SNJoin – A Scalable Join in Sensor-Network PhenomenaBases

Mohamed. H. Ali[1] Mourad Ouzzani[2] Walid G. Aref[1] Ibrahim Kamel[3]

[1]Dept. of Computer Science, Purdue University, West Lafayette, IN

[2]Cyber Center, Purdue University, West Lafayette, IN

[3]Dept. of Electrical and Computer Engineering, University of Sharjah, Sharjah, U.A.E.

{mhali, mourad, aref}@cs.purdue.edu kamel@sharjah.ac.ae

Abstract

A phenomenon appears in a sensor network when a group of sensors continuously generate a similar behavior over a period of time. PhenomenaBases (or databases of phenomena) are equipped with Phenomena Detection and Tracking *(*PDT*) techniques that continuously run in the background of a sensor database system to detect and track phenomena. The process of phenomena detection and tracking depends mainly on a multi-way join operator which is at the core of* PDT *techniques to report similar sensor readings. With the increase in the sensor network size and to address periods of heavy loads, the join operator and, consequently, query processing in the PhenomenaBase face several challenges. In this paper, we present a new join operator for PhenomenaBases called* SNJoin *that is specially-designed for dynamically-configured large-scale sensor networks with distributed processing capabilities. In particular, SNJoin introduces a new concept of join called variable-arity join that is best suited for PhenomenaBases. Variable-arity join reduces the number of probes that would have been necessary in a multi-way join. SNJoin allows the join to be performed at the sensor level and integrates query processing with relevance feedback to prune further the sensors to be probed to those only that are relevant to*

the join. Experimental studies illustrate the scalability and the performance gains of the proposed join operator in PhenomenaBases with respect to the number of detected phenomena and the output delay.

1 Introduction

With the evolution of large-scale sensor-network technologies, emerging sensor-network applications call for new online query processing techniques. Such techniques go beyond the traditional sampling, transmission, and processing of sensor data to the more complex paradigm of analyzing, understanding, and acting upon various forms of phenomena that develop in a sensor field. A phenomenon can be a pollution cloud in the air, an oil spill at the ocean surface, or a fire in a building. In general, a phenomenon [3] is a region of sensors generating similar behavior over a period of time. Various techniques have been developed to detect phenomena [3], estimate their boundaries [15], and utilize them in sophisticated data analysis [12].

A sensor-network *PhenomenaBase* [1] is a sensor database system that handles phenomena that develop in a sensor field. More specifically, a PhenomenaBase has two basic functionalities: First, it continuously executes *Phenomena Detection and Tracking* (*PDT*) techniques [3] at the background of a sensor database system to detect new phenomena and track the propagation of already-detected phenomena. Second, it uses the knowledge about detected phenomena to optimize subsequent user queries.

A key component in *PDT* techniques is an *outer multi-way* join operator that detects *similarities* among streams of sensor data over a sliding window of size ω. This join operation is "multi-way" because it detects similarities among multiple sensors and it is an "outer" join because phenomena are usually *localized*. Out of the large number of sensors in the space, only subsets of sensors generate the same values. Other sensors do not participate in the join output and are replaced by NULLs.

Usually, a multi-way join over data streams can be performed using trees of non-blocking binary joins (e.g., *symmetric hash join* [20], *XJoin* [18], or *hash merge join* [14]). Binary join trees perform the multi-way join in multiple steps (i.e., tree levels) and may incur several delays. Also, the output rate

of binary-join trees is sensitive to the join order. For this reason, binary-join trees are usually equipped with a dynamic scheme for tree reorganization (e.g., [5]). To overcome the shortcomings of binary-join trees, [19] introduces the *MJoin* operator, a *single-step* multi-way join operator that is symmetric with respect to all input streams. *MJoin* produces early output, maximizes the output rate, and avoids reorganization of the query plan at execution time. Based on these features, we used an outer *MJoin* operator in the previous design of *PDT* techniques [3].

MJoin has satisfactory performance for moderate system loads. However, with the increase in the network size, the sensor sampling rates, and the number of propagating phenomena, *PDT* techniques start losing many phenomena updates. A phenomena update is reported if a phenomenon appears, disappears, or changes its location. The number of detected phenomena updates per second reflects how fast the system is in tracking phenomena as they move in space. To tackle periods of heavy system loads, we identify three basic challenges that the current design of PhenomenaBases with *MJoin* faces:

1. *Scalability* – sensor networks are typically deployed in large scale with thousands of sensors.

2. *Dynamic-configuration* – Sensors can be dynamically added and removed from the sensor field based on the network conditions, the sensors' life time, and the availability of additional sensors.

3. *Distributed-execution* – the join operation should be performed in a distributed fashion to eliminate the bottlenecks of a centralized system.

In this paper, we introduce a novel join operator for PhenomenaBases, called *SNJoin* (or *Sensor-Network Join*) operator. In a nutshell, *SNJoin* handles the distributed execution of *continuous multi-way window join* queries over *dynamically-configured large-scale* sensor networks. In contrast to *MJoin*, *SNJoin* is not an outer multi-way join. *SNJoin* introduces a new concept of join called *variable-arity* join. Variable-arity join produces variable-size join output in response to the variable number of sensors contributing to a phenomenon. It exploits the *locality* characteristics of phenomena to reduce the number of streams that need to be joined. Moreover, the performance of *SNJoin* improves over time

through a *relevance feedback* (*RFB*) mechanism. *RFB* monitors the contribution of each sensor to the output. Then, *RFB* issues a *feedback note* to the join operator to indicate the relevance of each sensor to the output. This feedback note tunes query processing towards sensors that maximize the join output rate. With these two new notions of variable-arity join and *RFB*, *SNJoin* scales well with respect to the number of sensors and easily adapts to the dynamic configuration of the network. These two features make *SNJoin* a perfect match for the join operation in environments where a small number of sensors (relative to the huge number of sensors in the sensor field) are likely to join. Although not limited to PhenomenaBases, *SNJoin* suits the process of phenomenon detection since phenomena are usually localized in small portions of the sensor field. Fire, smoke, and oil spills usually span small portions of the sensor field. Other environments where such locality is expected call for the deployment of *SNJoin* over its streaming sources.

The contributions of this paper can be summarized as follows:

1. We introduce the new concept of variable-arity join and adopt it in the context of *SNJoin*.
2. We enhance *SNJoin* with distributed processing capabilities by performing the join at the sensor level.
3. We extend *SNJoin* with the ability to accept and process relevance feedbacks.
4. We provide an analytical study and an experimental study that is based on a real implementation of *SNJoin* inside *Nile-PDT* to prove its efficiency both in terms of the number of detected phenomena updates and the output delay.

