PHENOMENON-AWARE DATA STREAM MANAGEMENT SYSTEMS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Mohamed H. Ali

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2007

To my parents, Hassan and Afaf, my wife, Doaa, and my kids, Nour and Mariam

ACKNOWLEDGMENTS

I would like to express my gratitude to a large number of people who have contributed generously to my success. These people have made a huge difference and a noticeable impact on both my professional and personal life.

I would like to express my gratitude to my advisor Prof. Walid Aref for his advice and guidance throughout my Ph.D. studies. I thank him deeply for the endless hours he spent with me to get me on the right research track. Walid was not only an advisor but he was also a friend. I remember how happy he feels and how helpful he is when I knock his door asking for an advice. Walid taught me how to conduct high impact research and how to build a solid systems experience. I am really happy having Walid as my advisor.

I am really grateful to Prof. Ahmed Elmagarmid for being a basic source of support, advice, and guidance to me. Ahmed is a person that I value his advice and respect his extensive knowledge and unlimited experience.

My gratitude to my advisory and examining committee, especially, Prof. Sunil Prabhakar and Prof. Dongyan Xu for their constructive suggestions and informative comments. My appreciation to Prof. Elisa Bertino, Prof. Mohamed Mokbel, Prof. Ibrahim Kamel, and Dr. Mourad Ouzzani for their sincere collaboration in various research projects.

I would like to convey my sincere thanks to the Database group at Microsoft Research. Special thanks to Jonathan Goldstein for being a wonderful mentor during my summer intern. I would like to thank Roger Barga for the friendly atmosphere and the productive environment that I enjoyed while working with him. Many thanks to Paul Larson and David Lomet for being a live example for me of how a perfect researcher would be. I deeply thank Cesar Galindo-Legaria from the Microsoft SQL

Server Group for his interest in my research and for his guidance during my job search.

Special thanks are due to my colleagues who helped me a lot during my graduate life. In particular, I would like to thank Mohamed Eltabakh, Hazem Elmeleegy, Moustafa Hammad, Mohamed Elfeky, Hicham Elmongui, and Xiaopeng Xiong. Many thanks to everyone in the Indiana Center for Database Systems (ICDS) group at Purdue University.

I would never forget the support and kindness of my family. I will be always grateful to my parents for everything they made for me in life. I thank my wife for her support and patience. I would never forget the fun my kids gave to me during the tough periods of my study. I thank everyone and I appreciate their effort, patience, and support.

TABLE OF CONTENTS

Page

Page

Page

LIST OF FIGURES

Figure Page

ABSTRACT

Mohamed H. Ali. Ph.D., Purdue University, May, 2007. Phenomenon-aware Data Stream Management Systems. Major Professors: Walid G. Aref.

Recent advances in large scale data streaming technologies enabled the deployment of a huge number of streaming sources in the surrounding environment, e.g., sensor fields. Streaming sources do not live in isolation. Instead, close-by streaming sources experience similar environmental conditions. Hence, close-by streaming sources may indulge in a correlated behavior and generate a "phenomenon". A phenomenon is characterized by a group of streaming sources that show "similar behavior" over a period of time. Examples of detectable phenomena include pollution clouds in the air, oil spills at the ocean surface, fire zones in a building, water floods of a river, migration of birds, and epidemic spread of diseases. This dissertation proposes a framework to detect, track, and query various forms of phenomena in data streaming environments. This framework empowers data stream management systems (DSMSs) with phenomenon-awareness capabilities. Phenomenon-aware data stream systems use high-level knowledge about phenomena in the data streaming environment to optimize the execution of subsequent user queries.

To approach the above goal, this dissertation proposes the principle that "phenomenon detection guides query processing" and explores this principle's implications on DSMSs. Hence, user queries have the option to view the streaming environment at a higher level, i.e., the phenomenon level. In such a phenomenon-aware query processing paradigm, streams are prioritized and are processed based on a mechanism that tunes query processing towards data streams that contribute to detected phenomena.

This dissertation provides a formal definition for a phenomenon, models the phenomenon behavior, and proposes an extended syntax that enables the users to register their interesting phenomenon patterns with the system. Also, this dissertation adopts the concept of phenomenon-aware query processing by adding two major components to DSMSs: the Phenomenon Detection and Tracking module (PDT-module) and the phenomenon-aware optimizer. The PDT-module encompasses scalable techniques to detect the appearance of new phenomena and to track the propagation of already-detected phenomena. The phenomenon-aware optimizer is an adaptive optimizer that optimizes user queries continuously based on the feedback it receives from the PDT-module. Finally, this dissertation considers phenomenon awareness at the distributed setup of sensor networks by providing a phenomenon-aware data acquisition protocol and by extending the phenomenon detection process to the sensor-network platform. As a vehicle for this research, the *Nile-PhenomenaBase* system is prototyped as a framework for phenomenon-aware query processing inside *Nile*, a data stream management system developed at Purdue University.

1 INTRODUCTION

A large body of research in the database systems area focuses on handling massive amounts of data that come from data streaming sources. The main goal is to provide efficient query processing techniques for stream data. However, emerging data streaming applications call for new capabilities that are beyond traditional online query processing techniques. Examples of these applications include surveillance [1], object tracking [2], and environmental monitoring [3]. Mainly, these applications go past simple data retrieval to show their evolving interest in data analysis and field understanding.

In this research, we focus on extending data stream management systems (*DSMSs*) with phenomenon-awareness capabilities as a step towards the understanding of streaming environments. A phenomenon appears in a streaming environment if a group of data streams show "similar behavior" over a period of time. Phenomenon-aware DSMSs (or *PhenomenaBases*) are databases of phenomena that develop in the streaming environment. In particular, phenomenon-aware DSMSs have two major functions: First, phenomenon-aware DSMSs detect and track various forms of phenomena in space. Second, phenomenon-aware DSMSs utilize the knowledge about phenomena in the space to optimize the execution of subsequent user queries. Although individual stream readings can be useful by themselves, *phenomenon detection and tracking* (*PDT*) exploits various notions of correlation among data streams and provides a global view of the underlying environment. *PDT* does not only detect phenomena once they appear but also tracks the propagation of detected phenomena to continuously 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 stream readings and the high-level understanding of phenomena to answer user queries efficiently.

1.1 Phenomena and their applications

As a first step towards phenomenon awareness, we propose a high-level definition of a phenomenon. A formal definition for the phenomenon is provided in Chapter 2. Three parameters control the phenomenon definition: the *persistency* (α), the *spread* (β), and the *time span* (w). The *persistency* of a phenomenon indicates that a certain behavior should occur at least α times to qualify as a phenomenon. Reading a value less than α times is considered noise, e.g., impurities that affect the readings of a streaming source. The *spread* of a phenomenon is the number of streaming sources that participate in this phenomenon. The time span w limits how far a streaming source can be lagging in reporting a phenomenon. w can be viewed as a time-tolerant parameter, given the common delays in a streaming environment. In the light of these three parameters, a phenomenon can be defined as follows:

Definition 1.1.1 *A phenomenon P takes place only when a set of at least β streaming sources report similar reading values at least α times within a time window w.*

Several applications benefit from the detection and tracking of various phenomena in a streaming environment. Examples of these applications include:

1. Tracing pollutants in the environment, e.g., oil spills or gas leakage.
2. Reporting the excessive purchase of an item at different branches of a retail store in a specific sales period.
3. Detecting computer worms that strike multiple computer sub-networks over a certain period of time.

Notice that a phenomenon may or may not have spatial properties. The phenomenon in the first example has spatial properties, where an oil spill is a contiguous portion of the ocean surface. If a phenomenon has spatial properties, it is referred to by the term *cloud*. Retail store applications may not have the notion of spatial attributes, where retail stores can be spread arbitrarily in space. In the third application, the notion of spatial distance is relative to the network connectivity.

To generalize the concept of phenomena, a stream source may be a physical device (e.g., sensor) that acquires readings from the environment, (e.g., temperature, light, humidity, or substance identifiers as in the first example) or a virtual sensor like the cashier machine that reads item identifiers as in the second example. A stream source may even be a piece of software that detects computer worms as in the third example.

The benefits of phenomenon awareness span other application domains as well. For example, the spread of epidemic diseases is an example phenomenon in the medical domain. Triggering an alarm once excessive heating is detected in a building is an interesting and time-critical phenomenon to firefighters. Migration of birds is an environmental phenomenon that is detected through sensors in the nests of different bird species. Water floods and excessive rain are meteorological phenomena of interest to many applications including emergency monitoring and air traffic control. Although phenomena appear across many domains, they all share the notion of persistent similarity in the behavior of multiple stream sources over time.

1.2 Motivation

In this section, we identify five major points through which data streaming applications benefit from phenomenon awareness. These points can be summarized as follows:

1. **High-level description of the streaming environment.** With the aid of phenomenon detection and tracking (PDT) techniques, an application may ask "what is going on in a streaming environment?" instead of asking "what are the individual stream readings?" PDT techniques describe the underlying streaming environment using a higher level of knowledge (e.g., report a fire alarm instead of a bunch of high temperature readings).

2. **Phenomenon-guided data acquisition.** Data acquisition can be guided by detected phenomena in the sense that we reduce the sampling rate of *non-*

interesting streaming sources (i.e., streaming sources that do not contribute to any phenomena). Also, we reduce the sampling rate of streaming sources that are (and will remain) involved in a phenomenon. Such streaming sources with persistent phenomena are temporarily turned off with the assumption that their phenomena will not disappear instantaneously. Streaming sources that are on the *boundaries* of a phenomenon tend to be more interesting and are likely to change their values quickly. We increase the sampling rate of boundary data streams such that we capture the possible change in their state as quickly as possible. Reducing the sampling rate of a streaming source will result in a general reduction in the source's energy consumed in sampling, processing, and communication. Also, the processing load over the DSMS is reduced.

3. **Data compression.** Voluminous stream data can be compressed using *PDT* techniques. Instead of maintaining the readings of each individual data stream, we maintain phenomenon pairs (R, B), where R is the region that bounds a phenomenon with a representative behavior B.

4. **Prediction.** Tracking a phenomenon and predicting its future trajectory foresees the next state of the streaming environment. Based on the boundary of a phenomenon and its trajectory, we can predict the future movement of various phenomena in space. Consequently, safety measures are prepared in advance and necessary actions are taken on time.

5. **Phenomenon-guided query processing.** Given a query and given a set of phenomena, query processing can be guided to regions with phenomena that are likely to satisfy the query predicates. Hence, the query space is reduced. By maintaining all phenomena in the space and by indexing their contributing streaming sources, each query can be associated only with the streaming sources that generate values of interest to this query.

1.3 Platforms

In this dissertation, we consider two platforms for DSMSs. The first platform is the centralized setup where we have a single copy of the DSMS running on a centralized server. The centralized DSMS receives the data stream readings at its input buffer without taking into consideration the transmission medium between the streaming sources and the server. The DSMS neither makes any assumption on the processing capabilities of the streaming sources nor has the capability to control their sampling rates.

In the second platform, we consider the distributed setup of sensor networks. Due to the large number of emerging sensor-network applications, query processing over sensor networks has been investigated, e.g., see [2, 4–9]. Sensors are devices (usually wireless) that are capable of sensing, processing, and transmitting readings from the environment to a sink node. The processing capabilities of the sensors are usually limited while the sink node is a powerful node that is running a full-fledged DSMS. Because sensors are usually deployed in places where it is either difficult or dangerous for human beings to reach, e.g., habitat monitoring [3], battery life and power consumption are crucial issues in sensor-network processing. Because wireless transmission is a power-hungry operation, most of the sensor's power is disseminated in communicating with other sensors or with the sink node.

To reduce the communication cost, several techniques are proposed to configure the network topology dynamically, e.g., the HEED protocol [10]. These techniques exchange messages 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 with a higher energy level, is designated to serve as the cluster head. A cluster head receives readings from its cluster members and forwards them to the centralized DSMS, possibly through a multi-hop route. Cluster heads communicate with each other to achieve a distributed query processing at the senor-network level. Cluster heads may be recursively clustered

into a hierarchy of clusters such that each cluster head communicates with its cluster members and forwards the cluster members' readings to its parent. In our platform, we assume one level of clusters where cluster heads are capable of communicating with each other directly.

1.4 Contributions

To leverage DSMSs with phenomenon awareness, we carry out several contributions that touch various components of a DSMS. We summarize our contributions as follows:

1. **Formal Definition:** We provide a formal definition for a phenomenon, identify the parameters that control the phenomenon definition, and model the behavior of a phenomenon.

2. **Query Language:** We extend the SQL language with a phenomenon definition and manipulation language.

3. **Architecture:** We propose an architecture for phenomenon-aware DSMSs that emphasizes the newly-added components and highlights the changes in the already-existing components of the system.

4. **Phenomenon Detection and Tracking (PDT):** We introduce a framework for the detection and tracking of phenomena. More specifically, we present the following techniques:

   (a) **PDT using Multi-way Join:** We provide a phenomenon detection and tracking algorithm using a multi-way join operation and identify the scalability challenges that face the multi-way join operation.

   (b) **PDT using Variable-arity Join:** We develop a new join operator, the variable-arity join operator, to handle several scalability problems in multi-way joins.

(c) **Load Shedding:** We design a phenomena-guided load shedder to drop tuples that do not participate in desirable phenomena. Desirable phenomena are identified based on a set of preferences that are given by the user.

5. **Query Optimization:** We present a new paradigm for efficient query processing through an adaptive phenomenon-aware query optimizer.

6. **Phenomenon-awareness in Sensor Networks:** We address phenomenon awareness in a sensor-network setup. More specifically, we propose the following techniques:

   (a) **Data Acquisition:** We introduce *SPASS*, a phenomenon-guided data acquisition protocol over sensor networks. SPASS is a scalable and energy-efficient protocol that acquires sensor data with high accuracy.

   (b) **In-network Query Processing:** We make use of the in-network processing capabilities of sensor networks to shift the join operation from the centralized DSMS to the sensors' level. Moreover, we propose a relevance feedback mechanism to enhance the performance of the distributed join operation.

The above contributions are materialized in the context of the *Nile PhenomenaBase* prototype system [11]. Nile PhenomenaBase is a phenomenon-aware DSMS that is based on the *Nile* DSMS [12] developed at *Purdue University*. Inside Nile, the components that are rewritten to support phenomenon awareness include the SQL language parser, the data acquisition controller, the load shedder, the query plan generator, and the query executer. The variable-arity join is added to the query plan generator as new logical operator and, consequently, its corresponding physical implementation is added to the query executer. Two new components are introduced to Nile: the PDT module and the phenomenon-aware query optimizer. Based on a prototype implementation, we demonstrate experimentally the efficiency and

the scalability of *Nile PhenomenaBase*. We support the experimental results with mathematical verification wherever applicable.

1.5 Organization of the Thesis

The remainder of this dissertation is organized as follows. Chapter 2 provides a formal definition for a phenomenon, extends the SQL syntax of DSMSs with phenomenon definition/manipulation statements, and introduces the architecture of phenomenon-aware DSMSs. As we stated earlier in this chapter, phenomenon-aware DSMSs have two major tasks: (1) phenomenon detection and tracking (PDT) and (2) phenomenon-aware query processing and optimization. Chapters 3 - 5 address phenomenon detection and tracking. Chapter 3 discusses the phases of the phenomenon detection and tracking process. To address scalability challenges in the PDT process, Chapter 4 proposes a load shedding mechanism to drop tuples that do not contribute to desirable phenomena before they get into the PDT query processing pipeline. Chapter 5 introduces a new join operator, the variable-arity join operator, to address the same scalability challenges inside the PDT query processing pipeline. Chapter 6 addresses phenomenon-aware query processing and optimization in DSMSs.

While Chapters 2 - 6 focus on a centralized DSMS setup, Chapters 7 and 8 address phenomenon awareness in a sensor-network platform. Chapter 7 proposes a phenomenon-aware data acquisition protocol over sensor networks. Chapter 8 extends the variable-arity join operation to the sensor-network platform where it utilizes in-network query processing capabilities. Chapter 9 concludes the dissertation and provides directions for future work.

Parts of this dissertation have been published in workshops, conferences, and journals. The basic data streaming functionalities of the Nile DSMS are demonstrated in ICDE-2004 [12]. The phenomenon detection and tracking techniques are published in SSDBM-2005 [13] and SSDBM-2006 [14] and are demonstrated in VLDB-2005 [15].

The phenomenon-aware query optimizer is published in MDM-2007 [16]. The data acquisition protocol is published in MobiDE-2005 [17]. An overview of the Nile PhenomenaBase system is published in the EDBT Ph.D. Workshop [11] and is extended in the LNCS journal [18].

2 PHENOMENON-AWARE DSMSS

In this chapter, we provide a formal definition for the phenomenon and explore the parameters that control the phenomenon definition. Based on the phenomenon definition, we investigate several notions of similarity among streams' behavior and model the phenomenon overall behavior. To deal with various types of phenomena inside a DSMS, we extend the SQL syntax with a phenomenon definition and manipulation language. The phenomenon definition and manipulation language enables the user to register a set of interesting phenomenon patterns in the system and to interact with the detected phenomena later on. Then, we provide an architecture for phenomenon-aware DSMSs that can support the proposed extended syntax. We also highlight the basic idea behind the concept of phenomenon-aware query processing and optimization. Finally, we discuss the Nile PhenomenaBase experimental setups as our testbed for all the experiments that are conducted in the following chapters.

This chapter is organized as follows. Section 2.1 introduces a phenomenon definition and models the phenomenon behavior. Section 2.2 presents the extended SQL syntax. Section 2.3 discusses the proposed architecture. Section 2.4 highlights the idea of phenomenon-aware query processing and optimization. Section 2.5 explains the experimental setups. Section 2.6 overviews related work while Section 2.7 summarizes this chapter.

2.1 Phenomenon Definition

In Section 1.1, we defined a phenomenon P as a set of at least β streaming sources that report similar reading values at least α times within a time window w. In this section, we provide a formal definition for the phenomenon and discuss its parameters. Definition 2.1.1 defines a phenomenon at time instant τ to be an

R-B pair. R is a list of the streaming sources that contribute to the phenomenon and B is a representative behavior for the phenomenon over a sliding window of size ω. Notice that the phenomenon behavior is captured over a window of size ω to avoid the noise in the stream readings and to accommodate various sources of delay in the streaming environment. ω is defined to be the *time span* of the phenomenon. Each streaming source S_i contributes to the phenomenon (i.e., $S_i \in R$) if it exhibits (over the most recent time window of size ω) similarity with the other streaming sources that contribute to the phenomenon overall behavior at least α times ($Count(Value(S_i[\hat{\tau}]) \in B_\omega) \geq \alpha$). α is defined to be the *persistency* threshold of the phenomenon because it indicates how many times the streaming source has to persist in contributing to the phenomenon. In order for a set of streaming sources R to qualify as a phenomenon, the number of streaming sources in this set is required to reach a minimum threshold ($||R|| \geq \beta$). β is defined to be the *spread* threshold of the phenomenon. If the number of streaming sources is less than β, these sources are ignored and are not reported as a phenomenon.

Definition 2.1.1 *A phenomenon P at time instant τ is a binary tuple (R ,B_ω), where R is a list of the streaming sources with similar behavior and B_ω is the overall representative behavior of phenomenon P over the most recent time window of size ω, such that $\forall S_i \in R$, $Count(Value(S_i[\hat{\tau}]) \in B_\omega) \geq \alpha$, $\hat{\tau} \in [\tau - \omega + 1 \cdots \tau]$ and $||R|| \geq \beta$.*

Based on Definition 2.1.1, notice that the detection of a phenomenon is controlled by three parameters: α, β, and ω. α places a constraint on the frequency of the stream readings that constitute the phenomenon while β and ω place the spatial and the temporal constraints of the phenomenon. The phenomenon is identified by a list of streaming sources (R) with similar behavior and a representative behavior (B_ω). The phenomenon representative behavior B_ω captures the overall phenomenon behavior and is calculated using a summarization function over all the streaming sources in R. The remainder of this section details the proposed notions of similarity

among stream's behaviors (Section 2.1.1) and gives possible representations for the phenomenon overall behavior B_ω (Section 2.1.2).

2.1.1 Similarity Notions

Various notions of similarity among the streams' behavior control the way a phenomenon is detected. Examples of these similarity notions can be summarized as follows:

1. **Equality,** where similarity is simply reduced to equality. Two values v_1 and v_2 are considered similar if $v_1 = v_2$.

2. **Distance similarity,** where similarity is assessed based on a distance function "$dist$". Two values v_1 and v_2 are considered similar if $dist(v_1, v_2) \leq D$.

3. **Summary similarity,** where we extract summaries from the stream data (e.g., histograms, count sketches, or user-defined summaries) that capture the streams' behavior over a window of time. Similarity is assessed based on the distance between the summaries, i.e., $dist(Fn_1(v_1), Fn_2(v_2)) \leq D$, where Fn_1 and Fn_2 are summarization functions.

4. **Trend similarity,** where the increase/decrease in one stream readings implies the increase/decrease of another stream's readings. Generally, the change in the readings of one stream is related to the change in the other stream's readings by a correlation function F (i.e., $v_1 = F(v_2)$). For example, the increase in the readings of smoke detectors is usually accompanied by an increase in the readings of temperature sensors in case of a fire.

2.1.2 Phenomenon Representative Behavior

The phenomenon representative behavior expresses the overall phenomenon behavior and captures the content values of the phenomenon underlying data streams.

The similarity notion among the streams' behavior (as explained in Section 2.1.1) decides on how the phenomenon representative behavior is calculated. For example, if we reduce the similarity notion to equality, the phenomenon representative behavior is simply the value that is generated by the phenomenon underlying streams (Equation 2.1).

$$B_w = S_i[\hat{\tau}], \forall S_i \in R, \hat{\tau} \in [\tau - \omega + 1 \cdots \tau] \tag{2.1}$$

In case of distance similarity, we propose several ways to express the phenomenon representative behavior. First, we represent a phenomenon behavior by the average value of the streams contributing to the phenomenon. This behavior representation is obtained using Equation 2.2. The average value of each stream is obtained over the most recent window ω, then, the average over all streams is considered to be the behavior representation.

$$B_w = Avg_i(Avg^{\tau}_{\hat{\tau}=\tau-\omega+1}\ S_i[\hat{\tau}]), \forall S_i \in R \tag{2.2}$$

Second, a phenomenon behavior can be summarized and represented by the k most frequent elements across its underlying data streams (i.e., *top-k vector*), e.g., [19]. As in Definition 2.1.2, the top-k vector contains a subset of k elements such that the count of all elements in the top-k vector is equal or greater than the count of all other elements. In contrast to representing the phenomenon by a single value, the top-k vector provides a more accurate representation of a phenomenon behavior.

Definition 2.1.2 *Given a set of elements E, the top-k vector E_{topk} is a subset of elements such that $Count(e_i) \geq Count(e_j)$, where $e_i, e_j \in E$, $e_i \in E_{topk}$, $e_j \notin E_{topk}$, and $|E_{topk}| = k$.*

Third, a phenomenon behavior can be represented using a histogram of its underlying values. Histograms represent the item values in a data set along with their frequencies in a compact form with high accuracy. Histograms and top-k vectors,

similar to other summarization methods, are capable of capturing the intrinsic behavior of a phenomenon.

If the similarity notion among streams' behaviors is chosen to be a summary similarity, the overall phenomenon behavior can be calculated as an extra summarization level or as an addition of the streams' individual summaries. For example, histograms and top-k vectors are additive. If we have two data streams, each one of them is summarized using a histogram or a top-k vector, the two histograms/top-k vectors can be added together to obtain the histogram/top-k vector that represents the combined data of the two data streams [19].

Finally, for trend similarity, we do not only maintain a summary of the stream values but we also keep track of the parameters of the correlation function f. Given the reading values of a stream S_i, we can get a sense of the reading values of stream S_j if $S_j[\hat{\tau}'] = F(S_i[\hat{\tau}])$ (and $|\hat{\tau} - \hat{\tau}'| \leq \omega$). For example, in case of linear correlation among stream sources, we maintain the parameters a and b of the correlation function $S_j[\hat{\tau}'] = a \times S_i[\hat{\tau}] + b$.

Phenomena that are detected using the equality notion of similarity are termed *discrete phenomena*. Discrete phenomena are suitable for stream readings that are drawn from a discrete domain or that are drawn from a continuous domain but the readings are quantized by the sensing devices into discrete intervals. Unless mentioned otherwise, we limit the scope of this dissertation to the equality notion of similarity and we study the effect of discrete phenomenon awareness on various components of the DSMS.

2.2 Extended SQL Syntax

In a phenomenon-aware DSMS, an extended SQL syntax gives the user the ability to register a phenomenon definition of interest, to list the detected phenomena, and to pose queries against the detected phenomena. In this chapter, we overview the basic SQL statements of a DSMS (Section 2.2.1) and describe the extended

syntax for phenomenon awareness (Section 2.2.2) . More specifically, we use the Nile PhenomenaBase syntax as a vehicle to describe the semantics of the proposed syntactic constructs.

2.2.1 Basic SQL Syntax for Data Streaming

Similar to the create-table statement in traditional database management systems, Nile has a create stream statement. Q 2.2.1 gives the general form of the create-stream statement. The create-stream statement declares the expected schema of the incoming stream tuples. The stream tuples can be acquired either from a text data file or from a network port. Example 2.2.1 creates a data stream with a stream name of "EntranceSensor". The schema of the created stream is a pair of integer data types: one for the temperature and one for the humidity. The stream tuples are received at the network IP:128.10.9.155 port: 5600.

Q 2.2.1 *CREATE STREAM <stream-name>*
(· · · <schema> · · ·)
FROM [<file-name> | <data-port>];

Example 2.2.1 *CREATE STREAM EntranceSensor*
(int Temperature,
int Humidity)
FROM IP:128.10.9.155 PORT 5600;

If we have a huge number of streaming sources, it becomes a tedious process to register each stream source individually in the DSMS. The create-stream-bundle statement allows a bundle or an array of streams to be registered at once. The create-stream-bundle statement has the same syntax as the create-stream statement except that it takes the size of the expected stream array. Q 2.2.2 gives the general form of the create-stream-bundle statement while Example 2.2.2 creates a stream bundle for 200 temperature-humidity sensors. To refer to the stream tuples that are

coming from all the data streams in the bundle, we use the bundle name (e.g., SB). We use the subscript notation (e.g., $SB[i]$) to refer to an individual data stream in the bundle.

Q 2.2.2 *CREATE STREAM BUNDLE <stream-bundle-name> [size]*
(⋯ <schema> ⋯)
FROM [<file-name> | <data-port>];

Example 2.2.2 *CREATE STREAM BUNDLE SB[200]*
(int Temperature,
int Humidity)
FROM IP:128.10.9.155 PORT 5600;

Queries can be posed against a data stream or against a stream bundle using the select-from-stream and the select-from-stream-bundle statements, respectively. Q 2.2.3 and Q 2.2.4 present the general form of the select-from-stream and the select-from-stream-bundle statements while Example 2.2.3 and Example 2.2.4 give the corresponding examples. Notice that the select-from-stream-bundle statement retrieves tuples that satisfy the query predicates from the union of all individual streams in the bundle. The $SB.id$ attribute is a bundle-defined attribute that can be used inside the select statement to refer to the identifier of the stream that issues the tuple.

Q 2.2.3 *SELECT ⋯*
FROM STREAM ⋯
WHERE ⋯
WINDOW ⋯;

Example 2.2.3 *SELECT EnteranceSensor.Temperature*
FROM STREAM EnteranceSensor
WHERE EnteranceSensor.Humidity>30
WINDOW 60;

Q 2.2.4 *SELECT* $\cdots$
FROM STREAM BUNDLE $\cdots$
WHERE $\cdots$
WINDOW $\cdots$*;*

Example 2.2.4 *SELECT SB.id, SB.Temperature*
FROM STREAM BUNDLE SB
WHERE SB.Humidity>30
WINDOW 60;

2.2.2 Extended SQL Syntax for Phenomenon Awareness

The create-phenomenon statement defines a phenomenon pattern with a user-specified name over a system-registered stream bundle. Q 2.2.5 gives the general form of the create-phenomenon statement. The create-phenomenon statement defines a pattern of interest. The pattern clause states how two streams in the bundle share a similar behavior, i.e., Fn_1(SB[i]) is correlated to Fn_2(SB[j]) under a correlation function REL, where Fn_1 and Fn_2 are user-defined functions. Also, the create-phenomenon statement puts thresholds on the phenomenon persistency, spread, and time span parameters. The phenomenon persistency needs to be greater than or equal to α, the phenomenon spread needs to be greater than or equal to β, and the phenomenon should be detected over a time span that is less than or equal to ω. Finally, the where clause specifies a set of predicates that are applied on each data stream in the bundle to include/exclude tuples from being considered in the phenomenon detection process.

Q 2.2.5 *CREATE PHENOMENON <phenomenon-name>*
ON STREAM BUNDLE <stream-bundle-name>
PATTERN Fn_1*(SB[i]) REL* Fn_2*(SB[j])*
PERSISTENCY α

SPREAD β

TIME SPAN ω

WHERE <*other conditions*>;

Example 2.2.5 gives an example of a phenomenon definition that is interested in heat zones. Heat zones are temperature zones such that the temperature readings are greater than 90 (to be considered heat). In each zone, a stream reading should not be more than 3 degrees apart from other stream's readings to be considered as showing similarity in behavior. Each data stream should participate in the heat behavior at least 5 times to declare its persistency. A heat zone has a minimum size of 4 data streams. The behavior of the phenomenon is tracked over the most recent time window of size one minute (60 seconds). Notice that the create-phenomenon statement registers a phenomenon pattern in the system under which multiple phenomena are detected. For example, multiple heat zones can be detected; each heat zone has similarity in the behavior of its contributing data streams and, meanwhile, each heat zone has an overall behavior that is different from other heat zones' behaviors.

