) an updated SNJoin hash table and (2) the join output produced by tuple î
1. TVLEntry=TVL[(H(î)].Search(î)
2. VOLEntry=TVLEntry.vol-ptr.Insert(Ŝ, τ̂)
3. TVLEntry.Update-count-summary(Ŝ);
4. CSL.Append(VOLEntry)
5. if TVLEntry.ψ(c1, c2, ··· , cm) = m begin
        temp=TVLEntry.vol-ptr.first;
        while(temp≠NULL and τ̂ − temp.τ ≤ ω) begin
            if Ŝ ≠ temp.S include temp.τ in the join output of î
            temp=temp.next
        end
   end
6. Traverse CSL to delete expired tuples and update associated count summaries
```

Fig. 2. The SNJoin algorithm.

2.1 Data structure

Figure 1 illustrates the structure of the *SNJoin* hash table. The hash table is divided into partitions based on a suitable hash function H. In each partition, all tuple values that appear in the current window ω are chained in a *tuple-value list* (*TVL*), one entry per value. An entry in *TVL* is of the form:

1. t: the tuple value. Notice that a single entry is created per value even if t appears multiple times, whether in a single stream or in multiple streams.
2. $VOL - ptr$: a pointer to the *Value-Occurrence List* (or *VOL*). *VOL* records every occurrence of the value t. An entry in *VOL* contains the following:
   (a) S: an identifier of the stream that produced the value t.
   (b) τ: the timestamp at which t is produced.
   VOL is reverse-ordered based on the tuple's timestamp. A newly-incoming tuple is appended at the head of *VOL*.
3. $\psi(c_1, c_2, \cdots, c_m) \rightarrow \sum_{i=1}^{m}(min(1, c_i))$ (where c_i counts the number of occurrences of t in stream S_i over the last window ω): a count-summary function that keeps track of the tuple count in each stream and returns the number of streams that has the tuple t in the most recent window ω.

Finally, every single occurrence of a tuple $(t,[S, \tau])$ is chained chronologically, i.e., based on their timestamps, in a global *Clean-Sweep List* (or *CSL*)). *CSL* spans all partitions of the hash table to link all tuple occurrences from all streams with the oldest being at the head of the list. The purpose of *CSL* is to expire old tuples once they get outside the sliding window ω.

2.2 Complete join algorithm

The *SNJoin* algorithm is given in Figure 2. With the arrival of a new tuple $\hat{t}$ from stream $\hat{S}$ at timestamp $\hat{\tau}$, the hash function H is applied over $\hat{t}$ to determine the partition where the tuple should go. Then, the partition's *tuple value list (TVL)* is searched to return a handle to the tuple's entry in *TVL*, if the tuple is found. Otherwise, a new entry in *TVL* is created (Step 1). The tuple's stream ($\hat{S}$) and

the tuple's timestamp ($\hat{\tau}$) are inserted at the head of the *value occurrence list (VOL)* of *TVLEntry* to denote a new occurrence of that value (Step 2). Step 3 increments the counter of the stream that generated the tuple by one. In case the counter increases from zero to one, the function ψ increments the number of streams that contribute to producing the tuple by one. Step 4 appends the tuple's occurrence to the *clean-sweep list (CSL)* that maintains all tuples based on their arrival order for later clean-up purposes. Step 5 checks whether the tuple appears in all streams by investigating the count-summary function (ψ). If the tuple appears in all streams, a complete multi-way join operation is started by traversing the *value occurrence list (VOL($\hat{t}$))* to form the output from the value occurrences in other streams (i.e., $\hat{S} \neq temp.S$). Also, the condition ($\hat{S} \neq temp.S$) ensures that no duplicate tuples are generated because all the output tuples are formed by appending the newly-arrived value $\hat{t}$ to the cartesian product of other streams' qualifying tuples. We traverse *VOL* until we reach its end or until we reach a tuple that is far in the past by more than the window size, i.e., $\hat{\tau} - temp.\tau > \omega$. Notice that the count-summary function avoids generating partial results. If a tuple does not appear in all data streams ($\psi < m$), no further processing is pursued for the join operation. Also, the count summaries give a hint about the output size which is expected to be the product of the counters of all streams other than $\hat{S}$ (i.e., $\prod_{\substack{i=1 \\ i \neq \hat{S}}}^{m} c_i$).

Finally, in Step 6, we traverse the *clean sweep list (CSL)* to delete any tuple with a timestamp that is outside the most recent sliding time-window, i.e., $Current\,time - CSL.\tau > \omega + \epsilon$, where ϵ is an error factor that accommodates late tuple arrivals. ϵ represents the maximum delay in the tuple's arrival time to avoid expiring tuples that may join with late tuples. When a tuple is deleted, its associated counter is decreased by one. Consequently, if a counter reaches zero, the value of the count-summary function (ψ) is decremented by one to indicate that one more stream no longer participates in the join. Although we choose to perform the clean-sweep step (Step 6) with the arrival of every tuple, the clean-sweep step can be performed periodically or in a lazy fashion when there is plenty of system resources.

In addition to handling late tuple arrivals (by introducing ϵ), *SNJoin* is insensitive to out-of-order arrivals provided that we keep the *value occurrence list (VOL)* sorted by timestamps. We insert a delayed tuple in its proper position in *VOL*. Although the join output will be delayed by the maximum amount of delay in the components of the join output tuple, the output remains unchanged. On the other hand, the *clean-sweep list (CSL)* does not have to be kept sorted by timestamps. However, the expiration of a delayed tuple will be delayed because *CSL* is sorted based on the tuple's arrival time at the system. A delayed tuple will not be deleted unless all tuples that came before it are deleted. As a side effect, system resources will be slightly affected because delayed tuples occupy the system's memory for a longer period of time than they should do. For other techniques that handle out-of-order tuples, the reader is referred to [14].

