# Design and Implementation of an Efficient Data Stream Processing System

THÈSE N$^{O}$ 4611 (2010)

PRÉSENTÉE LE 29 MARS 2010

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DE SYSTÈMES D'INFORMATION RÉPARTIS

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Ali SALEHI

acceptée sur proposition du jury:

Prof. R. Guerraoui, président du jury
Prof. K. Aberer, directeur de thèse
Prof. T. Hara, rapporteur
Prof. M. Hauswirth, rapporteur
Prof. S. Spaccapietra, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2010

# Résumé

Dans le cadre standard de l'utilisation de bases de données, les données brutes telles que celles produites par un réseau de capteur sans fils (*Wireless Sensors Networks*) sont stockées en tous temps et peuvent être traitées simplement par l'intermédiaire de requêtes SQL ou de procédures stockées dont la complexité est arbitraire.

Cependant dans le cadre d'applications impliquant des flux de données (Data Streams), la quantité de données produite est si importante que le stockage puis le traitement de ces données est irréalisable pratiquement car les temps de traitement et les coûts de stockage deviennent trop conséquents.

Cette thèse décrit les résultats de nos travaux de recherche portant sur la conception et la mise en uvre d'une solution efficace de gestion en temps réel et à postériori de flux de données issus de mesures environnementales. Bien que nous nous sommes concentrés sur un domaine d'application précis, le résultat de nos recherches ont une portée générale et sont présentés dans cette thèse de manière générique afin que nos résultats puissent être appliqués à un large éventail d'applications impliquant le traitement de flux de données.

Cette thèse commence par dresser l'état de l'art de la recherche sur le traitement des flux de données. Les concepts de traitement par fenêtre (window processing), les requêtes continues (Continuous Queries), les languages de filtres pour flux de données (Stream Filtering Query Languages) ainsi que les méthodes de traitement à la source (in-network processing) sont présentés en détail. Nous présentons les principaux systèmes de traitement disponibles, leur architecture interne et leur comparaison avec notre système nommé : Global Sensor Network (GSN) middleware.

GSN offre une méthode flexible et rapide permettant de déployer des réseaux de capteurs et de les interconnecter entre eux. GSN offre un accès simple et standardisé à un ensemble complet de technologies hétérogènes. De plus, GSN offre un déployment simplifié à l'extrême (Zero-Programming), peut se reconfigurer et

s'étendre dynamiquement. Nous présentons le concept de capteur virtuel (Virtual Sensor), une abstraction permettant de représenter tout type de source de données et offrant des outils performants de recherche. Finalement, nous décrivons en détail les décisions concernant la conception, l'architecture et l'optimization de GSN. Un ensemble d'algorithmes groupant et traitant intelligemment différents types de requêtes continues et permettant le traitement de grandes quantités de flux de données sera principalement présenté.

La mise en uvre de GSN sur des réseaux comprenant un nombre important de nuds a dévoilé différents problêmes de performances liés au passage à l'échelle. Ainsi, nous avons pu définir un méthode permettant d'améliorer concidérablement les performances d'exécution des requêtes en les groupant par similarités afin que les coûts de traitements et de mémoire soient réduits au maximum. Aussi, nous avons pu définir un ordonnanceur de requêtes efficace, permettant dans le cas de requêtes continues, d'augmenter les performances de plus d'un ordre de grandeur tout en réduisant le temps de réponse et l'espace en mémoire.

Finalement, nous présentons comment la plateforme GSN peut être intégrée avec un système externe de partage et de visualisation de données tel que la plateforme SensWeb de Microsoft. Cette dernière offre une infrastructure de collection et de visualisation globalements accessible par les utilisateurs. Cette intégration démontre non seulement la capacité de passage à l'échelle de GSN mais aussi sa flexibilité.

**Mots clefs:**

Traitement de Flux de Données (Data Stream Processing), Global Sensor Networks, GSN, Traitement de Requêtes Continues (Continuous Query Processing), Traitement de Données par fenêtres (Window-Based Data Processing), Traitement Distribué de Flux de Données (Distributed Data Stream Processing)

# Abstract

In standard database scenarios, an end-user assumes that all data (e.g., sensor readings) is stored in a database. Therefore, one can simply submit any arbitrary complex processing in the form of SQL queries or stored procedures to a database server.

Data stream oriented applications are typically dealing with huge volumes of data. Storing data and performing off-line processing on this huge dataset can be costly, time consuming and impractical.

This work describes our research results while designing and implementing an efficient data management system for online and off-line processing of data streams in the field of environmental monitoring. Our target data sources are wireless sensor networks. Although our focus is on a specific application domain, the results of this thesis are designed in a generic way, so that they can be applied to wide variety of data stream oriented applications.

This thesis starts by first presenting the state-of-the-art in data stream processing research specifically window processing concepts, continuous queries, stream filtering query languages and in-network data processing (particular focus on TinyOS-based approaches). We present key existing data stream processing engines, their internal architecture and how they are compared to our platform, namely Global Sensor Network (GSN) middleware.

GSN middleware enables fast and flexible deployment and interconnection of sensor networks. It provides simple and uniform access to a comprehensive set of heterogeneous technologies. Additionally, GSN offers zero-programming deployment and data-oriented integration of sensor networks and supports dynamic re-configuration and adaptation at runtime. We present the *virtual sensor* concept, which offers a high-level view of arbitrary stream data sources, its powerful declarative specification and query tools. Furthermore, we describe design, conceptual, architectural and optimization decisions of GSN platform in detail.

In order to achieve high efficiency while processing large volumes of stream-

ing data using window-based continuous queries, we present a set of optimization algorithms and techniques to intelligently group and process different types of continuous queries.

While adapting GSN to large scale sensor network deployments, we have encountered several performance bottlenecks. One of the challenges we faced was related to scalable delivery of streaming data for high data rate streams. We found out that we could dramatically improve the performance of a query processor by performing simple grouping of user queries hence sharing both the processing and memory costs among similar queries. Moreover, we encountered a similar performance issue while scheduling continuous queries. Problem of efficiently scheduling the execution of continuous queries with window and sliding parameters is not addressed in depth in literature. This problem becomes severe when one considers large volumes of high data rate streams. In these cases, an efficient query scheduler not only increases the performance at least by an order of magnitude but also, decreases the response time and memory requirements.

Finally, we present how our GSN platform can get integrated with an external data sharing and visualization framework namely Microsoft's SenseWeb platform. Microsoft's SenseWeb platform, provides a sensor network data gathering and visualization infrastructure which is globally accessible to the end users. This integration (which is initiated by the Swiss Experiment project and demanded by GSN users) not only shows the scalability of GSN platform when combined with optimized algorithms, but also demonstrates its flexibility.

**Keywords:**

Data Stream processing, Global Sensor Network, GSN, Continuous Query Processing, Window-Based Data Processing, Distributed Data Stream Processing.

# Contents

# Chapter 1

# Introduction

## 1.1 Preface

Recent advances in embedded systems and mobile communication have led to the emergence of wireless sensor and actuator technologies. These new smart computing platforms based on wireless sensor network technology are paving the way for the next revolution in ubiquitous computing which is known as the "Internet of Things". In 1999, *Business Week* named networked micro-sensor technology as one of the 21 most important technologies of the 21$^{st}$ century. In 2009, Gartner group has predicted that by 2012, more than 20% of Internet traffic will be generated by sensor data streams.

Thanks to the small size of these devices, the pervasive computing concept is now far from just an imaginary idea. Wireless sensors (also known as motes and smart dust) are now available off-the-shelf from numerous vendors around the world. These new emerging computers are providing the base to enable the integration of the physical and the digital world.

Once started as a research tool for scientists, they are now becoming an indispensable part of the monitoring infrastructure in sensing, engineering and industry (e.g., monitoring the environment, infrastructure, supply chain,...). Today, cheap and smart devices with multiple on-board sensors, networked with wireless links are available from several research groups and companies (e.g., the MICA Motes from Crossbow Solutions[2], TMotes from MoteIV[8] and BTNodes from ETHZ[1]).

As smart dust is deployed in the environment, not only wireless sensor network technology brings a tangible value to the day-to-day lives of the people by providing detailed and real-time observations of the physical world, but also, it leads to

novel and challenging research problems. One of the interesting class of research problems is related to the management of sensor data generated by this new technology. For example, how one can integrate readings from different type of sensors in order to make more precise decisions. As another example, how one can process high volumes of incoming sensor readings in real-time and provide early warning information on-time and close to real-time as the events occur in the real world.

Another aspect related to the management of the devices is that, the motes are based on new software platforms[47][49] and networking technologies[90], which are specifically designed to enable low cost, high volume and low powered setups. These requirements result from the applications in which motes are used. Simply put, as the price of these devices is decreasing the demand for using this technology is increasing. In this situation, the maintenance cost has the potential of becoming one of the major factors and obstacles toward their large scale adaption. In the ideal case, theses smart devices should support adaptive and ad-hoc deployment with long battery life and a re-programming/configuration interface over the air.

As of today, sensors run specialized operating systems such as TinyOS[49] and Contiki[47]. For accessing sensor measurements (e.g., data stream of sensor measurements), there exists a variety of interfaces including, but not limited to, query based APIs such as the TinySQL and TAG[68]. While these interfaces are crucial to benefit from wireless sensors and are the bases for any advanced data management approach, research on data stream management for sensor networks is in its early stages.

Compared to the Internet, the infrastructure for processing, maintaining and publishing sensor data to a wide public is fairly limited. The main obstacles are the lack of standardization and the continuous development of novel sensor network technologies. As the price of sensors and motes are decreasing rapidly and given the current growth rate, we may expect to arrive at a situation comparable to the publication of documents on the Web, whose success is mainly based on sharing a few simple logical abstractions (URI, hyperlinks) and basic communication protocols (HTTP, more recently Web Services) to provide universal access and linking among autonomously published data sources. To translate the success of the Web's core technologies into wireless sensor network, one has to address the following requirement for any sensor data management solution:

**Simplicity.** A platform with a minimal set of powerful abstractions which can be configured and adapted easily to the user's needs.

**Adaptivity.** Allowing the user to add new types of sensor networks and facili-

tating dynamic (re-) configuration of the system during run-time without having to interrupt ongoing system operation.

**Scalability.** Supporting very large numbers of (distributed) data producers and consumers with a variety of application requirements.

**Light-weight implementation.** Having a software platform deployable in standard computing environments (no excessive hardware requirements, standard network connectivity, etc.) and portable (virtual machine based implementation) with minimal initial configuration.

To address these requirements, a carefully designed sensor data stream processing platform is necessary. Traditional database technology is not sufficient for handling the requirements of this new technology. This is because of the huge data volumes and real-time processing needs imposed by the applications of wireless sensor networks.

The research community have proposed various platforms for management and processing of streaming data such as [10][20][32][52][83][94] (a detailed discussion is provided in Chapter 2). However, we find that they frequently not match the above mentioned requirements of simplicity, Adaptivity and scalability. This motivated us to develop a novel data stream processing platform, Global Sensor Networks (GSN), with these objectives in mind. We will provide an overview of GSN in the following section 1.2 and illustrate its use in Section 1.3. While developing and using GSN we identified specific performance problems related to the processing of large numbers of time-based queries.

Despite a large research literature on data stream processing (see Chapter 2 for a detailed overview) this specific issue has not received sufficient attention. Therefore we developed as part of GSN novel techniques for processing large numbers of time-based queries and distributing efficiently the query results to large user communities, as described in Section 1.4.

## 1.2 Global Sensor Network

Since 2005, we are involved in multitude of sensor network data management projects involving several research institutes, different types of sensor networks and data capturing instruments. During this period, we are introduced to numerous deployments of sensor networks.

In order to address data stream management and processing needs of these applications, we gathered the major requirements of our users which are mainly

environmental scientists and hydrologists. These requirements are critical in order to propose a solution which can be adapted to real-world deployments. In the following, we present key challenges we identified in order to have an effective solution for managing and processing data streams for sensor networks.

**Flexible resource management.** A sensor network creates a dynamic environment in which nodes may join or leave at anytime during the deployment. This behavior is partly due to external factors such as being battery powered, erroneous code, and mobility of the device. Smooth resource management is an essential factor in adapting existing and limited resources of a middleware to current status of a sensor network dynamically. Adaption includes but not limited to, having unused resources freed and reallocated to other nodes, change of processing priority dynamically based on predefined exceptional events (e.g., avalanche warning in a ski resort) and finally identifying and exploiting shared computation.

**Integrated data acquisition and processing.** In sensor networks, data acquisition and processing are deeply coupled. Compared to off-the-shelf acquisition devices which normally don't have a programmable unit, a sensor node comes with a programmable unit which has a limited yet valuable processing and storage capabilities. In most of the real-world applications, streaming data generated by a sensor node also known as a mote, has to be cleaned, pre-processed (e.g., calibration, etc.) before data stream can be used effectively (e.g., getting integrated with other data sources). To this end, having a middleware which has the data acquisition layer integrated with a data stream processing engine is a decisive factor in expediting deployment and adaption process of wireless sensor networks.

**Runtime reconfiguration.** Application requirements may change several times during the life cycle of a sensor network deployment. We believe a middleware for sensor data should expedite the process of adapting an existing deployment with emerging needs. A middleware achieves this by providing means of reconfiguration. Such reconfigurations shouldn't be disruptive with regard of existing deployment. Middleware which can achieve adaptability in runtime can significantly assist mission critical applications by making sure that the middleware is run continuously without interruptions while addressing new needs through a runtime reconfiguration interface.

**Flexible data stream acquisition layer.** Domain of data acquisition instruments is wide and full of custom standards, proprietary protocols and interfaces. Data stream management middleware should be designed based on well-defined abstractions which are general enough to handle wide variety of data stream sources. These abstractions are key success factors in a wide adaption of any middleware for sensor networks. A system, which can be extended to be used with new emerging data sources would create a significant impact through bringing new technologies to existing deployments. Such a middleware saves resources through maintaining existing development and measurements while being integrated with new products, technologies and data sources.

**Common services in one package.** In practise, wireless sensor network deployments are not only about measuring several physical phenomena but also involve a full set of services. These services not only increase the value of measurements, but also, they provide essential building blocks for using sensor data with an existing infrastructure. To name a few key services, one may consider authentication services, storage and integration of streaming data with existing databases, multiple access APIs, etc.

**Peer-to-peer data streaming.** Extensive deployments of wireless nodes have generated a large number of streaming data around the Internet. Empowering data stream applications inside a middleware through bring the opportunity of integrating multiple data sources to provide rich services demands a careful design. Peer-to-peer topology can create a distributed stream processing environment in which nodes adhere a flat topology where each node is capable of acting both as a data producer and as a data consumer.

As presented in more detail in Chapter 2, research community in data stream processing and management have proposed a comprehensive set of solutions for different problems in this domain. While these solutions are proven to be valuable for specific problems under a certain circumstance, they are not naturally designed to be compatible with each other. In order to create a middleware which is an integrated solution addressing the above requirements, we have to address a number of difficult challenges. These challenges are resulted from incompatible abstractions, conflicting requirements and interfaces among the existing work.

Extensive analysis on existing work (more details in Chapter 2) convinced us that we need to revisit the design and architecture of stream processing middle-

wares. Our objectives are specifically focused on building a highly flexible and yet efficient stream data management infrastructure.

In this work, we present the Global Sensor Network (GSN) platform. GSN is a data stream processing platform which we built to address the requirements we had in hand. GSN has started in 2005 as a research project in EPFL and soon became one of the most pro-actively used[3][4][5][6][7] solutions for working with sensor generated data streams.

GSN, is a data stream processing engine which is designed to be deployed in a distributed context. GSN provides a solution for rapid deployment, maintenance and management of wireless sensors for end-users. In a typical setup, we have GSN users at the edge of the Internet (e.g., home users with off-the-shelf sensors connected to their computer), feeding data streams into the global network. GSN not only provides an extendible platform for handling streaming data but also supports important requirements for such a software platform.

These requirements include, but are not limited to, automated database and storage management, dynamic reconfiguration and resource management, variety of access interfaces (JSON, RestFul, WebServices,...), access control and a continuous query processor engine.

## 1.3 A Simple Sensor Data Integration Example

In order to illustrate the potential applications of sensor data integration we provide a simple application scenario in a (university) building as shown in Figure 1.1. We assume the following hardware setup in place.

- Wireless cameras with built-in HTTP access.
- Wireless sensors (motes) equipped with light, sound, temperature and pressure sensors. We assume that all motes of the same type form a sensor network
- RFID tags which are attached to the key rings of people, and to books, mobile phones and laptops in the buildings; and several RFID readers whose coverage ranges are shown in yellow (gray) in Figure 1.1.
- Mid-range RFID readers for covering a range of a few meters surrounding a reader.

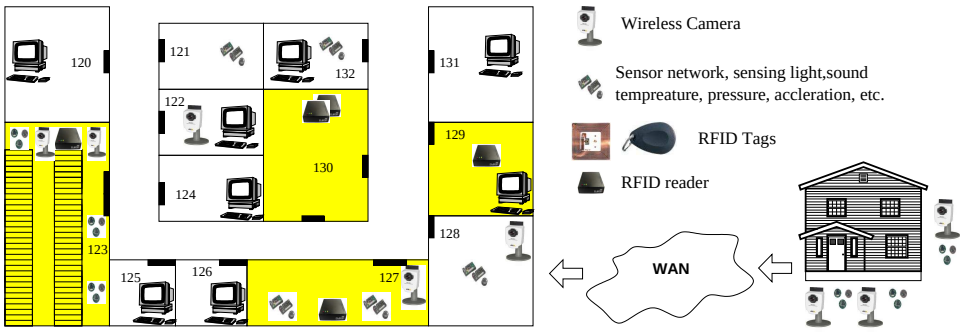



**Figure 1.1.** A simple scenario

Further we assume that each computer in the Figure 1.1 runs an instance of a stream processing engine. The wireless cameras are accessible directly via HTTP hyperlinks and the other sensors (motes) or a complete sensor network can either be accessed via the local area network (IP-based communication) or through a serial port connectivity.

In this setup having a stream processing engine which can process multiple data sources at the same time enables the users to accomplish a large variety of tasks. For example, the library manager can register a query to be notified when there are more than 15 books (assuming the books are all equipped with RFID tags) in one room in addition to the monthly report on the most popular books of the month (e.g., to order more of those to accommodate the need).

Individual users can post one-shot queries to the library (room 123) to get the status of certain books. In case a book of interest is currently not available, user can register a continuous query to be notified when the book is returned to the library. Another use-case of data integration among multiple heterogeneous sensors, one might be interested to receive a stream of camera images whenever a movement in the house is detected or a sound sensor observes some noise above a certain threshold (e.g., home surveillance application).

To extend the use-case, consider one can integrate different data streams using a stream processing engine to help people in finding their lost possessions. For instance, if one loses a mobile phone (with an RFID tag attached to its battery slot), one can check for the phone's last location in the building simply by posting a query on the previous observations provided by all RFID sensor networks deployed in the building. One can extend the processing chain through registering the continuous queries and connecting the results of the queries to a SMS service provider.

Having a stream processing engine in which users combine and process different data streams generated by different type of sensors, one can get (close to real-time) the short messages (SMS) with the most recent observations of the RFID networks filtered by the specific RFID tag.

## 1.4 Contribution of this Thesis

The primary focus of this thesis is a flexible middleware design and data processing abstractions for wireless sensor networks. In order to present our scalability and flexibility enhancements, we introduce the Global Sensor Network (GSN) platform. GSN platform constitutes the basis of this thesis. Design of GSN follows four basic goals:

**Simplicity.** The goal was to design the system based on a minimal set of powerful abstractions which could be easily configured and adapted to the user's needs. We targeted the possibility to define sensor networks and data streams in a declarative way by using SQL as data manipulation language. As a syntactic framework for system configuration we relied on XML.

**Adaptivity.** Adding new types of sensor networks and dynamic (re-) configuration of data sources have to be supported during run-time without having to interrupt the ongoing system operation (query processing, etc.). To that end we used a container-based implementation allowing dynamic reconfiguration.

**Scalability.** Targeting a very large number of data producers and consumers with a variety of application requirements, GSN has to consider scalability issues specifically for distributed query processing and distributed discovery of sensor networks. To meet this requirement, the design of GSN is based on a peer-to-peer architecture.

**Light-weight implementation.** GSN is intended to be easily deployable in standard computing environments (no excessive hardware requirements, standard network connectivity, etc.), be portable (Java-based implementation), should require minimal initial configuration, and provide easy-to-use, web-based management tools.

Building a generic data stream processing engine which can be integrated with many different types of sensor devices is both interesting and challenging in its

nature. On the technical side, we built GSN to provide a scalable infrastructure for integrating heterogeneous sensor network technologies using a small set of powerful abstractions.

While adapting GSN to large scale sensor network deployments, we have encountered several performance bottlenecks. One of the challenges we faced were related to scalable delivery of streaming data for high data rate streams. We found out that we could dramatically improve the performance of a query processor by performing simple grouping of user queries hence sharing both the processing and memory costs among similar queries (more details are provided in Chapter 5).

Moreover, we encountered a similar performance issue while scheduling continuous queries. Problem of efficiently scheduling the execution of continuous queries with window and sliding parameters is not addressed in depth in literature (more details are provided in Section 4.3). The problem becomes severe when one considers large volumes of high data rate streams. In these cases, an efficient query scheduler not only increases the performance at least by an order of magnitude but also, decreases the response time and memory requirements. A brief summary of what a reader should expect from each chapter is provided below.

### 1.4.1 Chapter 2: Related Work

We give an overview of the key data stream processing concepts such as window-based processing and continuous query processing languages. We present the major data stream processing engines, their internal architecture and how they are compared with GSN, specifically in architectural and design aspects. Moreover, we present different in-network data processing approaches (particular focus on TinyOS based approaches).

### 1.4.2 Chapter 3: Global Sensor Network

In chapter 3 we provide an overview of the conceptual model GSN is based on. We introduce the key abstraction of *virtual sensors* and GSN's approach to data stream processing. We conclude the chapter with the architecture, implementation and the evaluation of the GSN system.

GSN takes up the successful ideas of today's Web (e.g., Web Services, HTML documents, hyperlinks, ...) and aims at making publication and access to sensor networks and sensor data as simple, powerful, and flexible as accessing Web documents. To build such a system, we had to tackle a multitude of problems.

As an example, different sensor devices do provide different communication protocols, some of which are solely output oriented such as a poll-based wireless camera with no programmable configuration interface (e.g., capturing the pictures through calling a certain URI). Other, and more advanced sensing devices, such as the TinyOS based motes provide a much more verbose and interactive method of interacting with the capturing device (in our case, for environmental monitoring purposes). The TinyOS based protocol provides the end-application a full-fledged querying interface which one can use to send queries and receive sensor data as the response of those queries with in-network aggregation mechanisms.

GSN supports the integration of (distributed) sensor network deployments, provides distributed querying through a SQL-like language, complex filtering and combination of sensor data, supports dynamic adaptation of the system configuration during operation through a declarative XML-based language.