The remainder of this paper is organized as follows: Section 2 gives an overview of PhenomenaBases and *Nile PDT*, a framework for Phenomenon Detection and Tracking inside *Nile*. Section 3 gives an outline of *SNJoin* and investigates the underlying sensor platform. Section 4 presents the variable-arity notion of *SNJoin*. Section 5 empowers *SNJoin* with distributed processing capabilities. Section 6 describes the relevance feedback mechanism of *SNJoin*. Section 7 presents a mathematical analysis of

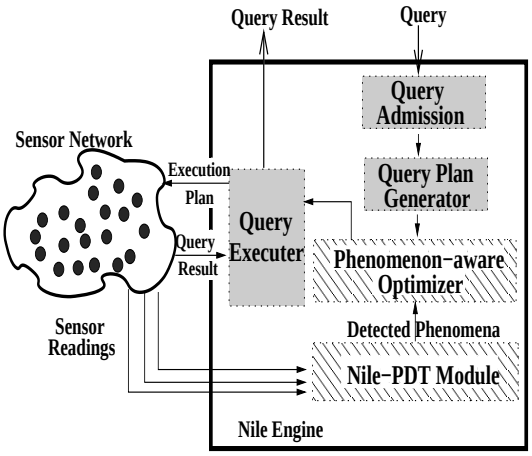


Figure 1. The architecture of Nile.

various join techniques and provides an experimental study of these techniques' performance. Section 8 overviews related join techniques and compares them to *SNJoin*. Finally, Section 9 concludes the paper.

2. PhenomenaBases

PhenomenaBases extend sensor database systems with phenomenon-awareness capabilities as a major step towards the understanding of sensor data. A phenomenon appears in a sensor field if a group of sensors show "similar" behavior over a period of time. In particular, phenomenon-aware sensor databases or PhenomenaBases, have two major tasks: First, detecting and tracking various forms of phenomena. Second, utilizing the knowledge about phenomena to optimize subsequent user queries. Although individual sensor readings can be useful by themselves, *phenomenon detection* exploits various notions of correlation among sensor data and provides actionable and meaningful information about the underlying environment. *Phenomenon tracking* monitors the propagation of detected phenomena to reflect the changes in the surrounding environmental conditions. Given the knowledge about phenomena in the surrounding space, phenomenon-aware optimizers bridge the gap between the low-level sensor readings and the high-level understanding of phenomena to answer user queries efficiently.

There are five major conduits through which sensor-network applications can benefit from *PDT* tech-

niques: (1) **High-level description of the sensor field** – With the aid of *PDT* techniques, an application may ask for "What is going on in a sensor field?" instead of asking "What are the sensor readings?" (2) **Phenomenon-guided data acquisition** – Data acquisition can be guided by detected phenomena, hence reducing the sampling rate of *non-interesting* sensors that do not contribute to any phenomena. (3) **Data compression** – where we maintain the boundaries of exiting phenomena (along with a brief summary of the phenomenon content) instead of maintaining every single reading from every single sensor. (4) **Prediction** – Tracking a phenomenon movement and predicting its future trajectory foresees the next state of the sensor field. (5) **Phenomenon-guided query processing** – Given a query and given a set of phenomena, query processing can be guided to regions with phenomena that satisfy the query predicates. Hence, the query space is reduced.

As an effort in the direction of phenomenon-aware systems, we have augmented an existing data stream management system, namely *Nile* [11], with the concept of PhenomenaBases. Figure 1 illustrates the architecture of *Nile*. *Nile* has three basic components: the query admission controller, the query plan generator, and the query executer. These basic components decide whether to accept or reject a query based on system resources, generate a query plan, and deploy the query plan over the sensor network for execution, respectively. To empower *Nile* with PhenomenaBase capabilities, we add two new components to the system: (1) The *Nile-PDT* (or Nile Phenomena Detection and Tracking) module [2] that continuously detects phenomena at the background of the *Nile* engine. (2) A phenomenon-aware optimizer that interacts with *Nile-PDT* to optimize user queries based on Nile-PDT's knowledge about the sensor field. In this paper, we focus on the *SNJoin* algorithm that comes at the core of the *Nile-PDT* module.

3. Outline of the SNJoin Algorithm

In this section, we describe the underlying sensor platform over which *SNJoin* operates and outline the basic steps of the proposed join algorithm. As illustrated in Figure 2, the sensor platform of *Nile* is

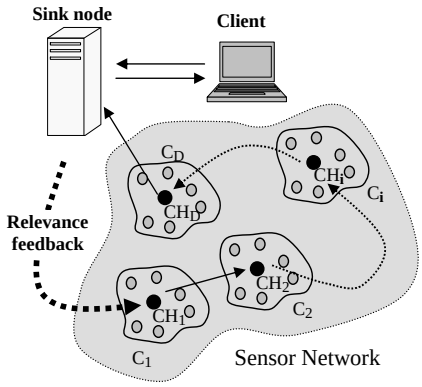


Figure 2. The sensor platform.

an ad-hoc network with resource-constrained sensor nodes. Each sensor generates a stream of readings. Stream tuples are timestamped at the source nodes before they are transmitted over the network to a sink node. However, tuples may arrive late or out-of-order due to network conditions.

Several techniques can be used to dynamically configure the network topology like those proposed in [4, 6, 23]. 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 each cluster, a specific node, usually one with a high-energy level, is designated to serve as the *cluster head* (the $CH_i's$ in the figure). Cluster heads communicate with each other to achieve a distributed execution of various queries over the sensor network. A cluster head receives partial results from sensors in its cluster or from other cluster heads. Then, the cluster head performs additional query processing and forwards the result to another cluster head or to the sink node, possibly through a multi-hop route. The sink node is assumed to be a node with high processing capabilities. The sink node analyzes the query result, assesses its relevance to the query, and returns relevance feedback to cluster heads seeking further optimizations.

We now lay down the basic steps of the algorithm that implements the proposed *SNJoin* algorithm:

Step1. Each sensor forwards its readings to its corresponding cluster head.

Step2. At each cluster head, a variable-arity join is performed among the readings of its cluster members

to generate join tuples of variable sizes (Section 4) where the size of the join output depends on the number of joining sensors. *SNJoin* handles late and out-of-order tuples in this step.

Step3. A distributed processing phase is initiated by cluster heads (Section 5). Each cluster head decides on a *probing sequence* to probe other cluster heads looking for matching tuples among members in their respective clusters. At the end of the probing sequence, the join result is shipped to the sink node.

Step4. The sink node measures the weight or contribution of each cluster in the output and returns a relevance *feedback note* to the cluster head that initiated the probing sequence. Based on the feedback, the cluster head adjusts future probing sequences by assigning high probability of being included to clusters with similar values (Section 6).

4 Variable-arity Join

In this section, we elaborate on the new variable-arity join that would produce variable-size join output in response to the variable number of sensors contributing to a phenomenon. We also compare it to the outer multi-way join that was initially implemented in previous versions of *Nile-PDT* [2]. In sliding-window multi-way join, upon the arrival of a new tuple, say $\hat{t}$, from stream $\hat{S}$, $\hat{t}$ probes other streams looking for matching tuples. $\hat{t}$ joins with tuples that have the same value from other streams provided that matching tuples are within ω time-window from $\hat{t}$. Deriving an outer join variant of an already existing *inner* join technique is straightforward. If the probing tuple is missing in one of the streams, simply append NULL in lieu of the missing stream and proceed to the next stream. This approach applies to binary-join trees and to *MJoin*. In a tree of binary joins, we propagate partial join results up the tree even if no matching values are found at some tree levels. In *MJoin*, the join probing sequence spans all streams. The join probing sequence does not terminate if no matching values are found in any of the streams.

