Scalability Management in Sensor-Network PhenomenaBases *

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]Department of Electrical and Computer Engineering, University of Sharjah, P.O. Box 27272, Sharjah, U.A.E.
kamel@sharjah.ac.ae

Abstract

A phenomenon appears in a sensor network when a group of sensors persist to generate 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 new phenomena and to track already existing phenomena. The process of phenomena detection and tracking is initiated by a multi-way join operator that comes at the core of* PDT *techniques to report similar sensor readings. With the increase in the sensor network size, the join operator (and, consequently, query processing in the PhenomenaBase) face several scalability challenges. In this paper, we present a join operator for PhenomenaBases (the* SNJoin *operator) that is specially-designed for dynamically-configured large-scale sensor networks with distributed processing capabilities. Experimental studies illustrate the scalability 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 alarm in a building. In general, a phenomenon (as defined in [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 database of phenomena that develop in a sensor field. In particular, a PhenomenaBase has two basic functionalities: First, it executes *Phenomena Detection and Tracking* (*PDT*) techniques (e.g., [3]) continuously at the background of a sensor database system to detect new phenomena and to track the propagation of already-detected phenomena. Second, it uses the knowledge about detected phenomena to optimize subsequent user queries.

At the core of *PDT* techniques is an *outer multi-way* join operator that detects similarities among streams of sensor data over a sliding window of size ω. Notice that the join operation is a "multi-way" join because it detects similarities among multiple sensors. Also, the join operation 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.

In general, 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 tress 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. Hence, *MJoin* produces early output, maximizes the output rate, and avoids reorganization of the query plan at execution time. Therefore, an outer *MJoin* operator has been selected in the previous design of *PDT* techniques [3].

MJoin has satisfactory performance for moderate system

*This work was supported in part by the National Science Foundation under Grants IIS-0093116 and IIS-0209120.

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 face periods of heavy system loads, we identify three basic challenges that face the current design of PhenomenaBases with *MJoin*. These challenges can be summarized as follows:

1. The *scalability* challenge, where sensor networks are typically deployed in large scale with thousands of sensors.

2. The *dynamic-configuration* challenge, where 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 *distributed-execution* challenge, where the join operation is performed in a distributed fashion to eliminate the bottlenecks of a centralized system.

In this paper, we introduce the *SNJoin* (or *Sensor-Network Join*) operator as the successor of *MJoin* inside PhenomenaBases. *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, the concept of *variable-arity* join. Variable-arity join produces variable-size join output in response to the variable number of sensors contributing to a phenomenon. 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 the notions of variable-arity join and *RFB*, *SNJoin* scales with respect to the number of sensors in space and adapts to the dynamic configuration of the network.

The contributions of this paper can be summarized as follows:

1. We introduce the notion of variable-arity join and adopt it in the context of *SNJoin*.

2. We promote the distributed processing capabilities of *SNJoin* by performing the join at the sensor level.

3. We extend *SNJoin* with the ability to accept and process relevance feedback.

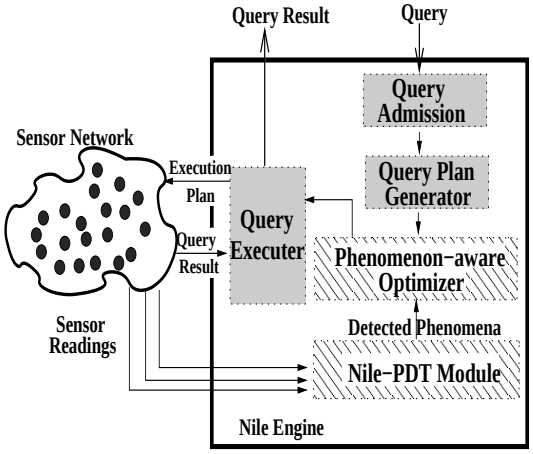


Figure 1. The architecture of Nile.

4. We provide an experimental study that is based on a real implementation of *SNJoin* inside *Nile-PDT* to prove its efficiency 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 outline of *SNJoin* and investigates the underlying sensor platform. Section 3 presents the variable-arity notion of *SNJoin*. Section 4 empowers *SNJoin* with distributed processing capabilities. Section 5 describes the relevance feedback mechanism of *SNJoin*. Section 6 provides an experimental study of the performance of various join techniques. Section 7 overviews related join techniques and compares them to *SNJoin*. Finally, Section 8 concludes the paper.