### 1.4.3 Chapter 4: Efficient Sliding Window Management

In order to build a highly scalable data processing engine, chapter 4 presents a set of optimization algorithms and techniques for handling large scale data stream processing. We provide a set of algorithms which can be used to efficiently decide on the processing time of the queries in the stream processing engines. This chapter presents in detail the optimization techniques used by GSN for handling window-based continuous queries.

The work in this chapter can be specifically useful for popular and high rate streams such as stock ticks, sensor values for a renowned location (e.g., snow height in a popular skiing resort in winter) in addition to resource constrained environments such as mobile phones and PDAs.

### 1.4.4 Chapter 5: Delivering Popular Streaming Data Efficiently

In stream processing, efficient delivery of data streams to end-users (e.g., subscribers) is one of the metrics for measuring the effectiveness of the system. The naive approach in data stream delivery involves evaluating every continuous query and delivering the sensor data once there is anything to deliver.

To make sure that GSN can efficiently interact with a large number of data stream consumers, GSN has to have a scalable data stream delivery mechanism. Chapter 5 provides the motivating scenarios with example use-cases for this kind of delivery system. In addition, the chapter provides the algorithms and the optimization techniques which can reduce the overall load of the system in the case of

having popular data streams (e.g., environmental information of a city or financial strategy's decisions in a highly volatile trading day) through intelligently grouping continuous queries thus significantly reducing the load and the latency.

### 1.4.5 Chapter 6: Sensor Data Sharing and Visualization

In this chapter, we show how a stream processing engine like GSN can get integrated with an external data sharing and visualization framework called Microsoft's SensorMap. Microsoft's SensorMap platform provides a sensor network data gathering and visualization infrastructure which is globally accessible to the end users[67][73][74].

In this chapter, we present the process of monitoring real-world deployments using a visual map-based interface in which users can inspect the measured data in the form of contour plots overlayed onto a high resolution map and a digital topographic model.

Thanks to this close interaction between these two systems, users can go back in time virtually to search for interesting events or simply to visualize the temporal dependencies of the data. The system presented is not only interesting and visually enticing for non-expert users but brings substantial benefits to environmental scientists. The easily installed data acquisition component as well as the powerful data sharing and visualization platform opens up a new ground in collaborative data gathering and interpretation in the spirit of Web 2.0 applications.

## 1.5 List of Publications

This thesis is based on the following research publications:

- Yongluan Zhou, Ali Salehi, Karl Aberer. **Scalable Delivery of Stream Query Result**. Very Large Data Bases (VLDB), Lyon, France, 2009.

- Ali Salehi, Mehdi Riahi, Sebastian Michel, Karl Aberer. **Knowing When to Slide - Efficient Scheduling for Sliding Window Processing.** Mobile Data Management (MDM), Taipei, Taiwan, 2009.

- Ali Salehi, Mehdi Riahi, Sebastian Michel, Karl Aberer. **GSN, Middleware for Stream World (Best Demonstration Award)**. Mobile Data Management (MDM), Taipei, Taiwan, 2009.

- Sebastian Michel, Ali Salehi, Liqian Luo, Nicholas Dawes, Karl Aberer, et al. **Environmental Monitoring 2.0. (Demonstration).** International Conference on Data Engineering (ICDE), Shanghai, China, 2009.

- Y. Zhou, K. Aberer, A. Salehi, K.-L. Tan. **Rethinking the Design of Distributed Stream Processing Systems.** International Workshop on Networking Meets Databases (NetDB), co-located with IEEE ICDE 2008 in Cancun, Mexico, 2008.

- Karl Aberer , Manfred Hauswirth , Ali Salehi. **Infrastructure for data processing in large-scale interconnected sensor networks.** Mobile Data Management (MDM), Germany, 2007.

- Karl Aberer, Manfred Hauswirth, Ali Salehi. **Zero-programming Sensor Network Deployment**. Next Generation Service Platforms for Future Mobile Systems (SPMS), Japan, 2007.

- Karl Aberer, Manfred Hauswirth, Ali Salehi. **A middleware for fast and flexible sensor network deployment**. Very Large Data Bases (VLDB) Seoul, Korea, 2006.

- Karl Aberer, Manfred Hauswirth, Ali Salehi. **Middleware support for the "Internet of Things".** 5. GI/ITG KuVS Fachgesprch "Drahtlose Sensornetze", Universitt Stuttgart, 2006.

# Chapter 2

# Related Work

In this chapter, we present the essential background for data stream processing systems and wireless sensor networks. We also present the current approaches toward building platforms to facilitate sensor network deployment and management. In this chapter, we intentionally limit our focus to high-level architectural and system design issues of key existing data stream systems for sensor data processing, integrating and publishing. Detailed related work with regard to various approaches for delivering data streams and techniques for processing and scheduling window-based continuous queries are presented in Section 4.3 and Section 5.2 respectively.

We start by introducing the key data stream processing techniques in the literature. Section 2.2 introduces the key data stream processing engines in the literature. In section 2.3, we present the major data stream management platforms. A data stream management platform is an integrated system composed of a data acquisition layer, a data stream processing engine and data publishing and distributing. In section 2.4, we present the basic concepts behind data stream processing engines, namely window-based data processing and the continuous query languages. In section 2.5, we present the distributed publish/subscribe systems and their difference with stream processing engines.

## 2.1 Data Stream Processing Techniques

The topic of data stream processing is a recent but a highly active research area. Data stream oriented applications are typically dealing with huge volumes of data. Storing data and performing off-line processing on such data can be costly and time consuming which is normally undesirable for most of data stream applications. Data streams occur in different types of real-time (or close to real-time) applica-

tions. These include data flows generated by sensor networks, financial markets, news feeds, monetary transactions and IP-networks. Sensor networks play an important role in this field. The possibility of fine-grained monitoring of physical environment and providing services such as early warning, rapid risk analysis and online pattern detection have led researchers to propose a variety of architectures and techniques for data processing.

Data processing techniques requiring repeated access to the same data are typically not applicable to the processing of data streams (e.g., data streams from wireless sensor networks). Moreover, approaches relying on availability of complete historical data can not address timing requirements of applications in this domain, such as a fire alarm network deployed in a building or an avalanche warning system at a ski resort. In these applications, the quality of a result is directly depending on a short processing time. Distributed computing techniques, such as in-network data processing and operator placement are used to efficiently reduce data load on a node by either pushing the processing logic toward data sources or by distributing load among multiple nodes. In-network data processing, exploiting internal processing capabilities of wireless sensor networks, opens a great opportunity in addressing performance bottlenecks, but also reveals new technical challenges. Specifically, wireless sensor nodes normally have limited resources, specifically memory, storage and energy, which implies that algorithms demanding high processing or memory usage is not suitable.

In the following we present key data stream processing problems and discuss state-of-the-art developments associated with each problem. An area that received substantial attention is real-time data mining and analysis of data streams. Analysis algorithms have in particular to consider the limitations that the same data item in a data stream can only be accessed once. This constraint of real-time data stream processing have led to new approaches for performing data mining.

Adaptions of the *k-means* clustering algorithm are proposed by [34][58][76]. The authors in [14] introduce *HPStream* which is a hierarchical method for clustering data streams. [13] presents the idea of dividing the clustering process into an online component which periodically gathers and stores a detailed summary statistics and an off-line component which only uses summary statistics. Density-based clustering approaches for data streams are introduced by [27] and [36].

Thanks to dynamic nature of data streams, the problem of data stream classification has to be revisited to adapt to the constraints introduced by data streams. A classification process may require simultaneous model construction and testing

in an environment which is constantly evolving. Decision tree based approaches which are involving only one-pass over a data stream are introduced in [46] and [60]. The authors in [15] propose an on-demand classification process which dynamically selects the appropriate window of past training data to build the classifier.

The problem of frequent pattern mining over data streams is investigated in [22][71]. In [71] authors introduce sticky sampling and lossy counting approaches for maintaining approximate counts over a sliding window using a limited space. [22] presents algorithms for computing approximate frequency counts over a data stream with a parameterizable error threshold.

As real-time data streams are evolving constantly, it is critical to analyze and predict changes in trends quickly. In [12][62][77] authors provide methods for detecting and quantifying changes in distribution of values over a streaming data.

Data streams are normally generated by external sources which may have different data rates for various circumstances. The rate of data stream typically depends on external conditions (e.g., sensor data burst in case of fire) which are out of the control of stream processing engine. In order to deal with uncertain data rates, several load shedding approaches have been proposed. These approaches normally involve dropping unprocessed tuples to reduce overall system load and latency with the cost of degrading the accuracy of results.

The authors in [38] introduce the Loadstar system which uses a metric known as quality of decision (QoD) to measure the level of uncertainty. Resources are then dynamically allocated to sources where uncertainty is high. [89] introduces a technique for dynamically inserting and removing drop operators into query plans as required by the current system load.

Large volumes of streaming data combined with real-time requirements led to the introduction of approximate solutions which are typically based on a synopsis structure. Approaches using synopses trade accuracy with performance and storage. Synopsis computation based on sampling of a data stream is proposed by [54], [53] and [71]. Sketch-based synopsis computation approaches are presented in [40][41][72].

Joining data streams is a fundamental operation for combining and correlating data produced by multiple sources. Section 1.3 provides a sample scenario involving multiple join operations among several heterogeneous streaming data. The continuous nature of data streams in combination with variable data rates implies novel challenges in query planning. Blocking operations such as sorting can no longer work effectively. Storage and indexing operations using non-volatile mem-

ory are undesirable due to timing requirements of streaming data and blocking nature of these operations. Moreover, thanks to the long-running nature of continuous queries, uncertainty associated with data rates and the continuously evolving stream elements, more adaptive solutions are desirable. [86] presents non-blocking versions of conventional join methods. Similarly, [56] proposes algorithms for multi-way incremental nested loop joins and multi-way incremental hash joins. [18][45] proposes sketch-based solutions for stream joins and multi-join queries. [43] shows that semantic load shedding (adapting to resource shortages by dropping tuples based on their values) is superior in terms of the quality of join result to random load shedding at the cost of a small overhead for maintaining simple stream statistics. [44] proposes *PWJoin* which is a 3-operation-based algorithm for binary window join which exploits value-based constraints that may hold in a data stream. Authors in [50] propose *GrubJoin* which is an adaptive, multi-way, windowed stream join that effectively performs time correlationaware CPU load shedding.

To handle the sheer volumes of streaming data, cost-based operator placement approaches, which dynamically move operators based on current system load, have been extensively studied by [78][84][95]. Moreover, in context of sensor network, authors in [84] propose a cost-based in-network operator placement method for wireless sensor networks which involves progressively increasing computational power and network bandwidth up a hierarchy of processing nodes.

Stream processing engines are proposed by [10][20][32][52][94]. Architectural and design differences between these systems and this work is presented in Chapter 2. Data stream processing engines support continuous queries. Different continuous query language proposals are described in detail in Section 2.4.

## 2.2 Data Stream Processing Engines

In this section, we review how different systems do query planning and query optimization for processing data streams. Before starting, we want to stress that, all the following systems support receiving data from distributed stream sources such as wireless sensor networks.

The Aurora[10] and STREAM[20] systems, are based on a centralized model where all processing takes place at a single node. In Aurora, streams are modeled as sequences of time-stamped tuples, and users can compose stream relationships and construct queries in a graphical tool which is then used as input for the query

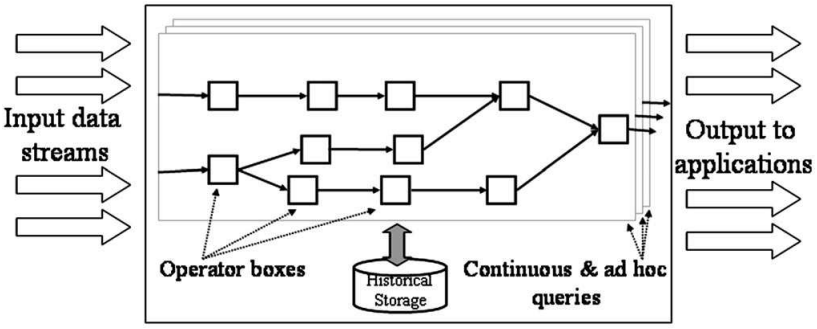



**Figure 2.1.** Aurora's Data Stream Processing Model from [10]

planner. This is shown in Figure 2.1. With distributed stream sources, moving some processing toward the data sources instead of moving the raw data to a central system may lead to more efficient use of processing and network resources. The TelegraphCQ[32] achieves this goal by running several TelegraphCQ instances on different machines and each machine receives stream elements from the closest stream producers, performs the filtering and forwards the processed data to the other TelegraphCQ nodes. Therefore, a TelegraphCQ node can receive several input data streams, process them further and produce a new output stream. The data flow in a TelegraphCQ based stream data processing is statically planned and should conform a tree topology. Currently, TelegraphCQ relies on PostgreSQL[1] (a centralized DBMS) for creating execution plans and optimization.

In GSN, stream data processing logic at each node is not statically planned hence using a cost-based operator placement systems, once can easily distribute the processing and communication loads in the network. Moreover, the data flow topology between nodes is not limited to a tree topology. In GSN, in contrast with TelegraphCQ, the topology of data flow is flat. This combined with the dynamic allocation (and deallocation) of resources creates a rich ecosystem in which application designers have the opportunity to fine-tune the deployment of GSN with the internal data flow patterns.

The Aurora* and Medusa[94] systems are aiming at designing a distributed version of Aurora[10], which is a centralized stream processing engine. In the Medusa distributed stream-processing system [94], Aurora is being used as the processing engine on each of the participating nodes. Medusa takes Aurora queries and distributes them across multiple nodes and particularly focuses on load management using economic principles and high availability. The Borealis stream processing

[1] `http://www.postgresql.org`

engine [9] is based on the work in Medusa and Aurora and supports dynamic query modification, dynamic revision of query results, and flexible optimization. These systems focus on distributing the query processing among multiple nodes. The actual query processing at the nodes is performed by Aurora system. In GSN, we provide an integrated platform which not only can be used to process data streams at a single node, but also, can be used as a distributed data stream processing platform in which nodes are communicating with each other through a symmetric peer-to-peer approach.

So far only few architectures to support interconnected sensor networks exist. Sgroi et al. [82] suggest basic abstractions, a standard set of services, and an API to shield application developers from the details of the underlying sensor networks. However, the focus is on systematic definition and classification of abstractions and services, while GSN takes a more generic approach and provides a complete integrated solution encompassing an acquisition layer, a continuous query processor, data stream storage over multiple storage models (e.g., relational databases, flat files, distributed file systems, etc.) and publishing systems. Capabilities of GSN are exposed both through APIs and a declarative interface.

Rooney et al. [79] propose so-called EdgeServers to integrate sensor networks into enterprise networks. EdgeServers filter and aggregate raw sensor data (using application specific code) to reduce the amount of data forwarded to application servers. The system uses publish/subscribe style communication and also includes specialized protocols for the integration of sensor networks. While GSN provides a general-purpose infrastructure for sensor network deployment and distributed query processing, the EdgeServer system targets enterprise networks with application-based customization to reduce sensor data traffic in closed environments.

GSN's approach is similar to TelegraphCQ's approach. In TelegraphCQ, the authors modified PostgreSQL database engine to introduce the streaming processing concepts right inside the engine. In GSN, we decided to have the modifications externalized from the database engine therefore, giving end-users the option of freely choosing their underlying database engine. This option provides the best of both worlds, as in most of the real-world deployments, end-users are more interested in closed source database engines such as Oracle database server.

GSN's stream processing engine is built on top of a relational database engine, thus standard database tables are used for storing and retrieving the streaming data during GSN's data processing. In GSN, we introduced the concept of *virtual sensor*

which abstracts from implementation details of access to sensor data and defines data stream processing to be performed. Local and remote virtual sensors, their data streams and the associated query processing can be combined in arbitrary ways and thus enable the users to build a data-oriented "Sensor Internet" consisting of sensor networks connected via GSN. In the relational view of the streaming data, each sensor reading corresponds to a new tuple in the related table. As GSN employs a standard relational database as its low-level query processing engine, the question is how to present the streaming logic in a form understandable by a standard database engine. We address this problem by using a query translator which gets an SQL query and the stream processing directives as provided in the virtual sensor definition as inputs and translates these inputs into a query executable in a standard database. The query translator relies on special support functions which emulate stream-oriented constructs in a database. These support functions are database dependent. With GSN, we provide adapters for various database engines.

## 2.3 Data Stream Management Platforms

With the advent of new sensing devices and recent advances in wireless sensor network technology, the demand for providing a large scale stream processing platform for processing data produced by these devices is higher than ever before. Data stream processing platforms are systems which not only include a data stream processing engine but also include the data acquisition layer, data publishing and delivery mechanisms. In this section we present the major data stream processing platforms and their differences compared to GSN.

### 2.3.1 HiFi

HiFi[48] provides efficient, hierarchical data stream query processing to acquire, filter, and aggregate data from multiple devices in a static environment as shown in Figure 2.2. In contrast to HiFi, GSN takes a peer-to-peer perspective assuming a dynamic environment and allowing any node to be a data source, data sink, or data aggregator.

HiFi system is using TelegraphCQ[32] as its stream processing engine, enabling dispersed, widely distributed organizations to continuously monitor, manage and optimize their operations.

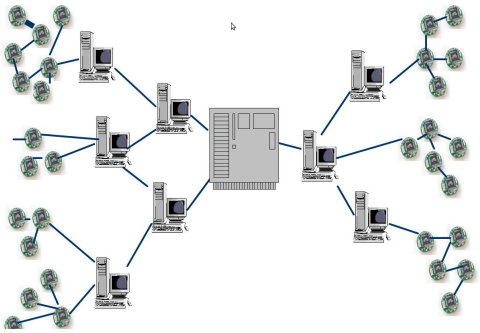


**Figure 2.2.** HiFi Data Stream Processing Platform [48]

The HiFi system is different from GSN in the following areas. First, TelegraphCQ moves data to a centralized processing point. Second, TelegraphCQ does not address *different types* of data streams produced by different sensor networks (HiFi just addresses different stream rates). The goal of GSN is providing a stream processing environment focusing on simplicity and rapid deployment.

The goal of HiFi system is providing a homogeneous stream processing for static environments with the focus on decreasing and filtering data by removing redundant and meaningless data elements from the observations as soon as possible (e.g., in the first steps) and transmitting merely the useful parts to the root of the hierarchy. HiFi doesn't address joining heterogeneous sensor networks problem and resulting issues such as integrating data streams with dynamic data structures.

### 2.3.2 IrisNet

The IrisNet[52] system is aimed to providing infrastructure that allows data consumers to access globally distributed collections of sensors which are attached to computers connected to the Internet (mostly high bit-rate sensors such as Webcams).

IrisNet consists of two layers. First, Sensor Agents (SAs) are the nodes implementing generic interface (API) provided by IrisNet. Second, Organizational Agents (OAs) are the nodes that implement a distributed database for storing observations from SAs. The OAs store sensor data in a hierarchical, distributed XML database. This database is modeled after the design of the Internet DNS and supports XPath queries. In contrast to that, GSN follows a symmetric peer-to-peer approach as already mentioned and supports relational queries using SQL.

### 2.3.3 HourGlass

HourGlass[83] is aimed to be an Internet based overlay network of well-connected computers providing services such as registration, discovery, routing, filtering, aggregation, compression and buffering. One of the main emphasis in HourGlass is providing an infrastructure that handles intermittent disconnections smoothly by keeping the data inside the network using buffering service, and retransmitting those data to the client when a new connection is established or previous connections are reestablished.

HourGlass consists of three components, *circuit manager*s, *registry* and *service provider*s. A service is a stream of data which has a topic with a number of addressing predicates. A topic is a mutually agreed upon name and a predicate is a logical statement which could be either true or false for a given service. A service provider is a computer which hosts several services. A client interested in a service must either provide the circuit manager, the exact address of the service provider in form of an IP-address and port number or a topic and predicate(s) which should hold true for the service. In the latter case, it's the responsibility of the circuit manager to map the topic and predicates pair to an appropriate base computer. Similarly to GSN, HourGlass tries to hide internals of sensors from the user, but focuses on maintaining quality of service of data streams in the presence of disconnections, while GSN is more targeted at flexible configuration, generic abstractions, and distributed query support.

### 2.3.4 TinyDB and Cougar

In the context of sensor networks, there exist different methods for retrieving data from a sensor network. The naive way is to use a low level programming API of the operating system (e.g., TinyOS[49]), to do the sensing and sending the data. This approach is time consuming and hardware dependent.

The Cougar[92] and TinyDB[69] systems are designed to facilitate this process and hide the underlying details by providing declarative query languages for getting the data from the sensor network. In these systems, when the user posts a query, the underlying system generates an optimized and efficient (in terms of communication cost and energy) query execution plan for the in-network query processing. Therefore these systems not only reduce the deployment costs (e.g., providing high-level query languages) but also reduce the energy consumption.

Based on the TinyOS operating system, the TinyDB[69] platform is a query

processing system for extracting information from a network of TinyOS-based motes using a declarative SQL-like query language named Tiny-SQL. The goal of TinyDB is providing an interface for getting data without specifying how to get it. TinyDB collects that data from motes, filters, aggregates and routes the data packets using a multi-hop power-efficient algorithm.

In GSN, we are pursuing a different goal. We are interested in efficient integration of multiple heterogeneous sensor networks in addition to posting complex queries on the underlying data stream. GSN gets the streaming data from the sensor network using the wrappers.

In GSN, a wrapper is an interface between the platform and an actual stream producer (e.g., a sensor network). If the data coming into GSN is produced by a physical sensor network, Cougar and TinyDB systems can be used for acquiring the data from the sensor network and delivering it to the sink node, which in turn delivers the data to the appropriate wrapper inside GSN. Therefore, Cougar and TinyDB systems are in fact complementary to GSN platform.

## 2.4 Continuous Query Languages

In this section we present the basic concepts behind data stream processing. We also describe different query languages designed for processing streaming data.

In order to process streaming data, the standard approach is to specify a query with at least two extra properties associated with it, window size and sliding value[2]. The window size is used to limit the actual data used for the processing (execution) to a certain range in time or number of values. The sliding predicate is introduced to specify the execution condition for the query. The execution of the query is triggered whenever the sliding condition is satisfied, implying a possibly infinitely long periodic execution of the query.

For instance, one can express the interest of obtaining the average of a temperature sensor over the last 10 minutes, and doing so periodically every 2 minutes, by simply providing the window size of 10 minutes and sliding value of 2 minutes to the stream processing engine. As indicated before, each time the sliding condition is satisfied (e.g., 2 minutes passed from the previous execution) the actual action, computing the average over the last 10 minutes, is performed. Note that in some cases the execution of the action is also called *movement of the sliding window*.

[2]In this thesis, we use the terms *sliding predicate*, *sliding value* and *sliding condition* interchangeably.