¿From a performance point of view, deploying outer joins over large-scale sensor networks is cost pro-

hibitive. To detect subsets of joining sensors using outer join, every sensor in the network has to be probed. Given the fact that phenomena are usually localized (e.g., an oil spill in a specific area), we may end up probing thousands of sensors to find out that only tens of sensors have a similar behavior. To reduce the number of probes involved in an outer multi-way join, we propose the concept of variable-arity join as given by Definition 1.

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

We should notice that variable-arity join is different from outer join both at the conceptual and implementation levels. At the conceptual level, variable-arity join *omits* streams that do not participate in the join to produce a variable-size tuple. The variable-size tuple contains (1) the join value $\hat{t}$, (2) the source stream and the timestamp of the tuple [$\hat{S}$, $\hat{\tau}$], and (3) a variable-size list of streams that produce matching tuples along with the timestamps of the matching tuples ([S_{o_1}, τ_{o_1}], [S_{o_2}, τ_{o_2}], $\cdots$). In contrast, outer join produces a fixed-size tuple with *NULL* values in lieu of missing streams (even in the presence of many of these NULL values). At the implementation level, variable-arity join probes only streams that participate in the join. Streams with no matching tuples do not infer any additional cost. However, outer join probes every stream to check the existence of the join value (even in the presence of many of such streams.)

4.1 Data Structures

Usually, hash-based join techniques maintain one hash table per stream. A new input tuple is inserted, based on a hash function, into its own stream's hash table and a probe is then launched to look for

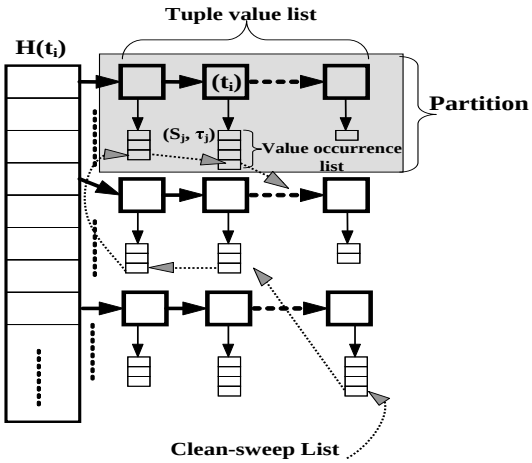


Figure 3. The SNJoin hash table.

matches in other streams' hash tables. With the increase in the number of streams, managing a large number of hash tables becomes costly. To avoid a lengthy join probing sequence, the variable arity join uses a single global hash table where all incoming tuples are hashed and inserted regardless of their streaming sources. Grouping tuples of the same value from various streams in the same partition of a hash table prepares candidates for the join output in advance.

Figure 3 illustrates the proposed SNJoin hash table that is used by the variable arity join. The hash table is divided into partitions based on a suitable hash function H. The hash function is only applied over the value of the join attribute in case the tuple has multiple attributes. 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's value of the join attribute. 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* stores 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 was produced.

VOL is reverse-ordered based on timestamp (i.e., τ). A newly-incoming tuple is appended at the head of *VOL*.

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

4.2 Variable-Arity Join Algorithm

The algorithm for the proposed variable-arity join is given in Figure 4. This algorithm is executed by the cluster head whenever it receives a readings from one of its cluster members. In Step 1, 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 not found, a new entry in *TVL* is created. In Step 2, the stream that generated the tuple ($\hat{S}$) and the tuple's timestamp ($\hat{\tau}$) are inserted at the head of the *value occurrence list (VOL)* that is associated with *TVLEntry* to denote a new occurrence of $\hat{t}$. Step 3 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. In Step 4, we traverse the *value occurrence list (VOL($\hat{t}$))* until we reach its end (temp=NULL) or until we reach a tuple that is far in the past by more than the window size ($\hat{\tau} - temp.\tau > \omega$). As we traverse VOL, we form the join output from the value occurrences in other streams (i.e., $\hat{S} \neq temp.S$). The join output is formed by separating the values in *VOL* based on their source stream into k sublists, i.e., a sublist per stream. Then, we compute the Cartesian product of $k + 1$ sublists: the k sublists plus a sublist of one tuple, the probing tuple $\hat{t}$. The Cartesian product of the sublists is equivalent to the join output since the join condition (i.e., equality on the tuple value) has

PROCEDURE *Insert-Probe*

INPUT: *(1) a new input tuple ($\hat{t}$,[$\hat{S}$, $\hat{\tau}$]) and (2) an SNJoin hash table*

OUTPUT: *(1) an updated SNJoin hash table (2) the join output produced by tuple $\hat{t}$*

1. *TVLEntry=TVL[$(H(\hat{t})$].Search($\hat{t}$)*
2. *VOLEntry=TVLEntry.vol-ptr.Insert($\hat{S}$, $\hat{\tau}$)*
3. *CSL.Append(VOLEntry)*
4. *temp=TVLEntry.vol-ptr.first;*

   while(temp≠NULL and $\hat{\tau} - temp.\tau \leq \omega$)

   begin

   if $\hat{S} \neq temp.S$ Append $temp.\tau$ to $Sublist_{temp.s}$

   temp=temp.next

   end

   Output $\leftarrow \hat{t}$, Cartesian product($[\hat{S}, t\hat{a}u]$, $Sublist_i \forall i = 1..k$), where k is the total number of sublists

5. *Traverse* CSL *to delete expired tuples*

Figure 4. The SNJoin algorithm.

been already fulfilled by *pre-grouping* tuples by value in the same *VOL*. Finally, in Step 5, 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., $Current\,time - CSL.\tau > \omega$. Although we choose to perform the clean-sweep step 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.

4.2.1 Support for Multiple Window Sizes

Up to this point, we assumed that the join operation is performed over a sliding window ω such that ω is fixed for all sensors. However, some applications require a different window size for each sensor (i.e., ω_i is the sliding window over stream S_i). In literature, binary join is performed over two streams such that each stream has its own window size [17]. The generalization of having multiple window sizes in the multi-way join is legitimate as well. Allowing multiple window sizes gives the flexibility to vary the memory overhead over different regions of the sensor field and accommodates variable sensor rates and sensitivity to the occurrence of various events in the environment.

In the variable-arity join, it is straightforward to support multiple window sizes; We just need to change Step 4 of Figure 4 as follows:

temp=TVLEntry.vol-ptr.first;
while($temp \neq$NULL and $\hat{\tau} - temp.\tau \leq \underline{\omega_{MAX}}$) **begin**
    if $\hat{S} \neq temp.S$ and $\underline{\hat{\tau} - temp.\tau \leq \omega_{temp.S}}$
        include $temp.\tau$ in the join output of $\hat{t}$
    temp=temp.next
end

We make two modifications. First, we traverse the value occurrence list (VOL) till we reach the maximum ω (i.e., $\hat{\tau} - temp.\tau \leq \omega_{MAX}$). Second, for each entry in the VOL, the timestamp of an element of stream S_i is tested against its own window size $\omega_{temp.S}$ instead of ω, i.e., $\hat{\tau} - temp.\tau \leq \omega_{temp.S}$.

4.3 Variable-arity Join Versus Outer Join

The concept of variable-arity join has three major advantages over outer join. First, the variable-arity join avoids unnecessary long chains of probing sequences. Other techniques, i.e., binary join trees or *MJoin*, need to probe large numbers of sensors that may produce no output.