Example 2.2.5 *CREATE PHENOMENON HeatZones*

ON STREAM BUNDLE SB

PATTERN $|SB[i].Temperature - SB[j].Temperature| \leq 3$

PERSISTENCY 5

SPREAD 4

TIME SPAN 60

WHERE $SB.Temperature > 90$;

Under heavy load conditions, the DSMS becomes incapable of detecting all phenomena that develop in the environment. Instead of dropping tuples and, consequently, dropping phenomena randomly out of the system, we extend the create-phenomenon statement with a *with-preference* clause that indicates the user's preference of which phenomena to keep in the system. Consider a phenomenon with persistency, spread, and time span parameters of $\hat{\alpha}$, $\hat{\beta}$, and $\hat{\omega}$, respectively such that

$\hat{\alpha} \geq \alpha$, $\hat{\beta} \geq \beta$, and $\hat{\omega} \leq \omega$. The user may have preference either towards *persistent* phenomena (large $\hat{\alpha}$) or towards *intermittent* phenomena (small $\hat{\alpha}$). The asc/desc specifier selects the ordering in which the phenomena are prioritized inside the system. Similarly, the user may have preference either towards *stretch* phenomena (large $\hat{\beta}$) or towards *confined* phenomena (small $\hat{\beta}$). Finally, the user may have preference either towards *durable* phenomena (large $\hat{\omega}$) or towards *impulse* phenomena (small $\hat{\omega}$). Q 2.2.6 gives the general form of the with-preference clause.

Q 2.2.6 *CREATE PHENOMENON <phenomenon-name>*
ON STREAM BUNDLE <stream-bundle-name>
PATTERN Fn_1*(SB[i]) REL* Fn_2*(SB[j])*
PERSISTENCY α
SPREAD β
TIME SPAN ω
WHERE <other conditions>
WITH [*asc* | *desc*] *PREFERENCE IN*
[*PERSISTENCY* | *SPREAD* | *TIME SPAN*];

After the user registers a phenomenon pattern in the system, the user can issue a list-phenomena statement to query the system about detected phenomena. The syntax of the list-phenomena statement is given in Q 2.2.7. The system displays the list of detected phenomena in response to the list-phenomena statement by associating a phenomenon identifier with each phenomenon. Then, the system lists each phenomenon identifier, the streaming sources that contribute to this phenomenon, and the phenomenon representative behavior. Moreover, the user has the option to pose his subsequent queries against a set of phenomena through the select-within-phenomena statement. The select-within-phenomena statement limits the execution of the query to the streaming sources that contribute to the user-given list of phenomenon identifiers (list-of-ph-ids) as shown in Q 2.2.8.

Q 2.2.7 *LIST PHENOMENA;*

Q 2.2.8 *SELECT* $\cdots$
FROM $\cdots$
WHERE $\cdots$
WITHIN PHENOMENA <list-of-ph-ids>;

The "select-within-phenomena *" query (as given in Q 2.2.9) is a variation of the "select-within-phenomena <list-of-ph-ids>" query where the query execution is limited to the streaming sources that contribute to any phenomenon. If the streaming source is not contributing to any phenomenon, it is considered as noise and is not included in the query answer. More interestingly, the "select-within-phenomena" query gives a wide room for optimization. A smart optimizer makes use of the knowledge about phenomena in the space to guide the query to phenomena (and only to phenomena) that are likely to satisfy the query predicates. Phenomenon-aware query optimizers are discussed in detailed in Chapter 6.

Q 2.2.9 *SELECT* $\cdots$
FROM $\cdots$
WHERE $\cdots$
*WITHIN PHENOMENA *;*

2.3 The Architecture

In this section, we examine the basic components of a DSMS and discuss the new components that we add for phenomenon awareness. We focus on the Nile PhenomenaBase architecture as our prototype system for this research. Figure 2.1 shows the architecture of Nile PhenomenaBase. Nile [12] provides the basic data streaming functionality for Nile PhenomenaBase. In other words, Nile PhenomenaBase extends Nile with the phenomenon-awareness capabilities.

The basic components of *Nile* are the stream admission controller, the query admission controller, the stream monitor, the query plan generator, and the query

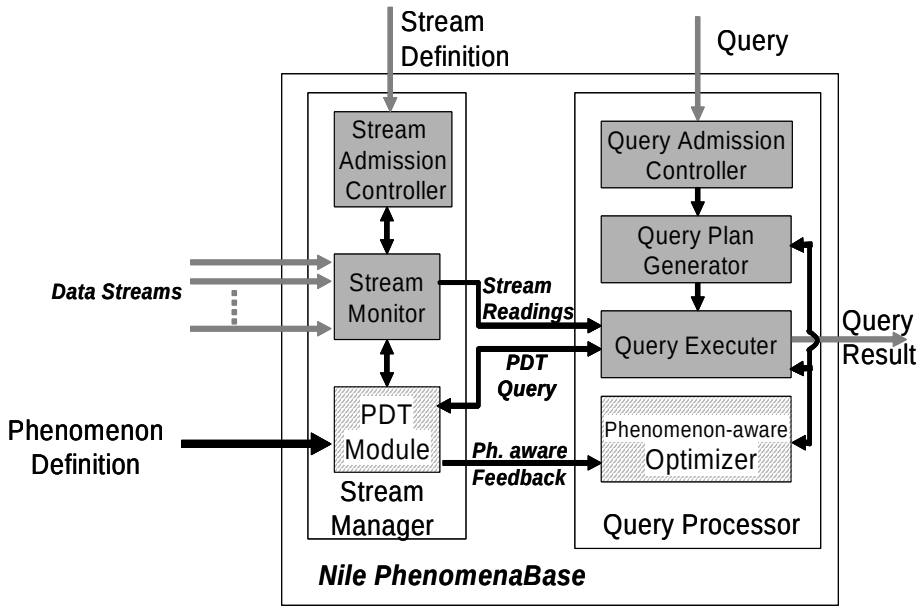


Figure 2.1. The Nile PhenomenaBase Architecture.

executer. The stream admission controller and the query admission controller decide on the admission of new data streams and new queries, respectively, based on system resources. The stream admission controller processes the *"CREATE STREAM"* and the *"CREATE STREAM BUNDLE"* statements. The query admission controller processes the *"SELECT $\cdots$ FROM STREAM"* and the *"SELECT $\cdots$ FROM STREAM BUNDLE"* statements. *Nile* has a stream monitor component to monitor streams as they join, leave, and change their location in the environment (in case of mobile streaming sources). The stream monitor is also responsible for receiving the stream data and for pushing this stream data inside the system's input buffers. The query plan generator generates corresponding query plans for the queries that are admitted into the system by the query admission controller. The query executer executes the generated query plans over the incoming data stream tuples and, meanwhile, streams the output result back to the user.

We adopt the concept of phenomenon-aware query processing inside *Nile* by adding two major components: the *Phenomenon Detection and Tracking module*

(*PDT-module*) and the *phenomenon-aware optimizer*. The *PDT-module* detects the appearance of new phenomena and tracks the propagation of already-detected phenomena. The *PDT-module* processes the *"CREATE PHENOMENON"* statement by converting the create-phenomenon statement into a continuous query (called, PDT query) that is shipped to the query executer. The PDT query detects the similarity pattern that is defined in the create-phenomenon statement and continuously tracks any changes in the streaming sources that contribute to this pattern. The output of the PDT query is streamed back to the PDT module to keep track of existing phenomena in the system.

The phenomenon-aware optimizer optimizes user queries based on the feedback it receives from the *PDT-module*. The phenomenon-aware optimizer processes the *"SELECT $\cdots$ WITHIN PHENOMENON"* statement by guiding query processing to the query's interesting phenomena. The phenomenon-aware optimizer is connected to both the query plan generator and to the query executer. Both the query plan generation and the query execution phases can be altered by the phenomenon-aware optimizer. In this dissertation, we focus on the effect of the phenomenon-aware optimizer on the query execution phase. The query execution phase is a continuous process that lasts as long as the continuous query is registered in the system. Hence, the phenomenon-aware optimizer continuously guides the query to interesting phenomena and adaptively adjusts itself based on the continuous feedback it gets from the PDT-module. Various components of the Nile PhenomenaBase architecture are detailed throughout the chapters of this dissertation.

2.4 Phenomenon-aware Query Processing and Optimization

By looking at existing phenomena within the streaming sources, the query processor will have a fine-resolution view over all the streams. Based on this fine view, the query optimizer decides which phenomena need to be investigated to answer a specific query. Initially, a user expresses in the extended SQL language the definition

of the phenomenon patterns that the user is interested in. Consequently, the system would detect and track all the phenomena of interest. User queries operate on the detected phenomena as well as on the raw stream readings. Hence, user queries have the option to view the streaming environment at a higher level, i.e., the phenomenon level. In such a phenomenon-aware query processing paradigm, streams are prioritized and are processed based on a mechanism that tunes query processing towards data streams that contribute to detected phenomena. Data streams that do not contribute to any phenomena are considered outliers and are not included in the query answer.

The main idea is to let the DSMS observe the input data streams at the phenomenon level. Then, each incoming user query is directed only to those phenomenon regions that are likely to satisfy the query predicates. More specifically, each incoming continuous query is directed only to the phenomenon underlying data streams that participate in the query answer. Detected phenomena act as an indexing scheme that direct the execution of continuous queries to only those streaming sources that can contribute to the query answer.

The *phenomenon-aware* query optimizer achieves a trade-off between the number of phenomena (and their underlying streaming sources) that participate in the query execution and the accuracy of that query. As a result of such phenomenon-guided query processing, the search space is optimized and is reduced to the phenomena that are likely to participate in the query answer. For example, given the heat zones that are detected by the phenomenon definition in Example 2.2.5, and given a user query that filters the sensor data based on a temperature predicate, the query processor limits its attention only to the heat zones that satisfy the query predicate.

Figure 2.2. The Nile PhenomenaBase sensor platform.

2.5 Experimental Setup

We base our study on two experimental setups from the Nile PhenomenaBase system [15]. The first setup is a *real* small-scale sensor board[1] with a grid of 5×5 temperature sensors (Figure 2.2). Due to hardware limitations, the number of sensors is limited to 25. However, we overload the system by increasing the sampling rate of each sensor. The sensor nodes stream their data to a centralized server running Nile PhenomenaBase through a wired connection. The processing capabilities of the sensor board are limited to controlling the sensors' sampling rates. The sensor board accepts control signals from the Nile PhenomenaBase and adjusts the sensors' sampling rates accordingly. We generate phenomena by moving a heat effect back and forth over the sensor board. Details about the hardware of the sensor board can be found in [20].

The second setup simulates a larger scale streaming environment (up to 2000 streaming sources). Simulation gives us both the flexibility to have a large-scale setup and the ability to control the processing capabilities of each streaming source. The number of generated values, the distribution of the generated stream values,

[1]The sensor board is designed and implemented in collaboration with the Mobile and Pervasive Computing Laboratory, University of Florida.

and the interarrival time between consecutive tuples can be tuned to adjust the required system load. The default parameters of the simulated setup are as follows. Each streaming source generates a stream of $10,000$ tuples where the tuple values follow the Zipfian distribution [21]. 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. To generate phenomena, streaming sources are grouped and each group is assigned the same parameter of the Zipfian distribution but with different seeds to generate (and to persist in generating) close by values. The persistency of a generated phenomenon is proportional to the inter-arrival time between consecutive tuples. To model the spread of a phenomenon, the number of streaming sources in each group is a random number that ranges from 0% to 10% of the total number of sources. A streaming source may participate in more than one phenomenon. The rate at which a phenomenon is formed is exponentially distributed with an average interarrival of 10% of the total experiment time to model the appearance of a phenomenon. To model the time span of a phenomenon, the duration of each phenomenon is another number that randomly ranges from 0% to 10% of the total experiment time. To model phenomenon movement, each time unit, a random action is selected among *shrink*, *grow*, *move*, and *do nothing*. The direction and the size of each action are decided such that the direction is randomly chosen as one of the eight basic directions (E, W, N, S, NE, NW, SE, and SW) and the increase/decrease in size is randomly chosen with a limit of 10% of the phenomenon size. Figure 2.3 illustrates a snapshot of the visualization tool for simulated setup.

2.6 Related Work

The research focus of the data streaming area has been directed to processing continuous queries over data streams that are generated by mobile objects, e.g., [22–26] and to track these objects as they roam the surrounding space. In our research,

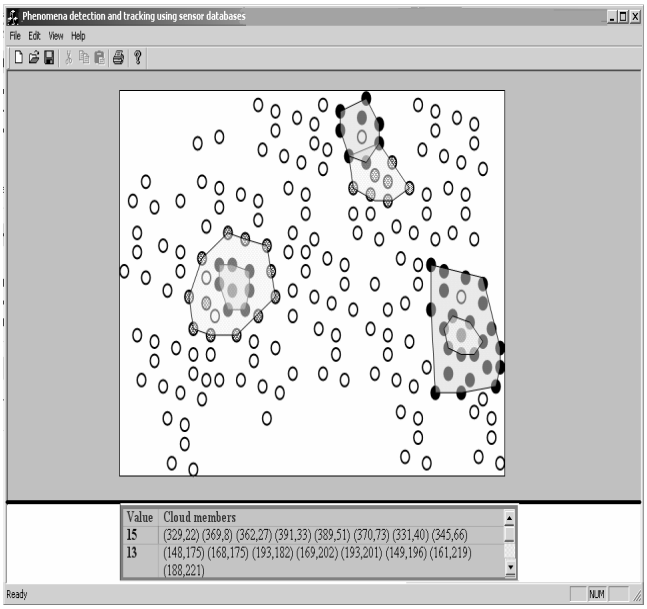


Figure 2.3. Snapshot of the visualization tool for the Nile PhenomenaBase simulated setup

we provide a framework to detect and track phenomena that span a bundle of data streams inside a DSMS instead of tracking a single object. Examples of tracking moving objects include the *Wjoin* operator, a window join operator over multiple streaming sources. [2] employs the Wjoin operator for the purpose of tracking moving objects in a sensor field. To optimize for the object tracking process, [27] reconfigures the tree-like communication structure of a sensor network dynamically. As sensors get closer to the moving object, they are moved towards the root of the communication tree to reduce the number of hops between the sensor and the sink. A prediction-based strategy is also proposed in [28] to reduce the power consumption of the sensor network by focusing on regions where moving objects are likely to appear.

The concept of a phenomenon has been addressed in the literature, yet, under different terminologies, e.g., homogenous regions, isobars, and moving clusters. Detection of boundaries that separate *homogeneous regions* of sensors is investigated in [29]. In [30], streams of sensor data that have approximately the same value are

grouped into continuous regions called *isobars*. Given a database of object trajectories, [31] refers to a set of objects that move close to each other over a period of time as a *moving cluster*. Our definition of a phenomenon enriches the concept of a phenomenon by spatiotemporal constraints through the persistency, the spread, and the time span parameters. Also, our definition of a phenomenon encompasses several notions of similarity among the phenomenon underlying data streams.

From a system's point of view, Nile PhenomenaBase is a phenomenon-based query processing system. Similarly, the *Complex Event Detection and Response* (*CEDR*) system [32] is an event-based system. The concept of event shares similarity with the concept of phenomenon and is defined to be a sequence of stream tuples with spatial and temporal constraints. CEDR proposes an event detection language that has a *where* clause and a *when* clause to express the spatial and the temporal constraints of an event, respectively. The focus of CEDR is to detect complex events, i.e., events that are composed of other events, and to provide consistency guarantees on the output. Consistency guarantees trade the stability of the output versus the output delay. Several differences distinguish between Nile PhenomenaBase and CEDR. The CEDR event does not have the notion of persistency in the event definition. Also, CEDR events are stationary. Once an event is detected there is no notion of event tracking. While CEDR focuses on consistency guarantees, Nile PhenomenaBase works on a best-effort basis and does not provide consistency guarantees on the output. On the other hand, the focus of Nile PhenomenaBase goes beyond phenomenon detection to the level of phenomenon-guided query optimization.

2.7 Summary

In this chapter, we provided a formal definition for the phenomenon. Three parameters control the phenomenon definition: the persistency α, the spread β, and the time span ω. Also, various similarity notions among streams' behaviors affect the way a phenomenon is detected. Examples of these similarity notions include

equality, distance similarity, summary similarity, and trend similarity. In addition, we presented several models that capture the overall phenomenon representative behavior.

This chapter introduced the extended SQL syntax of the Nile PhenomenaBase prototype system, its architecture, and its experimental setups. To support phenomenon awareness inside Nile, various components of the system are modified, e.g., the parser, the load shedder, the query plan generator, and the query executer. Moreover, a PDT-module and a phenomenon-aware optimizer are added and are connected to existing components of the system. Nile PhenomenaBase adopts the principle of phenomenon-aware query processing and utilizes the detected phenomena to guide the execution of subsequent user queries. Nile PhenomenaBase has both a real small-scale sensor board and a simulated large-scale data streaming setup as its experimental testbeds.

3 DETECTION AND TRACKING OF DISCRETE PHENOMENA

This chapter introduces a framework for Phenomena Detection and Tracking (PDT, for short) in DSMSs. We focus on discrete phenomena where stream readings are drawn from a discrete set of values, e.g., item numbers or pollutant IDs. Our proposed PDT framework accepts a phenomenon definition and converts that definition to a continuous SQL query (called a PDT query). PDT queries are continuously executed by the query executer to detect and track phenomena as they appear in the environment. The challenge for the proposed PDT framework is to detect as much phenomena as possible, given the large number of data streams, the overall high arrival rates of stream data, and the limited system resources. Experimental studies illustrate the scalability challenges that face PDT techniques and provide room for further optimizations that will be addressed in the following chapters.

3.1 Background and Motivation

As illustrated in Figure 2.1, the PDT-module takes a phenomenon definition from the user as its input. Then, the PDT-module takes the responsibility of processing the phenomenon definition to detect the phenomena that satisfy the defined phenomenon pattern. The PDT-module converts the phenomenon definition into a continuous query (called PDT query) and dispatches this query to the query executer. The execution of the PDT query detects phenomena in the incoming data streams and tracks the propagation of these phenomena once the PDT query is dispatched to the query executer. In this chapter, we present a typical phenomenon definition along with its corresponding PDT query. Then, we propose a framework for the execution of the PDT query. For simplicity, we consider only the equality

notion of similarity among streams' behavior that results in discrete phenomena and, consequently, the join operation reduces to an equality join.

In general, the proposed *phenomena detection* and *tracking* (*PDT*) framework has three phases: the *grouping* phase, the *joining* phase, and the *output* phase. The *grouping* phase takes the raw data from the streaming sources as its input and groups each stream readings based on value. The number of tuples in a group represents the stream's persistency ($\hat{\alpha}$) for a specific value. Persistent stream values (i.e., $\hat{\alpha} \geq \alpha$) are reported in the output of the grouping phase and are processed by the joining phase. The *joining* phase takes the persistent stream readings as its input and produces as output a set of join tuples with data stream IDs that have similar values. The *joining* phase is a multi-way join operation that spans the data streams that are monitored by the create-phenomenon statement. The output of the join operator is tested against the desired spread to ensure that the actual number of joining streams ($\hat{\beta}$) is above the spread threshold (i.e., $\hat{\beta} \geq \beta$). Join output tuples that satisfy the desired spread are denoted as phenomenon candidates and are passed to the output phase. Finally, the *output* phase constructs the overall phenomenon from the candidate join tuples that are produced by the joining phase. To avoid the infiniteness of incoming data streams, all the operations are based on window operators such that the window size is equivalent to the time span threshold that is specified in the phenomenon definition (ω).

The contributions of this chapter can be summarized as follows:

1. We express the phenomenon definition as a continuous SQL query that is executed by the query processor to detect phenomena and to continuously track their propagation.

2. We propose a three-phase framework for *phenomena detection* and *tracking* that adheres to the phenomenon definition.

3. We investigate the scalability challenges that face the proposed framework and provide, based on a prototype implementation inside Nile PhenomenaBase, an

experimental study that illustrates the performance of the *PDT* framework with multi-way joins.

The remainder of this chapter is organized as follows: The PDT continuous queries that initiate the processing of the PDT framework are presented in Section 3.2. Section 3.3 introduces the three phases of the proposed PDT framework. Section 3.4 discusses the scalability challenges that face the proposed PDT framework and illustrates these scalability challenges by an experimental evidence. Finally, Section 3.5 summarizes the chapter.

3.2 PDT SQL-Queries

In this section, we discuss how the PDT-module converts a phenomenon definition (as shown in Q 3.2.1) into a continuous query (i.e., a PDT query) that is responsible for the detection and tracking of phenomena as they develop in the field. The PDT query is given in Q 3.2.2 and its corresponding query plan is shown in Figure 3.1. A *window* operator is inserted at the bottom of the query plan to limit the query processor's attention to the most recent time window of size ω. The window operator promotes the concept of the positive and negative tuples [33] that is adopted by the Nile system. The window operator passes an incoming input tuple up the query plan as a positive tuple. After the elapse of ω time units, the window operator generates a corresponding negative tuple and propagates it up the query plan to denote its expiration and to undo its effect from the output. Performing the query processing over a window of size ω limits the phenomenon actual time span to $\hat{\omega}$ such that $\hat{\omega} \leq \omega$.

Q 3.2.1 *CREATE PHENOMENON <phenomenon-name>*
ON STREAM BUNDLE <stream-bundle-name>
PATTERN SB[i] = SB[j]
PERSISTENCY α
SPREAD β

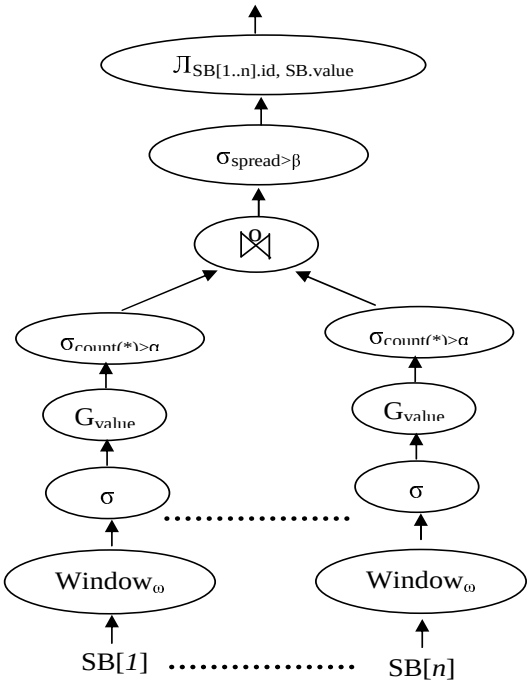


Figure 3.1. PDT query plan

TIME SPAN ω

WHERE <other conditions>;

Q 3.2.2 *SELECT SB[1..n].id, SB.value*

FROM OUTER JOIN

(

SELECT SB[i].id, SB[i].value

GROUP BY (SB[i].id, SB[i].value)

HAVING COUNT(*)$\geq \alpha$

) ON SB.value

WHERE Spread() $\geq \beta$ *AND <other conditions>*

WINDOW ω*;*

To enforce the persistency constraint, the PDT query groups each stream's readings using a group-by operator based on the readings' value. The count of each

group members, i.e., the number of times a data stream persists to generate a value, is recorded as the stream's persistency for that value. Stream readings with persistency that is less than α are filtered out.

The *pattern* clause in the phenomenon definition is transformed into an *outer multi-way window* join. The join operation detects similarity in behavior among data streams. The join operation is multi-way because there are multiple streaming sources co-existing in the streaming environment. Also, the join operation is an outer join because not all the streaming sources are expected to participate in the same phenomenon. Only subsets of the data streams are expected to join. Hence, the outer join detects similarity among the streaming sources and replaces non-joining data streams with the NULL values. To enforce the spread constraint, the number of non-NULL joining streams is evaluated using the Spread function and is tested against the spread constraint in the *where* clause. The *where* clause may contain any other constraints that appear in the phenomenon definition as well. The *where* clause in the phenomenon definition is preserved in the PDT query to filter out any stream tuples that do not satisfy the where clause predicates. Finally, the IDs of the data streams that contribute to the same phenomenon and the phenomenon value are reported by the select clause in the PDT query. Example 3.2.1 gives an example phenomenon definition that is interested in heat zones while Example 3.2.2 give sits corresponding PDT query.

Example 3.2.1 *CREATE PHENOMENON HeatZones*
ON STREAM BUNDLE SB
PATTERN $SB[i].Temperature = SB[j].Temperature$
PERSISTENCY 5
SPREAD 4
TIME SPAN 60
WHERE $SB.Temperature > 90$*;*

Example 3.2.2 *SELECT SB[1..n].id, SB.value*

FROM OUTER JOIN

(

SELECT SB[i].id, SB[i].value

GROUP BY (SB[i].id, SB[i].value)

HAVING COUNT(*)≥ 5

) ON SB.value

WHERE Spread$() \geq 4$ *AND* $SB.Temperature > 90$

WINDOW 60*;*

3.3 PDT Query Processing

The process of *phenomena detection* and *tracking* is initiated by issuing the PDT SQL-query as discussed in Section 3.2. *PDT* query processing is divided into three phases as illustrated in Figure 3.2. The main idea of the first phase (the grouping phase) is to feed the incoming stream tuples to a *group-by* operator that takes into consideration the *persistency* of the desired phenomenon. Then, we join persistent readings from various streaming sources using a *multi-way* join algorithm over data streams. The join output tuples are then checked for the spread constraint. If at least β data streams join together on the same value, an output is generated to denote a phenomenon candidate. Based on the output of the joining operation, the overall phenomena are constructed in the third phase and are reported to the PDT module. The third phase (the output phase) investigates the application semantics to form and report the phenomena to the user. The output phase is a good place for further phenomenon analysis and for post phenomenon-detection checks. The continuous execution of the group-by and the join operators *tracks* the movement of phenomena in the space. The remainder of this section discusses the phases the PDT query execution.

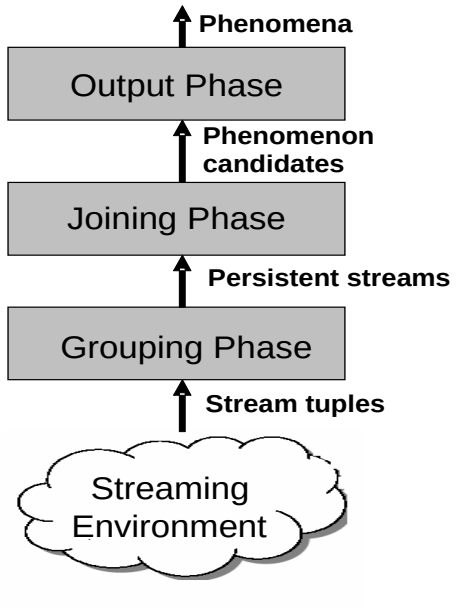


Figure 3.2. PDT query processing phases

3.3.1 Phase I: The Grouping Phase

The grouping phase employs a standard group-by operator to group the readings that are coming from the same stream based on the reading value. Therefore, the stream input tuples are grouped by (SB[i].id, SB[i].value). Within each group, the persistency of each reading value is measured as the count of the group members. Persistent readings are the readings with actual persistency $\hat{\alpha}$ that is greater than or equal to the persistency threshold (α).

The *group-by* operation and its corresponding *having clause* utilize the notion of positive and negative tuples [33]. The group membership and the count of each group are updated according to the arrival of tuples (positive tuples) and to the expiration of existing tuples (negative tuples). Hence, a stream reading may alternate between persistency and impersistency over time.

The output of the grouping phase is a stream of tuples of the form $\pm$(SB.id, SB.value). The output is of the form +(SB.id, SB.value) if stream number SB.id generates the value SB.value at least α times. Also, the output is of the form $-$(SB.id,

SB.value) if the number of occurrences of SB.value from stream number SB.id drops below α. The output of the grouping phase is pipelined into the joining phase.

3.3.2 Phase II: The Joining Phase

In the joining phase, we perform the multi-way join using an MJoin operator [34]. The main idea of MJoin is to maintain a hash table for each stream. Once a tuple arrives from one stream, it is inserted into the stream's corresponding hash table. Then, the incoming tuple probes the hash tables of the other streams. The original version of MJoin performs an inner multi-way join. Since an inner join output tuple is reported only if it appears in *ALL* streams, the MJoin algorithm stops probing hashing tables once the probed value is missing in one of the streams.

To employ MJoin inside PDT techniques, we make two modifications to the original MJoin algorithm [34]. These modifications are summarized as follows: First, we modify MJoin to perform an *outer* multi-way join. The modified MJoin algorithm does not stop if the join value is missing in one of the streams. Instead, it assumes a NULL value for the missing stream and continues to examine the remaining streams to produce partial results.

Second, we modify MJoin to process both the positive and the negative tuples that are coming from the grouping phase and to generate positive and negative tuples accordingly in the output. A positive tuple is reported when a join occurs. A negative tuple is reported when one of the previously-reported join tuple components expires, i.e., becomes old enough to get outside of the most recent time-window w. The negative tuples are important to invalidate the phenomenon, if the streaming sources stop showing the same behavior over a time-window w.