In the context of stream processing, three main approaches used for designing the continuous query language (query with window size and sliding value).

### 2.4.1 Data Flow Based Languages

In the Aurora[10] system, one can construct queries visually through a graphical user interface by arranging boxes and joining them with arrows representing the data flows between boxes. This graphical presentation is then used as the input for the query planner. The query planner further optimizes the processing flow internally in the optimization phase.

### 2.4.2 SQL Based Languages

The most popular stream querying languages are extensions of declarative relational query languages such as CQL[21] which is introduced by the STREAM[20] project and StreaQuel[55] which is introduced by the TelegraphCQ[32] system.

CQL's syntax is very similar to the standard SQL language with extensions defined for handling streaming data. For instance in CQL the query `SELECT * FROM S1 [ROWS 100] WHERE S1.A > 10` will use the last 100 values as the count based window from `S1` for processing the query. That query processing in CQL is data-driven which means the query will be executed for each stream element that arrives at the system.

The StreaQuel language isolates the streaming semantics from the query language. The window size used for the query is defined using a for-loop construct. Let S be a stream and let ST be the start time of the query. To specify the sliding window consisting the last 20 time units over stream S which runs for 100 time units :

```
for  ( t = ST; t < ST + 100; t++ )
  WindowIs ( S, t-20, t)
```

When using StreaQuel, the actual query for performing filtering and joins is specified in an SQL like syntax. Note that, compared to CQL, the StreaQuel supports streaming and periodic query processing. The other family of query languages designed for streaming data are Object based languages such as COUGAR. In this approach the system models the stream source as abstract data types (ADTs) whose interface consists of the sensor's data processing methods with a SQL like query language.

### 2.4.3 Declarative SQL Based Languages

GSN separates the stream related constructs from the query similar to the StreaQuel [55]. The data stream processing related concepts can be specified using standard XML syntax (e.g., window size), which makes the continuous query more like a standard SQL. Separating the filtering and the stream related constructs has two advantages. First, using standard SQL queries implies that more users can understand and use the system without having to learn a new language. Second, the separation of concerns implies that users can present the stream related processing logic separately from the filtering concepts (e.g., SQL query).

In GSN we support the standard window specifications[70] (described in detail in chapter 3) such as time and count based windows and sliding predicates. In GSN, we also introduced a simple load shedding extension which is aimed to bound the rate of the data stream through random sampling of the data stream within the current window when the rate exceeds a certain user defined threshold.

In GSN users can also post queries encompassing *continuous and historical data*. Users can use the standard SQL join on live data streams combined with the static data (e.g., static data mapping of GPS locations to the room numbers) allowing users to integrate several streams with the static data storage. In GSN, users can issue complex SQL queries (such as different type of joins, sub-queries, ordering, grouping, unions, intersections, etc.) within stream processing semantics.

## 2.5 Distributed Publish/Subscribe Systems

Distributed publish/subscribe systems (DPSS) is one of the extensively researched subjects in both the networking community and the data management community. Many research efforts have been focused on enabling scalable and efficient data dissemination services to a large number of users. For instance, efficient matching of events with subscriptions within a broker is studied in [16]. Authors in [28] presented the architecture design of a DPSS with a number of widely distributed brokers. Additionally, several systems providing publish/subscribe style query processing comparable to GSN exist, for example, [57].

Distributed publish/subscribe systems are one of the highly relevant systems to stream processing platforms. This commonality comes from the fact that both systems are sharing the goal of efficiently delivering massive amount of data to the end users. In the case of stream processing engines, having the continuous queries registered over the streams is very similar of having subscriptions on top of the

data providers. The main difference lies in the use of new stream processing constructs which don't exist in publish/subscribe systems. Predicates such as window size and sliding value define a new data processing paradigm. The window size limits the number of the data items used in the processing and the sliding value enforces a certain execution pattern over the data stream. There exist systems in which publish/subscribe systems are used as the basis for building a stream processing engine. For instance the approach proposed in [96] employs a distributed publish/subscribe system to disseminate the stream data from the data sources to the processing servers and was focused on optimizing the allocation of the queries to the servers.

# Chapter 3

# Global Sensor Network

## Overview

With the price of wireless sensor technologies diminishing rapidly we can expect large numbers of autonomous sensor networks being deployed in the near future. These sensor networks will typically not remain isolated but the need of interconnecting them on the network level to enable integrated data processing will arise, thus realizing the vision of a global "Sensor Internet." This requires a flexible middleware layer which abstracts from the underlying, heterogeneous sensor network technologies and supports the following requirements.

- Runtime reconfiguration.
- Flexible resource management.
- Integrated data acquisition and processing.
- Flexible data stream acquisition layer.
- Common services in one package.

This chapter presents the Global Sensor Network (GSN) middleware which addresses these goals. We present GSN's conceptual model, abstractions, and architecture, and demonstrate the efficiency of the implementation through experiments with typical high-load application profiles. The GSN implementation is available from http://gsn.sourceforge.net/.

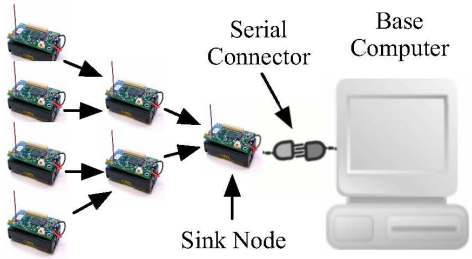



**Figure 3.1.** GSN model

## 3.1 Introduction

The availability of cheap and smart wireless sensing devices provides unprecedented possibilities to monitor the physical world. Until now, research in the sensor network domain has mainly focused on routing, data aggregation, and energy conservation inside a single sensor network while the integration of multiple sensor networks has only been studied to a limited extent. However, as the price of wireless sensors diminishes rapidly we can soon expect large numbers of autonomous sensor networks being deployed. These sensor networks will be managed by different organizations but the interconnection of their infrastructures along with data integration and distributed query processing will soon become an issue to fully exploit the potential of this "Sensor Internet." This requires platforms which enable the dynamic integration and management of sensor networks and the produced data streams.

The Global Sensor Network (GSN) platform aims at providing a flexible middleware to accomplish these goals. GSN assumes the simple model shown in Figure 3.1: A sensor network internally may use arbitrary multi-hop, ad-hoc routing algorithms to deliver sensor readings to one or more sink node(s). A sink node is a node which is connected to a more powerful base computer which in turn runs the GSN middleware and may participate in a (large-scale) network of base computers, each running GSN and servicing one or more sensor networks. The base computer can be used to perform further (application dependent) processing on data in addition to providing various services (such as the storage for keeping the history of readings) and interfaces (such as web services interface, http based interface, etc) to enable local and remote users to access the sensor readings and interact with the sensor network such as sending commands to the sensor network.

We do not make any assumptions on the internals of a sensor network other

than that the sink node is connected to the base computer via a software wrapper conforming to the GSN API. The wireless sensor networks can use any kind of energy saving (such as TMAC or SMAC protocols for the MAC Layer) and routing protocols. While the nodes inside the sensor network are most likely communicating with each other for various reasons such as building the internal routing table, depending on the application requirement, it is most likely that these internal communications are hidden from the base computer and hence the GSN. The base computer only receives the data packets from the sensor network whenever a packet specifically addressed to it or broadcasted.

On top of this physical access layer GSN provides so-called *virtual sensors* which abstract from implementation details of access to sensor data and define the data stream processing to be performed. Local and remote virtual sensors, their data streams and the associated query processing can be combined in arbitrary ways and thus enable the user to build a data-oriented "Sensor Internet" consisting of sensor networks connected via GSN.

In the following, we start with a detailed description of the virtual sensor abstraction in Section 3.2, discuss GSN's data stream processing and time model in Section 3.3, and present GSN's system architecture along with a discussion of essential implementation details in Section 3.4. Section 3.5 presents the major implementation decisions of GSN, specifically the optimization techniques for sharing internal resources. The network layer of GSN is presented in detail in Section 3.6. We evaluate the performance of GSN in Section 3.7.

## 3.2 Virtual sensors

The key abstraction in GSN is the *virtual sensor*. Virtual sensors abstract from implementation details of access to sensor data and correspond either to a data stream received directly from sensors or to a data stream derived from other virtual sensors. A virtual sensor can be any kind of data producer, for example, a real sensor, a wireless camera, a desktop computer, or any combination of virtual sensors. A virtual sensor may have any number of input data streams and produces exactly one output data stream (with predefined format) based on the input data streams and arbitrary local processing. The specification of a virtual sensor provides all necessary information required for deploying and using it, including:

- Metadata used for identification and discovery.

- The details of the data streams which the virtual sensor consumes and produces.
- Declarative SQL-based specification of the data stream processing (filtering and integration) performed in a virtual sensor.
- Processing class which performs the more advanced and complex data processing (if needed) on the output stream before releasing it.
- Functional properties related to persistency, error handling, life-cycle, management, and physical deployment.

To support rapid deployment, the virtual sensors are provided in a human readable declarative format (XML). Figure 3.2 shows an example which defines a virtual sensor that reads two temperature sensors and in case both of them have the same reading above a certain threshold in the last minute, the virtual sensor returns the latest picture from the webcam in the same room together with the measured temperature.

A virtual sensor has a unique name (the `name` attribute in line 1) and can be equipped with a set of key-value pairs representing the logical addressing of the virtual sensor (lines 12–17), i.e., associated with metadata. The addressing information can be registered and discovered in GSN and other virtual sensors can use either the unique name or logical addressing based on the metadata to refer to a virtual sensor. We have defined certain addressing keys which are specifically used by GSN's web interface. In GSN if a given virtual sensor has the addressing values for the both `latitude` (line 15) and `longitude` (line 16) keys, the default GSN web interface uses these geographical locations to show the sensor on the global map.

The example specification in Figure 3.2 defines a virtual sensor with three input streams which are identified by their metadata[1] i.e., by logical addressing. For example, the first temperature sensor is addressed by specifying two requirements on its metadata, namely that it is of type temperature sensor and at a certain physical location. By using multiple input streams Figure 3.2 also demonstrates GSN's ability to access multiple stream producers simultaneously. For the moment, we assume that the input streams (two temperature sensors and a webcam) have already

[1]Note that the support for distributed directory/registry service had been removed from GSN's source code thus as of January 26, 2010, we only support physical addressing for identifying the data sources.

```
<virtual-sensor name="room-monitor"
          protected="false" >
  <processing-class>
     <class-name>gsn.vsensor.BridgeVirtualSensor</class-name>
     <init-params/>
     <output-structure>
        <field name="image" type="binary:jpeg" />
        <field name="temp"  type="int" />
     </output-structure>
  </processing-class>
  <life-cycle pool-size="10" />
  <addressing>
      <predicate key="geographical">BC143</predicate>
      <predicate key="usage">room monitoring</predicate>
      <predicate key="latitude">46.5214</predicate>
      <predicate key="longitude">6.5676</predicate>
  </addressing>
  <storage history-size="10h" />
  <streams>
      <stream name="cam">
         <source name="cam"  storage-size="1" >
            <address wrapper="remote">
               <predicate key="geographical">BC143</predicate>
               <predicate key="type">Camera</predicate>
            </address>
            <query>select * from WRAPPER</query>
         </source>
         <source name="temperature1" storage-size="1m" >
            <address wrapper="remote">
               <predicate key="type">temperature</predicate>
               <predicate key="geographical">BC143-N</predicate>
            </address>
            <query>select AVG(temp1) as T1 from WRAPPER</query>
         </source>
         <source name="temperature2"  storage-size="1m" >
            <address wrapper="remote">
               <predicate key="type">temperature</predicate>
               <predicate key="geographical">BC143-S</predicate>
            </address>
            <query>select AVG(temp2) as T2 from WRAPPER</query>
         </source>
         <query>
             select cam.picture as image, temperature.T1 as temp
             from   cam, temperature1
             where  temperature1.T1 > 30 AND
                    temperature1.T1 = temperature2.T2
         </query>
      </stream>
   </streams>
</virtual-sensor>
```

**Figure 3.2.** A virtual sensor definition

been defined in other virtual sensors (how this is done, will be described later in this chapter).

In GSN data streams are temporal sequences of timestamped tuples (also known as `Stream Elements`). This is in line with the model used in most stream processing systems. The structure of the output data stream a virtual sensor produces is encoded in XML as shown in lines 6 – 9 (the `output-structure` part). The structure of the input streams is learned from the respective specifications of their virtual sensor definitions.

In GSN data stream processing can be performed in three levels:

- Directly on the sources (lines 26, 33, and 40).
- Combining data from the different input streams and producing the temporary output stream (lines 43-46).
- Passing the temporary output stream through a processing class (a processing logic represented in some programming language). This part is presented by lines 3 – 10. Note that since the final output of the virtual sensor is produced by the processing class, the actual output structure of the virtual sensor should strictly conform the output format of the processing class [2].

The data filtering logic on the data sources is specified through the SQL queries which refer to the actual data source by the reserved keyword `WRAPPER` (the data source is logically represented as a relational table which called `wrapper`). The attribute `wrapper="remote"` indicates that the data stream is obtained through the network from another virtual sensor, which can be located in any other GSN instance accessible through the network.

In the case of a directly connected local sensor, the `wrapper` attribute would reference the required wrapper[3]. For example, `wrapper="tinyos"` would denote a TinyOS-based sensor whose data stream is accessed via GSN's TinyOS wrapper [4]. GSN already includes wrappers for all major TinyOS platforms (Mica2, Mica2Dot, etc.), for wired and wireless (HTTP-based) cameras (e.g., AXIS 206W), several RFID readers (Texas Instruments, Alien Technology), Bluetooth devices,

[2] As of January 26, 2010, the order and the type of the fields should match.

[3] As of January 26, 2010, all the wrappers have to be written in the Java language. The actual code for accessing the sensor can be written in any language as long as there is a possibility of communicating the data to the hardware through Java (e.g., interfacing Java to the existing C code or the communicating through the serial ports).

[4] In GSN, we have multiple TinyOS wrappers each corresponding to different versions and packet formats. The details are out of the scope of this chapter but are fully documented in the GSN website.

Shockfish, WiseNodes, epuck robots, etc. The implementation effort for wrappers is rather low, for example, the RFID reader wrapper has 50 lines of code (LOC), the TinyOS wrapper has 120 LOC, and the generic serial wrapper has 180 LOC.

In the given example the output stream joins the data received from two temperature sensors and returns a camera image if certain conditions on the temperature readings are satisfied (lines 43–46). To enable the SQL statement in lines 43–46 to produce the output stream, it needs to be able to reference the required sources which is accomplished by the `name` attribute (lines 21, 28, and 35) that defines a symbolic name for each stream source.

The output structure definition of the virtual sensor is directly affected by the data processing logic that is performed by the virtual sensor's processing class. GSN provides multiple processing classes each of which is designed to perform different tasks (e.g., charts, network plots, filtering, ...). In our example we are using the `gsn.vsensor.BridgeVirtualSensor` as the processing class. The `gsn.vsensor.BridgeVirtualSensor` class is special in the sense that unlike most of the other GSN's processing classes, this class does not perform any further processing on its input stream thus it does not alter the data nor the structure of its input.

Since the structure of the virtual sensor output is not altered through using the `gsn.vsensor.BridgeVirtualSensor` processing class, the final structure of the virtual sensor's output is determined through the SQL statement at line 43. Therefore we need to make sure that, the data fields in the `select` clause matches the definition of the output structure in lines 6–9 (the order is also important). It is recommended to use `gsn.vsensor.BridgeVirtualSensor` as long as the processing performed in the virtual sensor through the SQL queries are sufficient and no further processing is required before publishing the sensor data to the outside.

In the design of GSN specifications we decided to separate the temporal aspects from the relational data processing using SQL. The temporal processing is controlled by various attributes provided in the input and output stream specifications, e.g., the attribute `storage-size` (lines 21, 28, and 35) defines the window size. Due to its specific importance the temporal processing will be discussed in detail in Section 3.3.

In addition to the specification of the data-related properties, a virtual sensor also includes high-level specifications of functional properties: The `<life-cycle>` element (line 11) enables the control and management of resources provided to a virtual sensor such as the maximum number of threads/queues available for pro-

cessing, the `<storage>` element (line 18) allows the user to control how the output stream data is persisted.

For example, in Figure 3.2 the `<life-cycle>` element in line 11 specifies a maximum number of 10 threads, which means that if the pool size is reached, data will be dropped (if no pool size is specified, it will be controlled by GSN depending on the current load), the `<storage>` element in line 18 defines that only the most recent 10 hours output of this virtual sensor to be stored. The `storage-size` attribute in line 21 defines the window size of 1 stream element. That's the most recent image taken by the webcam irrespective of the time it was taken.

In GSN, we can specify the set of values either by time or count. In the count based representation one provides the values through integers. For instance `slide='2'` or `history-size='100'`. The time based representation consists of an integer directly postfixed (without any space characters) with one of the predefined time units. As of January 26, 2010, we have `d`,`h`,`m`,`s` units which are corresponding to days, hours, minutes and seconds respectively. As a time based example, we might have `storage-size='1m'`.

The `storage-size` attributes in lines 28 and 35 define a window of one minute for the amount of sensor readings subsequent queries will be run on, i.e., the `AVG` operations in lines 33 and 40 are executed on the sensor readings received in the last minute which of course depends on the rate at which the underlying temperature virtual sensor produces its readings. Note that when the `storage-size` is anything other than *1*, the virtual sensor author should be aware of the possibility of duplicated stream elements (discussed in more detail in section 3.3).

The query that produces the output stream (lines 43–46) also demonstrates another interesting capability of GSN as it also mediates among three different flavors of queries: The virtual sensor itself uses continuous queries on the temperature data, a "standard" database query is performed on the camera data and the query produces a result only if certain conditions are satisfied, i.e., a notification analogous to pub/sub or active rules.

Virtual sensor is a powerful abstraction mechanism which enables the users to declaratively specify sensors and create arbitrary complex data processing chains. Virtual sensors can be deployed while a GSN instance is running without having to stop the system. Also dynamic unloading is supported but should be used carefully as unloading a virtual sensor may have undesired (cascading) effects.

## 3.3 Data stream processing and time model

Data stream processing has received substantial attention in the recent years in various application domains, such as network monitoring or telecommunications. As a result, a rich set of query languages and query processing approaches for data streams exist. A central building block in data stream processing is the time model as it defines the temporal semantics of data and thus determines the design and implementation of a system. Currently, most stream processing systems use a global reference time as the basis for their temporal semantics because they were designed for centralized architectures in the first place. As GSN is targeted at enabling a distributed "Sensor Internet," imposing a specific temporal semantics seems inadequate and maintaining it might come at unacceptable cost. GSN provides the essential building blocks for dealing with time, but leaves temporal semantics largely to applications allowing them to express and satisfy their specific, largely varying requirements. In our opinion, this pragmatic approach is viable as it reflects the requirements and capabilities of sensor network processing.

In GSN a data stream is a set of timestamped tuples also known as Stream Elements. The order of the data stream is derived from the ordering of the timestamps. GSN provides basic support for managing and manipulating the timestamps. The following essential services are provided:

1. A local clock at each GSN instance.
2. Implicit management of a timestamp attribute (reserved field called `TIMED`).[5][6]
3. Automatic timestamping of tuples upon arrival at the GSN in case the tuples (stream elements) don't have any timestamp (no `TIMED` field available).
4. Windowing mechanism which allows the user to define count- or time-based windows on data streams.
5. Sliding mechanism which allows the users to define count- or time-based sliding behaviors on the data streams.

In this way it is always possible to trace the temporal history of data stream elements throughout the processing history. Multiple time attributes can be associated with data streams (as long as only one of them is named `TIMED`) and can

[5] All timestamps in GSN are represented in milliseconds using 64-bit integers.

[6] As the timestamp (e.g., the `TIMED` field) is always present, it is not required to specify the `TIMED` field in the `output-structure` section of the virtual sensor. In fact, specifying the `TIMED` field in the output structure causes error and GSN refuses to load the virtual sensor.

be manipulated through SQL queries. Thus sensor networks can be used as observation tools for the physical world, in which network and processing delays are inherent properties of the observation process which cannot be made transparent by abstraction. Let us illustrate this by a simple example: Assume a bank is being robbed and images of the crime scene taken by the security cameras are transmitted to the police. For the insurance company the time at which the images are taken in the bank will be relevant when processing a claim, whereas for the police report the time the images arrived at the police station will be relevant to justify the time of intervention. Depending on the context the robbery is thus taking place at different times.

As tuples (sensor readings) are timestamped, queries can also explicitly deal with time. For example, the query in lines 43–46 of Figure 3.2 could be extended such that it explicitly specifies the maximum time interval between the readings of the two temperatures and the maximum age of the readings. This would additionally require changes in the source definitions as the sources then must provide this information (more detailed example below). The averaging of the temperature readings (lines 33 and 40) would have to be also changed to be explicit in respect to the time dimension.

In order to concretely show the time management inside GSN, we would like to simulate the above scenario through two different virtual sensors (only the input stream parts are presented). Say there exists a virtual sensor called *camera-vs* hosted on a GSN instance which listens to port `80` on a machine with an IP address of `1.2.3.4`. The virtual sensor used by the police and the one used by the insurance are depicted in Figures 3.3 and 3.4. The stream specified in Figure 3.3 has a query in line 7 for retrieving both the picture and the time stamp from the remote virtual sensor therefore the remote timestamp is used by GSN for the internal calculations. Now consider the stream specified in Figure 3.4 which has a small change compared to the one in Figure 3.3, the latter does not select the timestamp field hence GSN automatically adds the local reception time to every tuple it receives from the remote source.

In order to further elaborate the time management issue, consider the stream source specified in Figure 3.5. This example combines both the local time and remote time in order to measure the latency associated with each tuple and uses the latency as a condition inside the selection criteria (e.g., only accepting the tuples

```
<stream name="cam">
    <source name="cam"  storage-size="1" >
        <address wrapper="remote">
            <predicate key="host">1.2.3.4</predicate>
            <predicate key="port">80</predicate>
            <predicate key="name">camera-vs</predicate>
        </address>
        <query>select PICTURE, TIMED from WRAPPER</query>
    </source>
            <query>
        select PICTURE, TIMED from cam
    </query>
</stream>
```

**Figure 3.3.** A stream using the remote timestamp.

```
<stream name="cam">
    <source name="cam"  storage-size="1" >
        <address wrapper="remote">
            <predicate key="host">1.2.3.4</predicate>
            <predicate key="port">80</predicate>
            <predicate key="name">camera-vs</predicate>
        </address>
        <query>select PICTURE from WRAPPER</query>
    </source>
    <query>
        select PICTURE, TIMED from cam
    </query>
</stream>
```

**Figure 3.4.** A stream using the local (arrival) timestamp.

```
<stream name="cam">
    <source name="cam"  storage-size="1" >
        <address wrapper="remote">
            <predicate key="host">1.2.3.4</predicate>
            <predicate key="port">80</predicate>
            <predicate key="name">camera-vs</predicate>
        </address>
        <query>select PICTURE,
                TIMED as REMOTE_TIMED
                from WRAPPER</query>
    </source>
    <query>
        select PICTURE, REMOTE_TIMED AS TIMED from cam where
        (cam.TIMED - cam.REMOTE_TIMED) < 5
    </query>
</stream>
```

**Figure 3.5.** A stream using both local and remote timestamps.

which are not delayed by the network for more than 5 milliseconds).