Second, the variable-arity join avoids partial-result processing. Binary join tress or *MJoin* consume system resources in processing partial results. Consider a join probing sequence of k tables ($h_1, h_2, \cdots, h_k$). The partial result up to table h_i is the result of ($h_1 \bowtie h_2 \bowtie \cdots \bowtie h_i$). In binary join trees or *MJoin*, partial results have to be maintained (and padded with NULLs if no matching tuples are found) with every probe until the probing sequence is exhausted. The cost of a complete traversal over the partial-result tuples to pad them with NULLs becomes significant with the increase in the partial result size and with the increase in the number of sensors. In the variable-arity join, we retrieve tuples (and only tuples) that contribute to the output with a single traversal of *VOL*.

Third, the variable-arity join accommodates the dynamic reconfiguration of a sensor network at no additional cost. Since all sensor readings are hashed to the same global table, the addition and/or deletion of sensors affect neither the data structure nor the algorithm of the join. In contrast, binary-join trees require a reorganization of the join tree. Also, in response to changes in the number of sensors, *MJoin* creates and/or removes hash tables and adjusts the join probing sequence of incoming tuples accordingly.

5 Distributed Processing in SNJoin

Up to this point, *SNJoin* addressed the demands of large-scale dynamically-configured sensor networks through the notion of variable-arity join. However, if *SNJoin* requires all sensors to transmit their readings to a centralized sink node, the sink node will be a bottleneck, especially with the increase of the network size. Scalable query processing requires the *en-route* processing of sensor readings, i.e., while they

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 ($CH_{o_2}, CH_{o_3}, \cdots, CH_{o_D}$)*
3. $SeqNo = 1$
4. *Ship ($SeqNo$, [$\hat{t}, \hat{\tau}$], [$CH_{o_1}, CH_{o_2}, \cdots, CH_{o_D}$], r) To $CH_{o_{SeqNo+1}}$*

Upon receiving a probe request:

INPUT: *a probe request PR:($SeqNo$, [$\hat{t}, \hat{\tau}$], [$CH_{o_1}, CH_{o_2}, \cdots, CH_{o_D}$], R).*

OUTPUT: *the join output produced by PR and a an updated PR.*

1. *r=probe($\hat{t}, \hat{\tau}$)*
2. $SeqNo = SeqNo + 1$
3. *Ship ($SeqNo$, [$\hat{t}, \hat{\tau}$], [$CH_{o_1}, CH_{o_2}, \cdots, CH_{o_D}$], $R + r$) To $CH_{o_{SeqNo+1}}$*

Figure 5. The distributed SNJoin algorithm.

are being transmitted to the sink node. Examples of such in-network query processing include [7, 16, 22]. We now present a distributed variant of *SNJoin* that shifts the join operation from the sink node to the sensor-network level.

As illustrated in Figure 2, we model the sensor network as an ad-hoc network of sensor nodes grouped into clusters based on their energy level and spatial locations. *SNJoin* decomposes the entire join operation into multiple smaller 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. The probing sequence will then end by shipping the join result to the sink node.

Figure 5 gives the distributed *SNJoin* algorithm. A cluster head receives either an input tuple from one of its cluster members or a probing request from another cluster head. Upon receiving a new input tuple,

SNJoin probes the cluster head's local hash table to retrieve a local join result (r) (Step 1). The cluster head (CH_{o_1}) decides on a probing sequence (either arbitrarily or based on relevance feedbacks as we will show in the next section) that spans *some* or *all* of the other cluster heads, ($CH_{o_2}, CH_{o_3}, \cdots, CH_{o_D}$) such that $1 \leq o_i \leq D$ where D is the total number of clusters (Step 2). The cluster head sets a sequence number to one ($SeqNo = 1$) since the cluster head is the initiator of the join operation (Step 3). Finally, the cluster head ships the probing request to the next hop (i.e., Cluster head number $SeqNo + 1$) (Step 4). A probing request consists of a sequence number that indicates the last cluster head that processed the request, the probing tuple $\hat{t}$, the tuple's timestamp τ, a sequence of cluster heads, and the partial join result r computed from Step 1.

Upon receiving a probing request, the cluster head probes its own hash table (Step 1). Then, the cluster head increases the probing sequence number (Step2). Finally, the cluster head accumulates its local result r to the partial result R computed so far and forwards the probing request to the next hop.

5.1 Early, Late and Out-of-order Arrival

Due to network delays and un-synchronized clocks in the different cluster heads, three issues need to be addressed: late and early arrivals, out-of-order arrivals, and generation of duplicates in the output. Late tuple arrivals may occur when a tuple arrives at a cluster head's buffer past the cluster head's local clock timestamp. Early tuple arrivals may occur when a tuple arrives at a cluster head's buffer before the cluster head's local clock timestamp. In the case of out-of-order arrivals, not only tuples are late but their order has been also altered. Finally, duplicates may occur when the same tuple is reported twice in the output due to two different cluster heads starting two different probing sequences for the same tuple simultaneosuly.

To handle late, early, and out-of-order tuple arrivals, we buffer all tuples and probing requests for some time (i.e., safety factor) before they get processed by *SNJoin*. Let us assume that ϵ is the maximum delay

in tuple arrival from a given sensor and $Delay$ is the maximum delay of a probing sequence to go from one cluster head to another. Whenever a tuple arrives to the processing node, i.e., cluster head or sink node, it is added to a buffer based on its timestamp (i.e., reordered relative to other tuples). A tuple is then sent to its usual processing step by *SNJoin* (i.e., inserted in the hash table) as soon as its timestamp goes beyond ϵ with respect to the current time in the processing node. Similarly, a probing request is buffered for ϵ time units to give a chance for all late tuples to be inserted in the hash table before the actual probing takes place. Thus, the probing tuple or probing sequence is delayed by ϵ until all late tuples are inserted, hence, processed in order with respect to other inoming tuples. In addition, tuples in VOL are expired only if they fall outside a window of size ($w + max(\epsilon, Delay)$). The idea is to avoid expiring tuples that may eventually join with delayed tuples (delayed by ϵ time units) or delayed probing requests (delayed by $Delay$ time units). By increasing the window size, we ensure that the delayed probe will find all the tuples that are supposed to be retrieved and joined.

To avoid duplicates from appearing in the join output, we restrict the processing of a probing request to only tuples that came before τ, where τ is the timestamp of the tuple that initiated the probing request. When a cluster head receives a probing request with a timestamp of τ, the cluster head probes its internal hash table starting from timestamp τ backward, i.e., reterieve all tuples with timestamps (τ') that are less than τ. This precaution places an ordering on the timestamps of the join output components and avoids generating the same output tuple twice, i.e., once in each cluster head. For example if cluster head CH_1 generates value v at timestamp τ_1 while cluser head CH_2 generates the same value later on at timestamp τ_2 ($\tau_1 < \tau_2$). Regardless of the time at which their associated probing requests travelled in the network, the output join tuple is supposed to reported by CH_1. When the probing request comes from CH_2 to CH_1, CH_1 will scan its value occurrence list starting from τ_2 backward and will join the value v at timestamp τ_1. For equal time stamps, ties are broken using the cluster head id. For example, the cluster head with a smaller id is responsible for enerating the join output.