3.3.3 Phase III: The Output Phase

The *output* phase receives phenomenon candidates on the form of tuples that consist of the IDs of the joining streams and the join value. Each tuple can be positive

or negative to denote the appearance or the disappearance of a candidate member. The *output* phase has the flexibility to apply application-dependent semantics in the phenomenon detection process. For example, the output phase can determine the outer contour or the convex hull of a phenomenon as a first step towards the shape understanding of the phenomenon. Also, the application may consider the spatial connectedness of the streaming sources. For example, multiple disconnected oil spills may imply leakage out of more than one container. Upon receiving a positive tuple, based on the application semantics, the *output* phase may start a new phenomenon, add one streaming source to an existing phenomenon, or merge two phenomena together if they become connected. Similarly, upon receiving a negative tuple, the *output* phase may delete a phenomenon, remove a streaming source from an existing phenomenon, or split one phenomenon into two separate phenomena if they got disconnected from each other.

Application-dependent semantics can include the density of the phenomenon as well. The density is measured by the ratio of the number of streaming sources reporting the same phenomenon to the number of streaming sources not reporting that phenomenon in a specified region. All these issues are application-dependent and are addressed by the *output* phase through user-defined functions in the query's select clause.

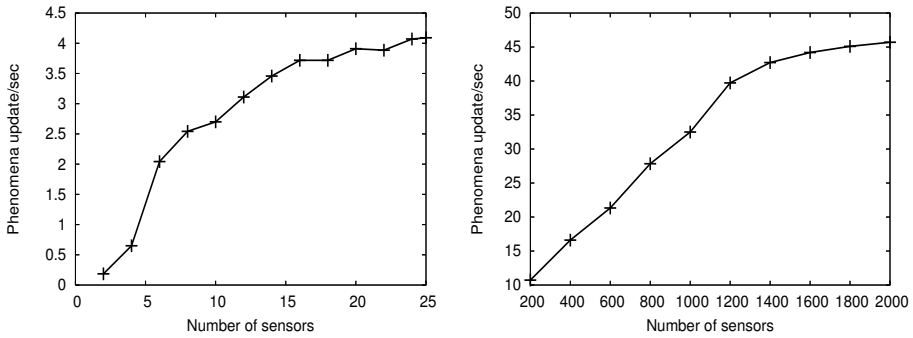
3.4 Scalability Challenges

In Section 3.3, we presented the SQL PDT continuous query that semantically tracks the required phenomenon pattern. However, the execution of the PDT query faces several scalability challenges. PDT query processing involves a join operation among multiple streaming sources. With the increase in the number of streaming sources and with the increase in the sources' streaming rate, an increase in the joining cost is inevitable. Query optimization techniques that move the selection and the group-by operators around the join (e.g., [35, 36]) reduce the cost of the

join operation. However, in this dissertation, we address the optimization of the join operator itself taking into consideration the phenomenon properties. In the remainder of this section, we give an experimental sense of the scalability challenges in the processing of PDT queries. We dedicate the following two chapters to enhance the performance of the joining phase of the PDT framework.

Throughout the chapters of this dissertation, we compare the performance of the PDT frame using MJoin against the proposed enhancements in terms of several performance measures. Examples of these performance measures are the *output delay*, the *input drop rate*, and the *output drop rate*. The output delay is the time difference between the arrival of a tuple and the time its effect appears in the output. The input drop rate is the number of tuples that are dropped out of the system's input buffer due to the system's limited resources that are incapable of accepting the continuous arrival of stream data. The output drop rate is the number of missed output tuples due to dropping some of the input tuples. In this section, we measure the overall system performance in terms of the number of *detected phenomena updates per second*. A phenomenon 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 tracks phenomena as they move in space. We base our study on the two experimental setups of Nile PhenomenaBase that are described in Section 2.5; a real small-scale sensor board and a simulated large-scale sensor network. The α, β, and ω parameters are set to be 30, 3, and 10, respectively, for the real setup while they are set to be 3, 30, and 10, respectively, for the simulated setup.

Figure 3.3 gives the number of detected phenomenon updates as we increase the number of sensors. Notice that the number of detected phenomenon updates gets saturated with the increase in the number of sensors. We refer this saturation to the increase in the number of dropped input tuples due to the incapability of the system to cope with the increase in the input load. For example, the number of dropped input tuples jumps from 10% of the total input tuples for 10 sensors to 30% with



(a) Small-scale sensor board (b) Simulated large-scale setup

Figure 3.3. Number of detected phenomenon updates

the increase in the number of sensors to 25 (for the real small-scale sensor board). Further analyses, comparisons, and experimental studies for the PDT performance are presented in the following chapters.

3.5 Summary

In this chapter, we proposed a framework for *phenomena detection* and *tracking* (*PDT, for short*) in DSMSs. To detect a phenomenon, we convert the phenomenon definition to a continuous SQL query, termed a PDT query. The framework to execute PDT queries encompasses three phases: the *grouping* phase, the *joining* phase, and the *output* phase. The *grouping* phase takes the raw data from the data streams and applies a filter to enforce the persistency constraint over the time span of the phenomenon. The *joining* phase employs a multi-way join algorithm that joins the output tuples of the grouping phase and produces a set of join tuples with similar values. Finally, the *output* phase is an application-specific phase where the application semantics are enforced. An experimental study based on a prototype implementation inside Nile PhenomenaBase shows that the proposed *PDT* technique with multi-way joins is *not* scalable in terms of the number of streams, the stream rates, and the number of detected phenomena. Hence, several enhancements and

optimizations need to be carried over to get a better performance. The following chapters of this dissertation address these enhancements and optimizations.

4 PREFERENCE-BASED LOAD SHEDDING IN PHENOMENON-AWARE DSMSS

The joining phase of the PDT query processing employs a multi-way join operation over a large number of, possibly high-rate, data streams. Hence, we need to take several measures to address the scalability problems that appear under periods of heavy system loads. In this chapter, we introduce the concept of preference-based load shedding to reduce the input load that is streamed into the DSMS. Preference-based load shedding tunes query processing towards tuples that satisfy a specific preference or a desired feature of a phenomenon definition. More specifically, we focus on three preference types that are based on the phenomenon persistency, the phenomenon spread, and the phenomenon time span. Experiments prove that the concept of preference-based load shedding inside Nile PhenomenaBase guides the PDT query processing towards phenomena of interest.

4.1 Background and Motivation

Some application domains favor persistent phenomena that are characterized by a highly-frequent appearance of its participating tuples while other application domains are interested in intermittent phenomena that feature low-frequency tuples. Similarly, some application domains favor stretch phenomena that are widely spread in the monitored environment while other application domains are interested in confined phenomena that are localized in narrow regions of the field. Also, some application domains favor durable phenomena that span a long period of time while other application domains are interested in impulse phenomena that occur over few time instants. Other preferences can be defined based on the requirements of the application domain. In this chapter, we elaborate on the *with-preference* clause of

the phenomenon definition as presented in Q 2.2.6 in Chapter 2. The *with-preference* clause in the phenomenon definition addresses the scalability problems of PDT techniques that appear during periods of heavy system load. As discussed in Section 3.4, the joining phase is based on a costly multi-way join operation among the large number of data streams in the environment. The *with-preference* clause acts as a prioritization mechanism for the incoming stream tuples. Based on the tuple's assigned priorities, tuples are ordered and are processed based on an ascending/descending order of their anticipated contribution to the overall phenomenon preference. Hence, in case of heavy-load periods, low-priority tuples are dropped from the query processor's buffers to reduce the overall system's load.

In general, preference-based load shedding can be based on any user-defined preference as long as a corresponding prioritization function is defined. However, in this chapter, we focus on three types of preferences that are expressed by the syntax given in Q 2.2.6. These three preference types are based on the parameters that control the phenomenon definition: the persistency, the spread, and the time span. Assume that $\hat{\alpha}$, $\hat{\beta}$, and $\hat{\omega}$ are the phenomenon actual persistency, spread, and time span. Based on Definition 2.1.1, the phenomenon actual persistency ($\hat{\alpha}$) has to exceed the persistency threshold (α), the phenomenon actual spread ($\hat{\beta}$) has to exceed the spread threshold (β), and the phenomenon actual time span ($\hat{\omega}$) has to be within the time span threshold (ω). However, with the increase in the system's load, we have the option to tune query processing towards highly-persistent phenomena (with large values of $\hat{\alpha}$) or towards intermittent phenomena (with small values of $\hat{\alpha}$). For example, an application may be interested in items that are *excessively* sold in multiple branches of a retail store while another application is interested in items that are sold in multiple stores with a quantity that is just above the persistency threshold α. Alternatively, we have the option to tune query processing towards stretch phenomena (with large values of $\hat{\beta}$) or towards confined phenomena (with small values of $\hat{\beta}$). For example, an application may be interested in oil spills that span large portions of the ocean surface while another application is interested in

small localized oil spills with a spread that is just above the spread threshold β. Also, we have the option to tune query processing towards durable phenomena (with large values of $\hat{\omega}$) or towards impulse phenomena (with small values of $\hat{\omega}$). For example, an application may be interested in regions that have been on fire for a long period of time (i.e., close to the time span threshold ω) while another application is interested in sparks with short duration that may be a cause for future fires.

The main idea of preference-based load shedding is to maintain stream summaries that approximate the intrinsic features of the stream's tuples. These summaries are maintained and are updated with the arrival of each stream tuple. Therefore, the cost of updating the summaries is required to be small relative to the actual query processing cost. Based on the global picture that is provided by the stream summaries, the load shedder decides on how relevant the tuple is to the phenomenon desired preference. The decision of the load shedder takes the form of a priority that is attached to each tuple in the stream. Tuples are inserted in a priority queue of limited size. The query processor proceeds with high-priority tuples from the head of the queue. Meanwhile, low-priority tuples are deleted from the tail of the queue in response to the insertion of high-priority tuples.

The contributions of this chapter can be summarized as follows:

1. We identify a general framework for preference-based load shedding and we emphasize on three preference specifiers that are of special interest to application domains.

2. We introduce two system components; the summary manager and the load shedder, that collaborate together to drop stream tuples not likely to match the phenomenon preference.

3. We provide an experimental study that is based on a prototype implementation of the summary manager and the load shedder inside Nile PhenomenaBase to explore the impact of load shedding on phenomenon detection.

The remainder of this chapter is organized as follows: Section 4.2 revisits the system architecture and focus on system components that are relevant to load shedding. Section 4.3 introduces the proposed summary manager and the proposed load shedder. Section 4.4 presents an experimental study of the proposed load shedding technique. Section 4.5 overviews related work. Finally, Section 4.6 summarizes the chapter.

4.2 System Architecture

As illustrated in Figure 2.1, the *stream monitor* receives the incoming data stream tuples into its own buffers. Then, the stream monitor forwards the tuples to the query executer for further processing. Also, the PDT-module receives a phenomenon definition that possibly includes a "with-preference" clause. The stream monitor is the perfect place to accommodate the load shedding components because it has access to the incoming stream tuples before they are pushed into the query plan buffers of the query executer. The PDT-module provides the stream monitor with the phenomenon preference to request the stream monitor and its load shedding component to favor tuples that are likely to satisfy the user's preference.

Figure 4.1 focuses on the architecture of the stream monitor and its connection to other system components. The stream monitor has three basic components: the buffer manager, the summary manager, and the load shedder. The buffer manager is the first point of contact between the DSMS and the streaming sources in the surrounding environment. Streaming sources push their stream readings into the input buffers of the buffer manager. The input buffers are circular queues of limited sizes. Therefore, if the incoming tuples are not consumed in an equal (or a higher rate) to their arrival rate, tuples are dropped out of the system's input buffers and are lost forever. This type of tuple dropping is called *random load shedding* [37–40] because tuple dropping occurs without taking into consideration the tuple values

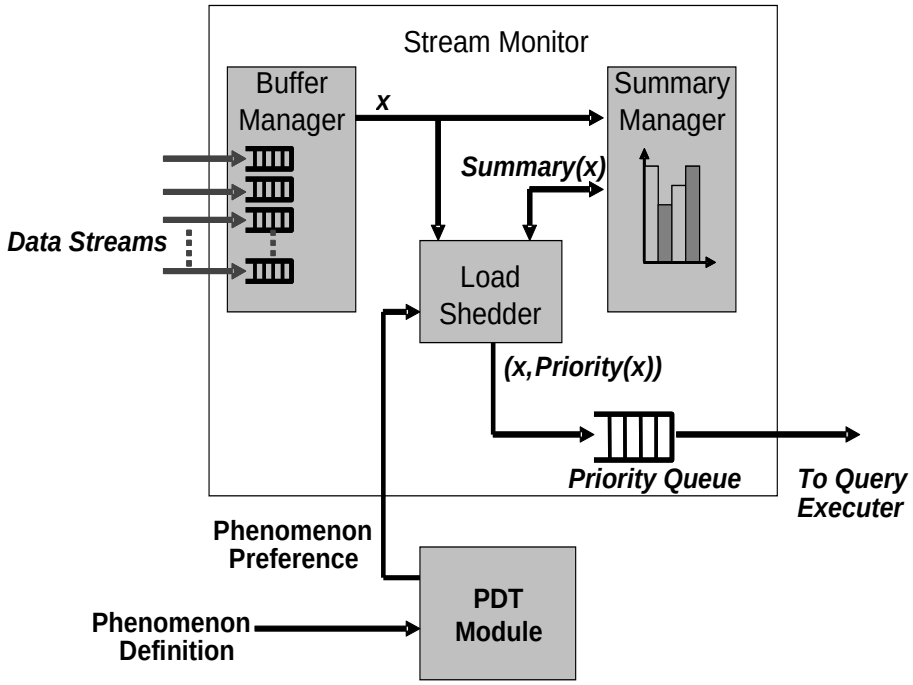


Figure 4.1. The architecture of the stream monitor.

and their semantics. In our work, we are interested in *semantic load shedding* [41–43] where tuple dropping occurs according to the semantics of the incoming tuples.

The summary manager maintains stream summaries over the incoming stream tuples that reflect the intrinsic properties of these tuples. With the arrival of each incoming stream tuple, we invest a small portion of the system's budget of time to update the stream summaries. Then, the stream tuple is sent to the load shedder.

The load shedder takes a phenomenon preference from the PDT-module and receives the incoming stream tuples from the buffer manager. The load shedder investigates the summary manager for each incoming stream tuple to assign a priority to that tuple. The tuple's assigned priority reflects the tuple's expected contribution to the phenomenon preference. Tuples are then inserted in a priority queue and are processed by the query executer in a descending order of their priorities.

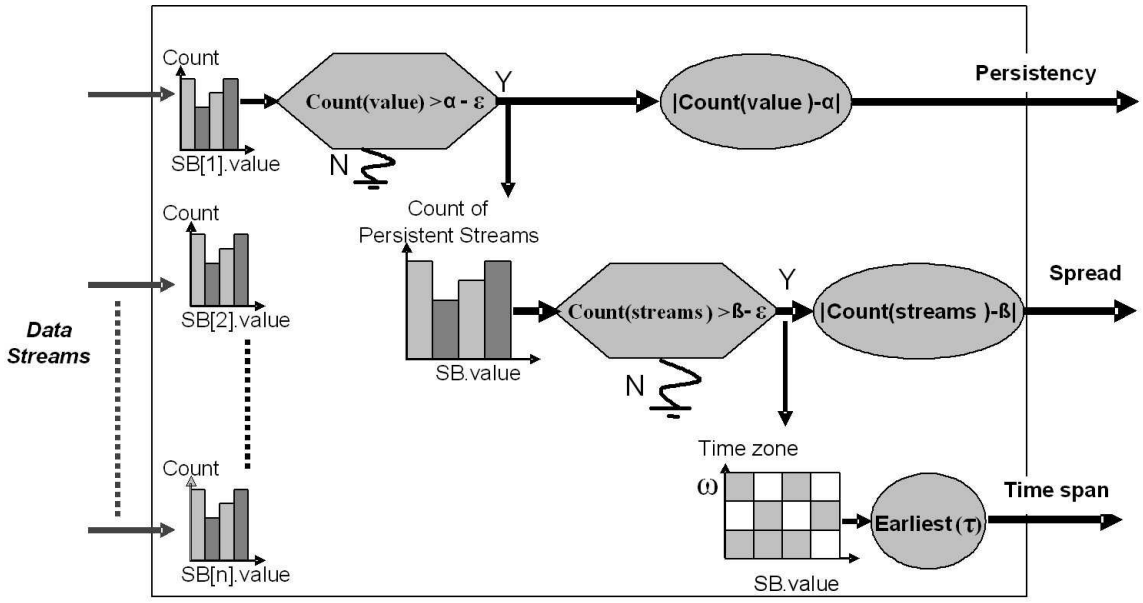


Figure 4.2. Data flow in the stream summarization and load shedding process.

4.3 Stream Summarization and Load Shedding

Stream summaries are essential to provide a rough but quick estimation for the expected gain of an incoming tuple. Various summarization techniques have been proposed in the literature, e.g., [19, 44–46]. Different DSMSs employ different summarization techniques. In this Section, we do not propose a new summarization technique. Instead, we propose a general framework that utilizes existing summarization techniques to help the load shedder identify and drop non-interesting tuples. In our work, the summarization technique is considered a black box as long as it provides the following set of interface functions:

1. *Summarize(x)*: Inserts a tuple x and adds the effect of its value to the existing summaries.

2. *Estimate(x)*: Estimates various properties of a stream tuple x from the summaries. These estimations are used in the context of the query preference function to specify how promising tuples are extracted.

3. *Confidence()*: Reports how much confident we are about the stream summaries. This function is used as a measure of accuracy of the properties that are estimated from the summaries.

There is a trade-off between the accuracy of the summary-estimated properties and the cost of updating the summaries. As the accuracy of the summaries increases, non-interesting tuples can be accurately identified and dropped out of the system. However, an increase in the maintenance cost of summaries implies a reduction in the time budget of the query processor. Detailed discussion on load shedding techniques is presented extensively in the literature, e.g., [37–43]. In our framework, we emphasize on the phenomenon semantics in the load shedding process. Therefore, we use simple summarization techniques, e.g., histograms and countsketches, that approximate the phenomenon actual persistency, spread, and time span.

Figure 4.2 illustrates the data flow of the proposed load shedding process. We maintain a summarization structure per data stream that is capable of estimating the count of a given value in the stream readings over the most-recent time window of size ω, e.g., histograms [44] or countsketches [19]. An incoming tuple updates the summarization structure of its own data stream. Then, the count of this tuple value is estimated from the summarization structure to decide on the persistency of this tuple. If the count of the tuple's value is above α-ϵ, the tuple is declared to be persistent. ϵ is an error factor that places a bound on the inaccuracy in estimating the count of the tuple's value and is returned by the $Confidence$ function. ϵ gives the tuple the benefit of doubt and avoids dropping persistent tuples due to summarization inaccuracies. The tuple's assigned priority of persistency is estimated as the absolute difference between the count of the tuple's value and the persistency thresh-

old ($Count(value) - \alpha$). The tuple's assigned priority is sorted either descending or descending to favor persistent phenomena or intermittent phenomena, respectively.

If the tuple is impersistent, it is dropped out of the system. However, if the tuple is impersistent, it is assigned a persistency priority. Moreover, for each incoming stream value, regardless of its issuing stream, we maintain a summarization structure that reports the number of data streams that persist to generate this value. If an incoming stream reading causes the stream to be persistent in this value, i.e., the number of occurrences of this value in that stream jumps from $\alpha - 1$ to α, the count of persistent streams is incremented by one. On the contrary, if a stream reading expires and causes the stream to be impersistent in this value, i.e., the number of occurrences of this value in that stream jumps from α to $\alpha - 1$, the count of persistent streams is decremented by one. If the number of persistent streams for a specific value exceeds β-ϵ, the spread constraint is likely to be satisfied for this value. The tuple's assigned priority of spread is estimated as the absolute difference between the count of persistent streams and the spread threshold ($Count(streams) - \beta$). The tuple's assigned priority is sorted either descending or descending to favor stretch phenomena or confined phenomena, respectively.

To estimate the time span of a phenomenon, we need a summarization structure that approximates, for each value (or group of values), the time stamps of the occurrences of this value. For each value (or group of values), we divide the timeline for the most recent tine window of size ω into n time zones such that each time zone is of size $\frac{\omega}{n}$ time units. Each time zone contains an on/off flag. Initially, all flags are set to off. With the arrival of a stream reading, the corresponding on/off flag of the reading value at the proper time zone is set to on. The timeline for the summarization structure slides every $\frac{\omega}{n}$ time units to remove the summaries of expired tuples and to build summaries for the new tuples. The earliest time zone of a specific value with an on flag reflects the time span of this value. The tuple's time span is sorted either descending or descending to favor durable phenomena or impulse phenomena, respectively.

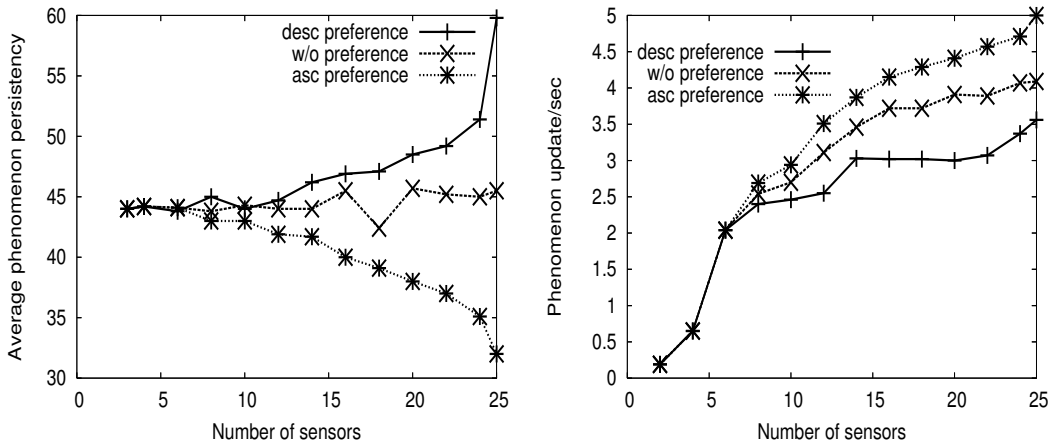
4.4 Experiments

In this Section, we investigate the effect of the proposed load shedding on the performance of the phenomenon detection and tracking process. We present the results of three sets of experiments that explore the persistency, spread, and time span preferences, respectively. In each set of experiments, we are interested in two measures of performance. The first measure is the average actual persistency, spread, and time span of the detected phenomena in response to changing the persistency, spread, and time span preferences, respectively. The second measure of performance is the number of detected phenomenon updates per second. A phenomenon update is reported if a phenomenon appears, disappears, or changes its location. The number of detected phenomenon updates per second reflects how fast the system is in tracking phenomena as they move in space.

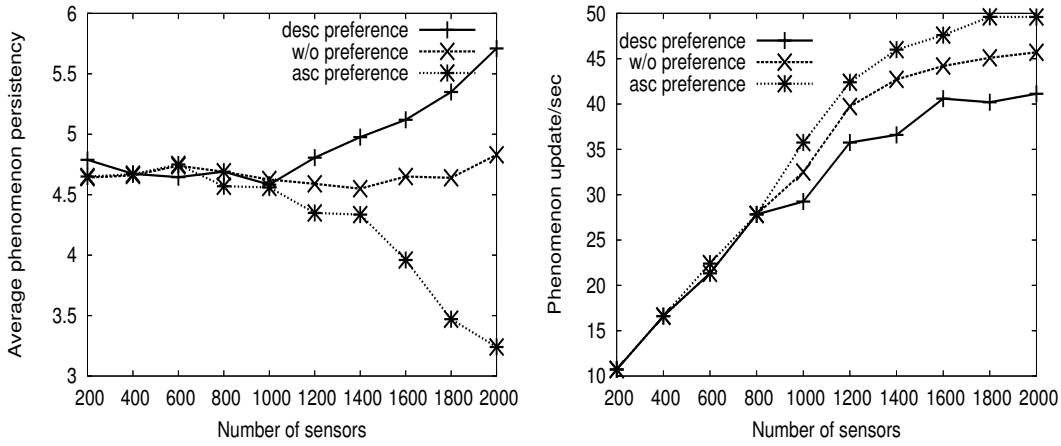
We compare the performance of the PDT module in three cases: (1) the case where load shedding is turned off, (2) the case of an ascending order of the preference, and (3) the case of a descending order of the preference. All the experiments in this section are based on a prototype implementation of the stream monitor and its components inside Nile PhenomenaBase. We base our study on the two experimental setups of Nile PhenomenaBase that are described in Section 2.5: a real small-scale sensor board and a simulated large-scale sensor network. The size of each stream's input buffer in the buffer manager and the size of the priority queue are set to 8 tuples. The α, β, and ω parameters are set to be 30, 3, and 10, respectively, for the real setup while they are set to be 3, 30, and 10, respectively, for the simulated setup. The Nile PhenomenaBase engine executes on a machine with Intel Pentium IV, CPU 2.4GHZ and 512MB RAM running Windows XP.

4.4.1 The Persistency Preference

Figure 4.3 illustrates the effect of load shedding with an ascending preference in persistency, without any preference in persistency, and with a descending preference



(a) Real small-scale setup



(b) Simulated large-scale setup

Figure 4.3. The effect of the persistency preference.

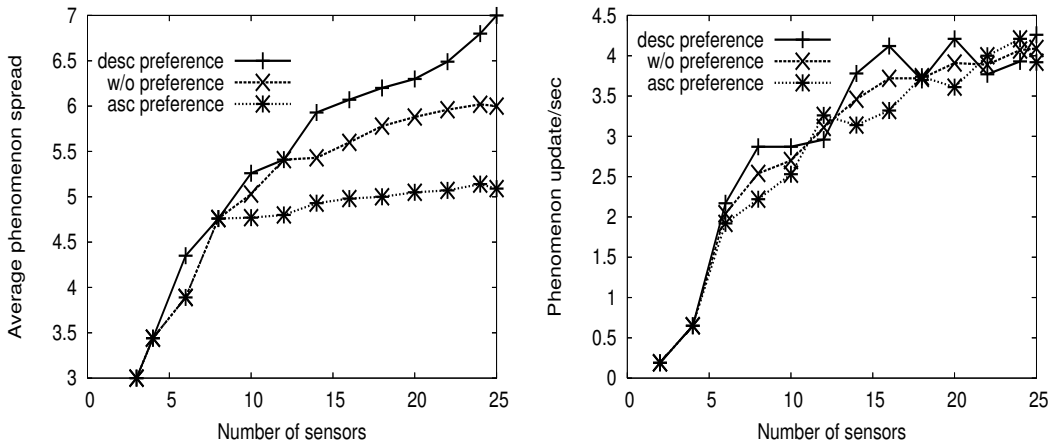
in persistency under various system loads. Figure 4.3a utilizes the real small-scale sensor board while Figure 4.3b utilizes the simulated large-scale setup. Both figures exhibit a similar trend. For a small system load (expressed in terms of the number of incoming data streams), almost every phenomenon is detected and, hence, the average persistency is almost the same for all of the three depicted cases. However, as we increase the number of data streams, tuples are dropped out of the system. PDT with a descending persistency preference favors persistent tuples and generates highly-persistent phenomena in the output. On the other hand, PDT with an ascend-

ing persistency preference favors less persistent tuples and generates less-persistent phenomena in the output. The deviation between the three PDT instances becomes apparent at the size of 25 sensors for the real sensor board and at the size of 2000 sensors for the simulated setup at Figure 4.3.

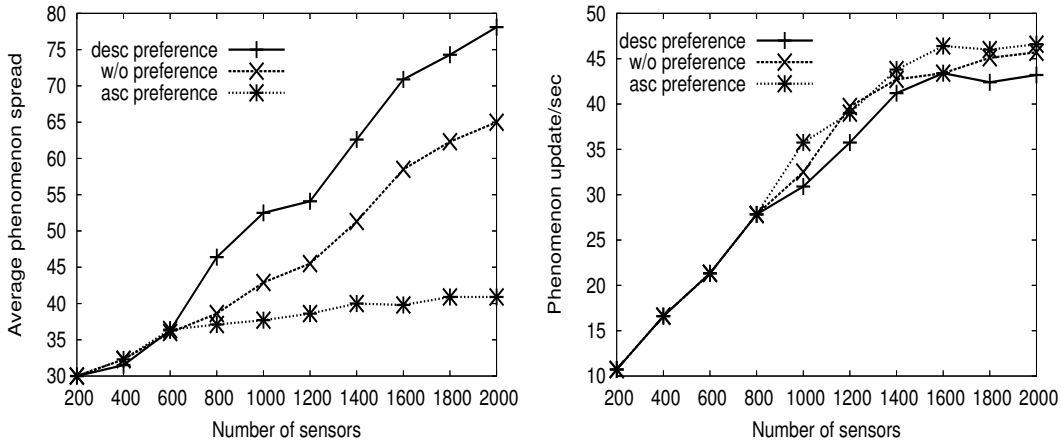
Notice that the number of tuples that contribute to the formation of a persistent phenomenon is larger than the number of tuples that contribute to the formation of an impersistent phenomenon. A persistent phenomenon update is not reported by the system unless the persistency constraint is met by the incoming tuples. Consequently, the total number of detected phenomenon updates per second decreases under a descending persistency preference compared to its corresponding ascending preference.