### 3.3.1 Window Size and Sliding Values inside GSN

In order to deal with the streaming data, the standard way is to specify a query with at least two extra properties associated with it, window size and sliding value. The window size is used to limit the actual data used for the processing (execution) to a certain range in time or number of values. The sliding value is introduced to specify the execution condition for the query. The execution of the query is triggered whenever the sliding condition is satisfied implying a possibly infinitely long periodic execution of the query, therefore in stream processing systems, continuous queries are executed whenever the sliding occurs.

For instance, one can express the interest of obtaining the average of a temperature sensor over the last 10 minutes, and doing so periodically every 2 minutes, by simply providing the window size of 10 minutes and sliding value of 2 minutes to the stream processing engine. As indicated before, each time the sliding condition is satisfied (e.g., 2 minutes passed from the previous execution) the actual action, computing the average over the last 10 minutes, is performed. Note that in some research papers the execution of the action is also called *movement of the sliding window*.

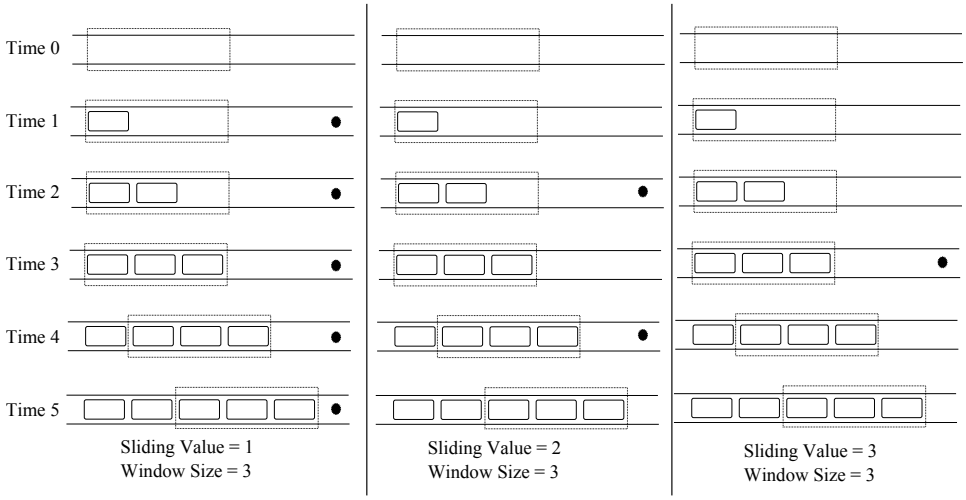



**Figure 3.6.** Illustration of the different sample sliding and window values.

The temporal processing in GSN is defined using the sliding value and the window size. Every data source in GSN can have one `slide`[7] and one `storage-size`[8] attribute. Both values can be represented in the form of count-based or time-based values (described earlier in this chapter). Figure 3.6 visually represents the query execution inside GSN with different sliding and window values. We used a black dot in the Figure to represent the triggering of execution. For instance, if both the window size and the sliding values are 3, and say we have received 5 stream elements in total, our continuous query have been executed only once (at the *timestamp 3*) during its life time. One can extend the above paradigm to create virtual sensors to support the integration of continuous and historical data. For example, if the user wants to be notified when the temperature is 10 degrees above the average temperature in the last 24 hours, he/she can simply define two sources, getting data from the same wrapper but with different window sizes, i.e., 1 (count) and 24h (time), and then simply write a query specifying the original condition with these sources.

The production of a new output stream element of a virtual sensor is always triggered by the arrival of a data stream element from one of its input streams, thus processing in GSN is data-driven. As described before, a stream can have multiple sources. Once the window of one of the sources of a stream slides, the following processing steps are performed:

[7] Default value is 1, therefore this attribute can be omitted.
[8] No default value defined.

1. By default the new data stream element is timestamped using the local clock of the virtual sensor provided that the stream element had no timestamp (this step is optional).

2. Based on the timestamps, the stream elements are selected according to the window size and the resulting sets of relations are unnested into flat relations.

3. The queries defined on the source are evaluated and the results are stored into temporary (in-memory) tables.

4. The stream query for producing the input for the processing class is executed based on the temporary tables.

5. The resulting stream elements are forwarded to the processing class.

6. The output of the processing class is stored and simultaneously forwarded (e.g., notifications) to all the consumers of this virtual sensor.

Figure 3.7 shows the logical data flow inside a GSN node.

Additionally, GSN provides a number of attributes in the virtual sensor definition to control the data stream rates. The values used for controlling these rates are usually presented as floating numbers between 0.0 to 1.0. The data rate management is useful whenever one wants to drop stream elements with some fixed probability to achieve load shedding. For instance, if one has a temperature source that keeps producing data with a very high rate, one might want to sample the produced values thus making the processing load lighter. For instance if one sets the sampling-rate to 0.75, any received stream element from the wrapper is going to be included in the window with a probability of 75 out of 100. Thus, on average 25 random stream elements will be dropped out of every 100 elements. In most of the cases one typically sets the rate control attributes to 1.0 to make sure no stream element is dropped.

The rate control can be applied in the following three different levels [9]:

- At the source level by providing `sampling-rate` attribute (real number in the range [0.0 . . . 1.0]).

- At the stream level by providing `rate` attribute (integer value above zero).

[9] Please refer to the virtual sensor quick reference for the syntactical information about different portions of the virtual sensor file. This document is available in the GSN's web site.

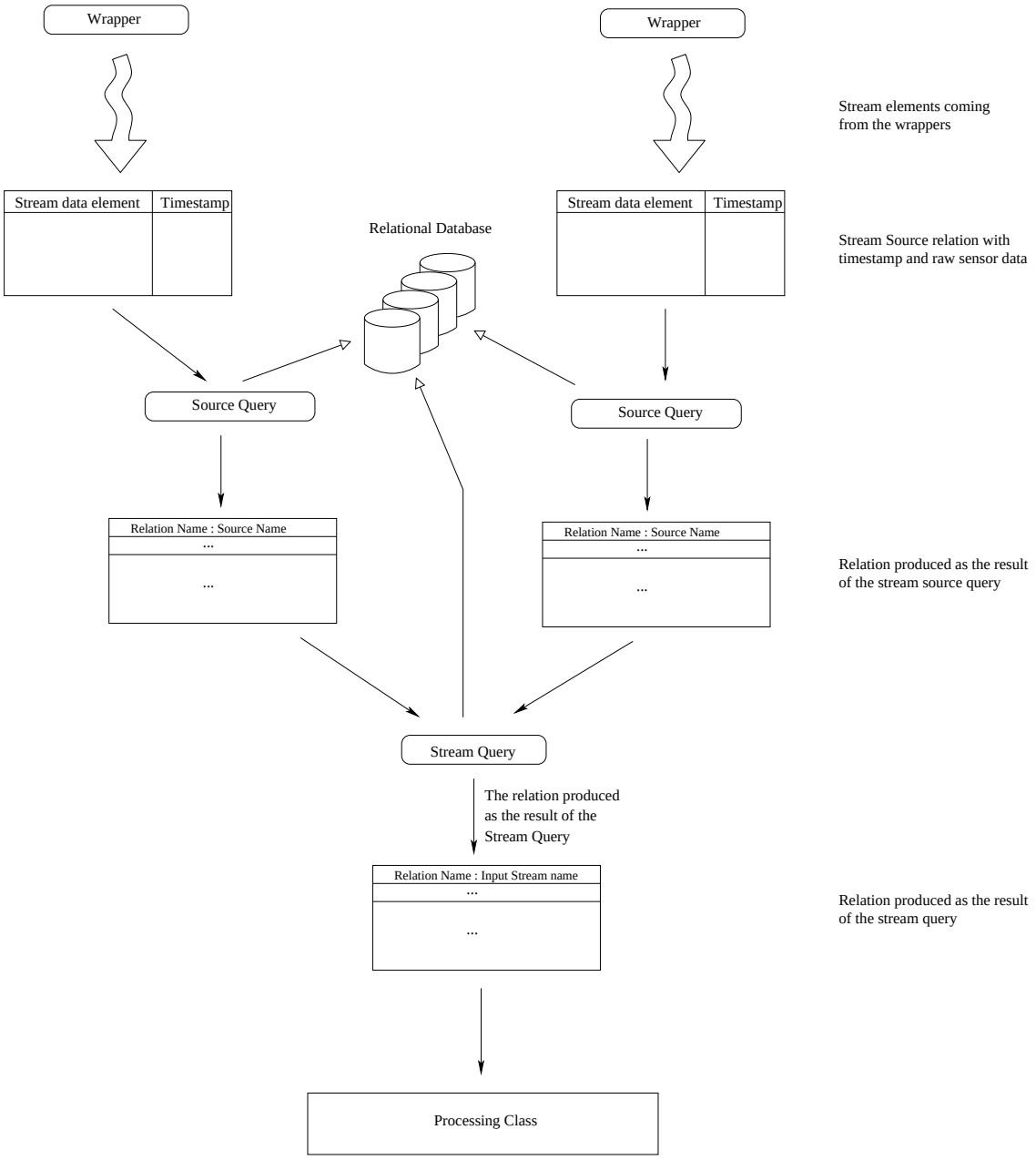



**Figure 3.7.** Conceptual data flow in a GSN node

- At the virtual sensor output level by providing `output-specification` → `rate` attribute (positive integer).

As noted above, if the rate control is a positive integer, it defines the minimum allowed time difference between to successive stream elements. For instance, if one is interested in receiving an average of a given sensor readings once an hour but the sensor underneath can produce arbitrary number of stream elements (e.g., due to uncontrollable packet losses in the internal network), he or she can express this behavior by setting the rate attribute of the virtual sensor output (`output-specification` → `rate`) to "3600000" (one hour is 3,600,000 milliseconds).

### 3.3.2 Continuous Query Language and GSN

To specify the data stream processing a suitable language is needed. A number of proposals exist and in this paragraph we compare the GSN approach to the major continuous query languages in the literature. In the Aurora project [37] (http://www.cs.brown.edu/research/aurora/) users can compose stream relationships and construct queries in a graphical representation which is then used as input for the query planner. The Continuous Query Language (CQL) suggested by the STREAM project [19] (http://www-db.stanford.edu/stream/) extends standard SQL syntax with new constructs for temporal semantics and defines a mapping between streams and relations. Similarly, in Cougar [93] (http://www.cs.cornell.edu/database/cougar/) an extended version of SQL is used, modeling temporal characteristics in the language itself. The StreaQuel language suggested by the TelegraphCQ project [33] (http://telegraph.cs.berkeley.edu/) follows a different path and tries to isolate temporal semantics from the query language through external definitions in a C-like syntax. For example, for specifying a sliding window for a query a *for*-loop is used. The actual query is then formulated in a SQL-like syntax.

GSN's approach is related to TelegraphCQ's as it separates the time-related constructs from the actual query. Temporal specifications, e.g., the window size and rates, are specified in XML in the virtual sensor specification, while data processing is specified in SQL. Using this design, GSN can support SQL queries with the full range of operations allowed by the standard SQL syntax, i.e., joins, subqueries, ordering, grouping, unions, intersections, etc. The advantage of using SQL is that it is well-known and SQL query optimization and planning techniques can be directly applied.

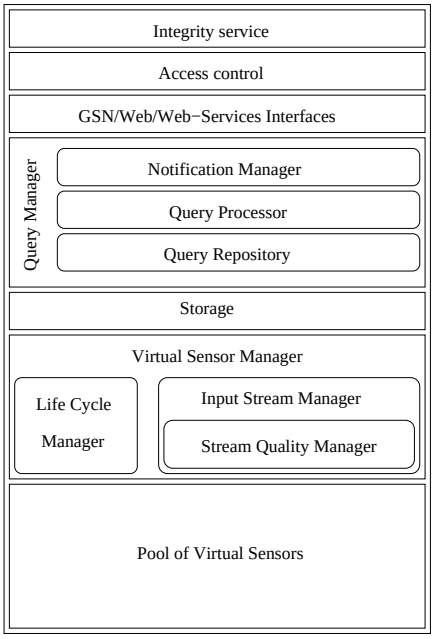



**Figure 3.8.** GSN architecture

## 3.4 System architecture

GSN uses a container-based architecture for hosting virtual sensors. Similar to application servers, GSN provides an environment in which sensor networks can easily and flexibly be specified and deployed by hiding most of the system complexity in the GSN instance. Using the declarative specifications, virtual sensors can be deployed and reconfigured in GSN instances at runtime. Communication and processing among different GSN instances is performed in a peer-to-peer style through standard Internet and Web Services protocols. By viewing GSN instances as cooperating peers in a decentralized system, we tried to avoid some of the intrinsic scalability problems of many other systems which rely on a centralized or hierarchical architecture. Targeting a "Sensor Internet" as the long-term goal we also need to take into account that such a system will consist of "Autonomous Sensor Systems" with a large degree of freedom and only limited possibilities of control, similarly as in the Internet.

Figure 3.8 shows the layered architecture of a GSN instance.

Each GSN instance hosts a number of virtual sensors it is responsible for. The virtual sensor manager (VSM) is responsible for providing access to the virtual sensors, managing the delivery of sensor data, and providing the necessary admin-

istrative infrastructure. The VSM has two subcomponents: The life-cycle manager (LCM) provides and manages the resources provided to a virtual sensor and manages the interactions with a virtual sensor (sensor readings, etc.). The input stream manager (ISM) is responsible for managing the streams, allocating resources to them, and enabling resource sharing among them while its stream quality manager subcomponent (SQM) handles sensor disconnections, missing values, unexpected delays, etc., thus ensuring the QoS of streams. All data from/to the VSM passes through the storage layer which is in charge of providing and managing persistent storage for data streams. Query processing in turn relies on all of the above layers and is done by the query manager (QM) which includes the query processor being in charge of SQL parsing, query planning, and execution of queries. The query repository manages all registered queries (subscriptions) and defines and maintains the set of currently active queries for the query processor. The notification manager deals with the delivery of events and query results to registered, local or remote virtual sensors. The notification manager has an extensible architecture which allows the user to largely customize its functionality, for example, having results mailed or being notified via SMS.

The top three layers of the architecture deal with access to the GSN server. The interface layer provides access functions for other GSN servers and via the Web (through a browser or via web services). These functionalities are protected and shielded by the access control layer providing access only to entitled parties and the data integrity layer which provides data integrity and confidentiality through electronic signatures and encryption. Data access and data integrity can be defined at different levels, for example, for the whole GSN server or at a virtual sensor level.

In connection with RFID tags this “plug-and-play” feature of GSN provides new and interesting types of mobility. For example, an RFID tag may store queries which are executed as soon as the tag is detected by a reader, thus transforming RFID tags from simple means for identification and description into a GSN instance for physically mobile queries which opens up new and interesting possibilities for mobile information systems.

## 3.5 Implementation

The GSN implementation consists of the GSN-CORE, implemented in Java, and the platform-specific GSN-WRAPPERS, implemented in Java, C, and Ruby, de-

pending on the available toolkits for accessing specific types of sensors or sensor networks. The implementation currently has approximately 80,000 lines of code and is available from SourceForge (http://gsn.sourceforge.net/). GSN is designed to be highly modular in order to be deployable on various hardware platforms from workstations to small programmable PDAs, i.e., depending on the specific platforms only a subset of modules may be used. GSN also includes visualization systems for plotting data and visualizing the network structure. In the following sections we are going to discuss some of the key aspects of the GSN implementation.

### 3.5.1 Adding new sensor platforms

For deploying a virtual sensor the user only has to define a virtual sensor in the form of a XML document as described in Section 3.2. GSN by default comes with a large set of hardware wrappers (drivers) which users can use in their virtual sensor files. Adding a new type of sensor or sensor network can be done by supplying the wrapper class (specified in `/conf/wrappers.properties`) conforming to the GSN API[10]. At the moment GSN provides the following wrappers:

**HTTP generic wrapper** is used to pull data from devices via HTTP GET or POST requests, for example, getting the pictures from the AXIS206W wireless camera.

**TinyOS wrapper** enables interaction with TinyOS compatible motes (versions 1.x and 2.x). This wrapper uses the serial forwarder which is the standard access tool for TinyOS based wireless sensor networks.

**USB camera wrapper** is used for dealing with cameras connected via USB to the local machine. As USB cameras are very cheap, they are quite popular as sensing devices. This wrapper supports cameras with OV518 and OV511 chipsets (see http://alpha.dyndns.org/ov511/).

**TI-RFID wrapper** enables access to the Texas Instruments Series 6000 and S6700 multi-protocol RFID readers.

**Generic UDP wrapper** can be used with any device with the UDP protocol support.

[10]The wrapper subsystem integrates with GSN through a callback API.

**Generic serial wrapper** can be used with any device with the serial port (COM ports) support.

New wrappers can be added to GSN without having to rebuild or modify the GSN server (plug-and-play). Upon startup GSN locates the wrapper mappings file through reading the `/conf/wrapper.properties` and loads each wrapper whenever needed by the system.

### 3.5.2 Dynamic Resource Management

The highly dynamic processing environment we target with GSN requires adaptive dynamic resource management to allow the system to quickly react to changes in the processing needs and environmental conditions. Dynamic resource management accomplishes three main tasks:

**Resource sharing:** As the user can modify/remove/add virtual sensors on-the-fly during runtime, the system needs to keep track of all the resources used by the individual virtual sensors and enforce resource sharing among sensors (wrappers) where possible.

**Failure management:** If GSN detects a faulty virtual sensor or wrapper, e.g., by runtime exceptions, GSN undeploys it and releases the associated resources.

**Explicit resource control:** The user can specify explicit memory and processing requirements and restrictions. While restrictions are always enforced, requirements are handled depending on the globally available resources of the GSN instance. GSN tries to share the available resources in a fair way taking into account the explicitly specified resource requirements, if provided.

Dynamic resource management is performed at several levels in GSN as shown in Figure 3.9. Separating the resource sharing into several layers logically decouples the requirements and allows us to achieve a higher level of reuse of resources. In the following we will discuss the different levels.

**Wrapper sharing.** Wrappers communicate directly with the sensors which involves expensive I/O operations via a serial connection or wireless/wired network communication. To minimize the costs incurred by these operations GSN shares wrappers among virtual sensors accessing the same physical/virtual sensors. To do so each GSN node maintains a repository of active wrappers. If a new virtual

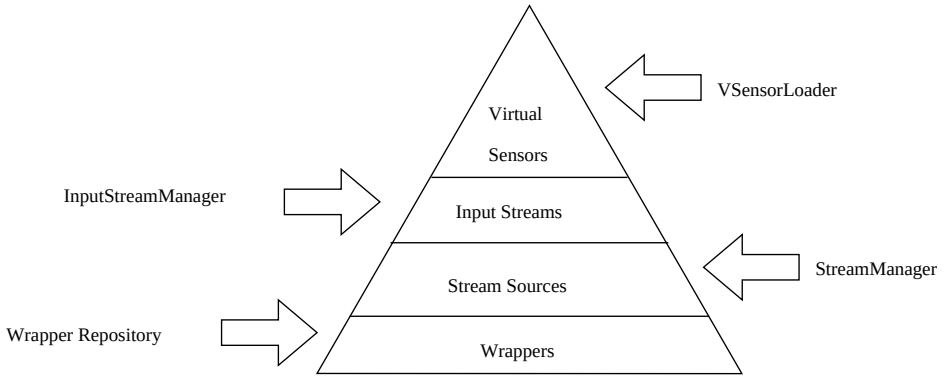



**Figure 3.9.** Hierarchical resource sharing in GSN

sensor is deployed, the node first checks with the wrapper repository whether an identical wrapper already exists, i.e., wrapper name and initialization parameters (and their corresponding values) of the `<wrapper>` element in the virtual sensor definitions are identical. If a match is found, the new virtual sensor is registered to the existing wrapper as a consumer. If not, a new wrapper instance is created and registered with the wrapper repository. In the case of remote sensor accesses this strategy is applied at both the sending and receiving sides to maximize the sharing, i.e., multiple virtual sensors on one GSN node share a wrapper for the same remote sensor and on the node hosting the sensor the wrapper is shared among all nodes accessing it.

**Data sharing.** The raw input data produced by the wrappers is processed and filtered by the source queries to generate the actual input data for the input streams of a virtual sensor. For this purpose a source defines what part of the raw input data is used by the associated source query to produce the source's output data, i.e., by defining the available storage, sampling rates, and window sizes a view on the raw data is defined on which the source query is executed. In terms of the implementation each wrapper is assigned a storage holding the raw data and source queries are then defined as *SQL views* on this data store.

This has a number of advantages: (1) It minimizes the storage consumption as raw data is only stored once. Especially if the sensor data is large, e.g., image data, this is relevant. (2) If the sensor data comes from a power-constrained or slow device, power is conserved and processing is sped up. (3) Different processing strategies can be applied to the same data without having to replicate it, for example, image enhancement algorithms and object detection can use the same raw image data.

In the same way as a wrapper can be shared by multiple sources, a source can also be shared among multiple streams at a higher level, and streams in turn are shared by multiple virtual sensors. In essence each of the layers in Figure 3.9 can be viewed as a resource pool where each of the individual resources in the pool can be shared among multiple resources at the next higher level. Conversely, each higher level resource can also use any number of lower level resources.

### 3.5.3 Query planning and execution

In GSN each virtual sensor corresponds to a database table and each sensor reading corresponds to a new tuple in the related table. As GSN uses a standard SQL database as its low-level query processing engine, the question is how to represent the streaming logic in a form which is understandable for a standard database engine (as already described, GSN separates the stream processing directives from the query). We address this problem by using a query translator which gets a SQL query and the stream processing directives as provided in the virtual sensor definition as inputs and translates it into a query that is executable in a standard database. The query translator relies on special support functions which emulate stream-oriented constructs in a database. These support functions are database dependent. Once the queries are translated, they are cached for subsequent use.

Upon deployment of a virtual sensor $VS$, all queries $Q_i$ contained in its specification are extracted. Each query $Q_i(VS_1,\ldots,VS_n)$ accesses one or more relations $VS_1,\ldots,VS_n$ which correspond to virtual sensors. Then the query translator translates each $Q_i(VS_1,\ldots,VS_n)$ into an executable query $Q_i^t(VS_1,\ldots,VS_n)$. Each $Q_i^t(VS_1,\ldots,VS_n)$ is declared as a view in the database with a unique identifier $Id_i$. This means that whenever a new tuple, i.e., sensor reading, is added to the database, the concerned views will automatically be updated by the database. Additionally, a tuple $(VS_j, Id_i, VS)$ for each $VS_j \in VS_1,\ldots,VS_n$ is added to a special view registration table. This procedure is done once when a virtual sensor is deployed.