2. Outline of the SNJoin Algorithm

The concept of PhenomenaBases has been adopted by the *Nile* data stream management system [11]. Figure 1 illustrates the architecture of *Nile*. The basic components of *Nile* are 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 the capabilities of PhenomenaBases, 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 accepts phenomenon-aware feedback from *Nile-PDT* and, hence, optimizes user queries based on its knowledge about the sensor field.

As illustrated in Figure 2, the sensor platform of *Nile* is an ad-hoc network with resource-constrained sensor nodes.

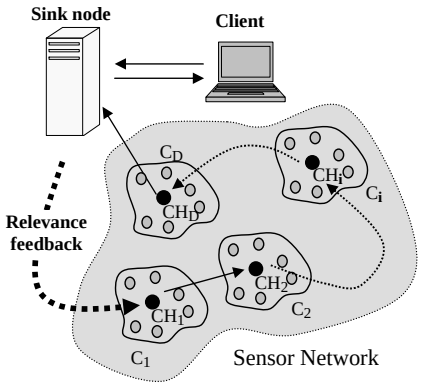


Figure 2. The sensor platform.

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 configure the network topology dynamically, e.g., [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 a node with high processing capabilities. The sink node analyzes the query result, assesses its relevance to the query, and returns feedback to cluster heads seeking further optimizations.

The basic steps of the *SNJoin* algorithm can be summarized as follows:

Step1. Each sensor forwards its readings to its 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 size (Section 3). The size of the join output depends on the number of joining sensors. Also, *SNJoin* handles late and out-of-order tuples in this step.

Step3. A distributed processing phase is initiated by cluster heads (Section 4). Each cluster head decides on a probing sequence to probe other cluster heads looking for matching tuples among members in their clusters. At the end of the probing sequence, the join result is shipped to the sink node.

Step4. The sink node measures the weight 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 to include clusters with similar values with high probability (Section 5).

3 Variable-arity Join

In this section, we introduce the concept of variable-arity join and compare it to the outer multi-way join that is 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 prohibitive. 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, we may end up probing thousands of sensors to detect tens of sensors with 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, below. Notice that variable-arity join is different from 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. 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}], $\cdots$). On the other hand, outer join produces a fixed-size tuple with NULL values in lieu of missing streams (even if we have many of these NULL values). At the implementation level, variable-arity join touches only streams that participate in the join. Streams with no matching tuples place zero cost. However, outer join probes every stream to check the existence of the join value (even if we have many of these streams). *SNJoin* avoids unnecessary long chains of probing sequences. Other techniques, i.e., binary join trees or

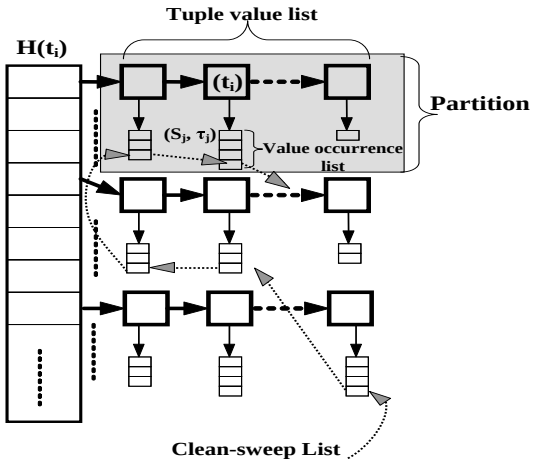


Figure 3. The SNJoin hash table.

MJoin, probe large numbers of sensors that may produce no output. The remainder of this section elaborates on the idea of variable-arity join in detail and discusses how it is implemented.

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 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, $o_i \in 1 \cdots m$, such that $\hat{t}$=t_{o_i} and $|\hat{\tau} - \tau_{o_i}| \leq \omega$, $S_{o_i} \neq \hat{S}$, $S_{o_i} \neq S_{o_j}$ $\forall i \neq j$ }.*

3.1 Data Structures

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 streams, 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 candidates for the join output in advance.

Figure 3 illustrates the hash table of *SNJoin*. The hash table is divided into partitions based on a suitable hash function H. The hash function is 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 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 is 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 chained chronologically, 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.

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 $\hat{t}$*
// TVL: Tuple Value List, VOL: Value Occurrence List,
// and CSL: Clean Sweep List

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 $\hat{S} \neq temp.S$ include $temp.\tau$ in the join output of $\hat{t}$
   temp=temp.next
   end
5. *Traverse* CSL *to delete expired tuples*

Figure 4. The SNJoin algorithm.