4.4.2 The Spread Preference

Figure 4.4 illustrates the effect of load shedding with an ascending preference in spread, without any preference in spread, and with an descending preference in spread under various system loads. Figure 4.4a utilizes the real small-scale sensor board while Figure 4.4b utilizes the simulated large-scale setup. For a small system load (expressed in terms of the number of incoming data streams), almost every phenomenon is detected and, hence, the average spread is almost the same for all of the three depicted cases. However, as we increase the number of data streams, tuples are dropped out of the system. PDT with a descending spread preference favors tuples that appear in multiple streams and generates stretch phenomena in the output. On the other hand, PDT with an ascending spread preference favors tuples that appear in fewer streams and generates confined phenomena in the output. The average spread of a phenomenon increases in all of the three depicted curves with the increase in the number of monitored data streams because more data streams tend to generate similar behaviors. However, the rate of increase deviates from one curve to another based on the specified preference as we increase the system load. Notice that



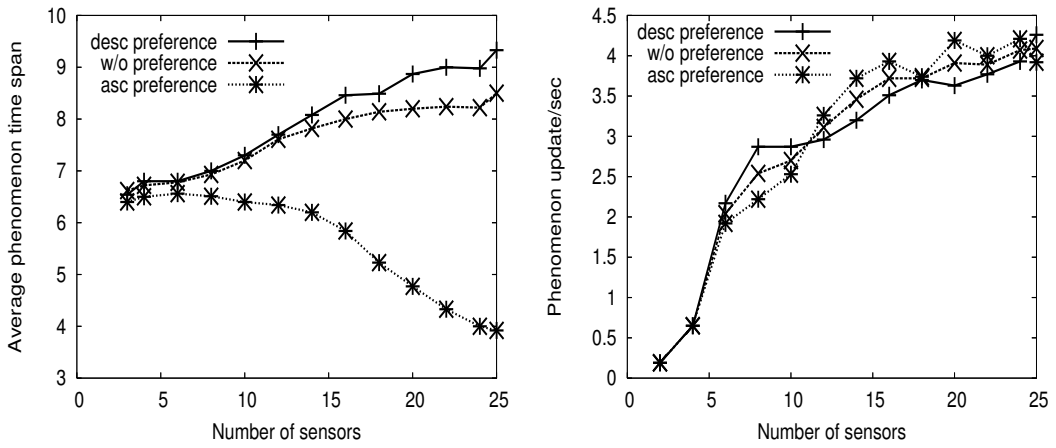
(a) Real small-scale setup



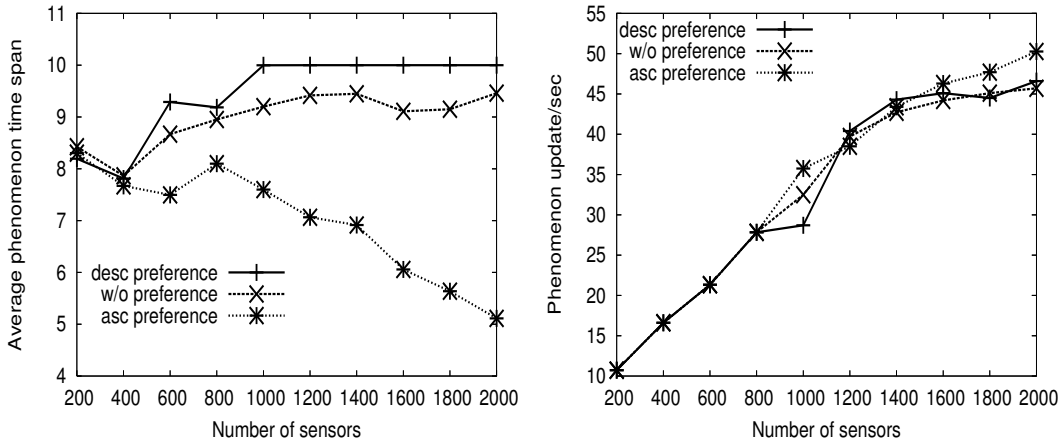
(b) Simulated large-scale setup

Figure 4.4. The effect of the spread preference.

changing the spread preference does not degrade the system's performance measured in terms of detected phenomenon updates per second. The spread preference does not affect the total number of detected phenomenon updates per second. However, the system focuses on the phenomenon updates that are associated with phenomena that have small/large spread in response to an ascending/descending spread preference.



(a) Real small-scale setup



(b) Simulated large-scale setup

Figure 4.5. The effect of the time span preference.

4.4.3 The Time-Span Preference

Figure 4.5 illustrates the effect of load shedding with an ascending preference in time span, without any preference in time span, and with an descending preference in time span under various system loads. Figure 4.5a utilizes the real small-scale sensor board while Figure 4.5b utilizes the simulated large-scale setup. For a small system load (expressed in terms of the number of incoming data streams), almost every phenomenon is detected and, hence, the average phenomenon time span is almost the same for all of the three depicted cases. However, as we increase the number of data

streams, tuples are dropped out of the system. PDT with a descending time span preference favors durable tuples that show up long time ago in the past and generates durable phenomena in the output. On the other hand, PDT with an ascending time span preference favors tuples with short history in the past and generates impulse phenomena that occur over a short period of time. Again, the deviation between the three PDT instances becomes apparent at the size of 25 sensors for the real sensor board and at the size of 2000 sensors for the simulated setup in Figure 4.5. Notice that changing the time span preference does not degrade the system's performance measured in terms of detected phenomenon updates per second. However, the system focuses on the phenomenon updates that are associated with phenomena that have small/large time span in response to an ascending/descending time span preference.

4.5 Related Work

In this section, we overview related work through three major lines. First, we refer to some research directions in the context of load shedding and discuss how load shedding is related to our work. Second, we highlight some summarization techniques that are deployed widely to enhance query performance. Third, we give example techniques that perform the join operation over data streams.

In data stream systems, load shedding is utilized to address memory and CPU limitations. There are two types of load shedding: random load shedding [37–40] and semantic load shedding [41–43]. Semantic load shedding considers the values of incoming tuples. Tuples are processed based on how productive their values are. In our work, we make use of semantic load shedding and the semantics of incoming tuples are assessed relative to the query preference. In the absence of semantic interpretation or distribution models for the tuple values, random load shedding takes place.

With the limited CPU time and bounded memory constraints, approximate answers that are obtained via summaries are accepted in place of the exact ones.

Sampling [47], histograms [44], and wavelets [45] can represent a stream using lower memory requirements. Streams are also summarized using sketches [46]. Some of the sketching techniques have the capability to maintain a stream's most frequent items [19]. Sketch-based processing and sketch sharing among multiple queries are presented in [48]. The concept of multiple granularities over data streams is proposed in [49]. The work in [50] suggests a gradual degradation of the summary resolution as data tuples move from one granularity to the next. This gradual degradation is referred to by the term *amnesic stream approximation*. Statistical models are utilized in [6] to query sensors interactively. Based on outstanding queries, statistical models provide an estimate for a sensor reading and tell how much this estimate is accurate. Consequently, one may decide to query the sensor for a fresh reading.

The join operation over data streams has been studied extensively in literature, e.g., [42, 51–54]. Many of these techniques are based on Symmetric Hash Join (*SHJ*) [55]. *SHJ* takes care of the infiniteness of the data sources. XJoin [56] is a non-blocking symmetric join that stores overflowing tuples on disk for later processing to avoid blocking. Hash-Merge Join (*HMJ*) [57] is another non-blocking join algorithm that produces early join results. Our proposed technique for load shedding extends *SHJ* by tuning the join output towards query preferences.

4.6 Summary

In this chapter, we introduced the concept of preference-based load shedding for phenomenon-aware DSMSs. Preference-based load shedding favors tuples that contribute to satisfying a specific preference of a phenomenon definition. In particular, we address three phenomenon preferences that are based on the phenomenon persistency, the phenomenon spread, and the phenomenon time span. The persistency-based preference tunes the PDT query processing towards either persistent or intermittent phenomena. The spread-based preference tunes the PDT query processing towards either stretch or confined phenomena. The time span-based preference tunes

the PDT query processing towards either durable or impulse phenomena. Other preferences can be flexibly defined to satisfy the domain requirements. We proposed a stream summary manager that is capable of identifying the stream tuples that contribute to a specific preference. Then, we proved the applicability of the summary manager in tuning the window join query that comes at the core of phenomenon detection and tracking techniques. Experimental results show that the concept of preference-based load shedding increases the effectiveness and the resource utilization of Nile PhenomenaBase.

5 PHENOMENON DETECTION AND TRACKING USING VARIABLE-ARITY JOINS

Phenomenon-aware DSMSs are equipped with *Phenomenon Detection and Tracking* (*PDT*) techniques that continuously run at the background of the system to detect and track phenomena as they propagate in the surrounding field. The process of phenomenon detection and tracking depends mainly on a multi-way join operator that is at the core of *PDT* techniques to report similar stream readings. With the increase in the number of data streams and with periods of heavy loads, the join operator and, consequently, query processing in the DSMS face several challenges. In this chapter, we present a new join operator for phenomenon detection and tracking techniques called *variable-arity join* (*VAJoin*) that is specially-designed for dynamically-configured large-scale streaming environments. Experimental studies illustrate the scalability and the performance gains of the proposed join operator inside Nile PhenomenaBase with respect to the number of detected phenomena and the output delay.

5.1 Background and Motivation

A key component in *PDT* techniques is an *outer multi-way* join operator that detects *similarities* among stream readings over a sliding window of size ω. This join operation is "multi-way" because it detects similarities among multiple data streams and it is an "outer" join because phenomena are usually *localized*. Out of the large number of data streams in space, only subsets of data streams generate the same values. Other data streams 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* [55], *XJoin* [56], or *hash merge join* [57]). 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., [58]). To overcome the shortcomings of binary-join trees, [34] 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 as described in Chapter 3.

MJoin has satisfactory performance for moderate system load. However, with the increase in the number of data streams, the stream sampling rates, and the number of propagating phenomena, *PDT* techniques start losing many phenomenon updates. A phenomenon update is reported if a phenomenon appears, disappears, or changes its location. The number of detected phenomenon updates per second reflects how fast the system is in tracking phenomena as they move in space. To tackle periods of heavy system load, we identify two basic challenges that the design of PhenomenaBases with *MJoin* faces: (1) *Scalability*, where streaming sources are typically deployed in large scale with thousands of sources. (2) *Dynamic-configuration* – Streaming sources can be dynamically added and removed from the streaming environment based on the system's conditions and the availability of additional sources.

In this chapter, we introduce a novel join operator for DSMSs, called *Variable-arity Join* (or *VAJoin*) operator. In a nutshell, *VAJoin* handles the execution of *continuous multi-way window join* queries over *dynamically-configured large-scale* streaming environments. In contrast to *MJoin*, *VAJoin* is not an outer multi-way join. VAJoin produces variable-size join output in response to the variable number of data streams contributing to a phenomenon. It exploits the *locality* characteristics of phenomena to reduce the number of streams that need to be joined. With the

notion of variable-arity output, *VAJoin* scales well with respect to the number of data streams and easily adapts to the dynamic configuration of the network. This feature makes *VAJoin* a perfect match for the join operation in environments where a small number of data streams (relative to the huge number of data streams in the streaming environment) are likely to join. Although not limited to PhenomenaBases, *VAJoin* suits the process of phenomenon detection since phenomena are usually localized in small portions of the environment. Fire, smoke, and oil spills usually span small portions of the monitored field. Other environments where such locality is expected call for the deployment of *VAJoin* over its streaming sources.

The contributions of this chapter can be summarized as follows:

1. We introduce the concept of variable-arity output and we equip PDT techniques with a variable-arity join operator.

2. We compare the performance of variable-arity join against the performance of outer multi-way join and we explore the cost models of both join techniques mathematically.

3. We provide an experimental study that is based on a prototype implementation of *VAJoin* inside Nile PhenomenaBase to prove its efficiency both in terms of the number of detected phenomena updates and the output delay.

The remainder of this chapter is organized as follows: Section 5.2 presents the *VAJoin* operator and its manipulating algorithms. Section 5.3 presents a mathematical analysis of the proposed join technique while Section 5.4 provides an experimental study of the proposed operator and compares its performance to the mathematical study. Section 5.5 overviews related join techniques and compares them to *VAJoin*. Finally, Section 5.6 summarizes the chapter.

5.2 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 data streams contributing to a phenomenon. We also compare it to the outer multi-way join that was initially implemented in the phenomeon detection and tracking module. 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 a 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 a large-scale streaming environment is cost prohibitive. To detect subsets of joining data streams using outer join, every streaming source in the environment 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 streaming sources to find out that only tens of streaming sources have 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 5.2.1.

Definition 5.2.1 *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=$\{(\hat{t}, [\hat{S}, \hat{\tau}], [S_{o_1}, \tau_{o_1}], [S_{o_2}, \tau_{o_2}], \cdots)$, where S_{o_i} is one of*

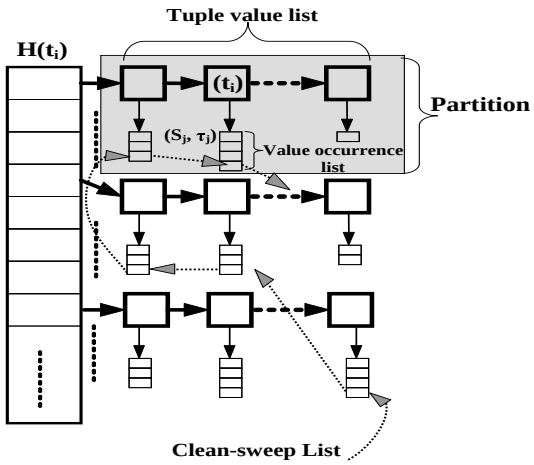


Figure 5.1. The VAJoin hash table.

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

We should notice that VAJoin is different from an outer join both at the conceptual and implementation levels. At the conceptual level, VAJoin *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 $([S_{o_1}, \tau_{o_1}], [S_{o_2}, \tau_{o_2}], \cdots)$. 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 such streams.)

5.2.1 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 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 VAJoin 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 5.1 illustrates the proposed VAJoin 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

PROCEDURE *Insert-Probe*

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

OUTPUT: *(1) an updated VAJoin 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 ← $\hat{t}$, 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 5.2. The VAJoin algorithm.

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.

5.2.2 Variable-Arity Join Algorithm

The algorithm for the proposed VAJoin is given in Figure 5.2. This algorithm is executed by the query processor whenever it receives a new reading from one of the incoming data streams. 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 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.

5.2.3 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 data streams. However, some applications require a different window size for each data stream (i.e., ω_i is the sliding window over stream S_i). In the literature, binary join is performed over two streams such

that each stream has its own window size [54]. 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 streaming environment and accommodates variable streaming 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 5.2 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., $\hat{\tau} - temp.\tau \leq \omega_{temp.S}$.

5.2.4 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 streaming sources that may produce no output.

Second, the variable-arity join avoids partial-result processing. Binary join trees 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 data streams. 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 streaming environment at no additional cost. Since all stream readings are hashed to the same global table, the addition and/or deletion of a streaming source affects 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 streaming sources, *MJoin* creates and/or removes hash tables and adjusts the join probing sequence of incoming tuples accordingly.

5.3 Mathematical 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 *VAJoin* 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 *VAJoin* to generate the output in the centralized case.

	Outer MJoin	**VAJoin**
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 5.3. Cost estimates of both *MJoin* and *VAJoin*.

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 *VAJoin*), *resolve collisions* in the hash tables (only one table for *VAJoin*), *separate* the different encountered tuples into their respective streams, this applies only to *VAJoin* since the tuples for all streams are in the same hash table, and finally *form* the output join tuples.

Given k input streams, Figure 5.3 provides the different formulas to compute the time estimate for each of the above components for both outer *MJoin* and *VAJoin*. 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 *VAJoin* 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 VAJoin ($distinct_2$) because the VAJoin 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 *VAJoin* 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 afterwards. 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

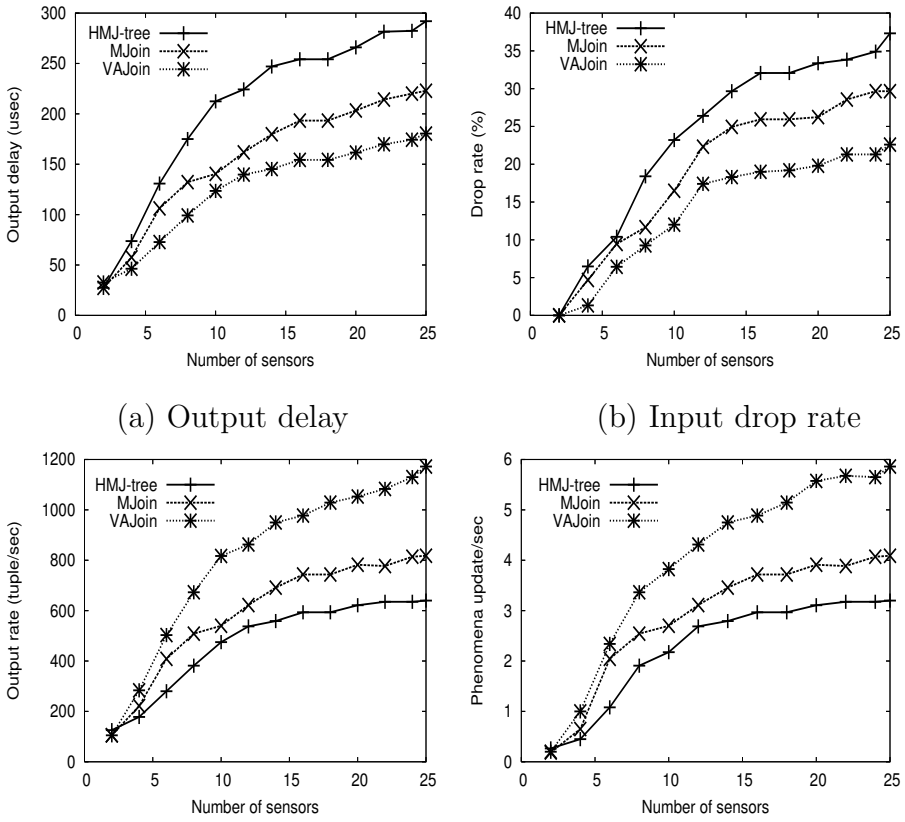
in that stream (σ_i) for all k streams, which results in $C_4\Sigma_{i=1}^{k}\sigma_i n_i$. The outer *Join* is not subject to this separation cost since tuples from the same stream are in the same table.

The tuple formation cost is computed based on the size of the output that is the product of the number of output tuples, $\prod_{i=1}^{k}\sigma_i n_i$ for both joins, by the tuple size that corresponds to the number of streams k for the outer *MJoin*, and $2kk'$ for *VAJoin*. 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 in Section 5.4 shows how this analysis compares to the actual experiments.

5.4 Experimental Analysis

We now present the experimental study we conducted to explore the performance of the proposed *VAJoin* operator. We base our study on the two experimental setups of Nile PhenomenaBase that are described in Section 2.5; a real small-scale sensor board and a simulated large-scale sensor network. In both setups, the join techniques are triggered through a multi-way join query with a sliding window of size 10 seconds. Two sets of experiments are performed. The first set of experiments (Section 5.4.1) investigates the performance under the real sensor-platform setup. The second set of experiments (Section 5.4.2) addresses the large-scale simulated sensor-network setup and examines the dynamic reconfiguration of the network. In Sections 5.4.1 and 5.4.2, we compare the performance of the following three techniques:

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



(a) Output delay (b) Input drop rate

(c) Output rate (d) Phenomena updates

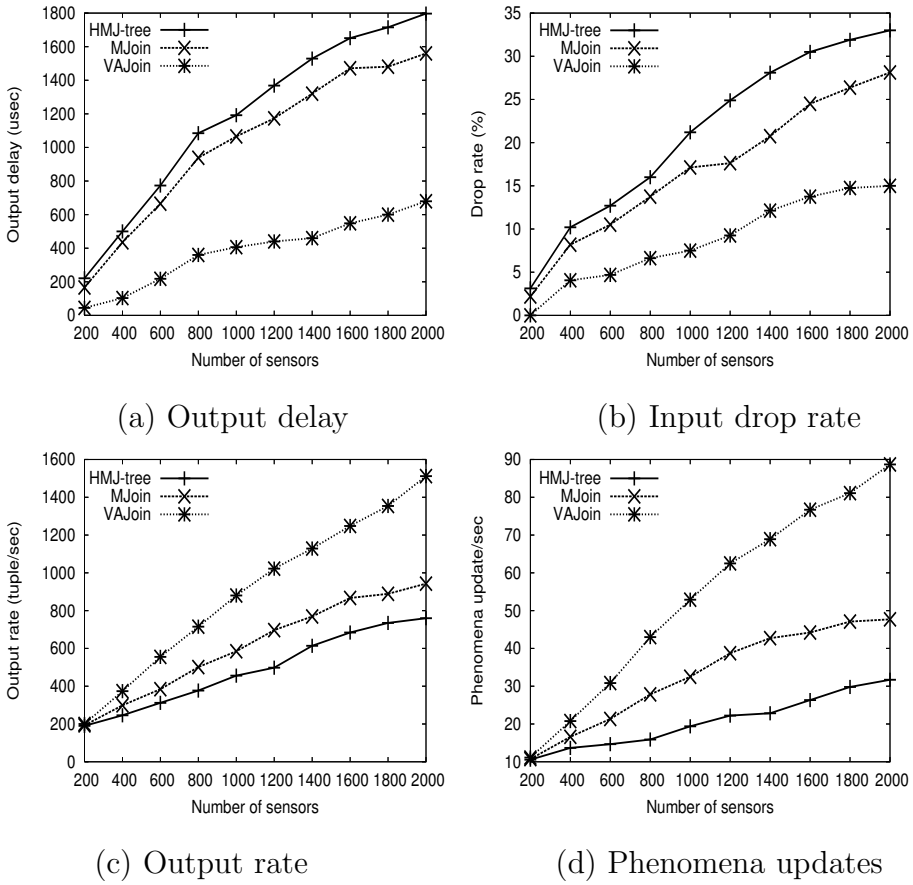
Figure 5.4. Performance under *real small-scale* data sets.

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

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

Section 5.4.3 compares the mathematical analysis of Section 5.3 against the experimental results that are obtained in this section.

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,



(a) Output delay (b) Input drop rate

(c) Output rate (d) Phenomena updates

Figure 5.5. Performance under *synthetic large-scale* data sets.

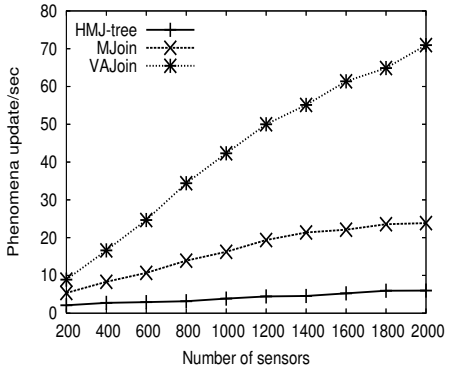


Figure 5.6. The effect of dynamic network configuration.

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 prototype implementation of the join operators inside Nile PhenomenaBase. The Nile PhenomenaBase engine executes on a machine with Intel Pentium IV, CPU 2.4GHZ and 512MB RAM running Windows XP.

5.4.1 Performance Analysis Using Real Data Sets

The performance of *a HMJ tree*, *MJoin*, and *VAJoin* under the real sensor-platform setup is given in Figure 5.4. As illustrated in Figure 5.4a, *VAJoin* 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 5.4b), and consequently produce a higher output rate (Figure 5.4c). From the point of view of the overall-performance, *VAJoin* detects up to 75% more phenomena updates than *HMJ trees* and up to 43% more phenomena updates than *MJoin* (Figure 5.4d).

5.4.2 Performance Analysis Using Synthetic Data Sets

Performance gains of *VAJoin* become more significant for large-scale sensor networks. In contrast to binary join trees and *MJoin*, *VAJoin* avoids unnecessary probes to a huge number of separate tables, and therefore, reduces its per-tuple processing time. The same experiments of Section 5.4.1 are repeated using the 2000 sensor simulated setup. Figure 5.5 illustrates the efficiency of *VAJoin* in terms of the output delay, the input drop rate, and the output rate. *VAJoin* doubles the output rate

of a *HMJ tree* and increases the output rate by up to 60% over *MJoin*. Moreover, *VAJoin* detects up to 180% more phenomena updates than *HMJ trees* and up to 85% more phenomena updates than *MJoin*.

Figure 5.6 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 5.5d and Figure 5.6, 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 *VAJoin* is reduced by only 20% (at 2000 sensors).

5.4.3 Comparison of the Analytical and the Experimental 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 5.7. 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 *VAJoin*, 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 (from the real sensor board experimental setup) and is found to be 40%.

Figure 5.8 gives the result of the comparison. The analytical and experimental output delays exhibit the same trend for both *VAJoin* and outer *MJoin*. We notice that *VAJoin* 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 *VAJoin* 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 5.7. Parameter and Constant Values for the Comparison.

5.5 Related Work

A large body of research in the data streaming area focuses on the join operation, e.g., [2, 51–53]. To highlight the reasons that make *VAJoin* applicable in phenomenon-aware DSMSs, we overview related multi-way join techniques and compare them to *VAJoin*. Multi-way join can be achieved through a tree of binary joins (either *symmetric hash join* [55], *XJoin* [56], or *hash merge join* [57]), a single *MJoin* operator [34], or a single *VAJoin* operator. Based on the experiments in Section 5.4, Figure 5.9 provides a comparison among various multi-way join techniques based on a key set of distinguishing features.

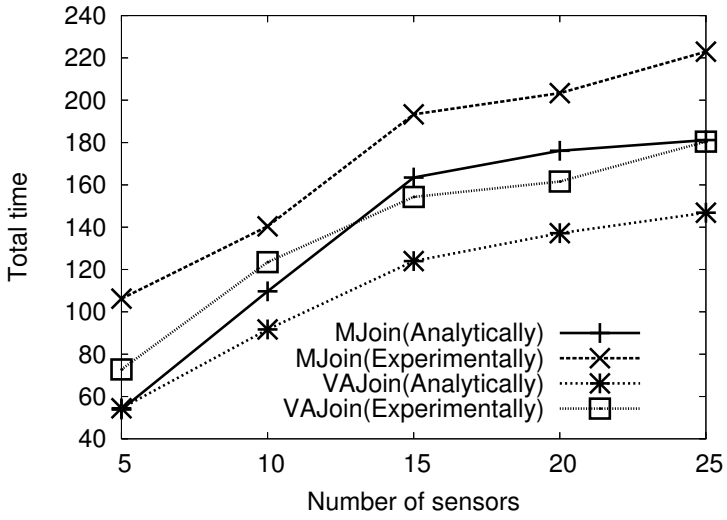


Figure 5.8. Comparison of Analytical and Experimental Output Delay for outer MJoin and VAJoin.

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 streaming environments (unless query plan reorganization is performed). On the other hand, *MJoin* and *VAJoin* 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 *VAJoin* results in a lower output delay. Moreover, *VAJoin* 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 [58]) to increase the output rate. All techniques handle outer joins by traversing the join probing sequence completely. On the other hand, *VAJoin* supports, by design, variable-arity joins to avoid long chains of probing sequences.

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

(×: feature not supported, √: feature supported, √√: feature supported and enhanced)

Figure 5.9. Comparison among various multi-way join techniques.

5.6 Summary

In this chapter, we presented the *VAJoin* operator, a variable-arity join operator for phenomenon detection and tracking techniques inside phenomenon-aware DSMSs. To meet the demands of streaming environments, *VAJoin* is designed to scale with respect to the number of streaing sources in the monitored environment without sacrificing the join output rate. Analytical and experimental studies that are based on a prototype implementation of the join operators inside Nile PhenomenaBase show the scalability of *VAJoin*. *VAJoin* increases the output rate over binary join trees and *MJoin*. Once *VAJoin* is adopted by Nile PhenomenaBase, the number of detected phenomena updates is increased while the output delay is reduced.

6 ADAPTIVE PHENOMENON-AWARE QUERY OPTIMIZATION

Data streams that are generated from close-by regions experience similar environmental conditions that result in distinct phenomena. Previous chapters of this dissertation are dedicated to detect and track various phenomena inside a data stream management system (DSMS). In this Chapter, we use the detected phenomena and their spatial properties to reduce the demand on the DSMS resources when executing large numbers of concurrent continuous queries. The main idea is to let the DSMS observe the input data streams at the phenomenon level. Then, each incoming continuous query is directed only to those phenomenon regions that are likely to satisfy the query predicates. More specifically, each incoming continuous query is directed only to those input streams that participate in the query answer. Two levels of indexing are employed, a phenomena index and a query index. The phenomena index provides a fine resolution view of all the input streams that participate in a particular phenomenon. An input stream that does not participate in any phenomenon is considered an outlier and is not involved in the query answer. The query index utilizes the phenomena index to maintain a query deployment map in which each input stream is aware of the set of continuous queries that the stream contributes to their answers. Experimental results show the efficiency of our approach with respect to the accuracy of the query result and the resource utilization of the DSMS.