With this setup it is now simple to execute queries over the data streams produced by virtual sensors: As soon as new sensor reading for a virtual sensor $VS_d$ becomes available, it is entered into the database. Then the database server queries the registration table using $VS_d$ as the key and gets all identifiers $Id_r$ registered for new data of $VS_d$. Then simply all views $V_r$ affected by the new data item can be retrieved using the $Id_r$ and all $V_r$ can be queried using a `SELECT * FROM V_r` statement and the resulting data can be returned to the virtual sensor containing $V_r$ (third column in the registration table). Since views are automatically updated

by the database, querying them is efficient. However, with many registered views (thousands or more) scalability may suffer. Thus GSN does not produce an individual query for each view but merges all queries into a large select statement, and the result will then be joined with the view registration table on the view identifier. Thus the result will hold tuples that identify the virtual sensor to notify of the new data. The reasons for applying this strategy are that (1) database connections are expensive, (2) with increasing number of clients and virtual sensor definitions, the probability of overlaps in the result sets increases which automatically will be exploited by the database's query processor, and (3) query execution in the database is expensive, so one large query is much less costly than many (possibly thousands) small ones.

Immediate notification of new sensor data, which is an eager strategy, is currently implemented in GSN. As an alternative also a lazy strategy could be used where the query execution would only take place when the GSN instance requests it from the database, for example, periodically at regular intervals. In practice the former can be implemented using views or triggers and the latter can be implemented using inner selects or stored procedures.

## 3.6 GSN-to-GSN communication Protocol

In this section we present the low level details of GSN-to-GSN communication protocol. In order to enable data sharing and distributed collaborative data stream processing, we have introduced two special types of wrappers in GSN. First, the `local` wrapper, which enables data stream sharing among virtual sensors on the same GSN instance. Second, the `remote` wrapper, which enables data stream sharing among multiple distributed virtual sensors each of which located on different computers accessible through the network.

In GSN, whenever a virtual sensor wants to use another virtual sensor located on a different GSN instance, the communication between two GSN instances is triggered during the loading process of the virtual sensor. Once GSN notices that a remote virtual sensor is required by a local virtual sensor, it temporary suspends the local virtual sensor's loading process to confirm the existence of the remote virtual sensor. Therefore, GSN-to-GSN communication is initiated whenever a virtual sensor in the node $A$ wants to use the data stream provided by another virtual sensor in the node $B$ ($A \neq B$).

Using this approach, GSN mediates all the outgoing and incoming connections

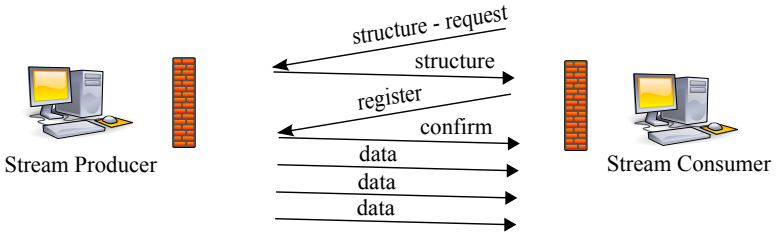



**Figure 3.10.** Experimental setup

therefore the local virtual sensor does not interact directly with the remote virtual sensor (and vice versa). The packets exchanged between two GSN instances during GSN-to-GSN communication are depicted in the Figure 3.10 (all communications are implemented using XML-RPC calls). Below, we provide a brief description of each packet:

**structure-request/structure** is used by the local GSN instance to discover the output structure of the remote virtual sensor. The response to this packet, confirms the existence and availability of the remote virtual sensor and contains the details of the output-structure of the remote virtual sensor.

**register/confirm** is used by the local GSN instance to send the query and the contact address of the stream consumer. The query will be added to the notification list associated with the prospective virtual sensor at the stream producer side, therefore whenever the remote virtual sensor produces a stream element, the query will be evaluated and the output of the evaluation, in case the output is not empty, is delivered to the stream consumer. The remote virtual sensor uses the addressing information (received in the registration packet) to contact the stream consumer in order to deliver the stream elements. As there might be multiple virtual sensors at the stream consumer side be interested in one virtual sensor hosted at the stream producer, any registration request has a UUID associated with it which is used by the stream producer whenever it wants to deliver stream elements to the stream consumer[11].

**data** presents the stream of tuples which are going to be delivered to the stream consumer. At the stream consumer side, the GSN instance receives the data and based on the UUID of the tuples, GSN instance disseminates the tuples to the appropriate local virtual sensor.

[11] One can think of the UUID in the GSN as the port number in the socket communication.

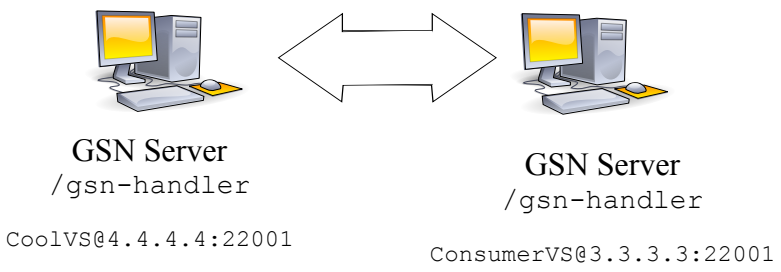



**Figure 3.11.** Simple GSN-to-GSN communication

In order to make the explanation of the GSN-to-GSN communication more concrete, we provide now more system level details. For using a remote virtual sensor, the first step is locating the *contact point* of the GSN instance which hosts the stream producer virtual sensor. By default, the contact point is `http://ip:port/gsn-handler`[12][13]. If the contact point is correctly identified, the response to a plain HTTP POST request returns an XML output.[14][15]

Correct identification of the contact point is crucial in success of using the remote virtual sensor. Once the contact points identified successfully, one can define a stream which consumes data from the other data source. Note that consuming data from a remote virtual sensor doesn't require any kind of modification at the remote host and in fact due to GSN's decoupled architecture, the remote virtual sensor is not even aware that its data is being consumed by others. In Figure 3.11, the virtual sensor `ConsumerVS` running at the GSN instance with the IP address of `3.3.3.3` under the port `22001` is interested in getting streaming data from the `CoolVS` virtual sensor running at the GSN instance with the IP address of `4.4.4.4` under the port `22001`. To enable this communication one has to use a source configuration similar to the one presented in the Figure 3.12.

In some deployments, GSN instances are hosted behind a NAT or backed by an Apache web server. This can be true for both the GSN data stream consumer and the GSN data stream producer. In these cases, one can use the more advanced form of the remote wrapper. Figure 3.13 presents a sample setup in which both of the GSN data stream consumer and data producer are behind the firewall. The firewall

[12]The port is specified in the `conf/gsn.xml` file.

[13]The exact mapping is specified in `webapp/WEB-INF/web.xml` file.

[14]The actual output represents an error as the request is not properly formatted.

[15]For sending plain HTTP POST requests to `http://ip-address:gsn-port/gsn-handler`, you may want to use http://code.google.com/p/rest-client/.

```
<address wrapper="remote">
  <predicate key="name">CoolVS</predicate>
  <predicate key="host">4.4.4.4</predicate>
  <predicate key="port">22001</predicate>
</address>
```

**Figure 3.12.** Source configuration for simple GSN-to-GSN communication.

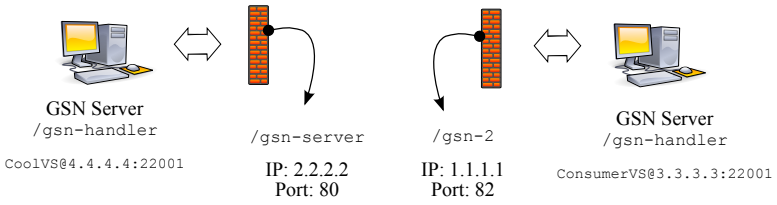



**Figure 3.13.** Simple GSN-to-GSN communication

at the consumer side has mapped `3.3.3.3:22001` into `1.1.1.1:82` and the firewall at the stream producer side has mapped `4.4.4.4:22001` into `2.2.2.2:80`. To enable this kind of communication one has to use a source configuration similar to the one presented in the Figure 3.14.

### 3.6.1 `local` wrapper

The `local` wrapper is a special version of the `remote` wrapper (host = "`127.0.0.1`") which is optimized for communication among two different virtual sensors inside the same GSN instance. By having the `local` wrapper optimized, we imply that most of the overhead associated with TCP/IP networking calls are eliminated by using internal GSN calls instead. The `local` wrapper is recommended whenever the end-to-end delay between two virtual sensors is important. In GSN, we have

```
<address wrapper="remote">
  <predicate key="name">CoolVS</predicate>
  <predicate key="local-contact-point">
          http://1.1.1.1:82/gsn-2</predicate>
  <predicate key="remote-contact-point">
          http://2.2.2.2:80/gsn-server</predicate>
</address>
```

**Figure 3.14.** Source configuration for NATed GSN-to-GSN communication.

implemented the notification system so that the GSN instance always gives priority to the local virtual sensors when it wants to disseminate the data stream elements thus the local virtual sensors usually get to be notified earlier.

## 3.7 Evaluation

GSN aims at providing a zero-programming and efficient infrastructure for large-scale interconnected sensor networks. To justify this claim we experimentally evaluate the throughput of the local sensor data processing and the performance and scalability of query processing as the key influencing factors. As virtual sensors are addressed explicitly and GSN nodes communicate directly in a point-to-point (peer-to-peer) style, we can reasonably extrapolate the experimental results presented in this section to larger network sizes. For our experiments, we used the setup shown in Figure 3.15.

The GSN network consisted of 5 standard Dell desktop PCs with Pentium 4, 3.2GHz Intel processors with 1MB cache, 1GB memory, 100Mbit Ethernet, running Debian 3.1 Linux with an unmodified kernel 2.4.27. For the storage layer use standard MySQL 5.18. The PCs were attached to the following sensor networks as shown in Figure 3.15.

- A sensor network consisting of 10 Mica2 motes, each mote being equipped with light and temperature sensors. The packet size was configured to 15 Bytes (data portion excluding the headers).

- A sensor network consisting of 8 Mica2 motes, each equipped with light, temperature, acceleration, and sound sensors. The packet size was configured to 100 Bytes (data portion excluding the headers). The maximum possible packet size for TinyOS 1.x packets of the current TinyOS implementation is 128 bytes (including headers).

- A sensor network consisting of 4 Tiny-Nodes (TinyOS compatible motes produced by Shockfish, http://www.shockfish.com/), each equipped with light and two temperature sensors with TinyOS standard packet size of 29 Bytes.

- 15 Wireless network cameras (AXIS 206W) which can capture 640x480 JPEG pictures with a rate of 30 frames per second. 5 cameras use the highest available compression (16KB average image size), 5 use medium compression (32KB average image size), and 5 use no compression (75kB average

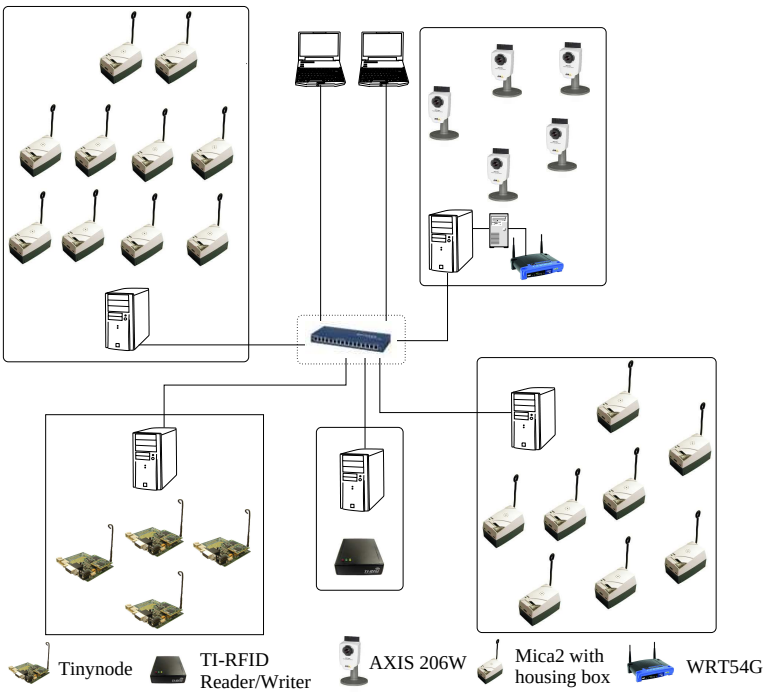



**Figure 3.15.** Experimental setup

image size). The cameras are connected to a Linksys WRT54G wireless access point via 802.11b and the access point is connected via 100Mbit Ethernet to a GSN node.

- A Texas Instruments Series 6000 S6700 multi-protocol RFID reader with three different kind of RFID-tags. Each RFID tag can store up to 8KB of binary data.

The motes in each sensor network form a sensor network and routing among the motes is done with the SURGE multi-hop ad-hoc routing algorithm[65] provided by TinyOS.

### 3.7.1 Internal processing time

In the first experiment we wanted to determine the internal processing time a GSN node requires for processing sensor readings, i.e., the time interval when the wrapper gets the sensor data until the data can be provided to clients by the associated virtual sensor. This delay depends on the size of the sensor data and the rate at which the data is produced, but is independent of the number of clients wanting to

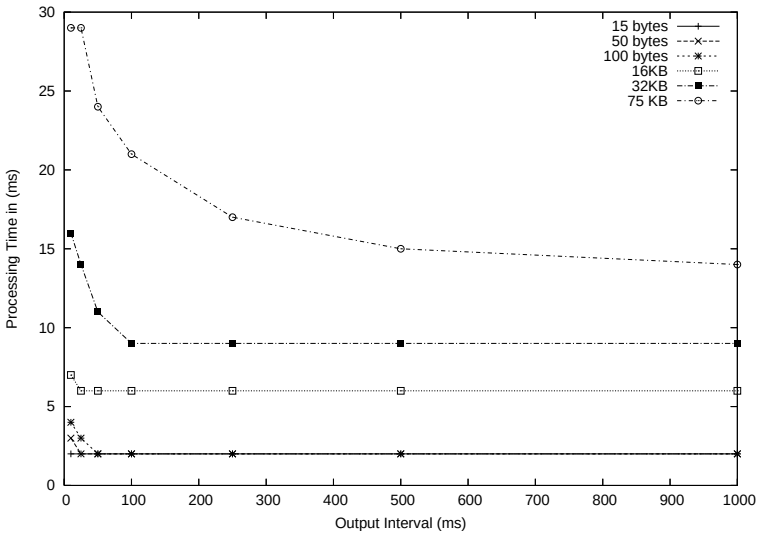



**Figure 3.16.** GSN node under time-triggered load

receive the sensor data. Thus it is a lower bound and characterizes the efficiency of the implementation.

We configured the output intervals of 22 motes and 15 cameras to produce data every 10, 25, 50, 100, 250, 500, and 1000 milliseconds. As the cameras have a maximum rate of 30 frames/second, i.e., a frame every 33 milliseconds, we added a proxy between the GSN node and the WRT54G access point which repeated the last available frame in order to reach a frame interval of 10 milliseconds. All GSN instances used the Sun Java Virtual Machine (1.5.0 update 6) with memory restricted to 64MB.

The experiment was conducted as follows: All motes and cameras were set to the same rate and produced data for 8 hours and we measured the processing delay. This was repeated 3 times for each rate and the measurements were averaged. Figure 3.16 shows the results of the experiment for the different data sizes produced by the motes and the cameras. This experiment presents the overhead of GSN in combination with storage of the streaming data (core GSN processing) without having any user defined processing logic present.

High data rates put some stress on the system but the absolute delays are still quite tolerable. The delays drop sharply if the interval is increased and then converge to a nearly constant time at a rate of approximately 4 readings/second or less. This result shows that GSN can tolerate high rates and incurs low overhead for realistic rates as in practical sensor deployments lower rates are more probable due

to energy constraints of the sensor devices while still being able to deal also with high rates.

### 3.7.2 Scalability in the number of queries and clients

In this experiment the goal was to measure GSN's scalability in the number of clients and queries. To do so, we used two 1.8 GHz Centrino laptops with 1GB memory as shown in Figure 3.15 which each ran 250 lightweight GSN instances. The lightweight GSN instance only included those components that we needed for the experiment. Each GSN-light instance used a random query generator to generate queries with varying table names, varying filtering condition complexity, and varying configuration parameters such as history size, sampling rate, etc. For the experiments we configured the query generator to produce random queries with 3 filtering predicates in the `where` clause on average, using random history sizes from 1 second up to 30 minutes and uniformly distributed random sampling rates (seconds) in the interval $[0.01, 1]$.

Then we configured the motes such that they produce a measurement each second but would deliver it with a probability $P < 1$, i.e., a reading would be dropped with probability $1 - P > 0$. Additionally, each mote could produce a burst of $R$ readings at the highest possible speed depending on the hardware with probability $B > 0$, where $R$ is a uniformly random integer from the interval $[1, 100]$. I.e., a burst would occur with a probability of $P * B$ and would produce randomly 1 up to 100 data items. In the experiments we used $P = 0.85$ and $B = 0.3$ to demonstrate a a sample setup. On the desktops we used MySQL as the database with the recommended configuration for large memory systems. Figure 3.17 shows the results for a stream element size (SES) of 30 Bytes. Using SES=32KB gives the same latencies. Due to space limitations we do not include this Figure.

The spikes in the graphs are bursts as described above. Basically this experiment measures the performance of the database server under various loads which heavily depends on the used database. As expected the database server's performance is directly related to the number of the clients as with the increasing number of clients more queries are sent to the database and also the cost of the query compiling increases. Nevertheless, the query processing time is reasonably low as the graphs show that the average time to process a query if 500 clients issue queries is less than 50ms, i.e., approximately 0.5ms per client. If required, a cluster could be used to improve the query processing times which is supported by most of the

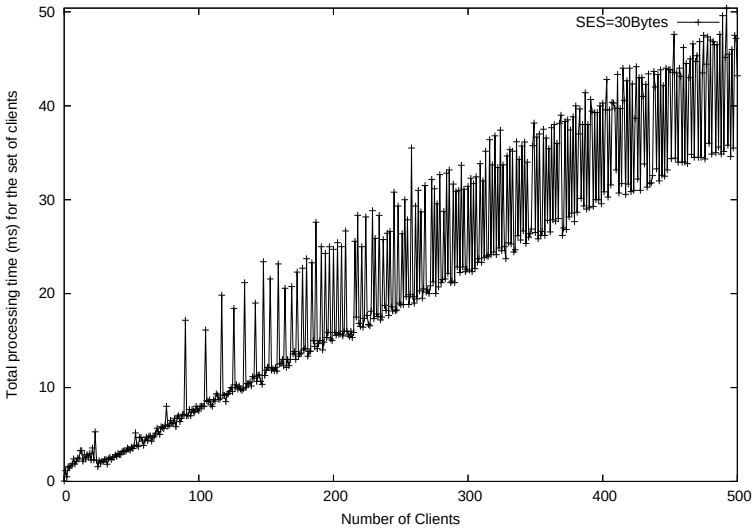



**Figure 3.17.** Query processing latencies in a node

existing databases already.

In the next experiment shown in Figure 3.18 we look at the average processing time for a client excluding the query processing part. In this experiment we used $P = 0.85$, $B = 0.05$, and $R$ is as above.

We can make three interesting observations from Figure 3.18:

1. GSN only allocates resources for virtual sensors that are being used. The left side of the graph shows the situation when the first clients arrive and use virtual sensors. The system has to instantiate the virtual sensor and activates the necessary resources for query processing, notification, connection caching, etc. Thus for the first clients to arrive average processing times are a bit higher. CPU usage is around 34% in this interval. After a short time (around 30 clients) the initialization phase is over and the average processing time decreases as the newly arriving clients can already use the services in place. CPU usage then drops to around 12%.

2. Again the spikes in the graph relate to bursts. Although the processing time increases considerably during the bursts, the system immediately restores its normal behavior with low processing times when the bursts are over, i.e., it is very responsive and quickly adopts to varying loads.

3. As the number of clients increases, the average processing time for each client decreases. This is due to the implemented data sharing functionalities

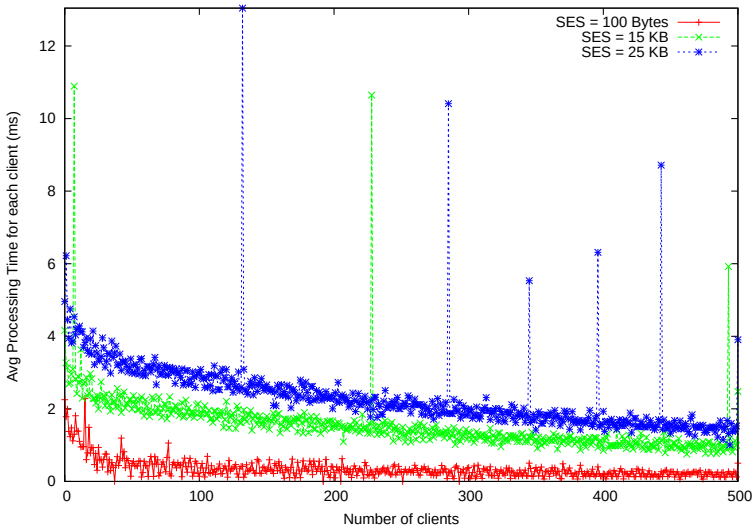



**Figure 3.18.** Processing time per client

as presented in Section 3.5.2. As the number of clients increases, also the probability of using common resources and data items grows.

## 3.8 Summary

The full potential of sensor technology will be unleashed through large-scale (up to global scale) data-oriented integration of sensor networks. To realize this vision of a "Sensor Internet" we propose our Global Sensor Network (GSN) middleware which enables fast and flexible deployment and interconnection of sensor networks. Through its virtual sensor abstraction which can abstract from arbitrary stream data sources and its powerful declarative specification and query tools, GSN provides simple and uniform access to the host of heterogeneous technologies. GSN offers zero-programming deployment and data-oriented integration of sensor networks and supports dynamic configuration and adaptation at runtime. Zero-programming deployment in conjunction with GSN's plug-and-play detection and deployment feature provides a basic functionality to enable sensor mobility. GSN is implemented in Java and is available (with detailed users and developers guides) at http://gsn.sourcefourge.net/. The experimental evaluation of GSN demonstrates that the implementation is highly efficient, offers good performance even under high loads and scales gracefully in the number of nodes, queries.