6 Query Processing with Relevance Feedback

A major challenge in multi-way join queries over sensor networks is that usually only a small fraction of the thousands of sensors in the network join with each other. This challenge is exacerbated in a *distributed* environment where a probe between two cluster heads requires a significant communication cost. Ideally, 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 more efficient to probe only clusters where it is more likely to find matches. The possibility of missing few matches from clusters with low contributing probabilities should not have a major impact on the process of detecting and tracking phenomena. The objective of the proposed query processing with *relevance feedback* is to guide the join operation to process only relevant cluster heads, i.e., clusters that are more likely to generate the same values. This selective probing reduces both the processing and communication costs at the price of losing some streams that could have participated in the join if they were included in the probing sequence.

With the arrival of a new tuple $\hat{t}$ at a cluster head, a join probing sequence has to be determined. In this case, the probing sequence will be (CH_{o_1}, CH_{o_2}, $\cdots$, CH_{o_k}) such that $k \leq D$, where D is the number of clusters. Each cluster head along the probing sequence performs the join operation over its data, then ships the result to the next cluster head in the probing sequence until the join result is received at the sink node. Based on the join result, the sink node decides on the contribution of each sensor to the output, i.e., how much each sensor along the probing sequence is effectively relevant to the output. The sink node forms a feedback array [w_1, w_2, $\cdots$, w_k] (where k is the arity of the join result) to represent the *contribution weight* of each sensor in the output and sends the array to the cluster head that initiated the probing sequence (i.e., CH_{o_1}). For simplicity, let us assume that w_i is the percentage of the output tuples in which cluster head CH_i appears. Each cluster head maintains a *Relevance Feedback Matrix (RFBM)* to record the relevance of all other cluster heads to its own input tuples. The *RFBM* is used to guide

future probing sequences. The *RFBM* is defined as follows:

Definition 2 *Given a hash function* $H(\hat{t}) \rightarrow [h_1, h_2, \cdots, h_n]$ *and* D *cluster heads* CH_1, CH_2, $\cdots$, CH_D, *a* Relevance Feedback Matrix (RFBM) *is a two dimensional matrix* ($n \times D$) *such that* $RFBM[H(\hat{t}), CH_i]$ *represents the relevance of cluster head* CH_i *to the join probing sequence of tuple* $\hat{t}$.

Using *RFBM*, the join probing sequence (Step 2 in Figure 5) for an input tuple $\hat{t}$ is formed such that the probability of including a cluster head in the probing sequence is proportional to its relevance to $\hat{t}$. The relevance probing sequence is defined as follows:

Definition 3 *Given* D *cluster heads* CH_1, CH_2, $\cdots$, CH_D *and an input tuple* $\hat{t}$, *the* Relevance Probing Sequence (RPS) *of* $\hat{t}$ *is a sequence of cluster heads* CH_{o_1}, CH_{o_2}, $\cdots$, CH_{o_k} *such that* $k \leq D$ *and the probability* $Pr\{CH_i \in RPS\} = \frac{RFBM[H(\hat{t}),CH_i]}{\sum_{i=1}^{D} RFBM[H(\hat{t}),CH_i]}$.

The *RFBM* entries are initially set to a base value (e.g., 50% to denote that each cluster head 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 notes based on the following equation:

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

The *RFBM* entries are affected by the cluster head weight in the output (w_i) relative to the average weights of all cluster heads in the output ($\frac{\sum_{j=1}^{k} w_j}{k}$). The algorithm of processing relevance feedback notes that are received from the sink node is given in Figure 6. Notice that as cluster heads contribute to the output, they *gradually* get a higher probability to be included in the probing sequence. Similarly, if cluster heads do not participate in the join output they *gradually* lose their *good reputation* and are excluded from the probing sequence.

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

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

Figure 6. Processing of relevance feedback.

7 Evaluation

7.1 Analytical Analysis

The time required to generate the output tuples is the key factor that differentiates among the performance of various join techniques. In this section, we analyze and compare the output delays for both *SNJoin* and outer *MJoin*. The output delay is defined as the time difference between the arrival of a tuple and the time its effect appears in the output. We now estimate the average time required by both outer *MJoin* and *SNJoin* to generate the output in the centralized case.

	Outer MJoin	**SNJoin**
Hash/Insert	C_1	C_1
Probe	$C_2(k-1)$	$-$
Collision	$C_3(k-1)(\frac{distinct_1}{size_H})$	$C_3(\frac{distinct_2}{size_H})$
Separation	$-$	$C_4\Sigma_{i=1}^{k}\sigma_i n_i$
Form	$C_5(\prod_{i=1}^{k-1}\sigma_i n_i)\times k$	$C_5(\prod_{i=1}^{k-1}\sigma_i n_i)\times 2kk'$

Figure 7. Cost estimates of both *MJoin* and *SNJoin*.

The time required to process a tuple, say t, from an input stream is the accumulated times taken to *hash*/*insert* t into its corresponding hash table, *probe* other streams' hash tables (this applies only to *MJoin* since there is only one hash table for *SNJoin*), *resolve collisions* in the hash tables (only one table for *SNJoin*), *separate* the different encountered tuples into their respective streams, this applies only to *SNJoin* since the tuples for all streams are in the same hash table, and finally *form* the output join tuples.

Given k input streams, Figure 7 provides the different formulas to compute the time estimate for each of the above components for both outer *MJoin* and *SNJoin*. The hashing and insertion steps for both joins are achieved in constant time, i.e., C_1. Outer *MJoin* probes all other hash tables than t's table ($k-1$) looking for matches even if the tuple value is missing in one of the hash tables. As a result, the *probe* cost corresponds to the product of a constant C_2 by $(k-1)$. In contrast, since *SNJoin* maintains only one hash table, all potentially joining tuples are accessible directly for the current entry in this hash table and thus the cost of *probe* is null.