2.3 Disk support for memory overflow

In response to fluctuations in the arrival rates of data streams, a *DSMS* experiences variable system loads. At high-load periods, some input tuples are dropped from the input buffers of the join operator due to memory and CPU constraints. On the other hand, at periods of low system load, the join operator may be blocking waiting for the arrival of new tuples. To make up for memory constraints and to increase the join output rate, *xjoin* [15] flushes partitions of the hash tables to disk at periods of high system load and joins these partitions later, i.e., at periods of low system load. *MJoin* [16] proposes a *coordinated memory flushing* policy that coordinates the tuple flushing across all hash tables. If tuples are flushed from partition P_i of one stream, the same partition (P_i) must be flushed from all other streams before another partition P_j is to be flushed. This approach increases the output rate by allowing tuples with the same values to be either left together in memory or flushed together to disk.

SNJoin utilizes one global hash table where tuples from *all* streams are hashed based on their values. A partition of the *SNJoin* hash table accommodates sets of tuples that have the same values, yet coming from different streaming sources. As a result, flushing a partition from the global hash table is a coordinated flushing by default where tuples of the same values are moved together to disk. *SNJoin* achieves a coordinated flushing without incurring any additional coordination cost. Although sensors are devices that have no disk in general, we investigate the disk support for the completeness of the algorithm and for comparison purposes with *MJoin* if both algorithms are applied in a centralized *DSMS*.

2.4 Support for multiple window sizes

In the *SNJoin* algorithm presented in Figure 2, we assume that the join operation is performed over a sliding window ω such that ω is fixed for all streams. However, many applications require a different window size for each group of streams or a different window size for each individual stream (i.e., ω_i is the corresponding sliding window over stream S_i). To support multiple window sizes, we define two variables ω_{min}=$MIN_i(\omega_i)$ and ω_{max}=$MAX_i(\omega_i)$. First, we set ω to ω_{min} and traverse the *value occurrence list* as in Figure 2. Second, we extend ω to ω_{max} and continue to traverse the *value occurrence list*. However, in the second step, we filter each tuple based on its timestamp to ensure that it falls within its specified time frame (i.e., $\hat{\tau} - temp.\tau \leq \omega_{temp.S}$). Although *SNJoin* handles multiple window sizes, it turns out to do a lot of work filtering tuples out if the gap between ω_{min} and ω_{max} gets large.

3 Variable-arity Multi-way Join

For large-scale sensor networks, a complete multi-way join among hundreds or thousands of sensors may result in no output tuples at all. A sensor network that spans a large area usually experiences variable conditions from one region

to another. Hardly all sensors will produce the same readings within the frame of a time-window. Moreover, it makes more sense to join sensors that show similar behavior even if this behavior does not span the whole sensor-network field. For example, a fire would trigger only a subset of heat sensors in the field that are exposed to an increasing temperature. Similarly, a gas leakage out of a container stimulates only close-by sensors to report the leaking gas. These applications require an immediate action once a join is detected among any number of sensors. Later on, the join output may be updated and amended with new joining sensors as time proceeds. For example, the first few seconds a gas leaks out of a container, there may be only two sensors reading that gas value. After a while, a third sensor will join the two sensors in reporting the same gas value. More and more sensors will be involved in the join as the gas cloud propagates in the field. Similarly, other sensors may stop joining as the gas cloud moves away from their regions.

An appropriate join strategy for large-scale sensor networks would be a variable-arity multi-way join that allows subsets of sensors join. Then, other sensors can be included or excluded from the join with the arrival of new tuples into the system. The *variable-arity* multi-way join definition is given in Definition 2. Notice that the variable-arity join operation produces a variable-size tuple that contains the arriving tuple value, its stream, its timestamp, and a list of all streams that produce the same tuple value along with their associated timestamp in no specific order. The size of the output tuple depends on the number of joining streams. The list of joining sensors is expressed as pairs of (S, τ) because this list is usually sparse. Only few sensors will be included in the joining list compared to the total number of sensors in the network. The total number of output tuples equals $\prod_{\substack{i=1 \\ i \neq \hat{S}}}^{m} max(1, c_i)$, which means that streams that do not generate the tuple $\hat{t}$ (i.e., $c_i = 0$) do not affect the total number of output tuples.

Definition 2. *Given m input streams, $S_1, S_2, \cdots, S_m$, each stream S_i generates tuples of the form $(t_i, [S_i, \tau_i])$ (where t_i is the tuple value that is produced by stream S_i at time τ_i). For a newly arriving tuple $(\hat{t}, [\hat{S}, \hat{\tau}])$, a* variable-arity *multi-way join over window ω produces an output O=$\{(\hat{t}, [\hat{S}, \hat{\tau}], [S_{o_1}, \tau_{o_1}], [S_{o_2}, \tau_{o_2}], \cdots)$, where S_{o_i} is one of the joining streams such that t_{o_i}=$\hat{t}$ and $|\hat{\tau} - \tau_{o_i}| \leq \omega$, $o_i \in 1 \cdots m$, $S_{o_i} \neq \hat{S}$, $S_{o_i} \neq S_{o_j}$ $\forall i \neq j$ $\}$.*

Notice that the variable-arity join is different from the outer join both at the conceptual and the implementation levels. At the conceptual level, variable-arity join omits streams that do not participate in the join to produce a variable-size tuple. Outer join produces a fixed-size tuple with NULL values in lieu of missing streams. At the implementation level, variable-arity join touches only streams that participate in the join. However, outer join probes every stream to check the existence of the join value.