6.1 Background and Motivation

Huge amounts of data are streamed out from large numbers of streaming sources that are widely and densely deployed in the surrounding environment. Having such large numbers of streaming sources poses new challenges to the query processor in data stream management systems. The highly-dynamic nature of the streaming

environments makes indexing the sources an infeasible solution. With the absence of efficient indexing schemes and the absence of any prior knowledge about the nature of streaming sources, queries may need to scan all the streaming sources to obtain an accurate answer. However, two main issues hinder the practicality of a sequential scan over input streaming sources: (1) The limited resources (e.g., computation and power resources) at the input streaming sources do not allow the participation of each streaming source in each outstanding continuous query. (2) Due to the real-time requirements of spatio-temporal queries and the large numbers of potential streaming sources, a database server cannot scale to process each single reading from each single source.

Recent research in exploiting the properties of spatio-temporal streams concluded that streaming sources that are spatially co-located tend to experience similar phenomenon conditions and provide similar readings over a certain period of time (e.g., see [11, 13, 15, 29–31]). Typically, in an environment with large numbers of streaming sources, there always exist a large number of various overlapped phenomena of different sizes and shapes. Due to the highly-dynamic nature of streaming sources, phenomena continuously change their sizes and locations over time.

In this chapter, we propose an adaptive stream query optimization paradigm, termed *phenomenon-aware* query optimizer. The main idea is to make use of the efficient techniques for phenomenon detection and tracking to optimize subsequent queries. In this chapter, we focus on one type of queries, namely the selection queries. Detected phenomena act as an indexing scheme that direct the execution of selection queries to only those streaming sources that can contribute to the query answer. By looking at the existing phenomena within the streaming sources, the query processor will have a fine resolution view over all the streams. Based on this fine view, the query optimizer decides which phenomena need to be investigated more to answer the selection predicate of a specific query. The *phenomenon-aware* query optimizer achieves a trade-off between the number of streaming sources participating in query execution and the accuracy of that query. One of the main attractive features of the

proposed *phenomenon-aware* query processor is that it is inherently equipped with an outlier-detection that makes it sustainable to the noisy environment of streaming sources. Outlier or isolated streaming sources that do not contribute to any phenomena do not appear in the finer resolution view. Thus, they do not contribute to the query answer.

To efficiently realize the *phenomenon-aware* query processor, we employ two indexing schemes, the *phenomenon index* and the *query index*. The *phenomenon index* keeps track of currently detected phenomena within the stream resources. The main assumption behind this indexing is that the change in the phenomena parameters (e.g., shape and location) is much less frequent than the change in the spatio-temporal streaming sources. Thus, while it is almost infeasible to index the spatio-temporal streaming sources, we can provide an efficient indexing scheme for the list of phenomena over the streaming sources. On top of both the *phenomenon index* and the *query index*, a *query deployment map* is maintained. The *query deployment map* allows each streaming source to subscribe only to a list of queries that the streaming source contributes to their answer. Moreover, the proposed phenomenon-aware optimizer handles both stationary streaming sources as well as mobile streaming sources. Mobile streaming sources are streaming sources that are attached to moving objects. Mobile streaming sources generate streams of readings as the moving objects move in space. Examples of mobile streaming sources include RFIDs that are attached to moving vehicles or temperature sensors that are attached to firefighters. Mobile sensors are of special interest to phenomenon-aware optimizers because mobile objects experience different phenomena as they move from one region to another. Consequently, the phenomenon-aware optimizer is required to reorganize its query deployment map dynamically. To address the mobility of streaming sources, we modify the phenomenon definition to include the spatial properties of the phenomenon as described in Section 6.3. Once a streaming source moves from a phenomenon region (R_1) to another phenomenon region (R_2), the query deployment map is reorganized to associate the queries that are interested in R_2 with the

streaming source. In summary, the contributions of this chapter are summarized as follows:

1. We introduce the concept of *phenomenon-aware query processing* and empower data stream management systems with an adaptive phenomenon-aware optimizer.

2. We propose two levels of indices at the core of the phenomenon-aware optimizer; a *phenomenon index* and a *query index*.

3. Given the phenomenon and the query indices, we generate an efficient *query deployment map* where each query is deployed over a small subset of data streams (e.g., 5.5% of the total number of sensors).

4. We provide an experimental evidence that phenomenon-aware query processing increases the output rate of continuous queries that are registered at the system (by up to 300%).

The rest of this chapter is organized as follows. Section 6.2 describes the architecture of the phenomenon-aware optimizer. Sections 6.3 and 6.4 describe the *phenomenon index* and the *query index*, respectively. Section 6.5 gives the implementation details while Section 6.6 provides an experimental study of the proposed indices inside Nile PhenomenaBase. An overview of the related work is given in Section 6.7. Finally, Section 6.8 summarizes the chapter.

6.2 System Architecture

Figure 6.1 gives the architecture of the proposed phenomenon-aware query optimizer. The phenomenon-aware optimizer has three main components: the *phenomenon monitor*, the *query plan analyzer*, and the *query dispatcher*. The *phenomenon monitor* keeps track of existing phenomena as they move in space through the *phenomenon index*. The phenomenon index indexes phenomena by their value

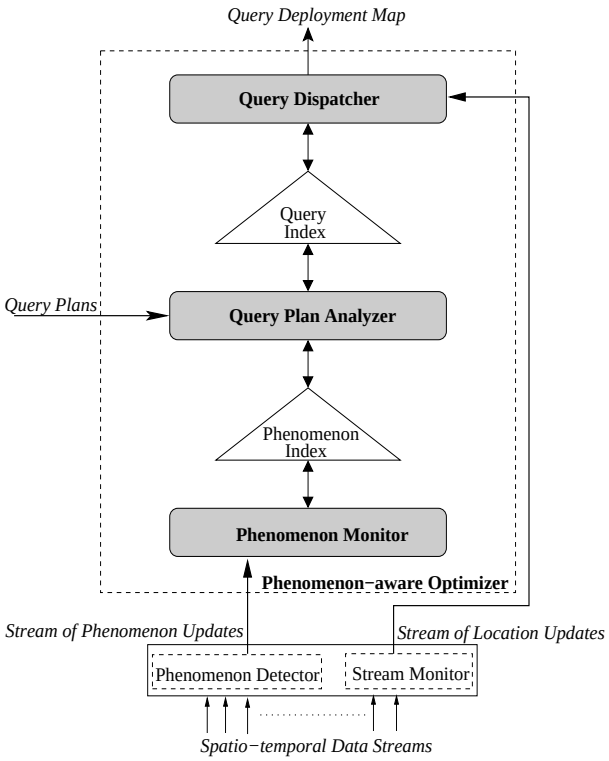


Figure 6.1. The Architecture of a phenomenon-aware Optimizer.

contents. The *query plan analyzer* traverses the phenomenon index for each query plan to decide which phenomenon regions are likely to satisfy the query predicates. Then, the *query index* is built to index queries spatially based on their regions of interesting phenomena. The *query dispatcher* updates the query deployment map according to the new locations of streaming sources. Then, the *query dispatcher* executes each query only over its regions of interest.

Inputs to the phenomenon-aware optimizer are of three types: a stream of *phenomenon updates*, a stream of *location updates*, and a *set of query plans*. The stream of *phenomenon updates* provides the optimizer with the necessary knowledge about phenomena in the space. A *phenomenon update* tuple has one of the following two forms: *(Phenomenon-id, Behavior-Update, B)* or *(Phenomenon-id, Region-Update, R)* to indicate a behavior or a region update, respectively. A phenomenon-behavior update implies a change in the readings of the phenomenon underlying streams, e.g.,

an increase in the temperature readings of a fire. A phenomenon-region update implies a displacement of the streaming sources contributing in the phenomenon, e.g., the movement of a fire in accordance with the direction of the wind. The stream of *location updates* provides the optimizer with the current locations of the streaming sources and it has the form *(StreamSource-id,x,y)*, where x and y are the location coordinates. The set of *query plans* is processed by the optimizer based on the knowledge of phenomena in the space. Recall from Chapter 2 that the phenomenon-aware optimization is triggered through select-within-phenomena queries.

The phenomenon-aware optimizer generates a *query deployment map* as its output. The *query deployment map* is represented as a sequence of commands with the form *(StreamSource-id* **SUBSCRIBES TO** *Query-id)* to indicate the streaming sources that are of interest to a particular query.

As described in Chapter 2, the basic components of *Nile* are the stream admission controller, the query admission controller, the stream monitor, the query plan generator, and the query executer. The *PDT-module* is added to Nile to detect the appearance of new phenomena and to track the propagation of already-detected phenomena. The phenomenon-aware optimizer, which is the focus of this chapter, optimizes user queries based on the feedback it receives from the *PDT-module*. Figure 2.1 illustrates how the phenomenon-aware optimizer is connected to other components of the system. The optimizer receives a stream of phenomenon updates from the *PDT-module*. Also, the stream monitor generates a stream of location updates and passes it to the PDT-module, which in turn forwards these location updates to the phenomenon-aware optimizer. Also, the optimizer receives a set of query plans from the query plan generator. Then, the optimizer generates efficient query plans and efficient query deployment maps (QDMs) that are sent over to the query executer.

6.3 Phenomenon Indexing

This section exploits the spatial properties of the phenomenon definition given in Section 2.1 and provides a description of the phenomenon index and its manipulating algorithms. Definition 2.1.1 defines a phenomenon at time instant τ to be an R-B pair, where R is a list of streaming sources with similar behavior and B is a representative behavior for the phenomenon over a sliding window of size ω. In this chapter, we utilize the spatial attributes of the streaming sources in R. We refer to R as the bounding box of all streaming sources that contribute to the phenomenon. Definition 6.3.1 extends the phenomenon definition of Chapter 2 with the phenomenon spatial attributes.

Definition 6.3.1 *A phenomenon P at time instant τ is a binary tuple (R ,B_ω), where R is the* bounding box *of the streaming sources contributing to P and B_ω is the representative behavior of phenomenon P over the most recent time window of size ω, such that $\forall S_i \in R$, $Count(Value(S_i[\hat{\tau}]) \in B_\omega) \geq \alpha$, $\hat{\tau} \in [\tau - \omega + 1 \cdots \tau]$ and $||R|| \geq \beta$.*

Based on Definition 6.3.1, a phenomenon has to be associated with a time instant τ, changes its location R and behavior B_ω with time, and is captured over a time window of time ω to ensure its persistency. Streaming sources that fall in the phenomenon region report values similar to the phenomenon behavior with high probability. As described in Section 2.1.2, B_ω captures the intrinsic features of the underlying phenomenon, e.g., values, frequencies, and trends of tuples contributing to the phenomenon. The exact choice of the parameters represented within B_ω is orthogonal to the phenomenon indexing. The only requirements from B_ω is to satisfy the following two properties: (1) Fast online processing. The phenomenon behavior should be captured and updated quickly to fit in the online data-streaming environment, (2) Adherence to the postulates of a metric space. The distance among the behavior of different phenomena should be positive, symmetric, and satisfy the

triangular inequality. Based on the phenomenon behavior properties, we identify the following two interface functions:

1. *P2P-Dist(P_1,P_2)*: A phenomenon-to-phenomenon distance function to compute the distance between the behaviors of two phenomena. The *P2P-Dist* function is used to maintain the phenomenon index upon insertion and deletion of phenomena.

2. *Q2P-Dist(Q,P)*: A query-to-phenomenon distance function to compute the distance between a given query Q and the behavior of a phenomenon P. The *Q2P-Dist* function is used by outstanding queries to search the index for interesting phenomena.

All the manipulation algorithms for both the phenomenon index and the query index are presented in terms of these two behavior interface functions.

6.3.1 The Phenomenon-Index Structure

Figure 6.2 illustrates the phenomenon index. The phenomenon index has two types of nodes: *leaf* nodes and *non-leaf* nodes. One *leaf* node is constructed per phenomenon to store the following information: (1) *Ph-id*, the phenomenon identifier, (2) R, the current phenomenon region, and (3) LSQ, a list of satisfied queries, i.e., queries with predicates that are satisfied by values of the phenomenon. *Non-leaf* nodes direct the search operation to leaf nodes by recursively grouping nodes with similar behavior together. Each non-leaf node maintains a list of <*child-ptr, B*> pairs where *child-ptr* is a pointer to the child whose behavior is B. As we go up the tree, the behavior field B in the non-leaf node is set to be a representative for the behavior of all the phenomena in the child subtree.

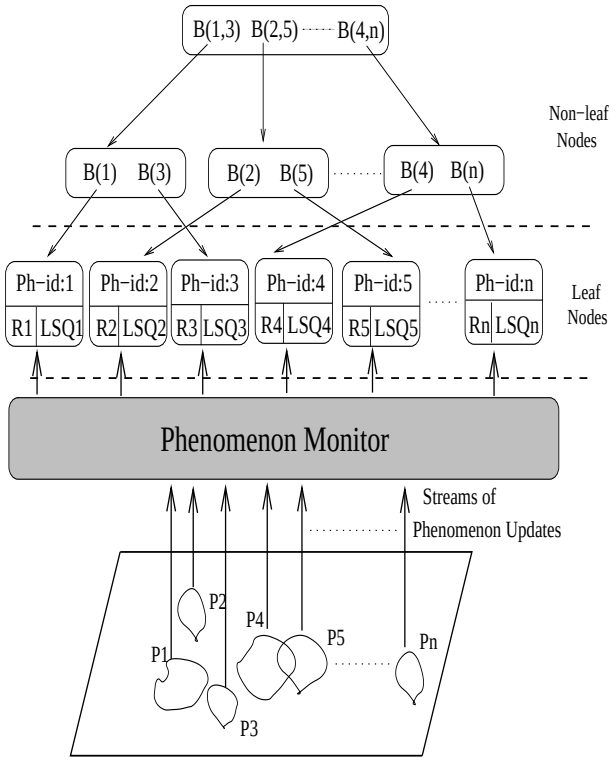


Figure 6.2. The phenomenon index.

6.3.2 Maintaining the Phenomenon Index

Figure 6.3 gives the algorithm of maintaining the phenomenon index when receiving a change in the phenomenon behavior. The inputs to the algorithm are the phenomenon identifier (*Ph-id*) and the new behavior (B_{new}). To avoid updating the phenomenon index for marginal phenomena changes, we compare the input behavior to the base phenomena behavior, i.e., the one used on building the phenomena index. Then, we process the incoming phenomena update only if it is more different than the base one by the *behavior tolerance parameter* (BTP) (Steps 1 and 2 in Figure 6.3). Examples of marginal phenomena update that we want to avoid processing include the temperature readings inside a fire region where temperature fluctuates up and down by small amounts. Updating the index with every behavior update may overload the system. Once the distance between the current and the base behaviors

PROCEDURE **Update-Phenomenon-Behavior** (*Ph-id*,B_{new})

1. *CurrentBehvior*(*Ph-id*)=B_{new}
2. if **P2P-Dist**(*CurrentBehvior*,*BaseBehvior*)$\leq BTP$
   exit
3. *BaseBehvior*(*Ph-id*)=B_{new}
4. Propagate (*Ph-id*,B_{new}) update to upper levels of the index
5. FOR (*i*=1 TO **sizeof**(*Ph-id.LSQ*))

   if **Q2P-Dist**(*Ph-id.LSQ[i]*,*Ph-id*)$> d$
   delete(*Ph-id.LSQ[i]*)
6. *Node-Ptr*=**LeafNode**(*Ph-id*)
   Do
   (a) *Node-Ptr*=**ParentNode**(*Node-Ptr*)
   (b) *Changed*=FALSE
   (c) FOR EVERY leaf nodes *LN* in *Node-Ptr* subtree S.T. *LN* is not visited before

   FOR (*i*=1 TO **sizeof**(*LN.LSQ*))
   if **Q2P-Dist**(*LN-Ptr.LSQ[i]*,*Ph-id*)$\leq d$
   add *LN-Ptr.LSQ[i]* **TO LeafNode**(*Ph-id*)
   Changed=TRUE;

   WHILE (*Changed*)

Figure 6.3. An algorithm to accommodate changes in queries' interest.

goes above *BTP*, the value of the current behavior is copied into the base behavior (Step 3 in Figure 6.3). Then, the base behavior is propagated up the phenomenon index causing updates in the non-leaf index nodes (Step 4 in Figure 6.3). For all

the queries that were interested in the phenomena (*LSQ*), we check whether they are still interested in the new value of the phenomena. This is performed by going through all the queries in *LSQ* and computing the distance between each query and the phenomenon's new behavior. If the distance becomes over d, the query is removed from the *LSQ* of this phenomenon region (Step 5 in Figure 6.3).

To discover the new queries that become interested in the new behavior of phenomenon *Ph-id*, we make use of the similarity in behavior among neighboring regions in the phenomenon index (Step 6 in Figure 6.3). The main idea is to backtrack the path from the leaf node of phenomenon *Ph-id* in the phenomenon index to the root node. Initially, let *Node-Ptr* point to the parent of the leaf node that contain the phenomenon with new behavior (i.e., phenomenon *Ph-id*). We identify the queries that are in the subtree of *Node-Ptr* and, are *not* in the phenomenon query list (i.e., *LSQ* of *Ph-id*). These identified queries are candidates to be added to the *LSQ* of the phenomenon *Ph-id* if they show interest in the phenomenon's new behavior. We measure the distance between every query in the *Node-Ptr* subtree and the new behavior of phenomenon *Ph-id*. If a query is within distance d from the phenomenon's new behavior, the query is added to the *LSQ* of the phenomenon *Ph-id*. We go up the phenomenon index one more level (*Node-Ptr=ParentNode(Node-Ptr)*) and repeat the same process. As we go up the phenomenon index, the similarity between the queries and the phenomenon's new behavior decreases. We stop ascending he index once we reach a level where no more queries are added to the *LSQ* of *Ph-id* leaf node.

6.3.3 Searching the Phenomenon Index

A query is executed over a phenomenon region if the phenomenon behavior is within distance d from the query (based on the *Q2P-Dist* function). For each query, a range selection (with the query in the center and with a radius of d) is executed over the phenomenon index. With the increase in d, the query is deployed over a larger

number of phenomenon regions. Consequently, more output tuples are produced at the expense of consuming more system resources. On the other hand, decreasing d conserves the system resources and produces less output tuples. Choosing the value of d depends on two factors: (1) The availability of resources (which are assigned to queries based on their priorities) and (2) The quality of the output. Varying d both over time and from query to query gives the flexibility to tune every query based on the quality of its output.

Figure 6.4 gives the algorithm that measures the quality of a query output in terms of the average number of output tuples per second per stream compared to other queries. This measure reflects the relative (i.e., to other queries) output gain (i.e., output tuples per second) per unit cost (i.e., deploying the query over one stream). The input to the algorithm is an initialization vector for the values of d, one entry per query. Initially, the value of d for each query Q_i is set to its corresponding initial value $d_{initial}[i]$ (Step 1a in Figure 6.4). In addition, we initialize a *safety vector* to the corresponding $d_{initial}$ multiplied by a *safety factor* (Step 1b in Figure 6.4). The *safety vector* is used to *prefetch* a larger number of regions than required when searching the phenomenon index, i.e., a superset of the result returned by vector d (Step 1c and 1d in Figure 6.4). The main idea is to avoid searching the index multiple times if the values in vector d increase over time.

For every existing query, we evaluate its quality measure (Step 2a in Figure 6.4). Then, we find the average value for the quality measure over all queries (Step 2b in Figure 6.4). For each query, the value of d is tuned based on the relative performance of each query $\frac{\mu \cdot QM_i}{AvgQM}$ where μ is a weight factor between 0 and 1 that indicates how fast we propagate updates to the values of d. If the new value of d exceeds the precomputed safe value d_{safe}, d_{safe} values have to be updated and phenomenon index has to be searched gain. Finally, the query is dispatched to the new set of phenomenon regions and the algorithm goes into a sleep period before it is executed again (Step 2d in Figure 6.4). The *safety factor*, μ, and the length of the *sleep period* are all system tuning parameters.

PROCEDURE **Tune-d** ($d_{initial}[1 \cdot \cdot \ \textit{No-of-Queries}]$)

1. FOR (i=1 TO *No-of-Queries*)
   (a) $d[i]$=$d_{initial}[i]$
   (b) $d_{safe}[i]$=$d_{initial}[i] \times \textit{SafetyFactor}$
   (c) **PhenomenonIndex.Search**(Q_i,$d_{safe}[i]$)
   (d) **Dispatch**(Q_i,$d[i]$)
2. WHILE (TRUE)
   (a) FOR ($i = 1$ TO *No-of-Queries*)
   QM_i=$\frac{Output-tuples-per-second_i}{No-of-Streams_i}$
   (b) $AvgQM$= $\frac{\sum QM_i}{No-of-Queries}$
   (c) FOR ($i = 1$ TO *No-of-Queries*)
      i. $d[i] = d[i] \times \frac{\mu \cdot QM_i}{AvgQM}$
      ii. if $d[i] > d_{safe}[i]$
         A. $d_{safe}[i] = d[i] \times \textit{SafetyFactor}$
         B. **PhenomenonIndex.Search**(Q_i,$d_{safe}[i]$)
      iii. **Dispatch**(Q_i,$d[i]$)
   (d) wait a number of seconds

Figure 6.4. An algorithm to optimize the parameter d.

6.3.4 Queries with no Interesting Phenomena

Queries with no interesting phenomena do not have any phenomenon behavior within a distance d. Such queries are not likely to produce results. To handle such queries, we block their execution for a specific amount of time. After the blocking period elapses, we reinvestigate the phenomenon index. The process is repeated until the query gains interest in one of the phenomenon regions. The blocking time is a

system-tuning parameter that takes into account how tolerable we are in losing some of the initial query results. This phenomenon-based processing may lose some of the query output tuples if they are not part of a phenomenon. However, it saves the system resources by drawing the query's attention only to regions with a satisfactory behavior. Alternatively, we can increase the parameter d by a specific factor and repeat this process until the query shows interest in some phenomenon regions.

6.4 Query Indexing

This section describes building and maintaining the query index over the outcome of the phenomenon index. The query index is used to index queries by their regions of interest to generate efficient query deployment maps (*QDMs*). A *QDM* maps and executes each query on a set of data streams. An efficient *QDM* deploys queries over regions that are likely to satisfy the query predicates. As an alternative to *QDM*, we define the stream's query working set (*QWS*) to capture the same information as *QDM*. A stream's *QWS* (as in Definition 6.4.2) is the set of queries that are executed on that stream. *QDMs* can be driven from the streams' *QWSs* and vice versa, and therefore, *QDMs* and *QWSs* are used interchangeably.

Definition 6.4.1 *A query deployment map (QDM) gives for each query Q a set of streams $\mathcal{S}$ such that $\forall\ s_i \in \mathcal{S}$, Q is executed on s_i.*

Definition 6.4.2 *A stream's query working set (QWS) gives for each stream s a set of queries $\mathcal{Q}$ such that $\forall\ Q_i \in \mathcal{Q}$, Q_i is executed on s.*

Figure 6.5 illustrates the query index. Leaf nodes store the phenomenon identifiers, their regions R, and their corresponding lists of satisfied queries (*LSQ*), one node per phenomenon. Non-leaf nodes are constructed to spatially index the *bounding boxes* (*bb*) of phenomenon regions. The phenomenon index is a typical spatial index (e.g., an R-tree or one of its variants) for phenomenon regions. However, the selected spatial index structure needs to remain robust under heavy updates,

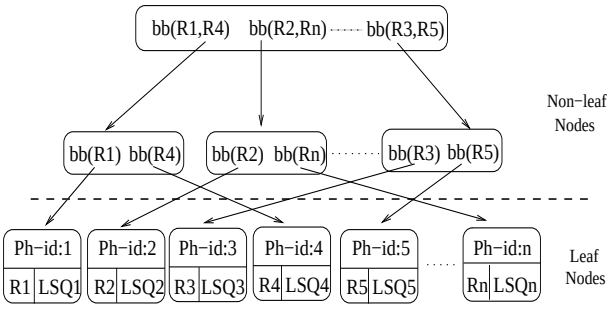


Figure 6.5. The query index.

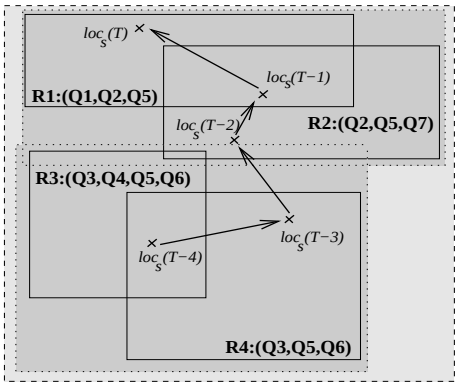


Figure 6.6. An example update in stream locations.

i.e., insertions and deletions. These updates correspond to the movements of the phenomenon regions in space over time.

The query index is constructed by the query plan analyzer and is searched by the query dispatcher (refer to Figure 6.1). The query plan analyzer propagates all updates in the region and the *LSQ* fields of the phenomenon-index leaf nodes to the query-index leaf nodes. If a phenomenon region is updated, this phenomenon region is deleted and is reinserted in the query index to adjust the index spatial properties. Updates to the *LSQ* fields are kept in the query-index leaf nodes and does not affect the index non-leaf nodes. For every stream, the query dispatcher searches the query index to retrieve all phenomenon regions that overlap with the stream's location. The stream's query working set (QWS) is the union of all *LSQs*

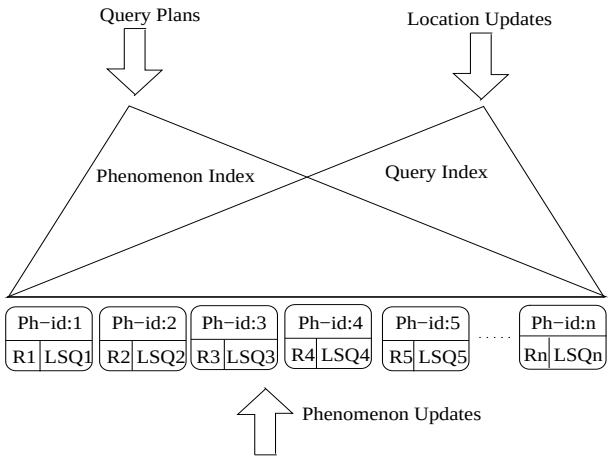


Figure 6.7. The combined phenomenon and query index.

that are associated with regions overlapping with the stream's location. The QWS of stream s_j is obtained as follows:

$$QWS(s_j) = \bigcup LSQ_i \; such \; that \; s_j.location \in R_i \tag{6.1}$$

Equation 6.1 implies that each stream subscribes to all queries that are interested in regions overlapping with the stream's location. Queries that have no interest in these regions are not executed on that stream. Therefore, no system resources are wasted to process queries that are not likely to be satisfied by the stream readings. The query dispatcher monitors changes in the streams' locations to update their $QWSs$ dynamically. Figure 6.6 gives an example of a moving stream over the last five time instants. At timestamp $\tau - 4$ the stream falls in regions $R3$ and $R4$. The stream's QWS is set to be the union of all queries that are interested in these two regions, i.e., $(Q3, Q4, Q5, Q6) \bigcap (Q3, Q5, Q6) = (Q3, Q4, Q5, Q6)$. At timestamp $\tau - 3$ the stream is in region $R4$ only and $Q4$ is no longer interested in the stream's readings. As the stream moves to region $R2$ and $R1$, the QWS becomes (Q2,Q5,Q7) and (Q1,Q2,Q5), respectively.

The leaf nodes in the phenomenon index have the same structure as the leaf nodes in the query index. Hence, the leaf nodes are shared by the two indices as illustrated

	Non-leaf nodes		**Leaf nodes**
	Phenomenon Index	**Query Index**	
Phenomenon behavior update	√	×	√
Phenomenon region update	×	√	√
Query plan update	×	×	√
Location update	×	×	×

Figure 6.8. Summary of updates to leaf and non-leaf nodes of the proposed indices.

in Figure 6.7. Upon receiving a phenomenon update, the update is propagated up the phenomenon index to reflect the phenomenon new behavior. Query plans search the phenomenon index starting from the root downwards and update the *LSQ* fields of the leaf nodes accordingly. Upon updating the leaf nodes, the query index has to be updated from the bottom up to accommodate any changes in the region fields of leaf nodes. Finally, the query index is searched from the top down with every update in the network configuration to associate each stream with a set of phenomenon regions.

Figure 6.8 summarizes the effect of various types of updates on the leaf and the non-leaf nodes of the phenomenon and the query indices. A behavior update affects the non-leaf nodes of the *behavior-based* phenomenon index. Also, with a change in the behavior of a phenomenon, queries change their interest in this phenomenon region and the *LSQs* fields of the leaf nodes are updated. Therefore, a behavior update affects both the leaf nodes and the non-leaf nodes of the phenomenon index. A region update affects the non-leaf nodes of the *region-based* query index and the region fields of the leaf nodes. The phenomenon index is searched using query plans to annotate the leaf nodes with *LSQs*. Hence, a change in the query plans affects the leaf nodes without affecting the non-leaf nodes. Also, the query index is searched using the locations of streams to find the streams' query working sets. An update in

the location of a stream implies a new search on the query index without any index updates.