# Chapter 4

# Efficient Sliding Window Management

## Overview

Sliding windows are the essential building blocks to limit the query focus at a particular part of the stream, based either on value count or time ranges. These so called sliding window predicates specify the execution condition for the query. Due to the often massive amount of registered queries, efficient algorithms to check these predicates are essential. While there exist a comprehensive set of works on the stream processing techniques, the actual algorithms to intelligently decide on the sliding behaviors is not extensively addressed in the existing works. In this chapter we propose a set of algorithms for managing and sharing sliding schedules. This chapter introduces the concept of the batch sliding and sliding graphs to improve the sliding decision of the stream processing engines. We introduce the algorithms that can be used efficiently in large-scale stream processing systems where data arrives at high rates and a large number of user queries are registered to these data streams. We conclude the chapter by presenting the evaluation results of this approach in the real world applications.

## 4.1 Introduction

Today, researchers and enterprises are playing an active role in the streaming world by publishing real time data ranging from financial information (e.g., stock ticks) to entertainment information such as real time scores for a soccer match. These streaming data can be produced either by real sensors such as RFID readers (e.g., tracking parcels on a web site) or virtual sensors with no direct connection to physical world (e.g., network traffic).

While there exist a comprehensive set of related work both on stream processing techniques and middlewares, the actual algorithms to intelligently decide on the sliding behaviors while the streaming data are arriving to the system has not been sufficiently addressed in the existing works. However, the problem becomes severe in scenarios with thousands of users registered to hundreds of high frequency data stream. One of our motivation applications, which suffers from the same issue, is called *NexTick*[81] which is a real time stock tick processing application architected to identify variety of trends by performing multiple technical analysis (TA) over the market to identify the best entry and exit points (lots of queries). NexTick is designed to be used on standard PCs and laptops and sends its recommendation messages in real time as the market moves which means it has to very efficiently handle the processing and notification events. Our second motivating use case is initiated by the environmental scientists whom we are collaborating in the context of the *Swiss Experiment*[6] project (described in detail in section 4.2).

In the above applications scenarios (among many other similar high demanding use cases) having algorithms to efficiently use the resources in order to decide *when and which* queries have to be executed can save both processing time and memory consumption, as verified later in the Section 4.6.

In this chapter, we provide a set of algorithms which can be used to efficiently decide on the processing time of the queries in the stream processing engines. We introduce a new query organization technique based on the sliding attributes of the queries and we provide algorithms for performing *batch sliding*. This work can be specifically useful for popular and high rate streams such as stock ticks, sensor values for a renowned location (e.g., snow height in a popular skiing resort in winter) in addition to resource constrained environments such as mobile phones and PDAs.

The rest of the chapter is organized as follows. Section 4.2 presents our motivation scenario. Section 4.3 presents the related work. Section 4.4 presents the stream processing model which we consider in this chapter. Section 4.3 presents

a discussion on scalability issues. Section 4.5 presents algorithms. Section 4.6 presents the evaluation results of the algorithms and finally we conclude in Section 4.7.

## 4.2 Motivating Scenarios

As mentioned in the introduction section, this chapter focus on two of the high data rate applications, the *NexTick* and the *Swiss Experiment*. While these two applications appear at the first glance to be fundamentally different (one is using market information while the other one is monitoring the physical world), in our design, architecturally, they are exhibiting the same behavior. Both applications deal with the integration and processing of high rate data, and achieving that in a highly efficient way to support effective decision making process (e.g., buy or sell actions or sending alerts in the case of environmental monitoring sensors).

Thanks to strong acceptance of the Wireless Sensor Networks technology, more and more applications and user groups outside the core technology of wireless sensor networks started to benefit from it. One of the major user bases are in environmental science. Wireless sensors bring environmental scientists the opportunity of fine grained monitoring of physical phenomena and that explains to some extent why environmental scientists are among the early adopters of this new technology.

In this section, we focus on the Swiss Experiment project as our motivation use case (very similar arguments can be applied to NexTick). In the context of Swiss Experiment, computer science researchers work closely in an inter-disciplinary collaboration with environmental scientists across multiple research centers in Switzerland. Interestingly, most researchers from the environmental science side are from different sub domains including snow and avalanche research, water quality, earthquake, understanding rapid mass movements, climate change, weathering, soil formation and ecosystem evolution. As one can imagine, introducing systems to address needs of over 10 different subgroups (although all are related to environmental science) can be both very interesting and challenging in nature.

Once the sensor data is captured (e.g., by means of wireless or satellite links) and transferred to the storage system deployed across the relevant research institute, it gets streamed to GSN. GSN offers several services such as data sharing between multiple GSN instances, data storage, processing and filtering. Scientists can use GSN to express the processing logic and have GSN taking care of storage of the events and distributing the notification messages among other services

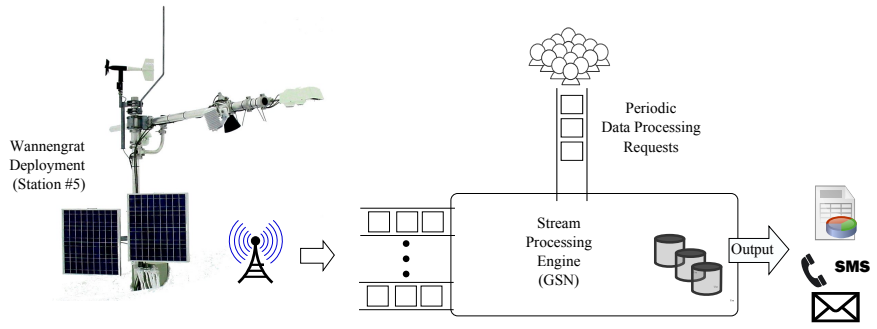



**Figure 4.1.** Data Acquisition, Processing and Management Chain in the Swiss Experiment Project.

(depicted in figure 4.1[1]).

The queries posted by the scientists in the Swiss Experiment, are including but not limited to data aggregation, statistical analysis (median, average, max and min), data quality estimation (through median average deviation) and analysis of extreme values (extreme statistics). Each of the aforementioned calculations has to be performed at different time granularities (e.g., 15 seconds, 1 minute, 15 minutes, etc.) on the data streams generated by each sensor on every station. For instance, a wind sensor is deployed on each weather station with 8 other sensors (e.g., sun radiation, snow height, etc.), typically generates streaming data at 50 hertz. Considering an environmental research center such as Swiss Federal Institute for Snow and Avalanche Research in Davos with over 150 scientists, all of whom are interested in different aspects of above calculations and considering the sheer amount of sensor data streamed into this institute (around 500 weather stations deployed around Switzerland) one can easily see that the need for scalable management of the resources on the underlying data stream processing system is not only beneficial but also essential to achieve reasonable response time. Note that the output of GSN is typically delivered through reports and emails. For applications which need immediate attention of the scientists, GSN can deliver its output through sending SMS or making VoIP phone calls (through using Text-To-Speech engines). Applications which may need immediate attention are including but not limited to sending early warning (forecasting) and detecting the broken sensors (by comparing the patterns of the sensor data measured by a station to its previous measurements and other stations covering an overlapping region).

[1]The Wannengrat deployment consists of a set of solar powered weather stations deployed on the Wannengrat mountain in Davos, Switzerland. The stations are communicating through a long range point to point wireless network and through GPRS satellite link (the backup link).

Consider a financial market data processing application. In these systems (e.g., NexTick[81]) quick response time has the utmost importance. The actual processing of the market ticks is done in two stages. The first stage involves identifying which queries to execute at a given time. The second stage involves the actual execution process. In a typical setup of 10,000 symbols (the NYSE and Nasdaq together) with 100 different types of technical analysis (TA) performed on the price movements plus having these TAs performed at multiple (typically 20) time granularities (e.g., 1 second, 5 seconds, 15 seconds, 1 minute, etc.), one can easily see how scalability can become a real issue. Just in the aforementioned application, one has to deal with over 20 Million queries. Of course each query has to be also processed which itself implies huge lag between the decision time and the stock tick arrival time.

In these setups the queries are posted to the system before data streams arrive to the system. For instance, the number of queries and TAs are typically fixed before the market opens. Thus the scheduling of window queries can already be prepared in advance. Whenever a tick is delivered to the system from the market, a typical naive approach has to evaluate all the registered queries to identify which ones to slide (e.g., 20 million queries in this case). Our approach optimizes this step by providing data structure and algorithms to intelligently handle the sliding values. The execution process of queries is orthogonal to this step simply because in typical scenarios like above, we can easily use data grids and grid computing techniques to parallelize processing and thus reducing latency. Note that in these applications, our algorithms for the query candidate construction phase combined with other optimization techniques at the execution time (likes those introduced by [35] and [66]) can provide a comprehensive toolkit to handle the performance issues in these kind of applications.

## 4.3 Related Work

As of today, there exist a few dozen of stream processing engines developed by different research groups. Aurora [10], STREAM [20] and TelegraphCQ [32] are some of the existing stream processing systems which support sliding windows on data streams. The different types of windows have been classified in [55] based on works in [51]. Processing data using the sliding concept is also recently added to commercial databases thanks to the new notations in SQL99 for specifying logical and physical windows on relations. Data stream query languages, such as CQL

[21], StreaQuel [32], GSQL [42], and AQuery [64], often define their own notion of windows based on SQL99.

The work in this chapter is also highly relevant to the data aggregation sharing methods introduced in [35] and [66].

In [35] the authors exploit similarities among the queries in order to share resources. The focus of our work is on sharing streaming aggregate queries with different periodic windows without any up-front multi query optimization which are relying on complex static analysis. The authors introduce *Shared Data Shards* for sharing the processing cost among queries with different window sizes and predicates. The Shared Data Shards itself is actually the combination of two other optimizations, the *Shared Time Slices* approach which is designed for sharing processing cost among continuous queries with differing window size and the *Shared Data Fragments* approach for sharing processing resources among queries with different predicates.

[66] proposes a technique called *panes* which reduces both the space and computation cost of evaluating sliding-window queries by sub-aggregating and sharing computation. The paper divides overlapping windows into disjoint panes, computes sub-aggregates over each pane, and compute a window-aggregates by *rolling up* the sub-aggregates to compute the window-aggregates. The concept of sub-aggregation and super-aggregation is used originally by the ROLLUP operator in SQL 99 specification and the data cube operator to express aggregates at different granularities.

In order to handle timestamps in a distributed environment, [85] proposes a flexible heartbeat technique for application-defined time in a DSMS (Distributed Stream Management System). The proposed solution can handle time skew between streams, out-of-order data items within streams, and latency in streams reaching the data stream management systems.

In stream processing systems, continuous queries are executed whenever the sliding occurs. While all the aforementioned works provide different insights in defining and using window and sliding parameters on the streaming data, the actual window processing and sliding is always considered to be handled in a per stream bases thus there is no optimization performed on the way stream processing engines deal with these parameters. To clarify the difference, for instance, in the window-data aggregation works mentioned above, the authors focus on sharing data in the aggregate queries at the execution time of the queries. Our work is focused on introducing data structures and algorithms to optimize the sliding

decisions (sliding decisions simply act as the triggers for the actual execution of a continuous queries). To the best of our knowledge, this work is the first attempt to optimize the way stream processing engines are handling the sliding parameters in large scale environments.

In this chapter, we exploit the similarities between the sliding specifications of the queries to plan a smart execution schedule. Once the query is scheduled to be executed, the optimizations in the above papers can be applied to share data and processing costs among the queries. We address both the time and count based sliding behaviors which are independent of the actual streaming data and execution plan. The ideas presented in this chapter are designed so that they can be applied easily to any stream processing engine including those mentioned above. We assume that data elements arrive in the correct order to the sliding manager component. For handling out of order data elements, we can benefit from the techniques introduced by [85].

## 4.4 System Model

In this section we briefly review the system model that we consider in our work. As we aim at introducing a new layer of optimization to stream processing systems, our goal is to be as general as possible.

Figure 4.2 represents a sample stream processing engine which uses two wireless sensor networks as its input streams. The first network is based on the TinyOS platform while the second network uses hand-held RFID readers communicating through WiFi with a base station (typically a wireless router). In contemporary stream processing engines, there exists a data interface component which is typically connected to a permanent storage such as a relational database, and a load shedder to monitor the current input rate in order to adapt the existing resources (which are consumed by the stream producers and users queries) with the preferred quality of service level.

In standard stream processing systems, users can post their queries to the system in order to get notified about the occurrence of specific conditions and patterns on the data streams. These queries are stored inside the query repository that closely interacts with the query planner which in turn uses the stream statistics directory to generate an optimal execution plan for each query. The stream statistics directory contains the most recent statistical information (e.g., income rate and stream data size) regarding all the streams registered to the system.

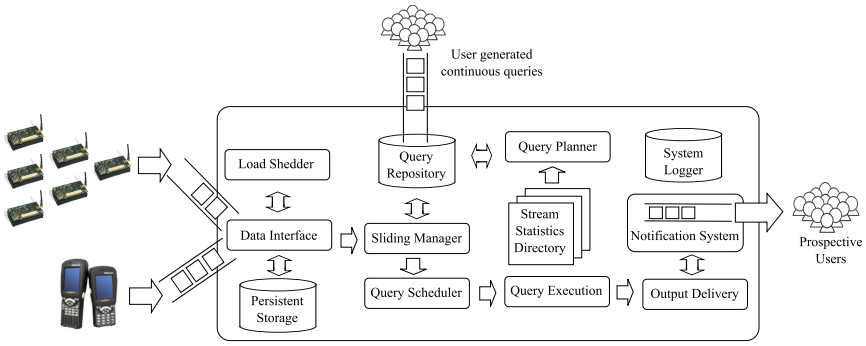



**Figure 4.2.** General Model For Large Scale Stream Processing

Once the streaming data arrives, the sliding manager uses the query repository to generate a list containing all queries for which the sliding is imminent. This list is called the candidate query list. The query list is then delivered to the query scheduler which schedules the queries for execution by using arbitrary scheduling algorithms. Since we post a list of queries for the scheduler, it can consider batch processing of the queries. The queries are evaluated in the query execution unit that emits the results to the output delivery module. The notification system and output delivery module communicate with each other to notify registered users about new events. This model implies an essential role of the sliding manager in a stream processing system. Our focus in this work is on the sliding manager module of the stream processing engines. We propose algorithms to intelligently manage the sliding windows, thus improving the processing time and reducing the memory overhead.

## 4.5 Algorithms

In the context of data stream processing, there exist two types of sliding actions, time based and (tuple) count based. The time based sliding implies execution of the query in predefined (and possibly fixed) intervals. The count based sliding is used for triggering the query execution once a certain amount of data items (tuples) has arrived at the stream processing engine. The amount of the data on which the query is evaluated is specified through the window property. The window property can be also specified using time or tuples. Given the above stream processing constructs one can come up with four different combinations listed below:

1. Count based window, count based slide (CBW-CBS)

2. Time based window, count based slide (TBW-CBS)

3. Time based window, time based slide (TBW-TBS)

4. Count based window, time based slide (CBW-TBS)

For sliding windows which have a count based slide (the first and second types), the case of *slide=1* is considered as a special case. In this case we simply do the sliding on arrival of each new tuple. There are also two different types of time which should be handled differently, *local time* and *remote time*. If the system time is used as the timestamp of tuples, we say these tuples are using local time. If the timestamp is set by the remote data source, we say tuples have remote time. Since each of these two types of time requires different treatment, sliding windows with time based slide are further divided into local time based and remote time based and therefore, we will have six different sliding window types: Two so called *count based sliding windows*, two so called *local time based sliding windows*, and in addition two *remote time base sliding windows*. In the following subsections we will explain different algorithms used for management of sliding for these three groups of sliding windows.

### 4.5.1 Sliding Graph

In this part, we start by describing the proposed sliding window management strategies and then we continue toward the concrete algorithms. The problem is to develop a method to reduce the processing time required for checking each sliding value to see whether its window must be slid or not. The straightforward method is to test all streams on arrival of each tuple. An obvious improvement is grouping those streams which have the same sliding value. This can already greatly reduce the number of comparisons on arrival of each new tuple.

We further improve the processing time by introducing a graph structure for sliding groups. This graph is based on the fact that in continuous queries that are issued by users, the sliding value of windows are often factors of each other. Suppose that after grouping of sliding windows, we have the following (count based) sliding groups: 2, 4, 8, 24, 15, 12, 5, 22, 3, 11, and 9. We know, for example, that 2 is a factor of 2, 4, 8, 12, 24, and 22; also 3 is a factor of 3, 9, 12, 15, and 24. These values are organized in a directed graph such that for each edge the start node's value divides the end node's value and there is no other node in between them. If after the construction of the graph there is more than one node without any edges, a dummy parent node is created which uses the greatest common divisor (*gcd*) of

its children as its sliding value. The only node without any incoming edges, which may be a dummy node, is called *root node*. Edges in the graph are either *strong* or *weak*; an incoming edge to a node is strong if its start value is the greatest among the start values of the other incoming edges. All other edges are weak edges and are used to simplify modifications to the graph. The resulting graph is called the *sliding graph*.

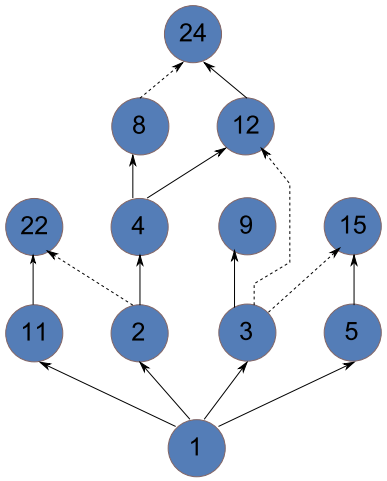



**Figure 4.3.** Sliding graph produced for sample sliding groups

Figure 4.3 shows the sliding graph for the sample sliding groups of the example given above. Strong edges are represented by solid line arrows and weak edges are represented by dashed line arrows. As it can be seen in the figure, if we remove weak edges from the graph, what remains is a tree which is called *sliding tree* and which we use in our sliding algorithms.

### 4.5.2 Sliding for Count based Sliding Windows

Count based sliding is the simplest among the three sliding window types. After constructing the sliding graph, we only need to keep track of the number of tuples received so far. On arrival of each new tuple Algorithm 1 is executed to create the candidate query list.

In Algorithm 1 we actually perform a pre-order traversal of the sliding tree. The search is stopped at each node of which the sliding value is not a factor of the current tuple count. In this way we can eliminate a (possibly) large number of unsuccessful sliding value tests and hence reduce the execution time. For example,

1: $tc \Leftarrow tc + 1$ {updating current number of received tuples}
2: **if** $tc$ mod $rootNode.slide = 0$ **then**
3: Push $rootNode$ to stack $S$
4: **while** $S$ is not empty **do**
5: Pop $n_s$ from $S$
6: Add requests of $n_s$ to the candidate query list
7: **for** each child node $n_{ch}$ of $n_s$ **do**
8: **if** $tc$ mod $n_{ch}.slide = 0$ **then**
9: Push $n_{ch}$ to $S$
10: **end if**
11: **end for**
12: **end while**
13: **end if**

**Algorithm 1**: Creating Candidate Query List

in the sample graph in Figure 4.3, when the tuple count is an odd number after testing divisibility by 2, the nodes 4, 8, 12, and 24 no longer need to be tested, while nodes 9, 15, and 22 are tested if the tuple count is divisible by 3, 5, or 11, respectively.

Window sizes are not considered in the algorithms and window size checking is left to the query execution system. If window sizes were considered in the sliding algorithms, the sliding groups might become very limited because we cannot put the sliding windows with the same sliding values and different window sizes into the same sliding group. Algorithm 1 does not consider the dynamic behavior of addition and deletion of queries to/from the system. It only uses the provided sliding graph. A separate (simple) algorithm is needed to update the sliding graph when a new query is introduced to the system or when an existing query leaves it. Since new sliding windows could be added to the sliding graph when the system is running, the first sliding round of new sliding windows may not be accurately scheduled, which is generally acceptable. After the first round, Algorithm 1 and other algorithms which are described in the next sections work as expected. For example, if 958 tuples have received so far and a new count based sliding window with sliding value of 10 is added to the sliding graph, this sliding window is scheduled for its first execution just after the arrival of two new tuples. After this first sliding round, the sliding window will be scheduled for execution after each new 10 tuples. The maximum performance of the algorithms is when the system is in a relatively steady state and there are a large number of registered queries on input streams.

### 4.5.3 Sliding for Local Time based Sliding Windows

In the simplest way, we use a local timer for each sliding window (or sliding group). The time unit or timer tick of each timer is set to its associated sliding value. This approach requires a large number of timers in the system and leads to more processor and memory usage. However we can use a single timer for all sliding windows defined over a data stream by setting its time unit to the *gcd* of the sliding values. Again we can use Algorithm 1 to reduce the number of slide tests on each timer tick provided that the tuple counter in the algorithm is replaced with a time unit counter. The time unit counter keeps the sum of time units passed up to now. Suppose that the sliding values in Figure 4.3 are in seconds, so we can use them as an example for time based sliding windows. The *gcd* of these sliding values, and hence the timer tick, is 1 *sec*. The timer is scheduled to check the sliding windows every 1 *sec*.

### 4.5.4 Sliding for Remote Time based Sliding Windows

Handling remote time based sliding windows is more complicated than local time based sliding windows. The variable delays by what networks affect delivery of packets are the main source for the complication of the time management. These delays along with unsynchronized clocks may lead to out of order reception of tuples. We assume that a separate component is responsible for dealing with these out of order tuples. Using any approach to deal with out of order tuples, this subsystem delivers correctly ordered tuples to the sliding manager component.

One simple solution is to check the sliding on arrival of each new tuple without using any local timer. On arrival of the first tuple, the *next slide time* is computed for all sliding windows and then the sliding windows are sorted in increasing order of next slide times. For each next tuple, the tuple's timestamp is compared with the updated next slide time of the first sliding window. If it is not greater than the timestamp, the window is slid and its next slide time is updated and then the next window is tested. If the test is not passed, other sliding windows won't be checked. At the end of slide testing, the sliding windows must remain sorted. Algorithm 2 represents these steps.

This algorithm has some drawbacks. First, window sliding may not be done at exact times. In the worst case, the algorithm is postponing sliding of windows for a possibly long time. Suppose that the sliding value for a sliding window is 150 *sec* and new tuples arrive each $60 \pm 10$ seconds. In some cases we must wait for 60

```
1:  for each new tuple, tpl do
2:    if tpl is the first tuple then
3:      for each sliding window, SW do
4:        SW.nextSlide ⇐ SW.slide + tpl.timestamp
5:      end for
6:      Sort sliding windows in increasing order of their nextSlides
7:    else
8:      for each sliding window, SW do
9:        if SW.nextSlide ≤ t.timestamp then
10:         Add requests of SW to the candidate query list
11:         SW.nextSlide ⇐ SW.slide + tpl.timestamp
12:       else
13:         Ensure sorting of sliding windows
14:         Exit for
15:       end if
16:     end for
17:   end if
18: end for
```

**Algorithm 2**: Computing The Next Slide Time.

*sec* to receive the next tuple and then slide the window.