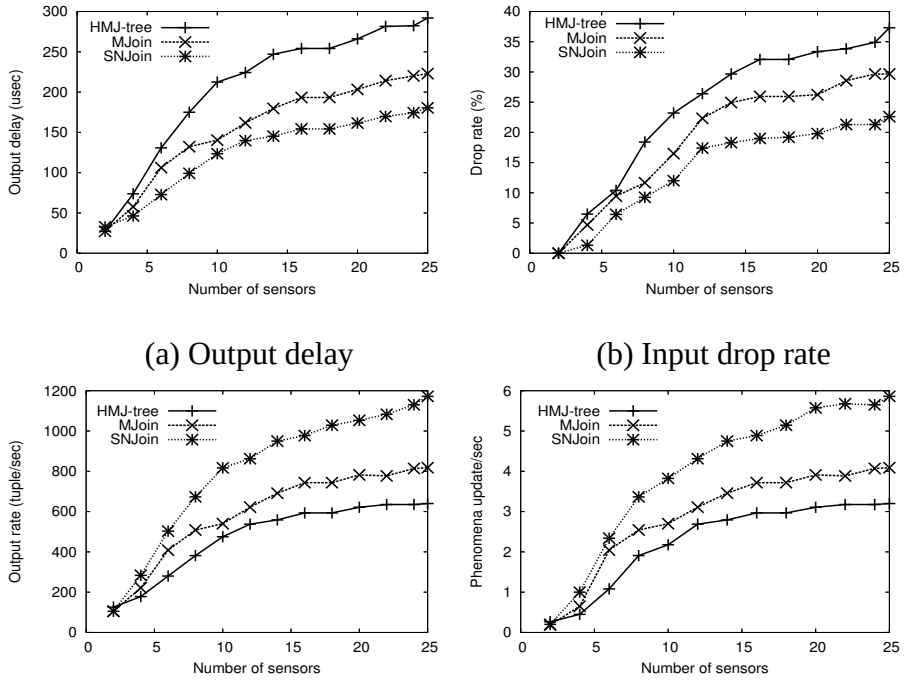
Both joins are subject to collisions in the hash table, the cost of these collisions corresponds to the average number of possible distinct values in the hash table divided by the number of buckets in this hash table ($size_H$). Notice that the number of distinct values in outer MJoin ($distinct_1$) is different from the number of distinct values in SNJoin ($distinct_2$) because the SNJoin hash table receives tuples from all streams while, in MJoin, each hash table maintains the values that are coming from a single stream. For the outer *MJoin*, this cost is repeated ($k-1$) times, i.e., for probing all the hash tables except the hash table of the stream that is producing the value.

Since *SNJoin* groups all tuples in one single hash table it needs to separate the tuples coming from different streams into k lists to be able to join them afterward. This cost is equivalent to a single traversal of the value occurrence list (*VOL*). The size of the *VOL* on average for a specific value equals the summation of the average number of tuples per stream (n_i) multiplied by the average selectivity of this value in that stream (σ_i) for all k streams which results in $C_4 \Sigma_{i=1}^{k} \sigma_i n_i$. The outer *Join* is not subject to this separation cost since tuples from the same stream are in the same table.

The tuple formation cost is computed based on the size of the output which is the product of the number of output tuples, $\prod_{i=1}^{k} \sigma_i n_i$ for both joins, by the tuple size, which corresponds to the number of streams k for the outer *MJoin*, and $2kk'$ for *SNJoin*. The parameter $k' < 1$ is usually very small. It represents the fact that only a small percentage of streams will join (locality of phenomena). This is what will

reduce the size of the output which will be limited to only those streams that contribute to the join. The factor 2 in the formula is needed since the variable-arity join requires both the tuple's timestamp and its corresponding stream id, i.e., $([\hat{S}, \hat{\tau}])$, to be reported in the output join tuple. The experimental study will show how this analysis compare to the actual experiments.

In the distributed case, *SNJoin* performs the join over D clusters of input streams. The output delay is dominated by the communication cost incurred by the probing sequence that needs to travel throughout all cluster heads or a subset of them if we are using the relevance feedback. This communication cost is proportional to the size of the probing sequence. Thus, to evaluate the output delay for the distributed case, we compute the size of the probing sequence. For each cluster head, the information that is generated locally, and added to the probing sequence, is calculated using the same formula as in the tuple formation phase in the centralized case. Thus, if the number of sensors in a cluster j is k_j and the percentage of joining streams is k'_j, then the size of the local output tuples is $2k_jk'_j \prod_{i=1}^{k_j} \sigma_i n_i$, where $2k_jk'_j$ is the average number of columns and $\prod_{i=1}^{k_j} \sigma_i n_i$ is the average number of rows in the partial join output at cluster j. Subsequently for D clusters, the size of the output tuples corresponds to accumulating the output of each cluster all the way along the probing sequence till we reach the last cluster (i.e., cluster number D). Accumulating the output means concatenating the columns of the partial results and computing the cartesian product of the partial result rows. The total output size is estimated to be $(\Sigma_{j=1}^{D} 2k_jk'_j) \times (\prod_{j=1}^{D} \prod_{i=1}^{k_j} \sigma_i n_i)$. This cost is calculated by adding the number of columns and multiplying the number of rows in each cluster head probe along the D cluster sequence. Again, the reduction in size is mainly due to the parameter k'_j that reflects the locality characteristic of phenomena where only very few streams contribute the the join. This is in contrast to the outer *MJoin* where we need to carry tuples about *all* streams throughout all cluster heads even if those streams do not contribute to the join. In addition, *SNJoin* can achieve better performance through relevance feedbacks which will reduces the number of clusters that need to be visited, hence reducing the parameter *D* in the formula that computes the size of the probing sequence.



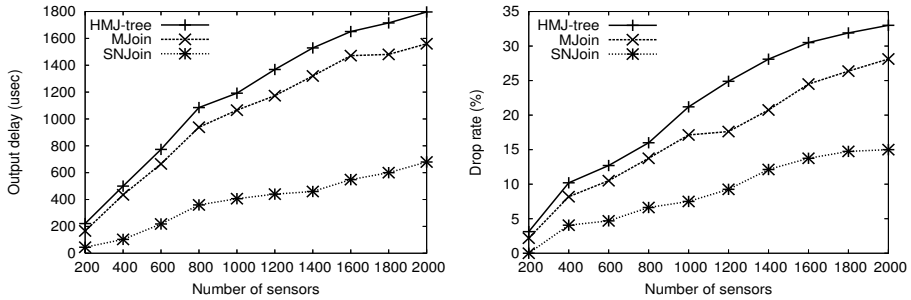
(a) Output delay (b) Input drop rate

(c) Output rate (d) Phenomena updates

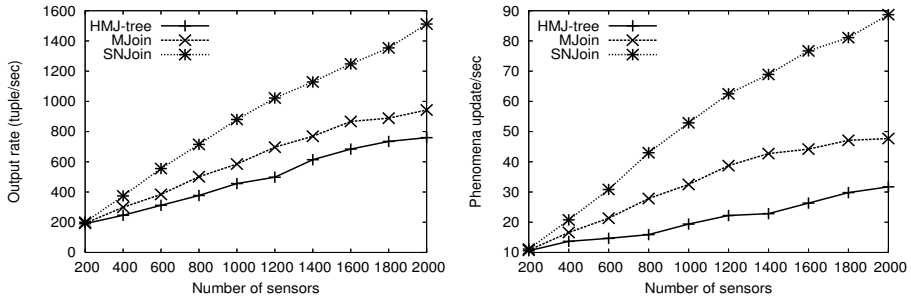
Figure 8. Performance under *real small-scale* **data sets.**

7.2 Experimental Analysis

We now present the experimental study we conducted to explore the performance of the proposed *SNJoin* operator. We base our study on two experimental setups from the *Nile-PDT* system [2]; a real small-scale sensor board and a simulated large-scale sensor network. The first setup is a *real* small-scale sensor board with a grid of 5×5 temperature sensors. Due to hardware limitations, the number of sensor is limited to 25. However, we overload the system by increasing the sampling rate of each sensor to one reading every 10 milliseconds-seconds. We run each experiment for 10 minutes and we move a heat effect back and forth over the sensor board to generate phenomena. The second setup simulates a large-scale sensor network (up to 2000 sensors). Each sensor generates a stream of 10,000 tuples where the tuple values follow the Zipfian distribution [24]. For each stream, the Zipfian parameter is an integer value chosen randomly between 1 and 5. The inter-arrival time between two consecutive tuples



(a) Output delay (b) Input drop rate



(c) Output rate (d) Phenomena updates

Figure 9. Performance under *synthetic large-scale* **data sets.**

coming from the same source follows the exponential distribution with an average of 1 second. In both setups, the join techniques are triggered through a multi-way join query with a sliding window of size 10 seconds.