6.5 Extensibility

The *Nile* system can be extended to support various representations of a phenomenon behavior. Section 2.1.2 describes three possible representations of a phenomenon behavior in a metric space. In this section, we investigate how the phenomenon-aware optimizer handles these behavior representations. More specifically, we give possible implementations of the *P2P-dist* and the *Q2P-dist* functions that accommodate various behavior representations. First, we represent a phenomenon behavior by the average value of streams contributing to the phenomenon. This behavior representation is obtained using Equation 2.2. The average value of each stream is obtained over the most recent window ω, then, the average over all streams is considered to be the behavior representation. The distance between two behaviors (*P2P-dist*) is the difference between the two behavior representative average values. The distance between a query and a phenomenon behavior (*Q2P-dist*) is the distance between a representative of the query and the behavior representative value. A point query, i.e., queries that ask for a single value point (e.g., select * where temperature=100°C), is represented by its point. A range query, i.e., queries that ask for a range of values (e.g., select * where temperature in $[50 \cdot \cdot 100]$°C), is represented by the center of its range. Although the average-value representation is suitable for point queries, it does not fit well into range queries. This representation lacks information about the range size of the query and the range of values covered by a phenomenon.

Second, a phenomenon behavior can be summarized and represented by the its k most frequent elements (or *top-k vector*), e.g., [19]. As in Definition 2.1.2, the top-k vector contains a subset of k elements such that the count of all elements in the top-k vector is equal or greater than the count of all other elements. In

contrast to representing the phenomenon by a single value, the top-k vector provides a more accurate representation of a phenomenon behavior. The distance between two phenomenon behaviors (*P2P-dist*) is measured as the distance between their top-k vectors. The work in [59] proposes several metric functions to measure the distance among top-k vectors. The distance between a query and a phenomenon (*Q2P-dist*) is the number of elements in the top-k vector that satisfy the query predicates.

Third, a phenomenon behavior can be represented by a histogram of its underlying values. The *P2P-dist* function is the L_2 distance between the normalized histogram buckets. Equation 6.2 gives the distance between the equi-width histograms of two phenomena: P_1 and P_2. Each histogram contains n buckets of equal width. Phenomena P_1 and P_2 contain a total of N_1 and N_2 reading values coming from their underlying streams over the most-recent window ω, respectively. The number of values in each histogram bucket (h_{1i} and h_{2i}) is normalized by the total number of values (N_1 and N_2) because two phenomena may have similar behaviors but with a different number of underlying stream readings. Then, we measure the distance between corresponding histogram buckets. The *Q2P-dist* function (Equation 6.3) measures the selectivity of the query over the histogram buckets. Then, we divide the selectivity by the total number of streams in the phenomenon to assess the number of expected output tuples relative to the query deployment cost.

$$\textit{P2P-dist}_H(P_1, P_2) = \sqrt{\sum_{i=1}^{n} (\frac{h_{1i}}{N_1} - \frac{h_{2i}}{N_2})^2} \tag{6.2}$$

$$\textit{Q2P-dist}_H(Q, P) = \frac{\sum_{i=1}^{n} \textit{Selectivity}_Q(h_i)}{\textit{No-of-Streams}} \tag{6.3}$$

Having a function that measures the distance among phenomenon behaviors in a metric space enables the indexing of phenomena by their behaviors. We propose to build the phenomenon index using an *M-tree* [60]. The *M-tree* is an efficient structure that indexes large data sets represented in a generic metric space. To implement the query index, we need a spatial index to maintain the bounding boxes of the queries'

interesting regions. The query index is required to accommodate frequent updates in the indexed regions. We implement the query index as an *R-tree with update memo* [61]. The *R-tree with update memo* is an R-tree variant that accommodates frequent updates by using an update-memo approach. This approach buffers updates in an update-memo structure and propagates these updates up the R-tree index from time to time.

6.6 Experiments

In this section, we explore the performance of the proposed phenomenon-aware optimizer experimentally. All the experiments in this section are based on a prototype implementation of the proposed optimizer inside Nile PhenomenaBase. We use the Nile PhenomenaBase simulated sensor platform (as described in Section 2.5). The Nile PhenomenaBase engine executes on a machine with Intel Pentium IV, CPU 2.4GHZ and 512MB RAM runningWindows XP. As described in Section 6.5, the phenomenon index is implemented as an *M-tree* while the query index is implemented as an *R-tree* with update memo. The behavior of phenomena is represented using a histogram of the phenomenon underlying values. The distance functions $P2P\text{-}dist_H$ and $Q2P\text{-}dist_H$ are used to measure the distance between the two phenomenon behaviors and between a query and a phenomenon behavior, respectively. Unless mentioned otherwise, we deploy 100 range queries over a set of 1000 data streams randomly distributed in space. The average radius of the query range is 10% of the space.

We conduct three sets of experiments. The first set of experiments measures the increase in the output rate in response to the proposed phenomenon-aware optimization (Section 6.6.1). The second set of experiments measures the reduction in the system's resource consumption (Section 6.6.2). The third set of experiments evaluates the best values for the system's tuning parameters (Section 6.6.3). We measure the output rate and the system resources with respect to a variable number

of queries and a variable number of data streams. We also investigate the effect of varying the radius of a range query on the system's performance.

6.6.1 The Output Rate

In this section, we investigate the average output rate per query under three different implementations:

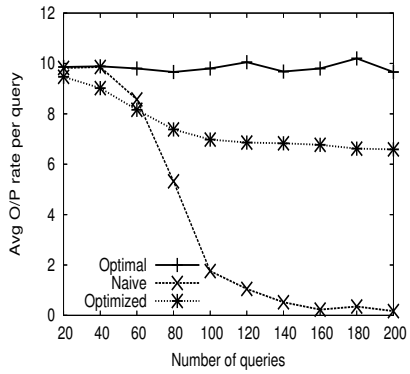
1. **Optimal query execution,** where the query result is computed as if we have infinite resources. The stream rates are slowed down such that no tuples are dropped out of the input buffers.

2. **Naive query execution,** where all queries are executed over all streams in the system.

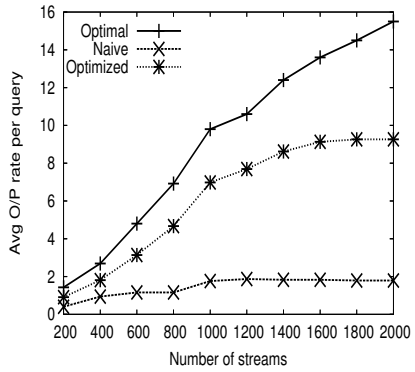
3. **Optimized query execution,** where a phenomenon-aware optimizer is implemented as described in this chapter.

Figure 6.9a illustrates the output rate per query of the three implementations with respect to a variable number of queries. In the optimal implementation, each query gets enough resources to process all tuples, and therefore, the output rate is not affected by the number of queries. In the naive implementation, the system resources are divided among all queries leading to a decrease in the output rate with the increase in the number of queries. In the optimized solution, each stream subscribes only to a small subset of queries (the stream's query working set) leading to a reduced processing load. Hence, the system resources are utilized efficiently to increase the output rate of each query.

Figure 6.9b illustrates the output rate per query with respect to a variable number of streams. The optimal output rate increases linearly with the increase in the number of streams because each additional stream contributes to the query result. However, the output rate of both the naive and the optimized versions saturate with the increase in the number of streams, yet, with different rates.



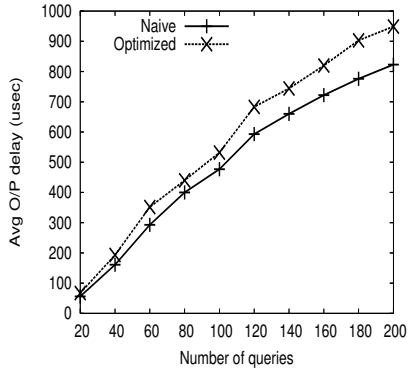
(a)



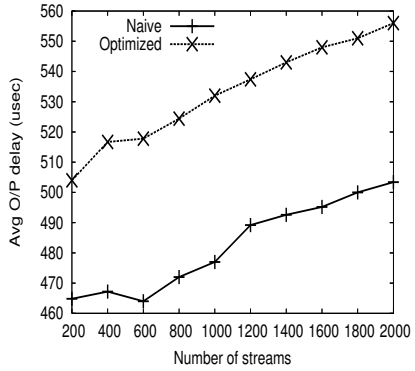
(b)

Figure 6.9. The performance of phenomenon-aware optimizers with respect to the output rate.

Figure 6.10a and Figure 6.10b illustrate the output delay (for the naive and the optimized cases) versus a variable number of streams and a variable number of queries, respectively. The output delay is defined to be the difference between the tuple's timestamp and the time the tuple's effect is apparent in the output. Because the optimized approach has an overhead that is associated with the phenomenon detection and tracking, the output delay of the optimized approach is higher than the output delay of the naive approach. For example, in the case of 100 queries that execute over 1000 streams, the average output delay of the optimized approach is 12% higher than the average output delay of the naive approach.



(a)



(b)

Figure 6.10. The performance of phenomenon-aware optimizers with respect to the output delay.

We investigate the output rate of the optimized solution further and identify the factors that result in a reduction in the output rate compared to the optimal output rate. Consider the case of 100 queries and 1000 streams, the optimized solution triples the output rate of the naive implementation, and meanwhile, the optimized output rate is 30% less than the output rate of the optimal solution. There are three factors that cause a loss in the output of the optimized version: (1) Latency in the phenomenon detection, (2) non-phenomenon or outlier tuples, and (3) random tuple dropping due to buffer overflow. The latency in phenomenon detection is attributed to the time a streaming source takes to persist in contributing

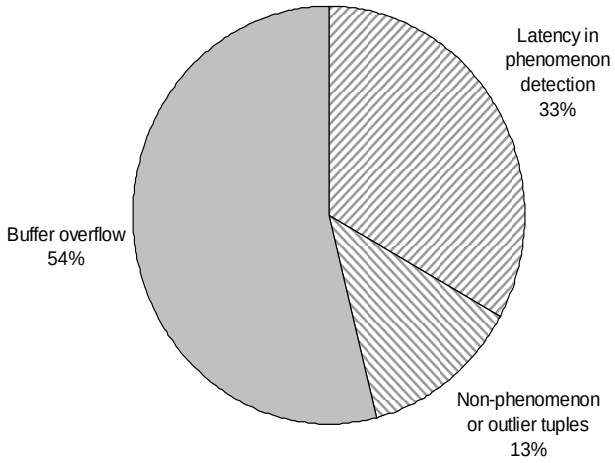
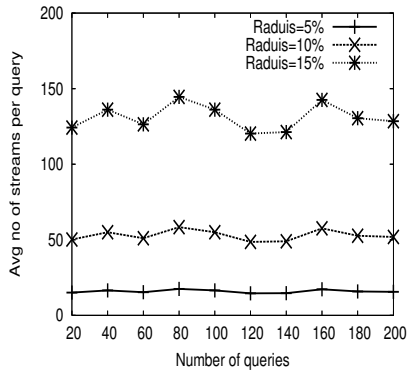
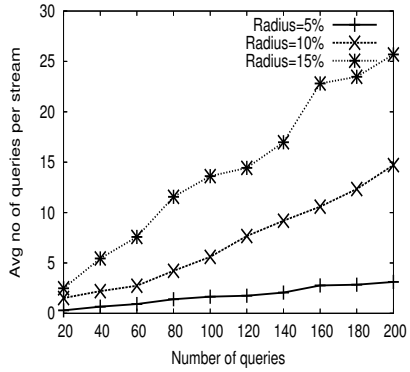


Figure 6.11. The factors of output dropping in the optimized solution.

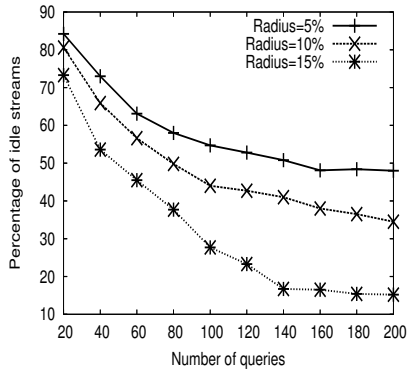
to a phenomenon taking into consideration the time the system takes to pay attention to a newly-developing phenomenon. Let the latency be the time between the first time a streaming source generates a tuple that contributes to a phenomenon and the time the streaming source is considered to be part of that phenomenon. The latency is experimentally found out to be 7.15 seconds on the average. During this warm-up period, the phenomenon-aware optimizer does not view the streaming source as part of the phenomenon and, therefore, the streaming source is not included in the query answer. Non-phenomenon or outlier tuples are tuples that satisfy the query predicates but do not participate in a phenomenon. Hence, these tuples are considered outliers and are not included in the query answer by the phenomenon-aware optimizer. Random tuple dropping occurs because the system is not able to catch up with the arrival rate of the input streams. Hence, tuples overflow out of the system's buffers and their corresponding output is lost. Figure 6.11 illustrates the effect of each factor as a percentage of the total number of dropped tuples in the output (in the case of 100 queries and 1000 streams). The latency in phenomenon detection, the non-phenomenon/outlier tuples, and the tuple dropping due to buffer overflow are responsible for 33%, 13%, and 54%, respectively, of the total number of dropped tuples in the output.



(a)

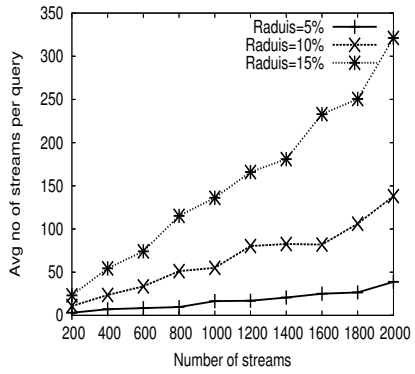


(b)

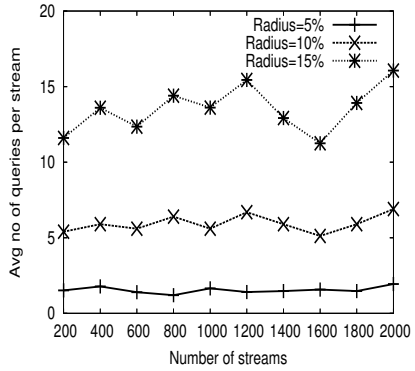


(c)

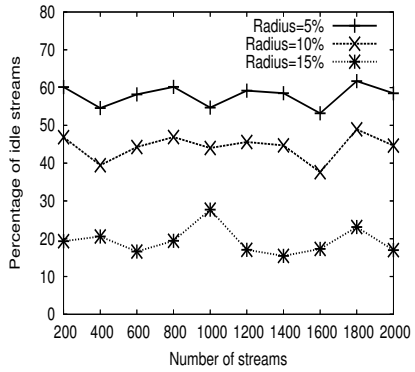
Figure 6.12. The effect of increasing the number of *queries* on the system resources.



(a)



(b)



(c)

Figure 6.13. The effect of increasing the number of *streams* on the system resources.

6.6.2 The System Resources

In the absence of a phenomenon-aware optimizer, every query is deployed over every stream in the system. In this section, we evaluate the amount of savings in system resources that are achieved by a phenomenon-aware optimizer. We measure the average number of streams that subscribe to the same query and, alternatively, the number of queries that are executed on the same stream. We also measure the percentage of idle streams, i.e., streams that subscribe to no queries. Idle streams can be sampled at a lower rate or can be safely turned off for some time. We repeat the experiment for the same data set after we vary the average radius of the search range.

Figure 6.12 measures the effect of increasing the number of queries on the system resources. With the increase in the number of queries, the average number of streams subscribed to the same query is not affected because each query is executed only on streams of interest (Figure 6.12a). However, the average number of queries that are executed on the same stream increases to accommodate the added queries (Figure 6.12b). As we increase the number of queries, queries are spread over various locations of the space and decreases the number of idle streams (Figure 6.12c).

Figure 6.13 measures the effect of increasing the number of streams on the system resources. With the increase in the number of streams, the average number of streams subscribed to a query increases in response to having more streams satisfying the query predicates (Figure 6.13a). However, the average number of queries executed on a stream remains fixed because each stream subscribes only to a subset of interested queries (Figure 6.13b). Consequently, the number of idle streams is not affected (Figure 6.13c). Notice that the increase in the radius of the range query increases the utilization of system resources quadratically.

To quantify the amount of savings in system resources, consider the case of having 100 queries (radius=10% of the space) running on 1000 streams. Each query is executed on 55 data streams out of the 1000 outstanding streams (i.e., 5.5% of the

total number of streams). Also, 44% of the data streams are not fed to the query processor because they have no associated queries.

6.6.3 System's Tuning Parameters

In Section 6.3, we presented several tuning parameters for the phenomenon index, e.g., the *behavior tolerance parameter* (*BTP*), the *safety factor*, μ, and the length of the *sleep period*. Each parameter controls the propagation of updates to the phenomenon index. The best values of these parameters are obtained experimentally by varying the value of one parameter while fixing the others. This tuning process is conducted repeatedly till we converge to the best values of the parameters. Also, the tuning process may be repeated upon changing the domain of underlying data streams.

Figure 6.14 investigates the average output rate per query versus multiple values of *BTP*. As we increase *BTP*, the index is updated less frequently, the update overhead is reduced, and the output rate increases till the optimal output rate is obtained at $BTP = 0.8$. As we continue to increase BTP, the index becomes too lazy to propagate updates in a timely fashion. Hence, the index does not reflect the underlying phenomenon behavior leading to a reduction in the output rate. Other tuning parameters have the same effect on the index performance. As a tuning parameter is varied, the output rate increases till an optimal point is reached where the performance starts to deteriorate afterwards. Based on the experiments, the typical values of *safety factor*, μ, and the length of the *sleep period* are 1.6, 0.7, and 6 seconds, respectively.

6.7 Related Work

The research focus of spatio-temporal data streams has been directed to process continuous queries over data streams that are generated by mobile objects, e.g., [22–26]. Moreover, some attention has been given to provide further analysis of data

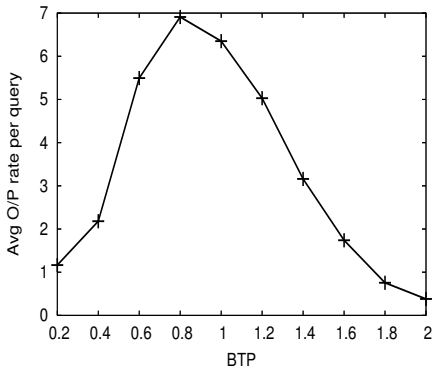


Figure 6.14. The effect of varying the value of BTP.

streams and to detect phenomena as they move in space, e.g., [29–31]. However, there are no DSMSs that employ a phenomenon-aware query optimizer that makes use of detected phenomena to optimize the execution of subsequent user queries.

In this chapter, we index phenomena as they move in space to optimize subsequent user queries. We use one of the recent moving object index structures, the *R-tree with update memo* [61]. However, indexing moving objects has been extensively studied in the literature, e.g., [62, 63] index historical trajectories of moving objects. Other index structures, e.g., [64–67], keep track of the current position of an object as it moves in space. The *TPR*-tree* [68] predicts the future trajectories of moving objects.

Mobility is an important issue in spatio-temporal data streams. Objects as well as queries can be mobile. A large body of literature addresses the execution of mobile queries over mobile objects, e.g., [69–72]. In the context of this chapter, an object generating a data stream can be stationary or, most probably, mobile. We have also mobile phenomena that appear and move in the surrounding environment. Queries are stationary and are deployed over the entire set of registered data streams in the system. However, we *artificially* move queries over regions of their interesting phenomena in response to changes in the monitored environmental conditions.

6.8 Summary

In this chapter, we explored the impact of phenomenon detection and tracking techniques on query optimization inside data stream management systems. Guided by the detected phenomena, we build a *phenomenon index* to track phenomena as they roam the surrounding space. By traversing the phenomenon index, another index is constructed, i.e., the *query index*, to index queries by the regions of their interesting phenomena. To optimize for system resources, we limit each stream to subscribe to a subset of queries (the stream's query working set). By investigating the query index, a query is added to a stream's query working set if the stream falls in one of the query's interesting phenomenon regions. A stream's query working set is updated dynamically as phenomena move in space or as the spatio-temporal properties of the data stream change. Experimental studies that are based on a prototype implementation of the proposed phenomenon-aware query optimizer show that we can achieve up to 70% of the optimal output rate while executing a query on 5.5% of the total number of streams in the system. The performance gains in the output rate are at the expense of a 12% increase in the output delay compared to the non-optimized query processing (in the case of 100 queries executing over 1000 data streams).

7 PHENOMENON-AWARE DATA ACQUISITION IN A SENSOR-NETWORK PLATFORM

In this chapter, we extend our discussion of phenomenon-aware DSMSs to include sensor-network databases. More specifically, we investigate the effect of phenomenon awareness on the acquisition of sensor data. Scalability and energy management issues are crucial for sensor networks. In this chapter, we introduce the *Sharing and PArtitioning of Stream Spectrum (SPASS)* protocol as a phenomenon-driven approach to provide scalability with respect to the number of sensors and to manage the power consumption efficiently. The spectrum of a sensor is the range/distribution of values read by that sensor. Close-by sensors are likely to indulge in the same phenomena. Hence, close-by sensors tend to give similar readings and, consequently, exhibit similar spectra. We propose to combine similar spectra into one global spectrum that is shared by all contributing sensors. Then, the global spectrum is partitioned among the sensors such that each sensor carries out the responsibility of managing a partition of the spectrum. Spectrum sharing and partitioning require continuous coordination to balance the load over the sensors. Experimental results show that the *SPASS* protocol relieves a sensor database system from the burden of data acquisition in large-scale sensor networks and reduces the per-sensor power consumption.

7.1 Background and Motivation

The recent advances in large-scale sensor-network technologies enable the deployment of a massive number of sensors in the surrounding environment. Each sensor consists of a small node with sensing, computing, and communication capabilities. Due to the limited processing capabilities of sensor nodes, the sensor readings are minimally processed at the sensor-network level. Then, the sensor data is trans-

mitted through a multi-hop communication route to a centralized DSMS for further processing. As sensor networks get larger, DSMSs are burdened with the massive amount of data that is streamed out of the sensors. Recent research focuses on sampling [6, 8, 73–75], communication [76–78], and query processing [4, 5, 7, 9, 79] techniques for sensor data. *Scalability* with respect to the size of the sensor network has been a major challenge in these techniques.

Sensors give the ability to monitor environments where the existence of a human being is either tough or dangerous, e.g., habitat monitoring [3]. Due to the nature of the environment, it is expected to perform the maintenance of the sensors only over large periods of time, e.g., on a yearly basis. The lifetime of the battery is crucial from the cost-effectiveness point of view. Hence, *energy* is one of the main resources that needs careful management. In this chapter, we address the effect of phenomenon awareness on the scalability and the energy management challenges in sensor networks.

Sensors are deployed densely in space to increase the reliability of monitoring the surrounding environment. The dense distribution of sensors achieves reliability via redundancy to decrease the likelihood of losing sensor readings. However, much redundancy results in *overhearing* the environmental measurements and, consequently, overloading the data stream management system. Things get worse if sensors indulge in a correlated transmission pattern where sensors transmit the same readings successfully while other readings are not delivered by any sensor. As a result, the system receives duplicates of the same value while losing other values totally.

A crisp observation of the sensor data reveals similarities in the distribution of the sensor readings that are coming from close-by sensors. This fact is understandable because close-by sensors are exposed to the same environmental conditions and are involved in the same phenomena. We refer to the distribution of readings from a sensor as the sensor's *spectrum* (A formal definition of the sensor's *spectrum* is given in Section 7.2). The similarities in the sensors' spectra bear redundant information allowing a wide room for optimizations.

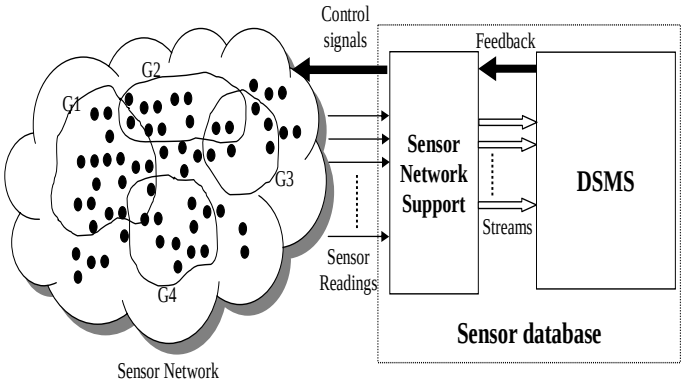


Figure 7.1. Sensor network support layer.

Motivated by similarities in the spectra of sensors, we propose to cluster close-by sensors into groups as illustrated in Figure 7.1. The spectra of sensors in the same group are merged to form one global spectrum. The global spectrum is partitioned among the sensors in the group to assign each sensor the responsibility of transmitting a partition of the spectrum to the sensor database. A sensor gives its own partition the highest priority and processes other partitions based on the availability of resources.

To detect similarities in the sensors' spectra, we push some of the data stream management system functionalities to the sensor network level. In particular, we move the summary manager in part from the data stream management system to the sensor network level to provide early summarization of the sensor data and to discover the associated sensor spectrum. In general, summaries at the core of the data stream management system reduce the processing cost by providing approximate answers in lieu of the exact ones. We propose to shift some of the summarization tasks to the sensor network level to provide an early approximation of the sensor data and to reduce the transmission cost as well as the processing cost. The proposed protocol (*SPASS*) saves energy by reducing the number of transmitted readings across the network. Also, the reduction in the number of transmitted readings has the advantage of reducing the load over the sensor database and achieves scalability.

The contributions of this chapter can be summarized as follows:

1. We extend the phenomenon-awareness capabilities to include sensor-network databases.

2. We propose a phenomenon-based data acquisition protocol, the *Sharing and PArtitioning of Stream Spectrum (SPASS)* protocol, that acquires a *faithful* representation of sensor data and handles energy management and scalability challenges.

3. We implement the proposed *SPASS* protocol inside Nile PhenomenaBase.

4. We provide experimental evidence that the *SPASS* protocol acquires sensor data faithfully and enhances the performance of sensor databases in terms of scalability and power consumption.

The rest of this chapter is organized as follows: Section 7.2 gives the definition of the stream *spectrum* and its properties. Section 7.3 introduces the proposed *SPASS* protocol while Section 7.4 presents a variation of the *SPASS* protocol that is optimized for adaptivity. Experimental results that are based on a prototype implementation of the proposed *SPASS* protocol inside *Nile* are presented in Section 7.5. Section 7.6 highlights related work. Finally, this chapter is summarized in Section 7.7.

7.2 Stream Spectrum and its Properties

The term *spectrum* refers to the distribution of a physical characteristic. The *spectrum* of a certain characteristic is the value ranges over which this characteristic is distributed. For example, the spectrum of the visible light is the continuous frequency ranges of its components. In the context of data streams, we define broadly the stream spectrum as the value ranges from which the stream tuples are drawn.

Before we proceed to a formal definition of the *stream spectrum*, we discuss the underlying structure of sensor databases (Section 7.2.1) and how the stream sum-

maries can be partially pushed to the sensor-network level to help derive the stream spectrum of each sensor (Section 7.2.2). We give a formal definition of the *stream spectrum* and how it can be derived from various summarization techniques in Section 7.2.3. By the end of this section, we highlight the benefits of merging the individual spectra of close-by sensors to form one global shared spectrum. Then, we provide mathematical bounds on the amount of savings that can be obtained by sharing the global stream spectrum (Section 7.2.4).

7.2.1 Sensor Network Support Layer

A data stream management system (*DSMS*) comes at the core of a sensor database system (Figure 7.1). The *DSMS* provides a pipelined query execution of continuous queries over the data streams that are generated by the sensors. We extend DSMSs with an additional layer, namely the *Sensor Network Support Layer (SNSL)*, to support the functionalities of sensor networks. The main purpose of the *Sensor Network Support Layer (SNSL)* is (1) to reduce the processing load at the system's side for the sake of scalability, and (2) to reduce the power consumption at the sensors' side.

The *SNSL* accepts feedback from the *DSMS* about the sensor data and sends control signals to the sensors to control their behavior. The *SNSL* instructs the sensors on how to sample, aggregate, and transmit their readings. As one of the proposed *SNSL* components, the *Sharing and PArtitioning of Stream Spectrum (SPASS)* protocol provides scalable and energy-efficient acquisition of sensor readings that *faithfully* represent the sensor-network data.

7.2.2 Sensor-network Level Summarization

We propose two levels of summarization: sensor-network level summarization and system level summarization. Sensor-network level summarization guides the sensors to the tuples that are worth transmission while system level summarization guides

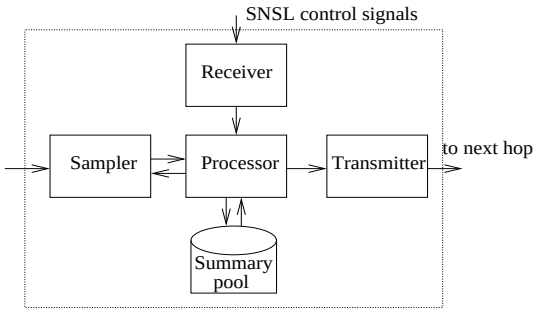


Figure 7.2. Basic components of a sensor.

query processing. In our work, sensor-network summarization is conducted at each sensor to capture the sensor spectrum. Guided by the sensor spectrum, we prioritize the stream tuples for transmission.

Each sensor has four basic components, as illustrated in Figure 7.2. The functionalities of these components can be summarized as follows:

1. The *sampler* has the capability of sensing the surrounding environment.

2. The *receiver* receives control signals from the *sensor network support layer (SNSL)* and forwards these signals to the processing unit.

3. The *processor* performs various tasks based on the *SNSL* signals. For example, the processor instructs the sampler on how to control its sampling rate. Data aggregation and filtration are performed at the processor to reduce data size. In addition, the processor performs the following *SPASS* functions: (a) build summaries over the incoming stream of tuples, (b) generate the stream spectrum from the summaries, (c) share the spectrum among other sensors, and (d) prepare tuples to be sent by the transmitter.