3.2 SNJoin Algorithm

The *SNJoin* algorithm is given in Figure 4. 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 in two steps: First, we separate the values in *VOL* based on their source stream into k sublists, i.e., a sublist per stream. Second, 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 becuase the join condition (i.e., equality on the tuple value) has 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.

3.2.1 Late and Out-of-order Arrival

In this section, we address two issues. First, we address late tuple arrivals, which means a tuple may arrive at the system's buffers ϵ time units past its timestamp. Second, we address out-of-order arrivals, which means tuples are not only late but also their order has been altered. To handle late tuple arrivals, we modify Step 5 in Figure 4. We do not expire tuples on the *clean sweep list (CSL)* unless they go outside the sliding window ω by a safety factor ϵ, i.e., $Current\,time - CSL.\tau > \omega + \epsilon$, where ϵ represents the maximum delay in the tuple's arrival [17]. The safety factor gives late tuples a chance to join with expired tuples.

In addition to handling late tuple arrivals, *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 arrived 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 [17].

3.2.2 Support for Multiple Window Sizes

Up to this point, we assume that the join operation is performed over a sliding window ω such that ω is fixed for all sensors. However, many applications require a different window size for each group of sensors or a different window size for each individual sensor (i.e., ω_i is the sliding window over stream S_i). In *SNJoin*, it is straightforward to support multiple window sizes. We change Step 4 of Figure 4 as follows to support multiple window sizes:

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 $\hat{\tau} - temp.\tau \leq \underline{\omega_{temp.S}}$
    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 widnow size $\omega_{temp.S}$ instead of ω, i.e., (i.e., $\hat{\tau} - temp.\tau \leq \omega_{temp.S}$).

4 Distributed SNJoin

Up to this point, *SNJoin* addresses 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, specially, with the increase in the network size. Scalable query processing requires the *en-route* processing of sensor readings, i.e., while they are being transmitted to the sink node. Examples of such in-network query processing include [7, 16, 22]. In this section, we present the 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 their 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.

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 a probing sequence that spans 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

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.

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.

clusters (Step 2). The cluster head sets a sequence number to one ($SeqNo = 1$) because 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 Query Processing with Relevance Feedback

In this section, we introduce the concept of query processing with relevance feedback. A major challenge in multi-way join queries over sensor networks is that actually, 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 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 practical to probe only clusters where it is more likely to find matches. The objective of query processing with relevance feedback is to guide the join operation to process only relevant cluster heads, i.e., clusters that generate the same values. This selective probing reduces both the processing cost and the communication cost 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 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 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 it back to the cluster head that initiated the probing sequence (i.e., CH_{o_1}). For simplicity, let w_i be 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 every other cluster head 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 given D cluster heads CH_1, CH_2, $\cdots$, CH_m, 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 given 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 11. 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.

6 Experiments

In this section, we conduct an experimental study to explore the performance of the proposed *SNJoin* operator. We use two sensor data sets that are extracted from the *Nile-PDT* system [2]. *Nile-PDT* has two experimental setups. 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 milli-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 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 6.1) investigates the performance under the real sensor-platform setup. The second set of experiments (Section 6.2) addresses the large-scale simulated sensor-network setup and examines the dynamic reconfiguration of the network. In Sections 6.1 and 6.2, 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.
3. *SNJoin*, where a variable-arity join is performed as described in this paper.

The third set of experiments (Section 6.3) 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*.

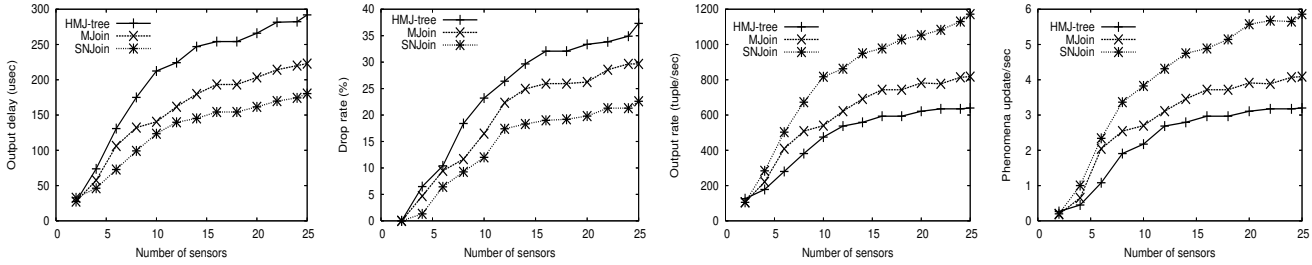
The overall system performance is measured in terms of the number of *detected phenomena updates per second*. Other measures of performance include the *output delay*, the *input drop rate*, and the *output rate*. The output delay is the time difference between the arrival of a tuple and the time its 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.