Three sets of experiments are performed. The first set of experiments (Section 7.2.1) investigates the performance under the real sensor-platform setup. The second set of experiments (Section 7.2.2) addresses the large-scale simulated sensor-network setup and examines the dynamic reconfiguration of the network. In Sections 7.2.1 and 7.2.2, we compare the performance of a *centralized* implementation of the following three techniques:

1. *HMJ-tree*, where an outer join is performed using a binary tree of binary *hash merge join* operators.

2. *MJoin*, where an outer join is performed using the single-step symmetric *MJoin* operator.

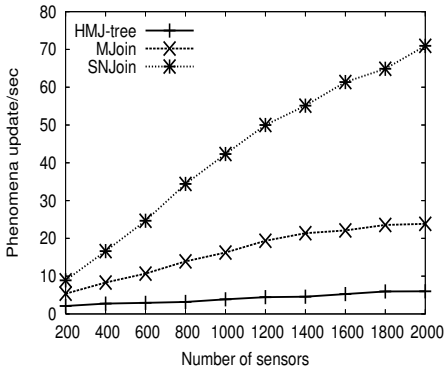


Figure 10. The effect of dynamic network configuration.

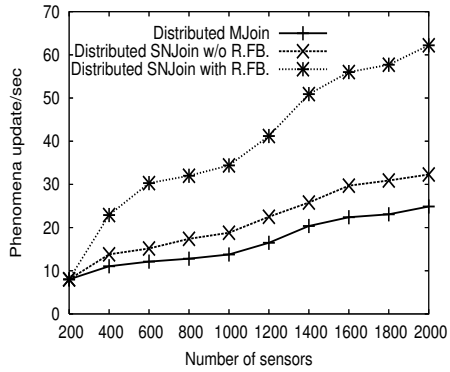


Figure 11. The effect of distributed query processing.

3. *SNJoin*, where a variable-arity join is performed as described in this paper.

The third set of experiments (Section 7.2.3) highlights (under the simulated setup) the advantages of query processing with relevance feedback and compares the performance of distributed *SNJoin* with a distributed variant of *MJoin*.

The overall system performance is measured in terms of the number of *detected phenomena updates per second*. Other measures of performance include the *output delay*, the *input drop rate*, and the *output 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 buffers to accommodate new tuples (i.e., random load shedding). In all experiments, we assume that tuple dropping occurs due to limited CPU time and not 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. The output rate is measured in terms of the number of output join tuples per second. All the experiments in this section are based on a real implementation of the join operators inside *Nile* [11]. The *Nile* engine executes on a machine with Intel Pentium IV, CPU 2.4GHZ and 512MB RAM running

Windows XP.

7.2.1 Performancevalues Using Real Data Sets

The performance of *a HMJ tree*, *MJoin*, and *SNJoin* under the real sensor-platform setup is given in Figure 8. As illustrated in Figure 8a, *SNJoin* reduces the output delay by up to 36% over the *HMJ tree* and by up to 19% over *MJoin* (in case of 20 sensors). The output delay reflects the per-tuple processing time (i.e., from the time a tuple arrives at the operator buffer till its effect appears in the output). Notice that operators with lower per-tuple processing time, exhibit a lower input drop rate (Figure 8b), and consequently produce a higher output rate (Figure 8c). From the overall-performance point of view, *SNJoin* detects up to 75% more phenomena updates than *HMJ trees* and up to 43% more phenomena updates than *MJoin* (Figure 8d).

7.2.2 Performance Using Synthetic Data Sets

Performance gains of *SNJoin* become more significant for large-scale sensor networks. In contrast to binary join trees and *MJoin*, *SNJoin* avoids unnecessary probes to a huge number of separate tables, and therefore, reduces its per-tuple processing time. The same experiments of Section 7.2.1 are repeated using the 2000 sensor simulated setup. Figure 9 illustrates the efficiency of *SNJoin* in terms of the output delay, the input drop rate, and the output rate. *SNJoin* doublvalueses the output rate of a *HMJ tree* and increases the output rate by up to 60% over *MJoin*. Moreover, *SNJoin* detects up to 180% more phenomena updates than *HMJ trees* and up to 85% more phenomena updates than *MJoin*.

Figure 10 gives the behavior of the join techniques with respect to the dynamic configuration of the network. Every minute, a group of sensors (randomly chosen between 1 and 100 sensors) is either added or removed from the sensor set. Comparing Figure 9d and Figure 10, notice that the dynamic behavior of the network reduces the number of detected phenomena updates by up to 80% in case of a *HMJ tree*

and by up to 50% in case of *MJoin*. However, the performance of *SNJoin* is reduced by only 20% (at 2000 sensors).

7.2.3 Performance of Distributed SNJoin

In this Section, we study the distributed execution of *SNJoin* over clusters of *uniformly-distributed* sensors in the space. Clusters of sensors are obtained using a simulation of the *HEED* clustering technique [23] 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). We construct a one-level clustering hierarchy where cluster heads communicate through a multi-hop communication link. The number of hops between two communicating cluster heads is determined by the routing protocol [21]. 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 11 gives a comparison between the performance of a distributed variant of *MJoin* and the performance of two distributed variants of *SNJoin*: one with relevance feedback and the other without relevance feedback. The distributed variant 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 11, notice that *SNJoin* increases the number of detected phenomena changes by up to 30% over *MJoin*. Moreover, query processing with relevance feedback enhances the performance of *SNJoin* by up to 90% (for 2000 sensors).

The relevance feedback allows the join operation to focus on sensors with similar behavior, and hence, reduces the number of probed streams. Consequently, the per-tuple processing time is 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. Hence, the arity of the output join tuple is reduced. Experimentally, this reduction in the arity of the tuple does not exceed 12% (at

	Percentage reduction in					
No of sensors	**no of probes**	**output delay**	**drop rate**	**O/P rate**	**tuple width**	**comm. cost**
200	0	0	0	0	0	0
400	29.1	23.6	3.5	2.2	3.4	25.3
600	41.2	30.4	5.1	4.7	6.8	38.6
800	50.3	37.7	6.2	6.0	7.2	46.3
1000	60.8	47.5	7.4	6.9	7.9	57.3
1200	65.2	54.1	14.0	12.0	8.1	62.3
1400	69.6	58.8	33.6	29.1	8.6	64.9
1600	74.4	65.4	43.7	42.6	9.3	72.2
1800	77.4	67.6	51.0	47.3	9.9	73.8
2000	79.4	70.1	52.3	50.3	11.5	75.5

Figure 12. The effect of relevance feedback.

2000 sensors). Figure 12 illustrates the effect of the relevance feedback on the performance of *SNJoin* with respect to the reduction in the number of probed streams, the output delay, the input drop rate, the tuple width, and the communication cost (measured in terms of the number of bytes transmitted per second). In general, if we compare the full fledged *SNJoin* operator (i.e., *SNJoin* with relevance feedback) to its predecessor inside *Nile-PDT* (i.e., *MJoin*), we find out that *SNJoin* reduces the output delay by 70% and increases the number of detected phenomena updates by 150%.