4. The *transmitter* is responsible for the physical transmission of (a) sensor data, (b) sensor spectrum, and (c) information about the sensor's transmission rate. The information about the transmission rate helps the *SNSL* coordinate the partitioning of the sensors' shared spectrum based on the load of each sensor.

7.2.3 Definition of a Stream Spectrum

In this section, we provide a formal definition of the stream spectrum and give examples of how the stream spectrum can be derived from stream summaries. The stream spectrum can be defined as follows:

Definition 7.2.1 *For a data stream S that consists of an infinite tuple sequence $\{x_1, x_2, x_3, \cdots\}$ that arrive at time instants $\{t_1, t_2, t_3, \cdots\}$, respectively, a stream spectrum SS, at time instant τ, is a finite set of stream representatives $R = \{r_1, r_2, \cdots, r_L\}$ such that R is obtained using a summarization function $\phi(x_{\tau-w+1}, x_{\tau-w+2}, \cdots, x_\tau) \rightarrow R$, where w is a sliding time-window. For any stream tuple x, $\exists\ r \in R$ such that $M(x) \rightarrow r$, where M is a mapping function that maps a stream tuple to one of the stream representatives.*

The stream spectrum is generated using a summarization function ϕ that captures the stream behavior over the most recent time-window w and produces a finite set of stream representatives. Notice that the stream spectrum is associated with a time instant τ because the stream spectrum may change with the slide of the summarization function window. A newly incoming tuple in the stream updates the stream summaries and is mapped to one of the stream representatives. All tuples that map to the same representative are processed and transmitted in the same way. A stream representative provides a unified processing and transmission interface for all tuples that map to that representative.

To give an example of how the stream spectrum can be extracted from stream summaries, we first consider *histograms* as our summarization technique. In histograms, the stream representatives are the histogram buckets. The summarization function ϕ updates the bucket frequencies based on the incoming tuples over the last w time-window. The tuple is mapped by the mapping function M to the bucket it falls in ($M(x) \rightarrow bucket(x)$). All the tuples in the same bucket are treated, i.e., processed and transmitted, uniformly.

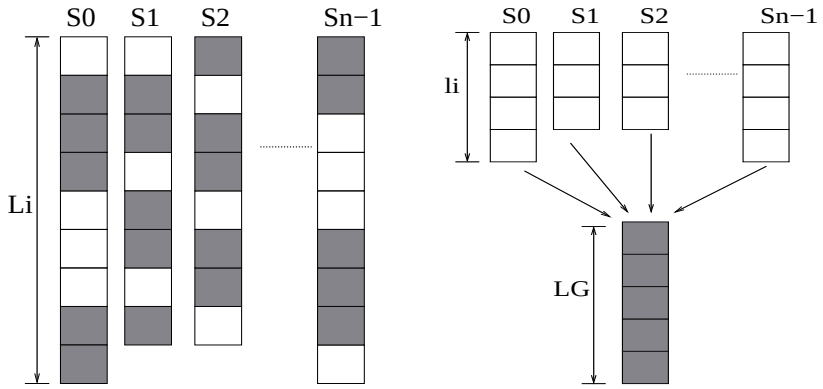


Figure 7.3. Individual sensor spectra versus a global sensor spectrum.

Maintaining the *top-k* list of a stream is another summarization technique. In the *top-k* approach, the stream is summarized using its k most-frequent elements. The summarization function ϕ updates the *top-k* list based on the tuple count over the last w time-window. The stream representatives (R) are the most frequent k elements themselves plus the *NULL* value. The mapping function M maps an element to itself ($M(x) \rightarrow x$) if x is frequent or ignores it otherwise ($M(x) \rightarrow NULL$).

Similarly, the stream spectrum can be extracted from other forms of stream summaries by defining their corresponding ϕ and M functions. The proposed *SPASS* protocol is general and can accommodate various summarization techniques. However, the mathematical analysis and the experimental evaluation assume that the sensor data follows the *Zipfian* distribution and is summarized using the *top-k* list approach as presented in [19].

7.2.4 Global Stream Spectrum

In real life, close-by sensors are exposed to similar environmental conditions. Hence, close-by sensors produce readings that share similar distributions over almost identical value ranges. The closer a sensor to its neighbors, the more correlated its

spectrum to the neighbors' spectra. The *correlation coefficient*, say ρ_{ij}, assesses how much the readings of a sensor (S_i) vary in response to the variation in the readings of another sensor (S_j). Equation 7.1 gives the correlation coefficient between the readings of two sensors, S_i and S_j, where μ and σ are the mean and the standard deviation of the stream tuples, respectively.

$$\rho_{ij} = \frac{E[(X_i - \mu_i)(X_j - \mu_j)]}{\sigma_i \sigma_j} \tag{7.1}$$

A *correlation matrix* is a two-dimensional matrix that records the *correlation coefficient* between every two sensors, S_i and S_j. By observing the correlation matrix of real sensor data, we find out that the correlation matrix contains *ones* along the diagonal because a sensor is fully correlated with itself. Also, the correlation coefficient between two sensors gets higher as they get closer to each other. The high correlation among a group of close-by sensors over time is our target area of optimization.

Figure 7.3a illustrates the spectra of a group of n sensors. Each sensor has a spectrum of length L_i. A gray box represents a value that appears in the spectra of *all* sensors. Figure 7.3b suggests to maintain only one global spectrum of length L_G that is shared by all sensors. The global spectrum accommodates all the items that appear in all sensors. Each sensor sees a spectrum of length l_i that accommodates its private non-shared spectrum elements plus the shared global spectrum.

Combining the common parts of the spectra of various sensors into one global spectrum reduces the overhead of processing the same value at different sites. The *stream compression ratio* (SCR) is the amount of savings achieved by merging the sensors' common spectra into one global spectrum. The (SCR) is given by Equation 7.2. Equation 7.2 calculates the ratio of the global spectrum size plus the sizes of private spectra to the summation of the sizes of the individual non-shared spectra, then subtracts this value from 1 to yield the compression ratio.

$$SCR = 1 - \frac{\sum_{i=1}^{n} l_i + L_G}{\sum_{i=1}^{n} L_i} \tag{7.2}$$

The *stream compression ratio* (SCR) is of great significance because it denotes the amount of processing overhead that can be distributed and balanced among sensors. In the remainder of this section, we give a mathematical bound on the SCR that can be achieved for sensor data that follows the *Zipfian* distribution. We derive the sensors' spectra from summaries that are based on the most frequent item list (the *top-k* list) as discussed in [19]. In this case, the shared spectrum will contain the k most-frequent elements that are common to all sensors (Equation 7.3).

$$L_G = k \tag{7.3}$$

For simplicity, assume that (1) sensors are close enough to each other such that they have a common k most-frequent item list, and that (2) sensors maintain spectra that are of the same length (i.e., $L_i = L_j \ \forall i, j$, and consequently, $l_i = l_j$ because $l_i = L_i - L_G$). The SCR in Equation 7.2 reduces to Equation 7.4.

$$SCR = 1 - \frac{nl + L_G}{nL} \tag{7.4}$$

The technique in [19] captures the k most-frequent items using a list of length L such that the most-frequent item that is *not* captured in the list (element number $L + 1$) occurs with a frequency that is less than $(1 - \epsilon)$ of the frequency of the k most-frequent element (Equation 7.5).

$$f_{L+1} < (1 - \epsilon) f_k \tag{7.5}$$

To capture the k most-frequent elements, we have to maintain a list of length $L = O(k)$. In particular, L is given by Equation 7.6, where z is the *Zipfian* distribution parameter.

$$L = \frac{k}{(1 - \epsilon^{\frac{1}{z}})} \tag{7.6}$$

The non-shared part of the stream spectrum at each sensor, l, is given by subtracting the shared spectrum length from the original spectrum length as in Equation 7.7.

$$l = L - L_G = \frac{k}{(1 - \epsilon^{\frac{1}{z}})} - k \tag{7.7}$$

Substitute for both L (Equation 7.6) and l (Equation 7.7) in Equation 7.4, we obtain Equation 7.8 that expresses the *stream compression ratio* in terms of the number of sensors (n), the summarization technique parameter (ϵ), and the *Zipfian* distribution parameter (z).

$$SCR = \frac{n-1}{n}(1 - \epsilon)^{\frac{1}{z}} \tag{7.8}$$

Notice that the SCR that is obtained in practice can be less than the value of Equation 7.8 because sensors may not share all the *top-k* elements during the life time of the experiment.

7.3 The SPASS Protocol

The *SPASS* protocol coordinates the sharing and the partitioning of the global spectrum among a cluster of close-by sensors. Sensors may be clustered by the nature of the problem. For example, sensors that read the temperature of the same room are exposed to similar conditions and are considered one cluster. Otherwise, clustering techniques that aim at minimizing the power consumption due to packet routing among sensors are deployed to cluster the sensors, e.g., [80, 81]. In our work, we are interested in sharing the spectrum of sensors that fall within the same cluster. This makes the clustering problem orthogonal to our work. Any clustering technique can be applied as a preprocessing phase to our protocol. For the sake of implementation, we use the *HEED* clustering technique as presented in [10].

Clustering techniques select a dedicated sensor among the sensors that fall in the same cluster to be the cluster head. All communication messages that go out of a sensor are forwarded to the cluster head as their next hop. The cluster head is

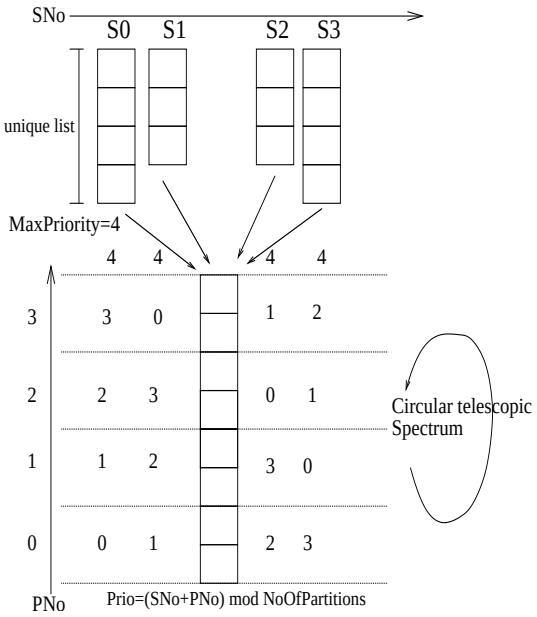


Figure 7.4. An example circular telescopic spectrum for four sensors.

responsible for the coordination among the sensors in its cluster as well as routing their messages to the centralized sensor database.

The goal of the *SPASS* protocol is to maintain a global spectrum over each cluster of sensors as shown in Figure 7.4. Each sensor maintains a small list of its unique items that are not shared among other cluster members. This portion of the spectrum is given the highest priority, *MaxPriority*, because no other sensors are expected to transmit these items. The shared spectrum is divided into n partitions of equal sizes (i.e., shares) where n is the number of sensors in the cluster. Each sensor takes the responsibility of transmitting one share. The sensor gives its own share the next highest priority, *MaxPriority-1*, and reduces the priority of other sensors' shares by one in a circular fashion. The priority of partition PNo at sensor SNo is given by Equation 7.9.

$$P = (SNo + PNo) \quad mod \quad n \tag{7.9}$$

We refer to the global spectrum as presented in Figure 7.4 by the term *circular telescopic spectrum* because each sensor processes its own share and extends the processing *telescopically* to other partitions based on the availability of time. The *scope* of a sensor is the average depth of items being transmitted by that sensor. The depth of an item is the difference between the item's index in the shared spectrum vector and the index of the beginning of the sensor's share, given the circular direction of movement. The *scope* parameter of a sensor provides information about how much of the global spectrum is covered by that sensor.

The *SPASS* protocol is divided into two major components: (1) the *summarize and transmit* procedure that is placed at all sensor nodes to build their individual spectrum and to control the transmission of their sensor readings, and (2) the *share and partition* procedure that is placed at the cluster head to form the shared global spectrum of the cluster and to coordinate the partitioning of the global spectrum among sensors.

Figure 7.5 summarizes the processing at each sensor node. A sensor either (1) generates a sensor reading or (2) receives a global spectrum from the cluster head. Upon receiving a new reading, the sensor uses this reading to update the summaries (Step 1) and update the local individual spectrum based on the change in the summaries (Step 2). If the distance between the new spectrum and the spectrum at the cluster head exceeds α, a fresh copy of the spectrum needs to be transmitted to the cluster head (Step 3). The sensor probes the spectrum to find the partition where the sensor reading falls and, consequently, retrieves its transmission priority (Step 4). The sensor places the reading in the transmission priority queue to compete for the transmission bandwidth based on the priority (Step 5). Upon receiving a new version of the global spectrum, the sensor updates the transmission priorities based on the notion of the circular telescopic spectrum.

Figure 7.6 describes the role of the cluster head in sharing and partitioning the global spectrum. The cluster head receives from each sensor either (1) a sensor reading or (2) a sensor's local spectrum. When the cluster head receives a sensor

procedure *Summarize and Transmit*

Input: *(1) a stream of sensor readings x_1, x_2, x_3, $\cdots$ and (2) A global spectrum (GS)*

Output: *continuously maintain (1) the sensor individual spectrum and (2) the priority transmission queue.*

Description:

Upon receiving a sensor reading x_i

1. *UpdateSummaries(x_i).*
2. *UpdateSpectrum().*
3. *if $|NewSpectrum - OldSpectrum| > \alpha$ then*
   transmit the spectrum to the cluster head.
4. *P=GetPrio(x_i)*
5. *Transmit(x_i,P)*

Upon receiving the global spectrum
update the circular teslescopic spectrum

Figure 7.5. The SPASS protocol at each sensor node.

reading, it forwards the reading to the next hop on its way to the sensor database. When the cluster head receives a sensor's spectrum, it merges this spectrum with the spectra of other sensors in the cluster to compute the global spectrum (GS) (Step 1). Merging the spectrum is simply to find the common items in *all* sensors' spectra. Then, the common items are partitioned into n partitions of equal sizes (Step 2). Finally, the cluster head updates each sensor with a copy of the shared item list (Step 3).

7.4 The SPASS+ Protocol: An Adaptive Version of the Protocol

The *SPASS+* protocol promotes adaptivity by balancing the load among sensors in the same cluster based on their relative loads. In this section, we define the *sensor load* and develop an adaptive technique to partition the global spectrum dynamically

procedure *Share and Partition*

Input: *Given a cluster of n sensors S_0, S_2, $\cdots$, S_{n-1}. Each sensor generates: (1) a stream of readings and (2) an individual sensor spectrum SS_i.*

Output: *(1) the global spectrum of the cluster and (2) the share of each individual sensor.*

Description:

Upon receiving a sensor reading

forward the sensor reading to next hop

Upon receiving a sensor spectrum SS_i

1. *MergeSpectrum(GS, SS_i)*
2. *PartitionSpectrum(GS)*
3. *for i=0 to n − 1 SendtoSensor(S_i, GS)*

Figure 7.6. The SPASS protocol at the cluster head.

among sensors. Each sensor is assigned a share of the spectrum that is inversely proportional to the load over that sensor. We define the sensor load as follows:

Definition 7.4.1 *The sensor load is defined to be the total time required to transmit all items that are queued in the sensor's buffer.*

Two major parameters formulate the sensor load: (1) the throughput, which refers to the achieved transmission rate in terms of the number of transmitted messages per second and (2) the queue length, which specifies how many items are still queuing in the buffer waiting for transmission. The sensor load is computed as follows:

$$Load = \frac{Queuelength}{throughput} \tag{7.10}$$

Let Ld_i be the load at sensor S_i. The share of sensor S_i in the global spectrum is calculated at the cluster head using Equation 7.11. Notice that the more the sensor is loaded, the smaller the share it gets. To achieve this adaptive behavior, each sensor is required to report its load periodically to the cluster head. The cluster head keeps

track of the load over each sensor in its cluster and continuously repartitions the spectrum among sensors to balance the load within the cluster.

$$Share_i = \frac{\frac{1}{Ld_i}}{\sum_{j=0}^{n-1} \frac{1}{Ld_j}} \times SpectrumLength \tag{7.11}$$

SPASS+ requires two major modifications over the *SPASS* protocol. In the *Summarize and Transmit* procedure (Figure 7.5), each sensor transmits information about its current load periodically to the cluster head. In the *Share and Partition* procedure (Figure 7.6), the *Partition Spectrum* function (Step 2) is modified to divide the spectrum into non-equal partitions. The length of each partition is given by Equation 7.11.

7.5 Experiments

In this section, we perform an experimental study to explore the performance of the proposed *SPASS* protocol. Two sets of experiments are conducted. The first set of experiments (Section 7.5.1) addresses the performance of the *SPASS* protocol in terms of scalability and power consumption. The second set of experiments (Section 7.5.2) is concerned with the internal parameters of *SPASS*, e.g., the *correlation* and the *spectrum compression ratio*, with respect to different cluster sizes. We study three protocols:

1. *SIMPLE*, where each sensor simply transmits, based on the allowed bandwidth, a uniform sample of its own readings to the sensor database.

2. *SPASS*, where the *SPASS* protocol is deployed to manage the transmission of data, as described in Section 7.3.

3. *SPASS+*, where the *SPASS* protocol is optimized for adaptivity, as described in Section 7.4.

Our major measure of performance is *Hist-MSE* (Equation 7.12) that represents the *mean square error* between the global histogram of all the generated sensor data

(at the sensor side) and the global histogram of the transmitted sensor data (at the system side) after they are normalized by the data set size. A global histogram includes the streams coming from all sensors to give a global view of the whole sensor network. We do not care about the individual histogram of each sensor. Instead, we care about the collaboration of sensors to transmit a faithful view of the entire sensor field being investigated.

Let H_1 be the histogram of the original data and let H_2 be the histogram of the transmitted data. Each histogram is an equi-width histogram of n intervals (n is set to 100). H_1 is divided into $H_{11}, H_{12}, \cdots, H_{1n}$ and H_2 is divided into $H_{21}, H_{22}, \cdots, H_{2n}$. Let N_1 be the size of the original data set and let N_2 be the size of the transmitted data set ($N1 \geq N2$). *Hist-MSE* is defined as follows:

$$\textit{Hist-MSE} = \frac{\sum_{i=1}^{n} (\frac{h_{1i}}{N_1} - \frac{h_{2i}}{N_2})^2}{n} \tag{7.12}$$

In this experimental study, we use the Nile PhenomenaBase simulated sensor platform (as described in Section 2.5). In contrast to the hardware limitations of the Nile PhenomenaBase real-sensor platform, the simulated platform gives the flexibility of pushing some processing functionality at the sensor level. Unless mentioned otherwise, the parameters of the simulated setup are as follows. We maintain 1000 sensors uniformly distributed in the space. Each sensor generates a stream of 10,000 tuples where the tuple values follow the Zipfian distribution. The sensors are grouped into clusters, where each cluster is of size 5. The interarrival time of sensor data follows an exponential distribution with an average of one second. To model the scarcity of resources, the sensor database is capable of processing a bandwidth that is up to 200 tuples per second. The bandwidth is shared fairly among the 1000 sensors, which means that each sensor is granted to transmit a bandwidth that is up to 0.2 samples every second (i.e., the allowed bandwidth carries around 20% of the sensor readings). In other words, instead of reading a value per sensor every second, we read a value per cluster. All the experiments in this section are based on a real implementation of the *SPASS* protocol inside *Nile PhenomenaBase*. The *Nile PhenomenaBase* en-

gine executes on a machine with Intel Pentium IV, CPU 2.4GHZ with 512MB RAM running Windows XP.

7.5.1 Scalability and Power Consumption

In this section, we study the performance of the proposed *SPASS* protocol with respect to the *Hist-MSE* measure of performance under various conditions of the sensor network. We provide an experimental evidence that the *SPASS* protocol: (1) reduces the power consumption of the sensors, (2) is scalable in terms of the stream rates, and (3) is scalable in terms of the size of the sensor network.

The sensor's bandwidth denotes the maximum number of transmitted messages per unit time. The energy consumption of a sensor decreases with the decrease in its utilized bandwidth. In Figure 7.7, we vary the total system bandwidth (which is shared among the 1000 sensors) from 100 to 1000 tuples per second and compare the performance of the *SPASS* protocol, its *SPASS+* variation, and the *SIMPLE* protocol. For the same bandwidth, the *SPASS* protocol gives a better representation of the sensors' readings as indicated by the *Hist-MSE* measure of performance. The optimized *SPASS+* protocol gives a better *Hist-MSE* over the *SPASS* protocol because of its capability to balance the load among sensors dynamically. The *SPASS+* protocol reduces the *Hist-MSE* by up to 70% over the *SIMPLE* protocol and by up to 55% over the *SPASS* protocol (at 100 samples per second bandwidth). Notice that as we increase the bandwidth to 1000 samples per second, each sensor transmits its readings completely to the sensor database and *Hist-MSE* drops to zero.

In Figure 7.8, we control the stream rate by varying its average tuple interarrival time and measure *Hist-MSE* of various protocols. As we increase the average interarrival time, the system load decreases and *Hist-MSE* drops till it reaches zero. Increasing the stream's average interarrival time has a similar effect to increasing the bandwidth because an increased bandwidth implies more system resources while increasing the interarrival time implies less system load.

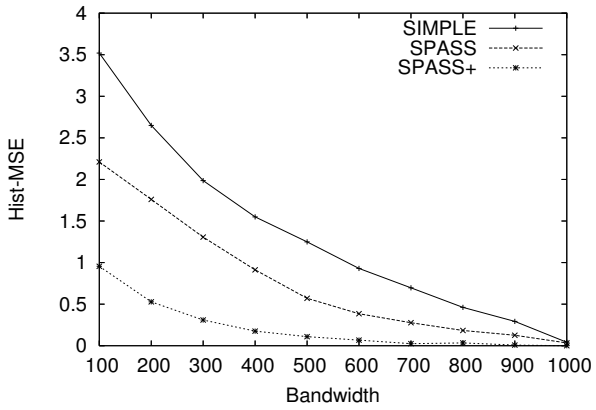


Figure 7.7. The effect of the allowed bandwidth.

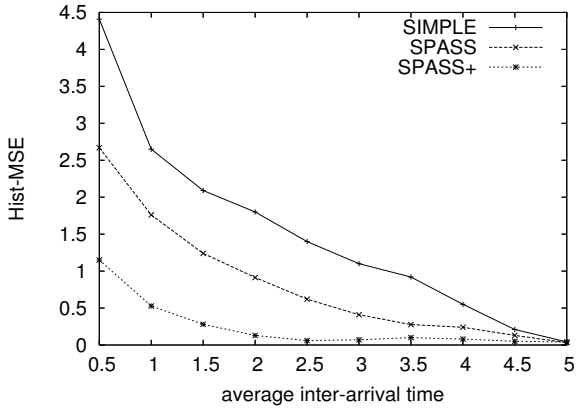


Figure 7.8. The effect of the stream average interarrival time.

Figure 7.9 illustrates the performance of the *SPASS* protocol under various sensor-network sizes. The size of the sensor network is expressed in terms of the number of sensors in the sensor network. We vary the number of sensors in the sensor network from 200 to 2000 sensors. In terms of *Hist-MSE*, the *SPASS* protocol gives a better performance over the *SIMPLE* protocol while its *SPASS+* variation is still capable of providing further reduction in the *Hist-MSE*. As the number of sensors increases, the performance gain of the *SPASS* protocol and its *SPASS+* variation becomes more significant. The *SPASS+* protocol reduces the *Hist-MSE* by up to 65% over

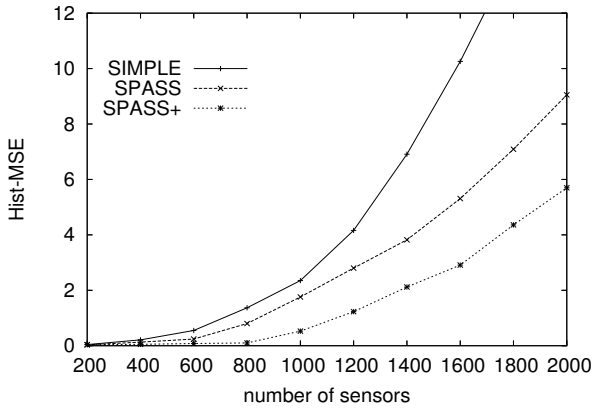


Figure 7.9. The effect of the number of sensors.

the *SIMPLE* protocol and by up to 35% over the *SPASS* protocol (for a network of size 2000 sensors).

7.5.2 Cluster Size

In this section, we study the effect of the cluster size on two internal parameters of the protocol, the average *correlation coefficient* between sensor pairs (ρ) and the *spectrum compression ratio* (SCR). The *correlation coefficient* assesses the similarities in the sensors' spectra while the *spectrum compression ratio* estimates how much saving can be obtained by combining their spectra into one global spectrum. We vary the cluster size from 1, which means no clustering is in effect, to 10 sensors per cluster. The *correlation coefficient* (ρ) and the *spectrum compression ratio* (SCR) are calculated offline based on the average of (ρ) and (SCR) in a one-minute sliding window over the sensor data.

Figure 7.10 gives the effect of the cluster size on the *correlation coefficient* and the *spectrum compression ratio* parameters. The maximum value of the *correlation coefficient* is one, which corresponds to full correlation among sensors. A cluster of size one is fully correlated because one sensor is 100% correlated with itself. As

we increase the cluster size, fewer items tend to be shared among *all* streams and, consequently, the *correlation coefficient* decreases.

The *spectrum compression ratio* (SCR) represents the ratio of the reduction in size of the global sensor spectrum relative to the summation of the sizes of individual spectra. With the increase in the cluster size, the global spectrum benefits from the shared items among individual spectra. The size of the global spectrum gets reduced relative to the summation of the sizes of individual spectra. The size reduction in the global spectrum affects SCR positively. However, as we keep increasing the cluster size, the number of shared items decreases, the size of the global spectrum increases, and SCR is affected negatively by the increase in the global spectrum size. The best SCR equals to 0.645 and is achieved for a cluster of size 5 (Figure 7.10).

Hist-MSE is measured for various cluster sizes in Figure 7.11. The *SPASS* protocol and its *SPASS+* variation give the same *Hist-MSE* as the *SIMPLE* protocol for a cluster of size 1 (no clustering). *Hist-MSE* decreases till we reach a cluster of size 5, then increases again with the increase in the cluster size. This behavior is accounted for by the behavior of *SCR*. For large cluster sizes, the benefit of sharing the sensors' spectrum is ruined by the overhead of maintaining a large global spectrum. A good tuning of the cluster size is the one that has the best *SCR*.

7.6 Related Work

Sensors are battery-equipped devices that are capable of sampling, processing, and transmitting readings from the surrounding environment.Various techniques have been proposed to save the sensor battery life at the *sampling*, *processing*, and *transmission* phases. In the remainder of this section, we give a brief overview of these techniques.

Research has been conducted to reduce the sampling rate, i.e., the *sampling power*, of the sensors. Statistical models have been utilized recently in [6] to provide estimates of the sensors' readings and to assess the uncertainty of these estimates.

Cluster size	ρ	**SCR**
1	1	0
2	0.851	0.285
3	0.68	0.42
4	0.58	0.54
5	0.483	*(0.645)*
6	0.382	0.62
7	0.325	0.546
8	0.275	0.428
9	0.24	0.24
10	0.218	0.078

Figure 7.10. The effect of cluster size on the internal paremeters of SPASS.

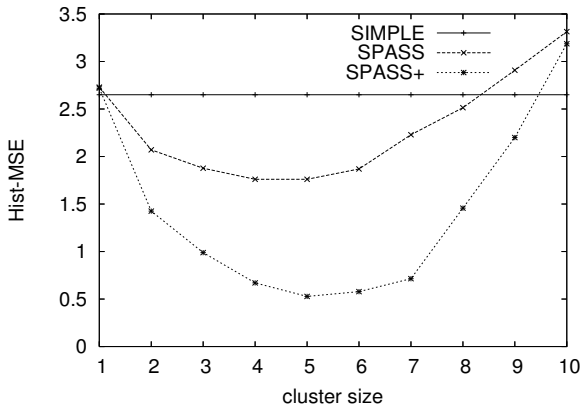


Figure 7.11. The effect of cluster size on the Hist-MSE.

A fresh sample is acquired from the sensors that exhibit high levels of uncertainty to refine their estimates. The *SPASS* protocol addresses a similar problem but without the requirement of a statistical model.

A framework to support an acquisitional query language is proposed in [8]. The proposed acquisitional query processor decides which sensors to query and how often

to sample from each sensor. The work in [74] suggests the use of quality-aware samplers to regulate the data rate at various levels of the system. Some work, e.g., [47], extends the traditional reservoir sampling [82] to fit in the streaming environment.

In-network processing is carried over at the sensor nodes to reduce the size of the data transmitted to the sensor database [9]. Data aggregation, e.g., max, min, and average, collapses a set of readings to one representative. Notice that our work aims at transmitting the actual sensor readings without performing any data aggregation. The work in [5] proposes approximate in-network aggregation using sketches. Sensors are clustered into groups and one member of the group, the *cluster head*, is responsible for collecting and managing the data of its cluster members. The cluster head is elected based on energy requirements as in [10].