3.1 Variable-arity join algorithm

One naive approach to turn a *complete* join algorithm into a *variable-arity* join algorithm is straightforward. If the joining tuple is missing in one of the stream hash tables, we ignore this stream and continue to join remaining streams. This approach applies to both trees of binary joins and *MJoin*. In a tree of binary joins, we propagate partial results up the tree even if no match is found at some steps. In *MJoin*, the join probing sequence spans all hash tables looking for matching values regardless of their existence in some tables. The major problem with this approach is that we achieve no performance benefits.Although this approach conceptually produces a variable-arity join, it is as costly as an outer join. All streams have to be probed anyway even if only a subset of the streams are to join.

Although *SNJoin* handles complete multi-way join efficiently, *SNJoin* is specially designed to perform a variable-arity multi-way join in large-scale sensor networks. Only one hash table is probed to retrieve all the tuples that join regardless of their generating stream. The variable-arity *SNJoin* algorithm is a down-sized version of the complete *SNJoin* described in Figure 2. In variable-arity *SNJoin*, maintaining count-summaries is useless because the join will be performed regardless of the number of streams that generate the tuple. Consequently, Step 3 and the first *if statement* in Step 5 are removed to yield the *SNJoin* algorithm to variable-arity join query processing.

Variable-arity *SNJoin* has two major advantages. First, it scales to networks with large number of sensors because it avoids unnecessary probes. Tuples that contribute to the output are the only tuples to be considered. Other techniques probe many streams that produce no output. Second, variable-arity *SNJoin* does not suffer from the dynamic configuration of sensor networks because all sensor readings are hashed to the same global table. Binary-join trees require reorganization of the join tree. Also, *MJoin* requires considering the change in the number of hash tables in the join probing sequence of incoming tuples.

4 Query Processing with Relevance Feedback

In this section, we take our first step to shift the join operation from the centralized *DSMS* to the sensor-network level. We introduce the concept of query processing with relevance feedback as a building block of the distributed *SNJoin** algorithm. A major challenge in variable-arity multi-way join queries comes from the fact that only a small number of sensors, compared to the thousands of sensors in the network, join with each other. The problem becomes more challenging in a distributed environment where a probe between two sensors requires a significant communication cost. The objective of query processing with relevance feedback is to guide the join operation to process only relevant sensors, i.e., sensors that generate the same values. With the arrival of a new tuple $\hat{t}$ at sensor S_i, a join probing sequence has to be determined. Each sensor along the probing sequence performs the join operation over its data, then ships the result to the next sensor in the probing sequence until the join result is received at the *DSMS*.

The *DSMS* performs any remaining query processing and forwards the result to the client. Based on the final query output, the *DSMS* decides how much each sensor contributes to the output, i.e., how much each sensor along the probing sequence is relevant to the output. In the query processing with relevance feedback paradigm, the *DSMS* forms a feedback array $[w_1, w_2, \cdots, w_k]$ (where k is the arity of the join output tuples) to represent the contribution weight of each sensor in the output and sends it back to the sensor that initiated the probing sequence. For simplicity, let w_i be the percentage of the output tuples in which sensor S_i appears. Each sensor maintains a *Relevance Feedback Matrix (RFBM)* to record the relevance of each sensor to its own input tuples.The *RFBM* is used to guide future probing sequences. The *RFBM* is defined as follows:

Definition 3. *Given a hash function $H(\hat{t}) \rightarrow [h_1, h_2, \cdots, h_n]$ and given m data streams $S_1, S_2, \cdots, S_m$, a* Relevance Feedback Matrix (RFBM) *is a two dimensional matrix $(n \times m)$ such that $RFBM[H(\hat{t}), S_i]$ represents the relevance of stream S_i to the join probing sequence of tuple $\hat{t}$.*

Using *RFBM*, the join probing sequence for an input tuple $\hat{t}$ is formed such that the probability of including a sensor in the probing sequence is proportional to its relevance to $\hat{t}$. The relevance probing sequence is defined as follows:

Definition 4. *Given m data streams $S_1, S_2, \cdots, S_m$ and given an input tuple $\hat{t}$, the* Relevance Probing Sequence (RPS) *of $\hat{t}$ is a sequence of data streams $S_{o_1}, S_{o_2}, \cdots, S_{o_k}$ such that $k \leq m$ and the probability $Pr\{S_i \in RPS\} = \frac{RFBM[H(\hat{t}), S_i]}{\sum_{i=1}^{m} RFBM[H(\hat{t}), S_i]}$.*

The *RFBM* entries are initially set to a base value (e.g., 50% to denote that each stream has an equal probability of being included/ excluded from the probing sequence). Then, the entries of the *RFBM* change dynamically with the arrival of relevance feedback from the *DSMS* based on the following equation:

$$RFBM[H(\hat{t}), S_i] = RFBM[H(\hat{t}), S_i] - \frac{\sum_{j=1}^{k} w_j}{k} + w_i$$

The above equation indicates that the *RFBM* is affected by the weight of a sensor in the output (w_i) relative to the average weights of all sensors in the output ($\frac{\sum_{j=1}^{k} w_j}{k}$). Notice that as sensors contribute to the output, they *gradually* get a higher probability to be included in the probing sequence of the values they generate. Similarly, if sensors do not participate in the join output they *gradually* lose their *good reputation* and are excluded from the probing sequence.