A different approach is to synchronize the local clock with the remote clock. The synchronization is based on the timestamps of the new tuples and therefore is an approximate method. The algorithm, which is executed for each sliding group, works as follows. At the arrival of the first tuple a timer is initialized, the timestamp of the tuple is set as the current time of the timer and the sliding value is set as its time unit. On each timer tick we delay the sliding by an estimated value for the delay $\lambda$. If a new tuple arrives during this period, we ignore the delay and do the sliding. Each time a new tuple arrives, the value of $\lambda$ is updated based on the following formula:

$$\lambda = \alpha\lambda + (1-\alpha)delay \tag{4.1}$$

where *delay* is the difference of the tuple's timestamp and the current time of the timer, and $\alpha$ is a value between 0 and 1 which specifies the weight of the previous value of $\lambda$ in the new value. In order to get a more accurate delay, the value of $\alpha$ can be refined during the execution of the algorithm to adjust the fraction of previous value of $\lambda$ that affects the current estimated delay. Note that in this approach a separate timer is used for each sliding group and also some late tuples may be discarded. It is easy to see that this approach requires more memory and processing time than the first one. It is up to the designer (or user) to choose

between the simplicity of the first algorithm and the better accuracy of the second one.

### 4.5.5 Optimizing the sliding graph

It is possible to have some slide values none of them is a factor of others but they may have some common factors. Assume that we have these sliding values: 7, 8, 12, and 20. The sliding graph produced for them has been shown in Figure 4.4(a). We know that 4 is the (greatest) common factor of 8, 12, and 20, so we can add a dummy node with slide value of 4 to the graph to get the sliding graph in Figure 4.4(b). To see the effect of adding an extra node to the graph, we can compare the number of node testing in these graphs. For $\tau$ tuples, the number of node testing is $C_a = 5\tau$ for graph (a) and $C_b = 3.75\tau$ for graph (b). Therefore, the number of eliminated nodes to be evaluated will be $C_a - C_b = 1.25\tau$, at the cost of adding one extra node to the original sliding graph. If the benefit of optimization is higher than the cost of extra nodes, we can optimize the graph by adding some dummy nodes to it. For each node $n$, the node optimization $O_n$ and the node optimization factor $\theta_n$ is defined as follows:

$$O_n = \frac{C_0 - C_1}{C_0} \times \frac{1}{S_n} \tag{4.2}$$

$$\theta_n = \frac{O_n}{cf(\sigma)} \tag{4.3}$$

where $C_0$ is the number of testing of the children of node $n$ before optimization, $C_1$ is the number of testing of the children of node $n$ after optimization, $S_n$ is the sliding value of the node, $\sigma$ is the number of extra nodes created for the optimization, and $cf(\sigma)$ is the *cost function* that takes $\sigma$ and returns the processing cost of $\sigma$ node(s) as a real number.

In order to optimize a sliding graph, two other optimization parameters are needed: the node optimization limit $\theta_l$, which is the minimum value of node optimization factor, and the graph optimization limit $\theta_g$, which is the lower bound on optimization of the graph. $\theta_l$ is a criterion for measuring the benefit of optimizing a node versus the cost of adding extra nodes. In other words, if the optimization factor of a node is less than $\theta_l$, then the cost of adding a node will become higher than the benefit we gain from the optimization and this optimization should not be applied to the sliding graph. If more than one node is optimized, we should inspect whether the sum of this optimization has a lower cost than the resulting benefit or it has a negative effect on running time of the algorithms. The $\theta_g$ parameter is used

not only to consider this issue but also to make a compromise between the number of extra nodes (and memory consumption increment caused by them) and the value of graph optimization. Therefore, for a realistic optimization of a sliding graph, the mentioned parameters should be carefully selected. This selection is done experimentally and based on the input rate of data, the way the graph is traversed, and the cost of extra nodes in this traversal.

Having these parameters, Algorithm 3 is used to optimize a sliding graph. The algorithm produces (in a greedy manner) an optimized sliding graph for the given parameters. Although it does not always produce the best optimization, it can build near optimal sliding graphs based on the given parameters. In this algorithm *n.sigma* is the number of extra nodes produced for optimizing node n. In order to compute possible optimizations at a node, it is necessary to compute all feasible combinations of child nodes based on their greatest common divisors. Although it is not a difficult problem, it might require heavy computation (exponential function of the number of children). In this case we can put a limit on the computation and continue with the best optimization resulted from this limited computation.

1: For each node $n$ in the graph compute $O_n$ ($n.opt$) and $\theta_n$ ($n.theta$).
2: If more than one optimization is possible for a node, select one optimization that its $\theta$ is maximum.
3: Add all nodes to the *nodeList*, except those nodes with $\theta < \theta_l$.
4: Sort *nodeList* in decreasing order of $\theta$s.
5: $sumOpt \Leftarrow 0$
6: $sumSigma \Leftarrow 0$
7: **for** each node $n$ in *nodeList* **do**
8:  **if** $((sumOpt + n.opt)/cf(sumSigma + n.sigma)) > \theta_g$ **then**
9:  Optimize $n$ in the graph
10:  $sumOpt \Leftarrow sumOpt + n.opt$
11:  $sumSigma \Leftarrow sumSigma + n.sigma$
12:  **else**
13:  **if** $n$ has multiple optimizations **then**
14:  Among those optimizations with $\sigma < n.sigma$ select one of them with the maximum $\theta$. If there exists such optimization, set it as the node's optimization and insert $n$ in the proper place in *nodeList*
15:  **end if**
16:  **end if**
17: **end for**

**Algorithm 3**:

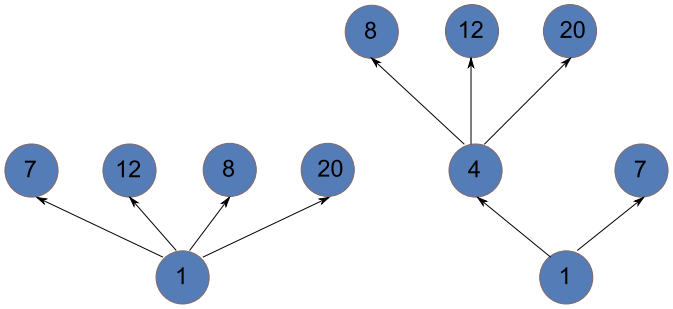



**Figure 4.4.** Original and optimized sliding graphs

## 4.6 Evaluation Results

To evaluate the effect of using sliding graphs in the sliding algorithms, the execution time of our count based sliding algorithm (without using any dummy nodes) has been compared with the execution time of the basic sliding algorithm which does not use sliding graphs. The basic sliding algorithm, which we call the *plain algorithm*, checks all sliding groups on arrival of a new tuple. We use the following configuration for the evaluation: The number of continuous queries with sliding windows defined over a single stream is 10000, 5000, 1000, 100, 50, 30, and 10. We used 200, 800 and 2000 as the maximum range for a count-based sliding values. Tuple production rate is 1000 tuples per second and each algorithm runs for 10 seconds (approximately 10000 tuples are produced). Sliding values are generated randomly in the range of 2 and maximum sliding value. We evaluate each algorithm 10 times independently, each time with different sliding values, and then the average execution time of the algorithms is used as the criteria for comparison of execution times. The evaluation was performed on a desktop with Intel dual core 2GHz processor, 2MB cache, 1GB memory, running Linux kernel 2.6.24. We used the GSN platform as our stream processing system.

Figure 4.5 shows the result of the evaluation when the maximum value of sliding values is 200. It can be seen that the processing time required for sliding is improved by the sliding graph algorithm (except for the case of 10 queries). As the number of queries increases the improvement of the sliding algorithm is getting more important. As the number of queries decreases, the time required for traversing the graph reduces the effect of using the sliding graph. The results of the evaluation are shown in Figure 4.6 and Figure 4.7 when sliding values are less

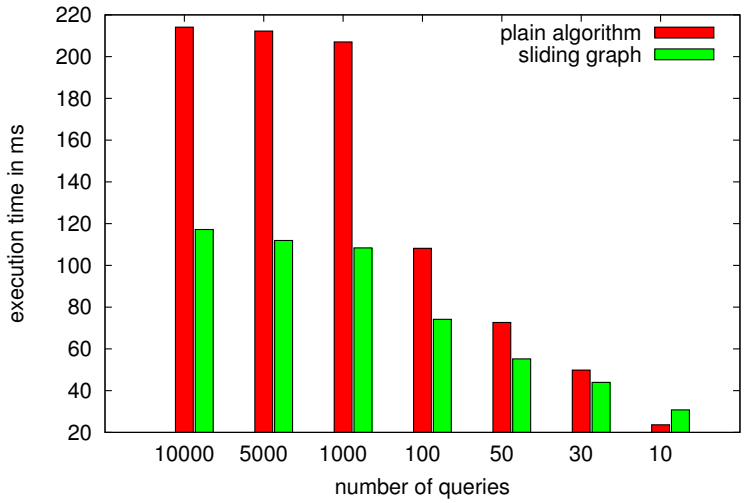



**Figure 4.5.** Average execution times of algorithms with sliding values are $\leq 200$.

than 800 and 2000, respectively. The execution time of the sliding graph algorithm is between 3 and 3.7 times better than the plain algorithm for 10000 and 5000 queries in Figure 4.6 and Figure 4.7, respectively. We can conclude that our sliding graph gives better performance as the number of queries increased and the range of sliding values gets expanded.

To view the effect of optimizing the sliding graph, each sliding graph has been optimized with two different optimization parameters. The first optimization has been done with $\theta_l = 0$, $\theta_g = 0$ and $cf(\sigma) = \sigma$ and the second optimization has been done with $\theta_l = 0.001$, $\theta_g = 0.01$ and $cf(\sigma) = \sigma$. The first optimization creates the optimal graph without any restriction with the minimum number of extra nodes. As in Algorithm 1, a stack is used when we search the sliding graph. To have a more accurate comparison, the number of comparison operations and stack pushes are calculated. Each sequence of push-pop is estimated to be equal to 10-12 comparison operations and is added to the actual comparison operations. In addition to the previous evaluation parameters, each algorithm is executed for 100000 tuples and the number of comparison operations and stack pushes is calculated.

Figure 4.8 shows the result of the evaluation for the original sliding graph and the optimized sliding graphs with the two series of parameters mentioned above for sliding values that are less than 200. This Figure shows that the optimized

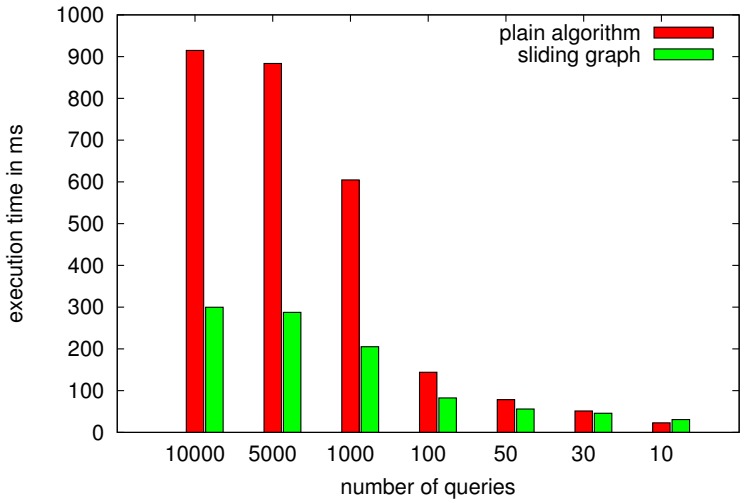



**Figure 4.6.** Average execution times of algorithms when sliding values are $\leq 800$.

sliding graph reduces the number of operations required in the sliding algorithms that uses the sliding graph (although the gains are not significant). Moreover, the first optimization works better than (or equal to) the second one. In general, we can conclude that by optimizing the sliding graph the execution time of the sliding graph algorithms is reduced, provided that the optimization parameters are correctly chosen.

Note that in the above evaluation results we only focused on count based sliding. As time is modeled using discrete integer values in the operating systems (e.g., number of milliseconds), one can use the exact same algorithms to handle time based sliding thus the evaluation results also covers the time based sliding queries.

In section 4.5.4 we proposed two different approaches for handling remote time based sliding windows and claimed that the second approach results in more accurate sliding times than the first one (Algorithm 2). To compare the accuracy of the algorithms, a stream source is used which produces a new tuple every 2 minutes. A random delay between 30 and 90 seconds is put on each tuple before sending them to the sliding manager. Two queries are defined on this data stream, one with a slide value of 3 minutes and other with a slide value of 5 minutes. The system

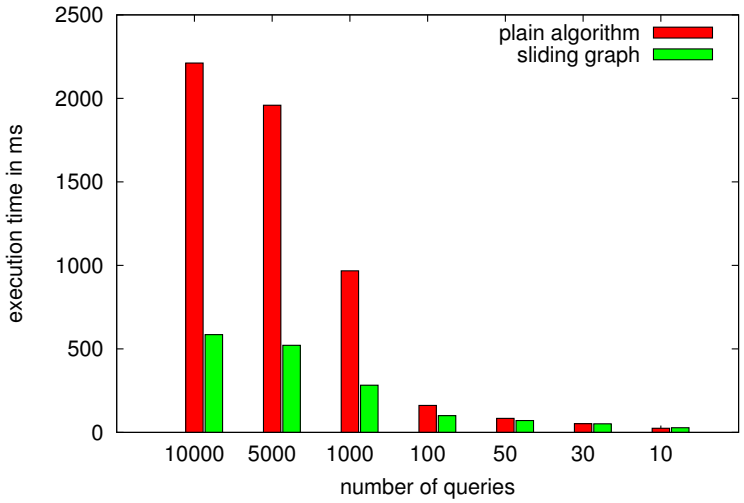



**Figure 4.7.** Average execution times of algorithms when sliding values are $\leq 2000$.

runs with both remote time based sliding algorithms and records the sliding times for each of them. Tables 4.1 and 4.2 show the time intervals between each sliding for each sliding algorithm. As can be seen, the algorithm using timers schedules the sliding windows to slide more accurately than the first algorithm. Algorithm 2 needs to wait for arrival of new tuples to decide on sliding while the second algorithm uses a timer to determine the sliding times and tries to synchronize this timer with the timer of the data source.

**Table 4.1.** Time intervals between each sliding in remote time based sliding algorithms when the slide value is 3 minutes.

| | | | | | | |
|---|---|---|---|---|---|---|
| *Algorithm 2* | 03:54 | 03:43 | 04:18 | 04:02 | 03:51 | 03:54 |
| *Using timers* | 02:56 | 03:39 | 02:58 | 03:06 | 03:12 | 03:29 |

**Table 4.2.** Time intervals between each sliding in remote time based sliding algorithms when the slide value is 5 minutes.

| | | | | | | |
|---|---|---|---|---|---|---|
| *Algorithm 2* | 06:03 | 05:49 | 06:22 | 05:48 | 06:20 | 05:52 |
| *Using timers* | 05:03 | 04:47 | 05:49 | 05:02 | 04:58 | 05:32 |

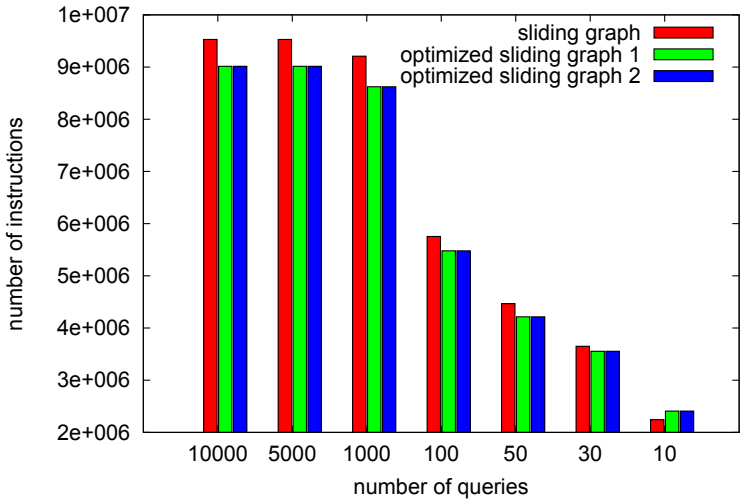



**Figure 4.8.** Comparison of original and optimized sliding graph with sliding values $\leq 200$.

## 4.7 Conclusion and Future Work

In this chapter we have presented a set of algorithms and techniques to deal with the management of sliding windows in stream processing systems. The proposed algorithms can be especially used in large-scale data stream processing systems in which there exist a large number of users registered to hundreds of high rate data streams. We address three possible types of sliding windows: count based, local time based, and remote time based. The sliding graph concept is introduced to reduce the processing time in sliding managers. Our evaluation results prove the efficiency of the sliding graph in the algorithms.

# Chapter 5

# Scalable Delivery of Stream Query Result

## Overview

Continuous queries over data streams typically produce a large volume of continuous result streams. To scale to a large number of users, one should carefully study the problem of how to deliver the result streams to the end users, which, unfortunately, is often overlooked in existing systems. In this chapter, we leverage Distributed Publish/Subscribe System (DPSS), a scalable data dissemination infrastructure, for efficient stream query result delivery. To take advantage of DPSS's multicast-like data dissemination architecture, one has to exploit the common contents among different result streams and maximize the sharing of their delivery. Hence, we propose to merge the user queries into a few representative queries whose results subsume those of the original ones, and disseminate the result streams of these representative queries through the DPSS. To realize this approach, we study the stream query containment theories and propose efficient query grouping and merging algorithms. The proposed approach is non-intrusive and hence can be easily implemented as a middleware to be incorporated into existing stream processing systems. A prototype is developed on top of the GSN system and results of an extensive performance study on real datasets verify the effectiveness of the proposed techniques.

## 5.1 Introduction

Stream processing systems are designed to evaluate complex continuous queries over high-rate data streams. The query results are typically in the form of continuous streams, which also have a very high data rate. Hence the delivery of query result streams from the processing server to the end users should be carefully handled. Unfortunately, this problem is often overlooked in existing systems. Most of them assume users are directly connected to the server and the result streams are sent to them directly. Such an architecture does not scale to a large set of users. To the best of our knowledge this is the first work that explicitly addresses the problem of scalable stream query result delivery, which is an important stepping stone towards massive stream query processing.

### 5.1.1 Motivating Scenario

The work in this chapter is motived by a performance issue that we faced for deploying a stream processing system shared by environmental scientists from multiple institutions in the Swiss Experiment project. In the project, environmental scientists are deploying a number of sensor stations to study the environmental changes and to provide alerts if needed (e.g., avalanche alert, etc). Depending on the purpose of each deployment, sensors of the stations are sampling at different rates. For instance, wind speed is sampled with 40 hertz or higher for each of the 3 directions (u,v,w) for each station, which generates very high-rate data streams (for more details visit Section 4.2).

An on-going effort of the project is to work with Microsoft Research to share the sensor data with other scientists and the world-wide public via Microsoft's SenseWeb (http://www.swiss-experiment.ch/index.php/MS:Home). This potentially requires the processing of a huge number of real-time stream queries. Delivering their results to a massive number of end users is one of the challenging problems of this platform (for detailed discussion about the SenseWeb project see Chapter 6).

As one of our efforts to solve the resulting issues, we propose to leverage an existing scalable data dissemination infrastructure, namely distributed pub/sub system (DPSS) [28], for query result delivery. A DPSS is typically supported by a number of brokers. In a DPSS, users express their data interest as user subscriptions which are propagated to the brokers. The data sources need not keep track of all the end users. Instead they only push the messages to their neighboring brokers,

**Table 5.1.** Example Queries

| | | |
|---|---|---|
| $Q_1$: | SELECT | S2.* |
| | FROM | Station1 [Range 30 Minutes] S1, |
| | | Station2 [Now] S2 |
| | WHERE | S1.snowHeight > S2.snowHeight |
| $Q_2$: | SELECT | S1.snowHeight, S1.timestamp, |
| | | S2.snowHeight, S2.timestamp |
| | FROM | Station1 [Range 1 Hour] S1, |
| | | Station2 [Now] S2 |
| | WHERE | S1.snowHeight > S2.snowHeight |
| $Q_3$: | SELECT | S2.*, S1.snowHeight, S1.timestamp |
| | FROM | Station1 [Range 1 Hour] S1, |
| | | Station2 [Now] S2 |
| | WHERE | S1.snowHeight > S2.snowHeight |

which cooperate with other brokers to disseminate the messages to the end users. Messages are routed within the network based on their content instead of explicitly specified destinations. With such a loosely coupled architecture, DPSS is shown to be scalable to a large number of users.

One can adapt a DPSS to disseminate the query result streams as follows. In a stream processing system, one query result stream is generated for each query. Hence, a unique identifier can be assigned to each query result stream. Then a user's subscription (i.e. the user's data interest) can be composed by specifying this unique identifier to retrieve the query result stream.

However, such a straight-froward approach is inefficient and involves large communication overhead. This is because the result streams could have overlapping contents. Disseminating these streams individually incurs many duplicate data transfers.

To illustrate the problem, Table 5.1 lists a few queries specified using CQL [19]. These queries are extracted and simplified from the typical snow drift monitoring tasks of the scientists.

Consider the join queries, $Q_1$ and $Q_2$, presented in Table 5.1. We can see the overlaps in the result streams generated for $Q_1$ and $Q_2$. Consider an overlay network structure depicted in Figure 5.1(a). Suppose nodes $n_3$ and $n_4$ post two queries $Q_1$ and $Q_2$ respectively and node $n_1$ is responsible for processing them. Using traditional techniques, their result streams, $s_1$ and $s_2$, are transmitted separately as shown in Figure 5.1(a). Hence the overlapping contents of $s_1$ and $s_2$ are transmitted twice over the link between $n_1$ and $n_2$ ($n_2$ is involved here because it is the neighboring broker of $n_1$ in the DPSS).


(a) Non-Share (b) Share

**Figure 5.1.** Result stream delivery

Note that existing multi-query optimization techniques, such as [70], also suffer from the same problem. For instance, one shared join operator can be created for the above two queries. However this join operator still generates two separate result streams for the aforementioned queries respectively.

To resolve this issue, we have to send one result stream $s_3$ to $n_2$, which is the superset of both $s_1$ and $s_2$, and "split" $s_3$ into two separate streams $s_1$ and $s_2$ at node $n_2$. This approach is illustrated in Figure 5.1(b). One can implement this approach by re-engineering a "specialized" stream processing engine to generate one result stream for multiple queries. However, such an intrusive approach is undesirable as it requires complex "low-level" software development and tightly coupled interactions between the processing engine and the overlay network.

This chapter proposes a query reformulation approach, which is relatively simple and easy to be implemented as a middleware between an existing stream processing engine and a DPSS. In our approach, for a group of queries that have overlapping results, the system composes a new query $Q$, called representative query, that contains all the queries in its group, i.e. the result of $Q$ is a superset of the result of each query in its group. For example, instead of submitting $Q_1$ and $Q_2$ individually, we create a new query $Q_3$ listed in Table 5.1, which contains $Q_1$ and $Q_2$, and we submit $Q_3$ to the processing engine at $n_1$. The result stream $s_3$ will be "split" at $n_2$ by using the filtering mechanism within the Distributed Publish/Subscribe System.