The transmission energy is conserved by techniques that configure the network topology dynamically [83, 84]. In these techniques, sensor nodes exchange messages among each other to acquire knowledge about their locations. Nodes are self-organized based on the acquired location information to reduce the communication cost. The routing decisions among nodes optimize power consumption [76–78].

7.7 Summary

In this chapter, we addressed the scalability and energy management issues in sensor networks. We introduced the *Sharing and PArtitioning of Stream Spectrum (SPASS)* protocol as a part of the *Sensor Network Support Layer (SNSL)*. The *SNSL* extends the functionalities of data stream management systems to support sensor networks.

We defined the spectrum of a sensor to be the distribution of values that are read by that sensor. Close-by sensors generate similar spectra because they are exposed to the same phenomena. In *SPASS*, we proposed to group close-by sensors and to combine their spectra into one global spectrum that is shared among all sensors in the group. The global spectrum is partitioned among sensors such that each sensor

transmits the data of its partition with a higher priority. Spectrum partitioning is continuously coordinated to balance the load over the sensor network in the *SPASS+* adaptive version of the protocol. According to the *Histogram Mean Square Error (Hist-MSE)* measure of performance, *SPASS+* achieves up to 70% improvement over the *SIMPLE* protocol for the same level of power consumption.

8 PHENOMENON DETECTION AND TRACKING IN A SENSOR-NETWORK PLATFORM USING RELEVANCE FEEDBACK

To increase the in-network processing capabilities of phenomenon-aware DSMS over sensor networks, we push further functionalities from the centralized DSMS to the sensor-network level. In this chapter, we address the joining phase of the PDT-module and introduce the *SNJoin* operator. SNJoin allows the join operation to be performed at the sensor level. SNJoin integrates query processing with a relevance feedback mechanism to prune the sensors to be probed to only those that are relevant to the join output. 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.

8.1 Background and Motivation

In Chapter 5, we address the demands of the join operation in large-scale dynamically-configured streaming environments through the notion of variable-arity join (VAJoin). However, in a sensor-network setup, if the join operation requires all sensors to transmit their readings to a centralized sink node, the sink node will be a bottleneck, especially 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 [5,9,85]. In this chapter, we present *SNJoin*, a distributed variant of the VA-Join operator that shifts the join operation from the sink node to the sensor-network level.

In this chapter, we consider the sensor network setup of Nile PhenomenaBase. As illustrated in Figure 8.1, the sensor platform of Nile PhenomenaBase is an ad-

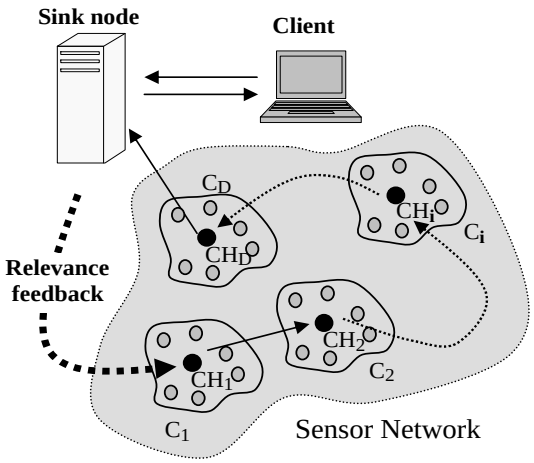


Figure 8.1. The sensor platform.

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

Several techniques can be used to dynamically configure the network topology, e.g., [10,80,81]. 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 Figure 8.1). 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.

The basic steps of the algorithm that implements the proposed *SNJoin* algorithm are as follows:

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 (as described in Chapter 5) where the size of the join output depends on the number of joining sensors.

Step3. A distributed processing phase is initiated by cluster heads (Section 8.2). 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 8.3).

The contributions of this chapter can be summarized as follows:

1. We enhance phenomenon detection and tracking techniques with the distributed processing capabilities of SNJoin that performs the join operation at the sensor-network level.

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

3. We provide an analytical study and an experimental study that is based on a prototype implementation of *SNJoin* inside Nile PhenomenaBase to prove its efficiency both in terms of the number of detected phenomena updates and the output delay.

The remainder of this chapter is organized as follows: Section 8.2 presents *SNJoin* and its distributed processing capabilities. Section 8.3 describes the relevance feed-

back mechanism of *SNJoin*. Section 8.4 gives a mathematical analysis of various join techniques while Section 8.5 provides an experimental study of these techniques' performance. Finally, Section 8.6 summarizes the chapter.

8.2 Distributed Processing in SNJoin

As illustrated in Figure 8.1, 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 8.2 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 feedback 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,

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 8.2. The distributed SNJoin algorithm.

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.

8.2.1 Early, Late and Out-of-order Arrival

Tuples are timestamped at their generating sources and are transmitted to the sink node over the network. 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 value simultaneously.

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 ϵ be 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). The 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 incoming 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 the tuples that has a timestamp 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., retrieve 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 cluster 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 traveled 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 generating the join output.

8.3 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 mechanism for 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 8.3.1 *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 8.2) 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 8.3.2 *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 $Pr\{CH_i \in RPS\} = \frac{RFBM[H(\hat{t}), CH_i]}{\sum_{i=1}^{D} RFBM[H(\hat{t}), CH_i]}$.*

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 8.3. Processing of relevance feedback.

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 in *RFBM* change dynamically with the arrival of relevance feedback notes based on the following equation:

$$RFBM[H(\hat{t}), CH_i] = RFBM[H(\hat{t}), CH_i] - \frac{\sum_{j=1}^{k} w_j}{k} + w_i \tag{8.1}$$

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 8.3. 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.

8.4 Mathematical Analysis

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 partial join result 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 of the VAJoin (as described in Chapter 5). Assume that the number of sensors in a cluster j is k_j and the percentage of joining streams is k'_j. Then, from the VAJoin analysis in Chapter 5, 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 to 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 feedback that will reduce 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.

8.5 Experiments

In this Section, we study the distributed execution of *SNJoin* over clusters of *uniformly-distributed* sensors in space. We use the Nile PhenomenaBase simulated sensor platform (as described in Section 2.5). Clusters of sensors are obtained using

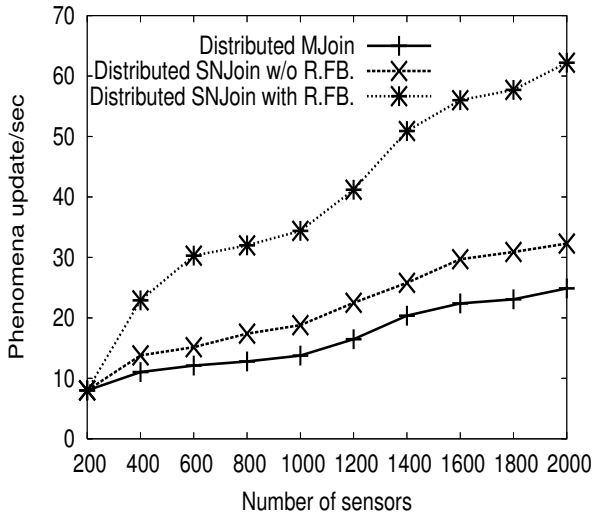


Figure 8.4. The effect of distributed query processing.

a simulation of the *HEED* clustering technique [10] 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 [86]. Cluster heads receive the sensor readings of their cluster members, perform the join operation, and communicate with other cluster heads to perform remote probes. Figure 8.4 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 8.4, 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).

	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 8.5. The effect of relevance feedback.

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 2000 sensors). Figure 8.5 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%.

8.6 Summary

In this chapter, 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 prototype implementation of the join operators inside *Nile PhenomenaBase* show the scalability of *SNJoin*. Once *SNJoin* is adopted by PhenomenaBases, the number of detected phenomena updates is increased while the output delay is reduced.

9 CONCLUSIONS AND FUTURE WORK

In this dissertation, we empowered DSMSs with phenomenon-awareness capabilities. Phenomenon awareness bridges the gap between user-defined queries and the data patterns that appear in the underlying streaming sources. A phenomenon is a group of streams that persist to exhibit *similar behavior* over time. Phenomenon-aware DSMS (or PhenomenaBases) are databases of the phenomena that develop in the streaming environment. Phenomenon-aware DSMS are tuned for large-scale streaming environments where it is challenging to discover interesting patterns among a huge number of, possibly high-rate, data streams. In a phenomenon-aware DSMS, the knowledge about detected phenomena in the surrounding environment guides the query processing to regions of interest in the streaming environment. In this research, we considered phenomenon awareness over two different setups. In the first setup, we are concerned with a centralized platform, where all stream readings are transmitted to a single sink node that is running a phenomenon-aware DSMS. In the second setup, we address phenomenon awareness in a sensor-network platform, where we shift some of the processing functionalities from the centralized DSMS to the sensor-network level. Throughout the chapters of this dissertation, we explore the effect of phenomenon-awareness on various components of DSMSs. For example, we study the effect of phenomenon-awareness on the parser and its extended SQL syntax, the stream monitor and its load shedder subcomponent, the query executer, the query optimizer, and the data acquisition controller.

9.1 Summary of Contributions

To achieve a flexible and efficient framework for phenomenon-awareness, this dissertation introduces several contributions. First, we provide a concrete definition

for the phenomenon and explore various notions of similarity among streams' behavior. Moreover, we formalize several parameters in the phenomenon definition. These parameters include the persistency, the spread, and the time span of the phenomenon. Based on the phenomenon definition, we extend the SQL language with the ability to register various types of phenomena in the system and to interact with detected phenomena. Consequently, we extend the architecture of DSMSs with additional components to handle phenomenon awareness. Meanwhile, we leverage the functionality of already existing components to cope with the concept of phenomena.

The second contribution of this dissertation is the scalable techniques for phenomenon detection and tracking. To be scalable with respect to the number of input data streams, to the stream rates, and to the number of detectable phenomena, we introduce a preference-based load shedder. This preference-based load shedder drops portions of the input streams that are less likely to participate in desirable phenomena, as specified in the user's preference. Moreover, we present a variable-arity join operator that detects similarity among streaming sources and, at the same time, conserves system resources during the costly join operation. The basic idea is to include in the join operation a variable number of streaming sources, i.e., only the streaming sources that are likely to join.

We introduce the adaptive phenomenon-aware optimizer as our third major contribution in this dissertation. The phenomenon-aware optimizer optimizes the execution of user-defined queries dynamically based on the knowledge of phenomena in space. The phenomenon-aware optimizer justifies the cost of running phenomenon detection and tracking modules continuously at the background of the system by guiding the execution of a query to phenomenon regions that are expected to satisfy the query predicates.

Finally, we consider phenomenon detection and tracking in a sensor-network platform that features in-network query processing. Sensor readings are transmitted to a sink node through a multi-hop route where sensor readings undergo *en-route* processing before they reach their final destination. In particular, we address phenomenon-

guided sensor data acquisition and sensor network join with relevance feedback. In phenomenon-guided data acquisition, sensor nodes that are within the same phenomenon collaborate in transmitting the sensor data to the sink node such that the overall transmission cost is reduced. In sensor network join, relevance feedback is computed based on the join output to tune future multi-hop join probing sequences. Relevance feedback includes sensors that generate similar data readings in the same probing sequence with high probability. To demonstrate the capabilities of phenomenon-aware DSMSs, we built the *Nile PhenomenaBase* prototype system, where we provided an experimental evidence for the performance gains of the proposed framework supported by mathematical verification wherever applicable.

9.2 Future Extensions

This dissertation triggers several directions for future research. This section highlights three directions and shows how these directions fit under the umbrella of phenomenon detection and tracking framework.

9.2.1 Detection and Tracking of Non-discrete Phenomena

In this dissertation, we focused on discrete phenomena where the notion of similarity among streams' readings reduces to equality. Then, we investigated the effect of discrete phenomenon awareness on various components of a DSMS. However, non-discrete phenomena that are based on general notions of similarity are apparent in many applications. For example, humidity, temperature, and light intensity readings are usually drawn from a continuous domain. Approximating non-discrete phenomena with discrete ones limits the similarity notions to equality and reduces the output accuracy.

To address non-discrete phenomena, phenomenon-aware DSMSs need to be revisited from three angles: (1) non-discrete phenomenon registration, (2) non-discrete phenomenon detection and tracking, and (3) query optimization based on non-

discrete phenomena. In Chapter 2, we introduced an extended SQL syntax that is capable of registering both discrete and non-discrete phenomenon in the system. Throughout the phenomenon detection and tracking process, we make use of an equality join operator that is followed by a group-by operator to detect equality among stream readings. To extend the similarity notion beyond equality, the equality join needs to be replaced by the more general similarity join operator [87, 88]. Also, the group-by operator should have the ability to tolerate the similarity (i.e., non-equality) in the join values.

On the query optimizer side, discrete phenomena guide the query execution to streaming sources that are likely to satisfy the query predicates. When streaming source S_i satisfies the predicates of a query and, meanwhile, streaming source S_i is in the same phenomenon as streaming source S_j, then streaming source S_j is likely to satisfy the query predicates as well. In the general paradigm of non-discrete phenomena, streaming source S_i is similar in behavior to streaming source S_j according to a similarity function F (i.e., $S_i = F(S_j)$). Given the value of S_i, the optimizer can get a sense of what the expected values of streaming source S_j would be (i.e., $S_j = F^{-1}(S_i)$) and, hence, decide whether streaming source S_j is of interest to the standing query or not. We expect that combining both discrete and non-discrete phenomena in the same system strengthens the phenomenon awareness and enhances the practicality of the model.

9.2.2 Statistical PDT Techniques

Statistical models have been utilized inside DSMSs to efficiently guide the data acquisition and to process several types of queries efficiently (e.g., aggregation queries). Examples of research efforts that address statistical models over data streams include [89–91]. A statistical model can be initially given as input to the system based on the knowledge about the surrounding environment. Then, the model is enhanced over time as the system processes portions of the incoming data streams.

In this dissertation, we use a simple self-trained statistical model, i.e., the relevance feedback matrix (*RFBM*), to guide the join probing sequence to relevant cluster heads. The RFBM is updated based on the past join output tuples and is used to optimize future join probes. Replacing the simple RFBM with a full-fledged statistical model would capture similarities in the stream distributions efficiently and would lead to better phenomenon detection results.

9.2.3 Phenomenon-aware Query Plan Reorganization

Query optimization inside a DSMS is a continuous process that dynamically adjusts the execution of user queries at run time to adapt to changes in the nature of the incoming data streams. Several optimization approaches reorganize the query plan at run time to obtain the query plan that best fits to the data stream distribution at a certain point of time. Then, the query plan is reorganized in response to changes in the data streams' underlying distributions. Examples of current approaches in query plan reorganization over data streams include [58, 92–94].

In this dissertation, the proposed phenomenon-aware query optimizer continuously updates the query deployment map and associates each query with a set of interesting phenomena at run time. However, each query is represented by a single query plan that is generated at compilation time. Combining phenomenon-aware query deployment with run-time query plan reorganization would result in a full-fledged stream query optimizer.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Suman Srinivasan, Haniph Latchman, John Shea, Tan Wong, and Janice McNair. Airborne traffic surveillance systems: video surveillance of highway traffic. In *the ACM international workshop on Video surveillance & sensor networks*, 2004.

[2] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Stream window join: Tracking moving objects in sensor-network databases. In *SSDBM*, pages 75–84, 2003.

[3] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan M. Mainwaring, and Deborah Estrin. Habitat monitoring with sensor networks. *Commun. ACM*, 47(6):34–40, 2004.

[4] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Mobile Data Management*, pages 3–14, 2001.

[5] Jeffrey Considine, Feifei Li, George Kollios, and John W. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, pages 449–460, 2004.

[6] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph M. Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *VLDB*, pages 588–599, 2004.

[7] Samuel Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, 2002.

[8] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD Conference*, pages 491–502, 2003.

[9] Yong Yao and Johannes Gehrke. Query processing in sensor networks. In *CIDR*, 2003.

[10] Ossama Younis and Sonia Fahmy. Heed: A hybrid, energy-efficient, distributed clustering approach for ad hoc sensor networks. *IEEE Trans. Mob. Comput.*, 3(4):366–379, 2004.

[11] Mohamed H. Ali. Phenomenon-aware sensor database systems. In *EDBT Ph.D. Workshop*, 2006.

[12] Moustafa A. Hammad, Mohamed F. Mokbel, Mohamed H. Ali, Walid G. Aref, Ann Christine Catlin, Ahmed K. Elmagarmid, M. Eltabakh, Mohamed G. Elfeky, Thanaa M. Ghanem, R. Gwadera, Ihab F. Ilyas, Mirette S. Marzouk, and Xiaopeng Xiong. Nile: A query processing engine for data streams. In *ICDE*, page 851, 2004.

[13] Mohamed H. Ali, Mohamed F. Mokbel, Walid G. Aref, and Ibrahim Kamel. Detection and tracking of discrete phenomena in sensor-network databases. In *SSDBM*, pages 163–172, 2005.

[14] Mohamed H. Ali, Walid G. Aref, and Ibrahim Kamel. Scalability management in sensor-network phenomenabases. In *SSDBM*, 2006.

[15] Mohamed H. Ali, Walid G. Aref, Raja Bose, Ahmed K. Elmagarmid, Abdelsalam Helal, Ibrahim Kamel, and Mohamed F. Mokbel. Nile-pdt: A phenomenon detection and tracking framework for data stream management systems. In *VLDB*, pages 1295–1298, 2005.

[16] Mohamed H. Ali, Mohamed F. Mokbel, and Walid G. Aref. Phenomenon-aware stream query processing. In *Mobile Data Management*, 2007.

[17] Mohamed H. Ali, Walid G. Aref, and Cristina Nita-Rotaru. Spass: scalable and energy-efficient data acquisition in sensor databases. In *MobiDE*, pages 81–88, 2005.

[18] Mohamed H. Ali. Phenomenon-aware sensor database systems. *Current Trends in Database Technology - EDBT 2006 Workshops - Revised Selected Papers, Lecture Notes in Computer Science*, 4254:1–11, 2006.

[19] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *ICALP*, pages 693–703, 2002.

[20] Sumi Helal, William C. Mann, Hicham El-Zabadani, Jeffrey King, Youssef Kaddoura, and Erwin Jansen. The gator tech smart house: A programmable pervasive space. *IEEE Computer*, 38(3):50–60, 2005.

[21] George Kingsley Zipf. Human behavior and principle of least effort: An introduction to human ecology. *Addison-Wesley Publishing Co., Reading, MA*, 1949.

[22] Xuegang Huang and Christian S. Jensen. Towards a streams-based framework for defining location-based queries. In *STDBM*, pages 73–80, 2004.

[23] Mohamed F. Mokbel and Walid G. Aref. Sole: Scalable online execution of continuous queries on spatio-temporal data streams. Technical Report CSD-05-016, Purdue University, Department of Computer Science, 2005.

[24] Rimma V. Nehme and Elke A. Rundensteiner. Scuba: Scalable cluster-based algorithm for evaluating continuous spatio-temporal queries on moving objects. In *EDBT*, pages 1001–1019, 2006.

[25] Yufei Tao, George Kollios, Jeffrey Considine, Feifei Li, and Dimitris Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, pages 214–226, 2004.

[26] Donghui Zhang, Dimitrios Gunopulos, Vassilis J. Tsotras, and Bernhard Seeger. Temporal and spatio-temporal aggregations over data streams using multiple time granularities. *Inf. Syst.*, 28(1-2):61–84, 2003.

[27] Wensheng Zhang and Guohong Cao. Optimizing tree reconfiguration for mobile target tracking in sensor networks. In *INFOCOM*, 2004.

[28] Yingqi Xu, Julian Winter, and Wang-Chien Lee. Prediction-based strategies for energy saving in object tracking sensor networks. In *Mobile Data Management*, pages 346–357, 2004.

[29] Robert Nowak and Urbashi Mitra. Boundary estimation in sensor networks: Theory and methods. In *IPSN*, pages 80–95, 2003.

[30] Joseph M. Hellerstein, Wei Hong, Samuel Madden, and Kyle Stanek. Beyond average: Toward sophisticated sensing with queries. In *IPSN*, pages 63–79, 2003.

[31] Panos Kalnis, Nikos Mamoulis, and Spiridon Bakiras. On discovering moving clusters in spatio-temporal data. In *SSTD*, pages 364–381, 2005.

[32] Roger S. Barga, Jonathan Goldstein, Mohamed h. Ali, and Mingsheng Hong. Consistent streaming through time: A vision for event stream processing. In *CIDR*, 2007.

[33] Moustafa A. Hammad, Thanaa M. Ghanem, Walid G. Aref, Ahmed K. Elmagarmid, and Mohamed F. Mokbel. Efficient execution of sliding-window queries over data streams. Technical Report CSD-03-035, Department of Computer Science, Purdue University, 2004.

[34] Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, pages 285–296, 2003.

[35] César A. Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD Conference*, pages 571–581, 2001.

[36] Weipeng P. Yan and Per-Åke Larson. Eager aggregation and lazy aggregation. In *VLDB*, pages 345–357, 1995.

[37] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *SIGMOD Conference*, pages 275–286, 1999.

[38] Ahmed Ayad and Jeffrey F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD Conference*, pages 419–430, 2004.

[39] Brian Babcock, Mayur Datar, and Rajeev Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361, 2004.

[40] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. On random sampling over joins. In *SIGMOD Conference*, pages 263–274, 1999.

[41] Ahmed Ayad, Jeffrey F. Naughton, Stephen Wright, and Utkarsh Srivastava. Approximating streamingwindow joins under cpu limitations. In *ICDE*, page 142, 2006.

[42] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Semantic approximation of data stream joins. *IEEE Trans. Knowl. Data Eng.*, 17(1):44–59, 2005.

[43] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.

[44] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Data-streams and histograms. In *STOC*, pages 471–475, 2001.

[45] Kaushik Chakrabarti, Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. In *VLDB*, pages 111–122, 2000.

[46] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. In *STOC*, pages 20–29, 1996.

[47] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.

[48] Alin Dobra, Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Sketch-based multi-query processing over data streams. In *EDBT*, pages 551–568, 2004.

[49] Donghui Zhang, Dimitrios Gunopulos, Vassilis J. Tsotras, and Bernhard Seeger. Temporal aggregation over data streams using multiple granularities. In *EDBT*, pages 646–663, 2002.

[50] Themistoklis Palpanas, Michail Vlachos, Eamonn J. Keogh, Dimitrios Gunopulos, and Wagner Truppel. Online amnesic approximation of streaming time series. In *ICDE*, pages 338–349, 2004.

[51] Lukasz Golab and M. Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, 2003.

[52] Moustafa A. Hammad, Michael J. Franklin, Walid G. Aref, and Ahmed K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, 2003.

[53] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, 2003.

[54] Utkarsh Srivastava and Jennifer Widom. Memory-limited execution of windowed stream joins. In *VLDB*, pages 324–335, 2004.

[55] Annita N. Wilschut and Peter M. G. Apers. Pipelining in query execution. In *Proceedings of the International Conference on Databases, Parallel Architectures and their Applications, PARBASE*, 1990.

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

[57] Mohamed F. Mokbel, Ming Lu, and Walid G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, pages 251–263, 2004.

[58] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272, 2000.

[59] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math.*, 17(1):134–160, 2003.

[60] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[61] Xiaopeng Xiong and Walid G. Aref. R-trees with update memos. In *ICDE*, page 22, 2006.

[62] V. Prasad Chakka, Adam Everspaugh, and Jignesh M. Patel. Indexing large trajectory data sets with seti. In *CIDR*, 2003.

[63] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, pages 395–406, 2000.

[64] Reynold Cheng, Yuni Xia, Sunil Prabhakar, and Rahul Shah. Change tolerant indexing for constantly evolving data. In *ICDE*, pages 391–402, 2005.

[65] George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. On indexing mobile objects. In *PODS*, pages 261–272, 1999.

[66] Sunil Prabhakar, Yuni Xia, Dmitri V. Kalashnikov, Walid G. Aref, and Susanne E. Hambrusch. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Computers*, 51(10):1124–1140, 2002.

[67] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD Conference*, pages 331–342, 2000.

[68] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, 2003.

[69] Sangeeta Bhattacharya, Octav Chipara, Brandon Harris, Chenyang Lu, Guoliang Xing, and Chien-Liang Fok. Mobiquery: a spatiotemporal data service for sensor networks. In *SenSys*, page 320, 2004.

[70] Bugra Gedik and Ling Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.

[71] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD Conference*, pages 623–634, 2004.

[72] Mohamed F. Mokbel and Walid G. Aref. Gpac: generic and progressive processing of mobile queries over mobile data. In *Mobile Data Management*, pages 155–163, 2005.

[73] Deepak Ganesan, Sylvia Ratnasamy, Hanbiao Wang, and Deborah Estrin. Coping with irregular spatio-temporal sampling in sensor networks. *Computer Communication Review*, 34(1):125–130, 2004.

[74] Iosif Lazaridis, Qi Han, Xingbo Yu, Sharad Mehrotra, Nalini Venkatasubramanian, Dmitri V. Kalashnikov, and Weiwen Yang. Quasar: quality aware sensing architecture. *SIGMOD Record*, 33(1):26–31, 2004.

[75] Jia-Yu Pan, Srinivasan Seshan, and Christos Faloutsos. Fastcars: Fast, correlation-aware sampling for network data mining. In *GLOBECOM*, pages 2167–2171, 2002.

[76] Jae-Hwan Chang and Leandros Tassiulas. Energy conserving routing in wireless ad-hoc networks. In *INFOCOM*, pages 22–31, 2000.

[77] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MOBICOM*, pages 56–67, 2000.

[78] Joanna Kulik, Wendi Rabiner Heinzelman, and Hari Balakrishnan. Negotiation-based protocols for disseminating information in wireless sensor networks. *Wireless Networks*, 8(2-3):169–185, 2002.

[79] Amol Deshpande, Suman Kumar Nath, Phillip B. Gibbons, and Srinivasan Seshan. Cache-and-query for wide area sensor databases. In *SIGMOD Conference*, pages 503–514, 2003.

[80] Alan D. Amis, Ravi Prakash, Dung Huynh, and Thai Vuong. Max-min d-cluster formation in wireless ad hoc networks. In *INFOCOM*, pages 32–41, 2000.

[81] Stefano Basagni. Distributed clustering for ad hoc networks. In *ISPAN*, pages 310–315, 1999.

[82] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[83] Alberto Cerpa and Deborah Estrin. Ascent: Adaptive self-configuring sensor networks topologies.. In *INFOCOM*, 2002.

[84] Ya Xu, John S. Heidemann, and Deborah Estrin. Geography-informed energy conservation for ad hoc routing. In *MOBICOM*, pages 70–84, 2001.

[85] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator placement for in-network stream query processing. In *PODS*, pages 250–258, 2005.

[86] Alec Woo, Terence Tong, and David E. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys*, pages 14–27, 2003.

[87] Christian Böhm and Hans-Peter Kriegel. A cost model and index architecture for the similarity join. In *ICDE*, pages 411–420, 2001.

[88] Gísli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD Conference*, pages 237–248, 1998.

[89] Amol Deshpande, Carlos Guestrin, Wei Hong, and Samuel Madden. Exploiting correlated attributes in acquisitional query processing. In *ICDE*, pages 143–154, 2005.

[90] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph M. Hellerstein, and Wei Hong. Model-based approximate querying in sensor networks. *VLDB J.*, 14(4):417–443, 2005.

[91] Amol Deshpande and Samuel Madden. Mauvedb: supporting model-based user views in database systems. In *SIGMOD Conference*, pages 73–84, 2006.

[92] Shivnath Babu and Jennifer Widom. Streamon: An adaptive engine for stream query processing. In *SIGMOD Conference*, pages 931–932, 2004.

[93] Shivnath Babu, Pedro Bizarro, and David J. DeWitt. Proactive re-optimization. In *SIGMOD Conference*, pages 107–118, 2005.

[94] Pedro Bizarro, Shivnath Babu, David J. DeWitt, and Jennifer Widom. Content-based routing: Different plans for different data. In *VLDB*, pages 757–768, 2005.

VITA

VITA

Mohamed Ali was born in Alexandria, Egypt in 1977. His interest in computer sciences started in his early years through a set of introductory computer courses. During high school, Mohamed nourished his interest in computers by a good amount of supplementary reading materials. In 1994, Mohamed joined the Faculty of Engineering at Alexandria University. After a very competitive freshman year, he joined the Computer Science Department. In 1999, Mohamed obtained his B.Sc. degree in computer science with the highest degree of honor from the Faculty of Engineering, Alexandria University. Mohamed was one of few students in his class who obtained a Distinction grade in all the undergraduate years of study. Mohamed obtained the highest GPA among the sixty students of his class upon graduation. Immediately after the completion of his B.Sc. degree, Mohamed joined the computer science program for graduate studies where he stared his research in computer vision, machine learning, and database systems. Two years later, Mohamed obtained his M.Sc. degree in computer science from the Faculty of Engineering, Alexandria University.

In 2002, Mohamed joined Purdue University as a research assistant with Prof. Walid Aref. Mohamed's research sharpened his experience in large scale database systems. Mohamed published several research papers in various areas of core database and data stream systems. In summer 2006, Mohamed interned with the database group at Microsoft Research, one of the top research labs world-wide. Mohamed's main research interests focus on advancing the state of the art in the design and implementation of database and data stream systems to cope with the requirements of emerging applications. Mohamed Ali graduated with a Ph.D. degree in computer science from Purdue University in May 2007 and joined the SQL Server group at Microsoft Corporation as a software design engineer.