5 SNJoin*

Up to this point, *SNJoin* is presented to meet the demands of the multi-way join operation over large-scale dynamically configured sensor networks. However, *SNJoin* is a centralized join algorithm that requires all sensors to transmit their data to a centralized *DSMS*. Hence, bottlenecks show up at the *DSMS*, specially, with the increase in the network size. Scalable query processing over sensor networks requires the *en-route* processing of sensor readings while they are transmitted to the *DSMS*. Examples of such in-network query processing

include [6, 13, 18]. In this section, we present the distributed variation of the algorithm, *SNJoin**, that shifts the query processing of the join operation from the centralized *DSMS* to the sensor-network level.

To reduce the communication cost among sensors, several techniques have been proposed to configure the network topology dynamically, e.g., [3, 5, 19]. These techniques involve message exchange among sensors to acquire knowledge about their locations and energy levels. Based on the acquired knowledge, sensors are grouped into clusters. Within the members of each cluster, a specific node, usually with a higher energy level, is designated to serve as the *cluster head*. The cluster head receives the readings of all sensors in its cluster and forwards these readings to the centralized *DSMS*, possibly through a multi-hop route. Cluster heads communicate with each other to achieve a distributed execution of various queries over the sensor network. Notice that cluster heads may be recursively clustered into head clusters to form a hierarchy of clusters such that each sensor node communicates with its head until its reading is received by the centralized *DSMS*. For the sake of simplicity, we assume one level of clusters where cluster heads can communicate with each other.

*SNJoin** divides the multi-way join operation that is performed over the entire sensor network into multiple multi-way join operations that are performed separately over each cluster at the cluster head. Then, each cluster head chooses a *cluster-head probing sequence* to probe other cluster heads looking for matches. Ideally speaking, the cluster-head probing sequence spans all cluster heads in the network to produce as much output results as possible. However, due to the large size of the network and its associated communication cost, it is practical to probe only clusters where it is more likely to find matches. This selective probing reduces both the processing cost and the communication cost at the price of losing some of the streams that could have participated in the join if they were included in the probing sequence. The concept of relevance feedback presented in Section 4 claims the responsibility of choosing an appropriate cluster-head probing sequence. In *SNJoin**, the relevance feedback is applied at the level of cluster heads not at the level of individual nodes.

Figure 3 gives the *SNJoin** algorithm. A cluster head receives an input tuple from one of its cluster members, a probing request from another cluster head, or a relevance feedback from the *DSMS*. The algorithm handles each case separately. Upon receiving a new input tuple, the *SNJoin** algorithm probes the cluster head's local hash table to retrieve a local join result (r) (Step 1). The cluster head decides a probing sequence that spans other cluster heads based on its local relevance feedback matrix (RFBM) (Step 2). The cluster head sets a *Last-Processed-Tuple* mark over tuple $\hat{t}$ to denote the last processed tuple (Step 3). This mark is used for processing the probing requests of other cluster heads as explained later in this section. The cluster head sets the sequence number to zero ($SeqNo = 0$) because the cluster head is the initiator of the join operation (Step 4) and prepares a probing sequence to be sent to the next hop (Cluster head number $SeqNo + 1$). A probing request consists of a sequence number that indicates the last cluster head that processed the request (zero in this case), the

PROCEDURE *Distributed-Insert-Probe*

Upon receiving a new input tuple:
INPUT: *a new input tuple* $(\hat{t},[\hat{S}, \hat{\tau}])$.
OUTPUT: *the join output produced by tuple* $\hat{t}$ *plus a cluster-head probing sequence.*

1. r=*insert-probe*$(\hat{t},[\hat{S}, \hat{\tau}])$
2. *Choose a cluster-head probing sequence* $(C_{s_1}, C_{s_2}, \cdots, C_{s_k})$
3. *Last-Processed-Tuple*$=(\hat{t},[\hat{S}, \hat{\tau}])$
4. $SeqNo = 0$
5. *Ship To* $C_{s_{SeqNo+1}}$: $(SeqNo, [\hat{t}, \hat{\tau}], [(C,|r|), (C_{s_1},0), \cdots, (C_{s_k},0)], r)$

Upon receiving a probe request:
INPUT: *a probe request* PR:$(SeqNo, [\hat{t}, \hat{\tau}], [(C_{s_0},|r_{s_1}|), (C_{s_2},|r_{s_2}|), \cdots, (C_{s_k},|r_{s_k}|)], R)$.
OUTPUT: *the join output produced by* PR *and a an updated* PR.

1. r=*probe*$(\hat{t}, \hat{\tau})$ *in* $[-\infty \cdot\cdot$ *Last-Processed-Tuple]*
2. $SeqNo = SeqNo + 1$
3. *Ship To* $C_{s_{SeqNo+1}}$: $(SeqNo, [\hat{t}, \hat{\tau}], [(C_{s_0},|r_{s_0}|), \cdots, C_{s_{SeqNo}},|r|), C_{s_{SeqNo+1}},0), \cdots, (C_{s_k},0)], R|r)$

Upon receiving a relevance feedback note:
INPUT: *a relevance feedback note:*$(\hat{t}, [(C_{s_1},w_{s_1}), (C_{s_2},w_{s_2}), \cdots, (C_{s_k},w_{s_k})])$.
OUTPUT: *an updated relevance feedback matrix.*

for i=1 to k

$$RFBM[H(\hat{t}), s_i]=RFBM[H(\hat{t}), s_i] - \frac{\sum_{j=1}^{k} w_{s_j}}{k} + w_{s_i}$$

Fig. 3. The SNJoin* algorithm.

joining tuple $\hat{t}$, a sequence of cluster heads, and the join result r. Notice that the output size produced by each cluster head is associated with the cluster head number to be used in the computation of the relevance feedback.