7.2.4 Comparison of Analytical and Experimental Studies Results

In this section, we compare the output delay obtained from the analytical study presented earlier with the output delay obtained through experiments. The values of different constants that appear in the analytical analysis are summarized in Table 13. These values are based on the real sensor platform setup that is presented earlier in Section 7.2.1. In this setup, we vary the number of sensors (k) from 5 through 25. We consider 1000 readings from each sensor ($Average_n$) such that the domain from which these readings are drawn is of size 100 ($Distinct_1$). We set the number of buckets in all hash tables to 13

($Size\ of\ hash\ table$). All the constants ($C_1 \cdots C_5$) along with the selectivity among sensor data are assessed experimentally based on the generated values. Notice that the selectivity varies for each value of k (the number of sensors). Similarly, the parameter $Distinct_2$, which represents the total number of distinct elements in the global hash table of *SNJoin*, has a different value for each value of k. If each sensor has a 100 distinct value in its own hash table, the global hash table is supposed to contain less than $k \times 100$ distinct values due to the overlap of these values among the k sensor readings. Finally, the average number of joining streams (k') is obtained experimentally and is found to be 40%.

Figure 14 shows the result of the comparison. The analytical and experimental output delays exhibit the same trend for both *SNJoin* and outer *MJoin*. We notice that *SNJoin* performs better than the outer *MJoin* even with a relatively large value for k', 40% in this case. The more the phenomena are localized, the smaller the k' is and the better performnace of *SNJoin* is.

Parameter	Value	Computed/Assumed
k	$[5, 10, 15, 20, 25]$	Assumed
$Average_n$	1000	Assumed
$Distinct_1$	100	Assumed
$Size\ of\ hash\ table$	13	Assumed
C_1	26.25	Obtained experimentally
C_2	06.93	Obtained experimentally
C_3	0.24	Obtained experimentally
C_4	2.72	Obtained experimentally
C_5	5.7	Obtained experimentally
$Selectivity$	$[0.00130, 0.00129, 0.00122, 0.00116, 0.00112]$	Obtained experimentally for each value of k
$Distinct_2$	$[210, 372, 455, 485, 494]$	Obtained experimentally for each value of k
k'	40%	Obtained experimentally

Figure 13. Parameter and Constant Values for the Comparison.

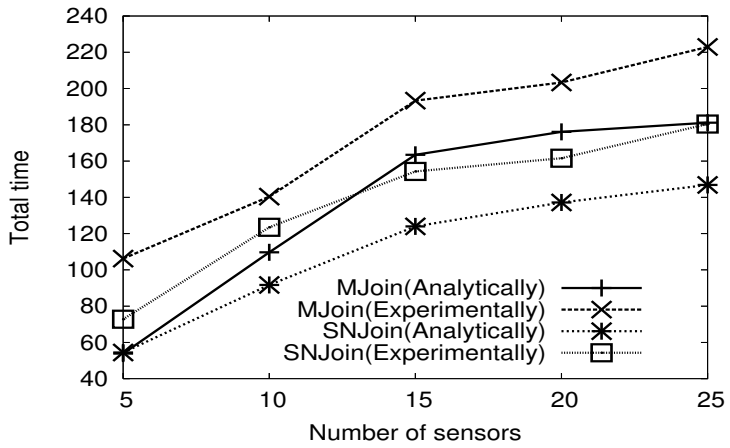


Figure 14. Comparison of Analytical and Experimental Output Delay for outer MJoin and SNJoin

8 Related Work

A large body of research in the data streaming area focuses on the join operation, e.g., [8, 9, 10, 13]. To highlight the reasons that make *SNJoin* applicable in PhenomenaBases, we overview related multi-way join techniques and compare them to *SNJoin*. Multi-way join can be achieved through a tree of binary joins (either *symmetric hash join* [20], *XJoin* [18], or *hash merge join* [14]), a single *MJoin* operator [19], or a single *SNJoin* operator. Based on the experiments in Section 7.2, Figure 15 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 variable input rates and require reorganization of the query plan operators (e.g., see [5]) to increase the output rate. All techniques handle outer joins by traversing the join probing

	Binary join Trees	**MJoin**	**SNJoin**
Scalability	×	√	√√
Dynamic configuration	×	√	√√
Symmetric Join	×	√	√
Reduction in output delay	×	√	√
Sensitivity to variable i/p rates	√	×	×
Query plan reorganization	√	×	×
variable-arity join support	×	×	√

Figure 15. Comparison among various multi-way join techniques (×: feature not supported, √: feature supported, √√: feature supported and enhanced)**.**

sequence completely. On the other hand, *SNJoin* supports, by design, variable-arity joins to avoid long chains of probing sequences.

9 Conclusions

In this paper, we presented the *SNJoin* (or Sensor-Network Join) operator, a variable-arity join operator for sensor-network PhenomenaBases. 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 probing sequence based on the selectivity between sensor pairs. *SNJoin* supports the distributed execution of the join operation with the capability to accept and process relevance feedback.

Experimental studies that are based on a real implementation of the join operators inside *Nile-PDT* show the scalability of *SNJoin*. *SNJoin* increases the output rate over binary join trees and *MJoin*. Once *SNJoin* is adopted by PhenomenaBases, the number of detected phenomena updates is increased while the output delay is reduced.

References

[1] M. H. Ali. Phenomenon-aware sensor database systems. In *Proc. of the EDBT Ph.D. Workshop*, March 2006.

[2] 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.

[3] M. H. Ali, M. F. Mokbel, W. G. Aref, and I. Kamel. Detection and tracking of discrete phenomena in sensor-network databases. In *SSDBM*, June 2005.

[4] A. Amis, R. Prakash, T. Vuong, and D. Huynh. Max-min d-cluster formation in wireless ad hoc networks. In *INFOCOM*, March 2000.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *ACM SIGMOD*, pages 261–272, May 2000.

[6] S. Basagni. Distributed clustering for ad hoc networks. In *the Intl. Symposium on Parallel Architectures, Algorithms and Networks (ISPAN)*, 1999.

[7] J. Considine, F. Li, G. Kollios, and J. W. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, pages 449–460, April 2004.

[8] L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, Sept. 2003.

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

[10] 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.

[11] 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.

[12] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Toward sophisticated sensing with queries. In *Proc. of IPSN*, 2003.

[13] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, March 2003.

[14] 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.

[15] R. Nowak and U. Mitra. Boundary estimation in sensor networks: Theory and methods. In *Proc. of IPSN*, 2003.

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

[17] U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. of VLDB*, 2004.

[18] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.

[19] 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.

[20] A. N. Wilschut and E. M. G. Apers. Pipelining in query execution. In *the Intl. Conf. on Databases, Parallel Architectures and their Applications*, 1991.

[21] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *ACM SenSys*, 2003.

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

[23] 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.

[24] G. K. ZIPF. Human behavior and principle of least effort: An introduction to human ecology. *Addison-Wesley Publishing Co., Reading, MA*, 1949.