6.1 Performance Using Real Data Sets

The performance of *a HMJ tree*, *MJoin*, and *SNJoin* under the real sensor-platform setup is given in Figure 7. As illustrated in Figure 7a, *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 7b), and consequently produce a higher output rate (Figure 7c). 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 7d).

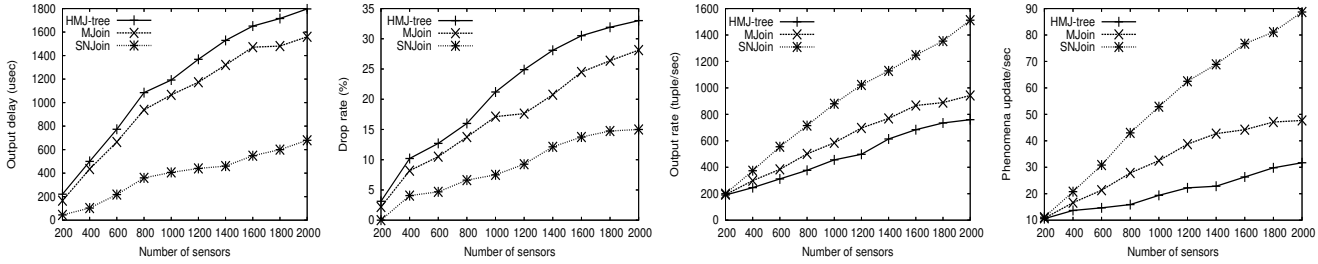
6.2 Performance Using Synthetic Data Sets

Performance gains of *SNJoin* become more significant for large-scale sensor networks. In contrast to binary join



(a) Output delay (b) Input drop rate (c) Output rate (d) Phenomena updates

Figure 7. Performance under *real small-scale* **data sets.**



(a) Output delay (b) Input drop rate (c) Output rate (d) Phenomena updates

Figure 8. Performance under *synthetic large-scale* **data sets.**

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 6.1 are repeated using the 2000 sensor simulated setup. Figure 8 illustrates the efficiency of *SNJoin* in terms of the output delay, the input drop rate, and the output rate. *SNJoin* doubles 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 9 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 8d and Figure 9, 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).

6.3 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 10 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 10, 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

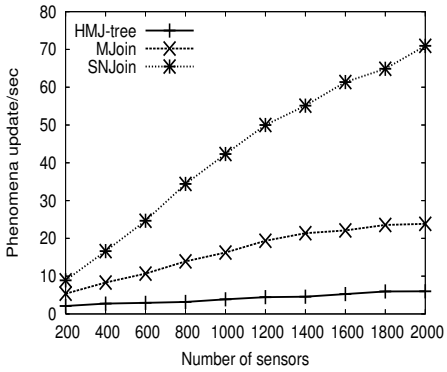


Figure 9. The effect of dynamic network configuration.

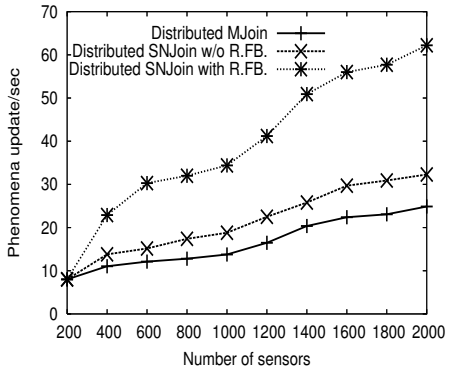


Figure 10. The effect of distributed query processing.

	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 11. The effect of relevance feedback.

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

Figure 12. Comparison among various multi-way join techniques ($\times$: feature not supported, $\surd$: feature supported, $\surd\surd$: feature supported and enhanced)**.**

the output join tuple is reduced. Experimentally, this reduction in the arity of the tuple does not exceed 12% (at 2000 sensors). Figure 11 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 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 6, Figure 12 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 sequence completely. On the other hand, *SNJoin* supports, by design, variable-arity joins to avoid long chains of probing sequences.

8 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 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. Flexible time management in data stream systems. In *PODS*, pages 263–274, 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.