Upon receiving a probing request, the cluster head probes its own hash table, accumulates its result to R, increases the probing sequence number, and forwards the probing request to the next hop. When a cluster head probes its local hash table, it starts from the *Last-Processed-Tuple* backward to avoid generating duplicate tuples. Otherwise, if the entire hash table is probed, the results associated with tuples that came after the *Last-Processed-Tuple* will be duplicated when they probe the hash tables of the cluster heads.

Upon receiving a relevance feedback note, the cluster head updates its own relevance feedback matrix (RFBM) accordingly. A relevance feedback note consists of a tuple $(\hat{t})$ that is being assessed for relevance and a sequence of cluster heads along with their relevance weight.

6 Analysis

In this section, we analyze the output rate of the *SNJoin* operator analytically and compare it to the output rate of the *MJoin*. The output rate is defined to be the number of output tuples divided by the time required to generate these tuples [16]. The number of output tuples depends on the input rates and the selectivity factors among various input streams regardless of the join technique. However, the time required to generate the output tuples is the key factor that differentiates among the performance of various join techniques. From now on, we focus on the average time required by both *MJoin* and *SNJoin* to generate

	MJoin	SNJoin	SNJoin*
Hash	$O(1)$	$O(1)$	$O(1)$
Insert	$O(1)$	$O(1)$	$O(1)$
Choose	$O((k-1)log(k-1))$	–	$O((D-1)log(D-1))$
Probe	$1+(1-(1-\sigma_1)^{n_1})+\cdots+\prod_{i=1}^{k-1}(1-(1-\sigma_i)^{n_i})$	$O(1)$	$1+(1-(1-S_1)^{N_1})+\cdots+\prod_{i=1}^{D-1}(1-(1-S_i)^{N_i})$
Form	$\sigma_1 n_1+\sigma_1\sigma_2 n_1 n_2+\cdots+\prod_{i=1}^{k-1}\sigma_i n_i$	$\prod_{i=1}^{k-1}\sigma_i n_i$	$S_1N_1+S_1S_2N_1N_2+\cdots+\prod_{i=1}^{D-1}S_iN_i+(\prod_{i=1}^{\frac{k}{D}-1}\sigma_i n_i)$

Fig. 4. Cost estimates of both *MJoin* and *SNJoin*.

the output tuples. For details on how to estimate the number of output tuples, the reader is referred to [16].

First, we consider the complete multi-way join. The time required to process a tuple, say t, from an input stream is the summation of the times taken to *hash* and *insert* t into its corresponding hash table, *choose* a join probing sequence, *probe* other streams' hash tables, and *form* the output join tuples. Figure 4 provides estimates of these time components for both *MJoin* and *SNJoin*, given k input streams. The hashing and insertion steps for both *MJoin* and *SNJoin* are achieved in a constant time, i.e., $O(1)$. However, *MJoin* maintains a hash table per input stream and the join probing sequence is computed by sorting the selectivity factors of the other $k-1$ hash tables in $O((k-1)log(k-1))$. The objective of choosing a probing sequence is to retrieve, for each input tuple, a join probing sequence $h_1, h_2, \cdots, h_{k-1}$ such that $\sigma_1 \leq \sigma_2 \leq \cdots \leq \sigma_{k-1}$, where σ_i is the selectivity factor of the input tuple in hash table h_i. *MJoin* probes hash table h_i and if a match is NOT found (with probability $(1-\sigma_i)^{n_i}$, where n_i is the number of tuples in hash table h_i), the join probing sequence is terminated. Otherwise, *MJoin* proceeds to probe the next hash table h_{i+1} (with probability $1-(1-\sigma_i)^{n_i}$). The probability that a probe will reach hash table h_j equals $\prod_{i=1}^{j-1} 1-(1-\sigma_i)^{n_i}$. The best case is to probe the first table only and the worst case is to probe all hash tables. The *expected* number of probed tables equals $1+(1-(1-\sigma_1)^{n_1})+\cdots+\prod_{i=1}^{k-1}(1-(1-\sigma_i)^{n_i})$. In contrast, *SNJoin* has only one global hash table and has no associated cost for deciding a probing sequence.

With each probe, *MJoin* produces partial results. By probing hash table h_j, the size of the partial result is the size of the cartesian product of all hash tables up to h_j multiplied by their selectivity factors, i.e., $\prod_{i=1}^{j}\sigma_i n_i$. These partial results are lost if the probing sequence terminates at this level. The cost of forming the output tuples equals the summation of all these partial results up to level $k-1$ (the last table to be probed), i.e. $\sigma_1 n_1+\sigma_1\sigma_2 n_1 n_2+\cdots+\prod_{i=1}^{k-1}\sigma_i n_i$. As for *SNJoin*, no partial results are produced. Instead, the count-summary function ψ is checked and the result is produced if and only if there is a match in every hash tables. The cost of the final result is the cartesian product of all input streams multiplied by their selectivity factors, i.e., $\prod_{i=1}^{k-1}\sigma_i n_i$. Figure 4 shows a noticeable reduction in the complexity of complete *SNJoin* over *MJoin*. A verification of the complexity analysis is provided in the experiments (Section 7).

For the variable-arity multi-way join operation, the *hash* and *insert* time components remain unchanged for both *MJoin* and *SNJoin*. However, *MJoin*

keeps probing all hash tables looking for matches even if the tuple value is missing in one of the hash tables. As a result, the *choose* cost vanishes and the *probing* cost increases to $O(k)$ where all hash tables are probed. *SNJoin* probes one table anyway and, hence, the probing cost remains $O(1)$ in all cases. The tuple formation cost remains unchanged for both *MJoin* and *SNJoin* with the exception that σ_i is never zero. At least one tuple from each input stream participates in the join (i.e., the NULL tuple). In this case, $\sigma_i = \frac{1}{n_i}$ contributing to the join output with a total of $\sigma_i n_i = 1$ tuple (the NULL tuple).

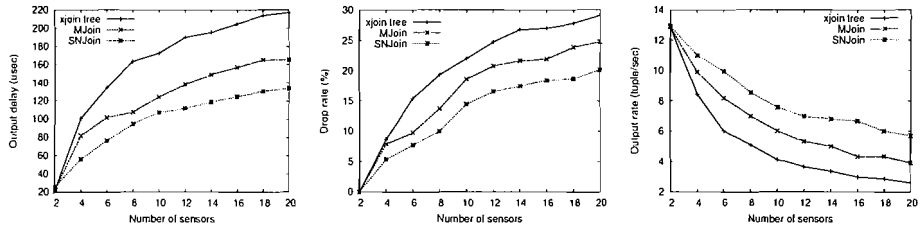
*SNJoin** performs multi-way join over D clusters of input streams. On the average, each cluster contains $\frac{k}{D}$ streams. Figure 4 summarizes the cost estimates of *SNJoin** where S_i and N_i are the average selectivity factor and the total number of tuples in cluster i, respectively. A stream tuple is hashed and is inserted into the hash table of its corresponding cluster in $O(1)$. *SNJoin** endures two costs: the cost of probing the hash table of tuple's cluster and the cost of probing other clusters' hash tables. The cost of probing its own cluster's hash table is the same as the cost of *SNJoin* but with a total number of streams that is equal to $\frac{k}{D}$ instead of k. The cost of probing other clusters' hash tables is the same as the cost of the *MJoin* but with a total number of hash tables that is equal to D instead of k.

7 Experiments

In this section, we conduct an experimental study to explore the performance of the proposed *SNJoin* and *SNJoin** operators. Three sets of experiments are performed. The first set of experiments (Section 7.1) investigates the performance of the complete multi-way join, as presented in Section 2. The second set of experiments (Section 7.2) addresses the variable-arity multi-way join support, as presented in Section 3, and examines the dynamic reconfiguration of sensor networks. The third set of experiments (Section 7.3) highlights the advantages of query processing with relevance feedback and investigates the performance of *SNJoin**, as presented in Section 5. In Sections 7.1 and 7.2 , we compare the performance of the following three techniques:

1. *XJoin tree*, where the multi-way join is achieved through a binary tree of *xjoin* operators with disk support for memory overflow.
2. *MJoin*, where the multi-way join is performed using the single-step symmetric *MJoin* operator with a *coordinated memory flushing* policy as described in [16].
3. *SNJoin*, where the multi-way join is performed as described in this paper.

In Section 7.3, we compare the performance of *SNJoin** with a distributed variation of *MJoin*. The output rate, measured in terms of the number of output tuples per second, is the major measure of performance. Other measures of performance include the *output delay* and the *input drop rate*. The output delay is the time difference between the arrival of a tuple and the time its effect appears in the output. Due to the system's limited CPU time and the continuous arrival of stream data, some input tuples are dropped randomly from the system's



(a) Output delay (b) Input drop rate (c) output rate

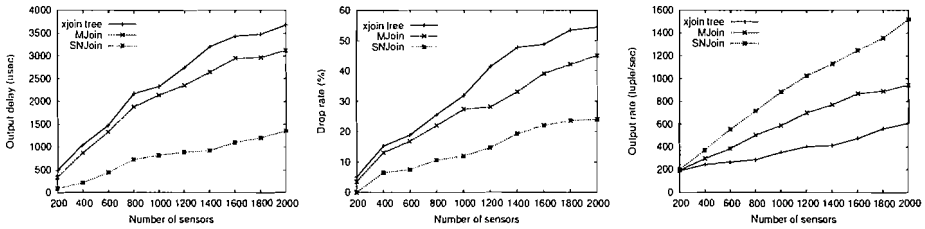
Fig. 5. Performance of *complete* multi-way join.

buffers to accommodate new tuples (i.e., random load shedding). In all experiments (except the experiment that deals with disk support for memory overflow in Section 7.1), we assume that tuple dropping occur due to limited CPU time not due to limited memory. We allocate enough memory to accommodate all tuples in the sliding window. We measure the number of dropped input tuples relative to the total number of input tuples as the input drop rate.

Unless mentioned otherwise, experiments are performed over variable-size sets of simulated sensors. Each sensor generates a stream of 10,000 tuples where the tuple values follow the Zipfian distribution. For each stream, the Zipfian parameter is an integer value chosen randomly between 1 and 5. The interarrival time between two consecutive tuples coming from the same source follows the exponential distribution. The average interarrival time is an experiment-controlled parameter. We have two experimental setups that produce similar system load. The first setup is directed to complete multi-way join (Section 7.1) and features a small number of high rate sensors (up to 20 sensors with an average interarrival time of 10 milli-seconds). Notice that complete multi-way join tends to produce fewer or no results as we increase the number of sensors. The second setup is directed to variable-arity multi-way join (Sections 7.2 and 7.3) and utilizes a large number of low rate sensors (up to 2000 sensors with an average interarrival time of 1 second). This setup simulates large-scale sensor networks. The join techniques are triggered through a multi-way join query with a sliding window of size 1 minute. All the experiments in this section are based on a real implementation of the join operators inside the *Nile* data stream management system [10]. The *Nile* engine executes on a machine with Intel Pentium IV, CPU 2.4GHZ and 512MB RAM running Windows XP.

7.1 Performance of complete SNJoin

The performance of the complete multi-way join operation under an *xjoin tree*, *MJoin*, and complete *SNJoin* is given in Figure 5. As illustrated in Figure 5a, complete *SNJoin* reduces the processing time per input tuple and reduces the output delay by up to 38% over *MJoin* and by up to 18% over the *xjoin tree* (in case of 20 streaming sources). As a result of reducing the per-tuple processing time, complete *SNJoin* reduces the input drop rate and, consequently, processes



(a) Output delay (b) Input drop rate (c) output rate

Fig. 6. Performance of *variable-arity* multi-way join.

more tuples (Figure 5b). The output rate of *SNJoins* is double the output rate of *xjoin trees* and exceeds the output rate of *MJoin* by up to 45% (Figure 5c).

To test disk support for memory overflow, we force some of the tuples to be either dropped or spooled to disk by reducing the main memory size that is available for the join techniques. We limit the main memory to accommodate around 50% of the total number of tuples that are coming from all streams during the one-minute sliding window. We conduct an experiment to test the performance of complete *SNJoin* with and without the disk support. The performance gains of the disk support are significant for small numbers of streams (up to 58% increase in output rate in case of 2 streams). However, with the increase in the number of streams, the disk support tends to be less beneficial. For large number of streams, there is hardly enough time to revisit the spooled tuples. In this case, spooling incurs a cost that is not justified if spooled tuples are not processed.

7.2 Performance of variable-arity SNJoin

Performance gains of *SNJoin* become more significant under variable-arity multi-way join. In contrast to binary join trees and *MJoin*, variable-arity *SNJoin* avoids unnecessary probes, and therefore, reduces its per-tuple processing time. Figure 6 illustrates the efficiency of variable-arity *SNJoin* in terms of the output delay and the input drop rate. From the figure, variable-arity *SNJoin* increases the output rate by up to 150% over binary join trees and by up to 60% over *MJoin*.

Figure 7 illustrates the output rate of the join techniques under a dynamically-configured set of sensors. This experiment studies the behavior of the centralized variable-arity *SNJoin* with respect to the dynamic configuration of the network in terms of additions and deletions to the sensor set. Every minute, a number of sensors (randomly chosen between 1 and 100) is either added or removed from the sensor set. Comparing Figure 6c and Figure 7, notice that the dynamic behavior of the network reduces the output rate of *xjoin tree* by up to 50% and reduces the output rate of *MJoin* by up to 25%. However, the output rate of variable-arity *SNJoin* is reduced by only 4% (in case of 2000 sensors).

7.3 Performance of SNJoin*

In this Section, we study the distributed execution of *SNJoin** over clusters of sensors. We conduct this experiment over various sensor-network sizes where

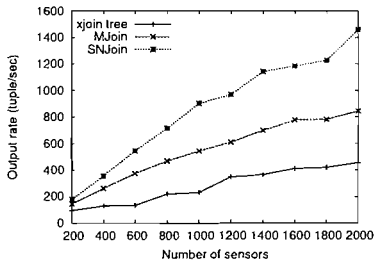


Fig. 7. Dynamic-configuration effect on SNJoin.

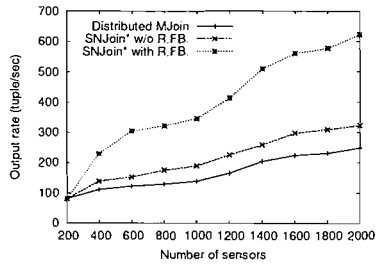


Fig. 8. Performance of SNJoin*.

	Percentage reduction in				
No of sensors	no of probes	output delay	drop rate	tuple width	comm. cost
200	0	0	0	0	0
400	29.1	23.6	3.5	3.4	25.3
600	41.2	30.4	5.1	6.8	38.6
800	50.3	37.7	6.2	7.2	46.3
1000	60.8	47.5	7.4	7.9	57.3
1200	65.2	54.1	14.0	8.1	62.3
1400	69.6	58.8	33.6	8.6	64.9
1600	74.4	65.4	43.7	9.3	72.2
1800	77.4	67.6	51.0	9.9	73.8
2000	79.4	70.1	52.3	11.5	75.5

Fig. 9. The effect of relevance feedback on SNJoin*.

sensors are *uniformly* distributed in the space. Clusters of sensors are obtained using a simulation of the *HEED* clustering technique [19] with the cluster range being set to 10% of the total sensor space (the number of clusters is decided by the algorithm based on the cluster range). For simplicity, we construct a one-level clustering hierarchy where cluster heads can communicate through a one-hop communication link. The join operation is performed at cluster heads. Cluster heads receive the sensor readings of their cluster members, perform the join operation, and communicate with other cluster heads to perform remote probes. Figure 8 compares the output rate of a distributed variation of the *MJoin* to the performance of two variations of the *SNJoin**: one with relevance feedback and the other without relevance feedback. The distributed variation of *MJoin* is obtained by performing the *MJoin* operation among members of the same cluster at the cluster head. Then, each cluster head probes other clusters in a descending order of the average selectivity of their members. From Figure 8, notice that query processing with relevance feedback increases the output rate of *SNJoin* by up to 90% for a sensor network of 2000 sensors.

Accepting relevance feedback allows the join operation to focus on sensors that show similar behavior, and hence, reduces the number of probed streams. Consequently, the per-tuple processing time and the input drop rate are reduced. As a negative effect of relevance feedback, not all cluster heads are probed and, consequently, the output join tuple may miss some streams that could otherwise participate in the join. This results in a decrease in the width of the output tuple. Experimentally, this reduction in the width of the tuple did not exceed 12% (at 2000 sensors). Figure 9 illustrates the effect of the relevance feedback on the performance of *SNJoin** with respect to the reduction in the number of

	SHJ Tree	XJoin Tree	MJoin	SNJoin
Scalability	×	×	√	√√
Dynamic configuration	×	×	√	√√
Symmetric Join	×	×	√	√
Reduction in output delay	×	×	√	√
Sensitivity to variable i/p rates	√	√	×	×
Query plan reorganization	√	√	×	×
Memory overflow support	×	√	√√	√√
variable-arity join support	×	×	×	√

Fig. 10. Comparison among various multi-way join techniques (×: feature not supported, √: feature supported, √√: feature supported and enhanced).

probed streams, the output delay, the input drop rate, the tuple width, and the communication cost. We measure the communication cost in terms of the total number of bytes transmitted per second. There is no performance gain in terms of communication cost between *SNJoin* and *MJoin* because all cluster heads are probed anyway. *SNJoin** increases the communication cost by one extra message per an input tuple. This message is sent from the DSMS to the stream that generated the tuple to carry the relevance feedback of other streams to that tuple. On the other hand, *SNJoin** reduces the communication cost as a consequence of reducing the number of cluster-head probes. The net reduction in communication cost is illustrated in Figure 9 where we can notice the correlation between the reduction in the number of probes and the reduction in the communication cost.

8 Related Work

The multi-way join operation can be achieved through a tree of binary joins (either *symmetric hash join* [17], *xjoin* [15], or *hash merge join* [12]), a single *MJoin* operator [16], or a single *SNJoin* operator. Figure 10 provides a comparison among various multi-way join techniques based on a key set of distinguishing features. Trees of binary joins are not scalable due to their multi-step non-symmetric processing. For the same reason, trees of binary joins do not allow the dynamic configuration of sensor networks (unless query plan reorganization is performed). On the other hand, *MJoin* and *SNJoin* are symmetric, scalable, and dynamically configurable. Also, the output delay in binary join trees increases with the increase in the number of tree levels. The single-step processing of *MJoin* and *SNJoin* results in a lower output delay. Moreover, *SNJoin* is specially designed for large-scale dynamically-configured sensor networks. Trees of binary joins are sensitive to the variable input rates and require reorganization of the query plan operators (e.g., see [4]) to increase their output rate. *XJoin* provides disk support to handle memory overflow. Similarly, *MJoin* and *SNJoin* support memory overflows and enhances the disk support further with a *coordinated flushing* policy. Although all techniques can be tweaked to handle variable-arity join processing, they do not make use of partial processing to reduce the processing cost significantly. However, *SNJoin* supports variable-arity multi-way join by design.

9 Conclusions and Directions for Future Extensions

In this paper, we presented the *SNJoin* (or Sensor-Network Join) operator, a multi-way join operator for sensor-network databases. To meet the demands of sensor networks, *SNJoin* is designed to scale with respect to the number of sensors in the network without sacrificing the output rate. We introduced the notion of query processing with *relevance feedback* to adjust the join selectivity between sensor pairs. *SNJoin** supports the distributed execution of the multi-way join operation with the capability to accept and process relevance feedback.

Experimental studies that are based on a real implementation inside a prototype data stream management system show the scalability of the *SNJoin* operator. For a sensor network of 2000 sensors, the proposed *SNJoin* operator increases the output rate by up to 150% over binary join trees and by up to 60% over *MJoin*. Also, the output rate of *SNJoin** increases by up to 90% with the deployment of relevance feedback.

References

1. M. H. Ali, W. G. Aref, R. Bose, A. K. Elmagarmid, A. Helal, I. Kamel, and M. F. Mokbel. Nile-pdt: A phenomena detection and tracking framework for data stream management systems. In *VLDB*, Sept. 2005.
2. M. H. Ali, M. F. Mokbel, W. G. Aref, and I. Kamel. Detection and tracking of discrete phenomena in sensor-network databases. In *the International Conference on Scientific and Statistical Database Management (SSDBM)*, 2005.
3. A. Amis, R. Prakash, T. Vuong, and D. Huynh. Max-min d-cluster formation in wireless ad hoc networks. In *INFOCOM*, March 2000.
4. R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *ACM SIGMOD*, pages 261–272, May 2000.
5. S. Basagni. Distributed clustering for ad hoc networks. In *the Intl. Symposium on Parallel Architectures, Algorithms and Networks (ISPAN)*, 1999.
6. J. Considine, F. Li, G. Kollios, and J. W. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, pages 449–460, April 2004.
7. L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, Sept. 2003.
8. M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *SSDBM*, July 2003.
9. M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, Sept. 2003.
10. M. A. Hammad, M. F. Mokbel, M. H. Ali, and et al. Nile: A query processing engine for data streams. In *ICDE*, page 851, April 2004.
11. J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, March 2003.
12. M. F. Mokbel, M. Lu, and W. G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, 2004.
13. U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *PODS*, June 2005.
14. U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.
15. T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.
16. S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, Sept. 2003.
17. A. N. Wilschut and E. M. G. Apers. Pipelining in query execution. In *the Intl. Conf. on Databases, Parallel Architectures and their Applications*, 1991.
18. Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR*, 2003.
19. O. Younis and S. Fahmy. Heed: A hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks. *IEEE Trans. Mobile Computing*, 3(4):366–379, 2004.
20. R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin. Habitat monitoring with sensor networks. *Comm. of ACM*, 47(6):34–40, 2004.
21. S. Srinivasan, H. Latchman, J. Shea, T. Wong, and J. McNair. Airborne traffic surveillance systems: video surveillance of highway traffic. In *the 2nd ACM Intl. workshop on Video surveillance & sensor networks*, pages 131–135, 2004.