### 5.1.2 Contributions

In summary, we make the following contributions in this work:

- We study the problem of stream query containment with a focus on window predicates which do not exist in traditional SQL queries. The containment theorems developed here are not limited to this work and may benefit future studies on

stream query processing, such as multi-query optimization.

• Based on the containment theorems, we propose query merging algorithms for both SPJ (Select-Project-Join) queries and aggregate queries. These algorithms are meant to be simple in order to be executed efficiently at run time.

• We consider the situation that queries are inserted and terminated frequently and propose an efficient query grouping optimization and re-optimization mechanism. Queries are organized into a multi-tree data structure based on their containment relationship. This enables the adaptation algorithm to efficiently determine whether it is necessary to re-optimize the current grouping.

• A prototype system is implemented on top of GSN. Extensive experiments running on real datasets show that our approach is both efficient and effective.

### 5.1.3 Roadmap

The rest of this chapter is organized as follows. Related work is first reviewed in Section 5.2. Then Section 5.3 presents the assumptions and the system model for this work. Section 5.4 addresses the problem of how to generate the representative queries and the user subscriptions. Query grouping and its maintenance issues are addressed in Section 5.5. Section 5.6 provides an extensive performance evaluation study to verify our approach. Finally, Section 5.7 concludes the chapter with a discussion on the future work.

## 5.2 Related Work

This chapter is mainly related to the research activities in two areas: data stream processing systems and distributed publish/subscribe systems.

Stream processing has attracted much attention from the database community due to its vast applicability. There exist many efforts to enhance the scalability of these systems. One direction is to exploit the sharing of computation among queries. For instance, TelegraphCQ [33] proposed to share the join and filter operations among multiple queries and STREAM [19] studied the computation sharing of sliding-window aggregates. While these methods are effective in making the use of computation resources scalable, they do not consider the data communication load at the network layer. This work is complementary to these approaches and endeavors to scale up the scalability of query result delivery by exploiting the sharing among queries.

Another direction in scalable stream processing is to shed the excessive workload when the data arrives much faster than what the system can handle. Reference [89] presented an input tuple shedding strategy to maximize the query result quality. Authors in [24] proposed another tuple shedding strategy to minimize the loss of aggregate accuracy that would be incurred by the shedding. While one can adopt a similar strategy when the server runs out of bandwidth to deliver query results, it sacrifices the accuracy of the results. Our approach tries to adopt a better result delivery architecture, namely DPSS, to avoid (or minimize) the occurrence of such cases.

Yet another approach to scale up a stream system is to employ a number of distributed servers to share processing load. There are many recent activities in this direction. The authors of [17] studied the problem of how to place the query operators to widely distributed servers. The authors of [91] investigated the operator placement problem in a locally distributed system. Our approach is also complementary to these type of efforts. It can be used to disseminate the result streams of the queries/operators allocated to a processing server to its downstream destinations in a distributed stream processing system.

Finally, distributed publish/subscribe systems have been studied extensively in both the networking community and the data management community. Many research efforts have been focused on enabling scalable and efficient data dissemination services to a large number of users. For instance, efficient matching of events with subscriptions within a broker is studied in [16]. Authors in [28] presented the architecture design of a DPSS with a number of widely distributed brokers. In this chapter, we propose to leverage these existing efforts to enhance the scalability of a stream processing system. There are also very recent efforts to extend pub/sub systems to support more complex subscription types, such as range-MIN (or MAX or DISTINCT) in [30] and select-natural-join in [31]. However, these works focus on specific query types; they lack a systematic study of general queries. Furthermore, operators such as window joins and window aggregates, which are heavily used in many stream applications, have not been discussed in previous work.

## 5.3 Preliminaries

This section presents the assumptions and the system model of this chapter. The whole system consists of a stream processing server (or a cluster of servers), a DPSS infrastructure, and a number of end users. Consistent with the trend of

"Cloud Computing", we assume that end users have limited computing power and can only perform simple operations such as projection and selection. Complex operations like window joins and window aggregates can only be processed at the stream processing server. Furthermore, to loosen the coupling between the server and the DPSS, we assume the server has little knowledge of the internal overlay network structure of the DPSS.

### 5.3.1 DPSS

A subscription in the DPSS is a triple $\langle \mathscr{S}, \mathscr{P}, \mathscr{F} \rangle$. $\mathscr{S}$ is a set of stream names, which indicates the streams that are of interest to the subscriber. Only data from these streams would match the subscription. In our system, a unique stream name is assigned to each result stream of a query running in the processing engine. Hence the users can retrieve their query results by subscribing to the corresponding result streams. $\mathscr{P}$ specifies a few selected attributes from the streams in $\mathscr{S}$ that are of interest to the subscriber. Finally, $\mathscr{F}$ is a set of filters over the streams within $\mathscr{S}$. Data from the streams that satisfy these filters will be sent to the subscriber.

In the DPSS, subscriptions are forwarded from the subscribers to the data source. On the way of the forwarding, an intermediate node aggregates all the subscriptions that are received before forwarding to its upper stream neighbor(s). Furthermore, each node will build their own routing table based on the subscriptions it has. Upon receiving a message, the routing table is used to determine which downstream neighbor(s) the message should be sent to. If the message matches any subscription forwarded from a neighbor, it will be delivered to that neighbor. It can be seen that, even if there are more than one subscriber behind that neighbor interested in the same message, it will be sent only once.

### 5.3.2 Continuous Stream Queries

For simplicity, all the queries are assumed to involve only data streams and no stored table is considered.

An SPJ query $Q$ is assumed to contain the following components:

1. $Strm(Q)$: the set of streams involved by the query $Q$, $\{s_1, s_2, \cdots\}$, which typically appear in the FROM clause of the SQL string.

2. $Window(Q)$: a set of window predicates $\{w_1, w_2, \cdots\}$, one for each stream in $Strm(Q)$. For brevity, this chapter only discusses time-based sliding window, while other types of window can be treated similarly. The value of a time stamp is assumed to be a non-negative integer. A window predicate is defined as follows:

**Definition 5.3.1** *A window predicate $w_i$ takes an input stream $s_i$ and three non-negative integer parameters:*

- $begin_i \in [0, +\infty)$*: the starting time of the query*
- $interval_i \in [0, +\infty)$*: the interval of the window*
- $slide_i \in [1, +\infty)$*: the sliding step of the window*

*It defines a temporal relation $\mathscr{R}(\tau) = \{t | t \in s_i \ \& \ 0 \leq \tau - t.timestamp < interval_i\}$ at each time instance of $\tau = begin_i + n \cdot slide_i$, where n is a non-negative integer.*

For example, take a look at the window predicate defined on $s_1$ in Figure 5.2. Here, the parameter values are $begin = 0$, $interval = 5$ and $slide = 10$. Hence, on each time instance $\tau \in 0, 10, 20 \cdots$, the window predicate defines a temporal relation. For instance, at $\tau = 20$ and $\tau = 30$, it defines two temporal relation containing tuples within the rectangle drawn in Figure 5.2(a) and Figure 5.2(b) respectively.

3. $Pred(Q)$: the predicate specified by $Q$, which appear in the WHERE clause of the SQL string. $Pred(Q)$ is assumed to be in the disjunctive normal form: $\sigma_1 \vee \cdots \vee \sigma_i \vee \cdots \vee \sigma_n$, where $\sigma_i$ is the conjunction of one or more atomic predicate. An atomic predicate could be in one of the forms $attr.op.value$ and $attr1.op.attr2$ and involves the attributes from one (a selection predicate) or two streams (a join predicate).

4. $Attr(Q)$: the set of attributes selected by $Q$.

An aggregate query $Q$ takes one input stream, which could be the output of a SPJ query[1]. In summary, $Q$ contains the following components:

1. $Strm(Q)$: the input stream of $Q$;
2. $Window(Q)$: a window predicate defined on the input stream;
3. $Groupby(Q)$: the set of attributes in the GROUP BY clause;
4. $Agg(Q)$: a set of aggregate functions;
5. $Having(Q)$: filters applied over the groups;
6. $Attr(Q)$: a set of selected attributes.

### 5.3.3 Approach Overview

In the system, the server partitions the queries into a number of groups such that queries inside each group have overlapping results and it is beneficial to rewrite these queries into one query $Q$ which contains all the member queries $Q_i$. Such a query $Q$ is called the representative query of the query group. Only the representative queries are inserted into the underlying query engine and result streams of

[1]Note that such *SPJ* part of the aggregate query will not be considered for merging with other *SPJ* queries in the system.

these queries are pushed into the DPSS.

To allow the users to retrieve the query result streams of the individual queries, subscriptions are also generated and sent to the users. The users register these subscriptions to the DPSS, which efficiently delivers the result streams back to the users.

In the example presented in Section 1, the following two subscriptions are sent to $n_2$ by $n_3$ and $n_4$ respectively:

- $p_1$: $\mathscr{S} = \{s_3\}, \mathscr{P} = \{S2.*\}, \mathscr{F} = \{-30(minute) \leq S1.timestamp - S2.timestamp \leq 0\}$.
- $p_2$: $\mathscr{S} = \{s_3\}, \mathscr{P} = \{S1.snowHeight, S1.timestamp, S2.snowHeight, S2.timestamp\}, \mathscr{F} = \{\}$

Tuples that pass $p_1$ are sent to $n_3$ and those that pass $p_2$ are sent to $n_4$.

## 5.4 Query Merging

This section first studies the stream query containment problem and then presents the query merging algorithms based on query containment theorems developed. Finally it presents the algorithm to generate the subscriptions for the users to retrieve the result from the DPSS.

### 5.4.1 Stream Query Containment

Query containment and equivalence is a fundamental problem which has been extensively studied in the literature. For example, [29] and [80] studied the conjunctive select-project-join queries and union thereof; [39] and [75] discussed the aggregate queries; [63] studied queries with arithmetic comparison predicates; [26] investigated problems of recursive queries. We, however, need to extend these techniques to the continuous stream query context.

On the other hand, some related literatures studied the use of views to answer user queries [59]. This direction addresses how to rewrite a query such that the given views of the underlying relations can be utilized to answer the original query. However, our work is the other way round. We have to compose a "view" of the streams that can be utilized to answer multiple queries using the simple filtering mechanism in a DPSS.

First of all, we have to extend the query containment and equivalence definition of traditional queries to continuous stream queries. Traditionally, query containment and equivalence are defined as follows.

**Definition 5.4.1** *A query $Q_1$ is contained by another query $Q_2$, denoted by $Q_1 \sqsubseteq Q_2$, if for all database instances D, $Q_1(D)$ is a subset of $Q_2(D)$, i.e. $Q_1(D) \subseteq Q_2(D)$, where $Q_i(D)$ is the result of evaluating $Q_i$ over D. $Q_1$ and $Q_2$ are equivalent if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.*

The above definition is based on set semantics. It can be extended to bag semantics in a straightforward way.

However, in the continuous stream query context, the result data are continuously generated and hence this traditional definition is no longer applicable. To address this problem, we extend the definition as follows. First it is assumed that there is an application discrete time domain $\mathscr{T}$ where the timestamps of the input stream tuples are shown from. We denote the temporal result data set of a query $Q$ evaluated on a stream instance $S$ at the time instance $\tau \in \mathscr{T}$ be $Q(S,\tau)$, which is the result of evaluating $Q$ over all the data from $S$ with timestamps smaller or equal to $\tau$. Furthermore, let $S$ be the whole set of streams. We have the following definition.

**Definition 5.4.2** *A continuous stream query $Q_1$ is contained by another continuous stream query $Q_2$, denoted by $Q_1 \sqsubseteq Q_2$, if for all stream instances S, $Q_1(S,\tau) \subseteq Q_2(S,\tau)$ at any time instance $\tau$. $Q_1$ and $Q_2$ are equivalent if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.*

Now the problem is how to determine the containment relationship between two continuous stream queries. The major difference between continuous stream queries and traditional database queries is the introduction of window semantics. Note that if all window predicates in a continuous stream query have an infinite time interval, then the two containment problems are equivalent. Here we assume that there is an approach to determine containment relationship between two traditional database queries, and develops the theorems to deal with the window predicates.

Furthermore, this work assumes that all the window predicates in a single query have a common sliding step and a common starting time. This covers most real application scenarios and simplifies the query merging algorithms. (Note that the sliding steps and starting times of different queries could be different.)

First, we have the following lemma stating the conditions that two tuples could be joined in a window-based join operator.

**Lemma 5.4.1** *For a query with only a window-based join operation of two streams $s_1$ and $s_2$ with window sizes of $interval_1$ and $interval_2$ respectively and a common*

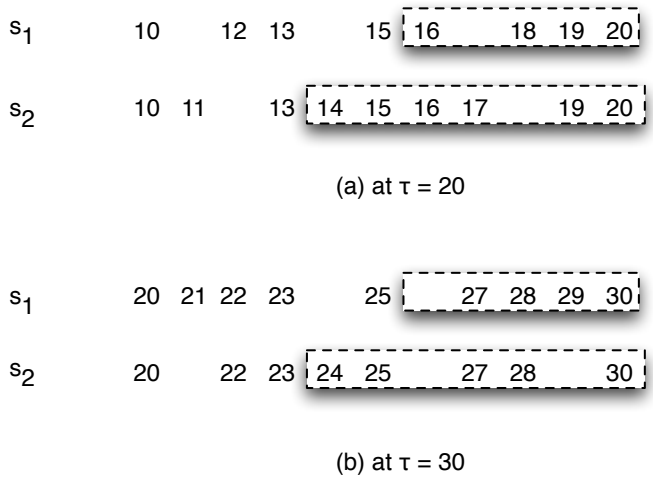



**Figure 5.2.** Window join query Q between two streams $s_1$ and $s_2$, with $begin(Q) = 0$, $slide(Q) = 10$, $interval_1(Q) = 5$ and $interval_2(Q) = 7$. Only the timestamps of the arrived tuples are shown.

*sliding step "slide" and a common query starting time "begin", two tuples $t_1$ from $s_1$ and $t_2$ from $s_2$ can generate a join result tuple $t$ if and only if all the following conditions are true:*

*(1) they satisfy the join predicates;*

*(2)* $-1 \cdot interval_1 \leq t_1.ts - t_2.ts \leq interval_2$.

*(3)* $t_1.ts > n_2 \cdot slide - interval_1$, *where* $n_2 = \lceil (t_2.ts - begin)/slide \rceil$

*(4)* $t_2.ts > n_1 \cdot slide - interval_2$, *where* $n_1 = \lceil (t_1.ts - begin)/slide \rceil$. □

Within Lemma 5.4.1, condition (2) basically says that the two joined tuples should appear in the corresponding window intervals. For instance, in Figure 5.2(a), tuple "14" from $s_2$ is the earliest tuple from $s_2$ that can be joined with tuple "20" from $s_1$. However, this condition alone cannot guarantee the correctness. For example, in Figure 5.2(b), tuple "23" from $s_1$ cannot join with tuple "22" from $s_2$ based on the window predicate definition. Condition (3) and (4) are used to deal with such cases. They ensure that there exists a pair of temporal relations at a particular time instance that contain the two tuples respectively. Based on this Lemma, we can get the following theorem.

**Theorem 5.4.1** *A select-project-join (SPJ) continuous query $Q_1$ is contained by another SPJ continuous query $Q_2$ iff they satisfy either conditions* $(1) \wedge (2) \wedge (3) \wedge (4)$ *or conditions* $(1) \wedge (2) \wedge (3) \wedge (5) \wedge (6)$*:*

*(1)* $Q_1^{\infty} \sqsubseteq Q_2^{\infty}$*, where $Q_i^{\infty}$ is a query resulted from setting all the window sizes of $Q_i$ as $\infty$;*

*(2)* $begin(Q_1) \geq begin(Q_2)$*;*

*(3)* $\forall i$*,* $inteval_i(Q_1) \leq interval_i(Q_2)$*, where $interval_i(Q_j)$ is the window size of the ith stream involved in $Q_j$;*

*(4)* $\forall i,\ 1 \leq slide(Q_2) \leq \max(1,\ interval_i(Q_2) - interval_i(Q_1))$*, where $slide(Q_2)$ is the sliding step of $Q_2$.*

*(5)* $\exists m \in [1, \infty)$*, s.t.* $begin(Q_2) + m \cdot slide(Q_2) - begin(Q_1) \leq \min_i(interval_i(Q_2) - interval_i(Q_1))$.

*(6)* $slide(Q_1) = k \cdot slide(Q_2)$*, where k is a positive integer.* □

The essential idea of Theorem 5.4.1 is, if $Q_1$ is contained by $Q_2$, then for every time instance $\tau_1$ at which a temporal relation $\mathscr{R}_1^i(\tau_1)$ for each stream is defined by the window predicates in $Q_1$, there exists at least another time instance $\tau_2$ at which a temporal relation $\mathscr{R}_2^i(\tau_2)$ for each stream is defined by the window predicates in $Q_2$ and contains $\mathscr{R}_1^i(\tau_1)$.

Conditions (1)-(3) are easy to understand. First, $Q_2$ has to contain $Q_1$ without considering the window predicates. Second $Q_2$ has to begin earlier than $Q_1$ and $Q_2$'s window intervals should be as large as the corresponding ones in $Q_1$.

Condition (4) says that the sliding step of $Q_2$ is smaller than the difference of the two window intervals defined on the same stream. Figure 5.3 illustrates the reasoning behind this. It can be seen that the temporal relation defined by $Q_2$ at $\tau = 20$, $\mathscr{R}_2(18)$ (shown in Figure 5.3(b)) does not contain the one defined by $Q_1$, $\mathscr{R}_1(20)$ (shown in Figure 5.3(a)). But, as long as the sliding step of $Q_2$ is smaller than the difference between the two window intervals, there would exist a time instance $\tau$ such that $\mathscr{R}_1(18) \subset \mathscr{R}_2(\tau)$. For example, in Figure 5.3(c) shows that actually $\mathscr{R}_2(22)$ contains $\mathscr{R}_1(20)$.

Conditions (5) and (6) state the case that the windows of the two queries slide synchronously. Figure 5.4 shows an example. Here, $\mathscr{R}_2(20)$ contains $\mathscr{R}_1(19)$ as shown in Figure 5.4(a) and (b). Furthermore, as their sliding steps fulfill Condition

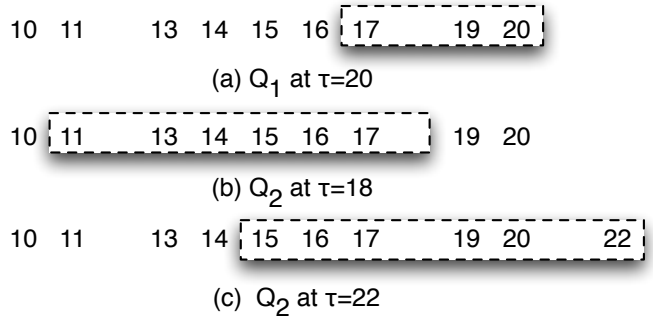



**Figure 5.3.** Temporal relations defined by two queries on the same stream. The parameters are $interval(Q_1) = 4, interval(Q_2) = 8$, and $slide(Q_2) = 4$.

(6), this containment pattern will repeat in the future time instances. Figure 5.4(c) and (d) illustrate the situation at another time instance.

**Theorem 5.4.2** *A continuous stream aggregate query $Q_1$ is contained by another continuous stream aggregate query $Q_2$ iff all the following conditions are true:*

*(1)* $Q_1^\infty \sqsubseteq Q_2^\infty$*, where $Q_i^\infty$ is a query resulted from setting all the window sizes of $Q_i$ as $\infty$;*

*(2)* $begin(Q_1) \geq begin(Q_2)$*;*

*(3)* $\forall i, interval_i(Q_1) = interval_i(Q_2)$*, where $interval_i(Q_j)$ is the window size of the ith stream in query $Q_j$;*

*(4)* $slide(Q_1) = k \cdot slide(Q_2)$ *where k is a positive integer.*

This reasoning of Theorem 5.4.2 is similar to Theorem 5.4.1. Hence, for brevity, we shall not reiterate here.

### 5.4.2 Query Merging Algorithms

Recall that, in our approach, the server maintains a number of query groups such that queries inside each group have overlapping results and it is beneficial to merge these queries into one representative query $Q$ that contains all the member queries $Q_i$.

With the lemma and theorems developed in the previous section, we can generate the representative query for a group of queries as presented in the following subsections.

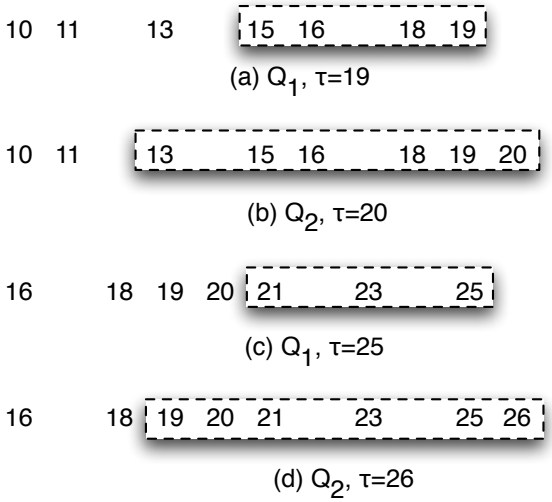



**Figure 5.4.** Temporal relations defined by two queries on the same stream. The parameters are $interval(Q_1) = 5, interval(Q_2) = 8, slide(Q_1) = 6$, and $slide(Q_2) = 3$.

#### 5.4.2.1 Merging SPJ Queries

The function to merge two SPJ queries is presented in Algorithm 4. It takes two queries as its input parameters and returns a query that contains them. In this chapter, we only consider merging queries involving the same set of streams to avoid incurring large processing overhead. Line 3 enforces this constraint.

Lines 4 – 12 deal with the situation that one of the input queries $Q_i$ contains the other $Q_j$. In this case, the function simply return $Q_i$. But note that the containment checking here does not consider which attributes are selected by the two queries. Therefore, we have to combine the attribute selection lists of both queries (line 29). Furthermore, to allow retrieval of the results of $Q_j$ from that of $Q_i$, the attribute list $Q$ is extended with those appear in $Pred(Q_j)$ (line 30).

Lines 13 – 28 perform the merging of two queries that do not contain each other They merge the predicates, stream windows and the selected attributes one after another. Among these, lines 19 – 28 refine the sliding step of $Q$ step by step based on Theorem 5.4.1. Note that the merged predicates might be further reduced if some of them are covered by the others. Minimizing the number of predicates is a traditional NP hard problem and is out of the scope of this work.

```
1  MergeSPJ(Q1, Q2)
2  begin
3      if Strm(Q1) ≠ Strm(Q2) then return error;
4      if Q1 ⊑ Q2 then
5          Q ← Q2;
6          Attr(Q) ← Attr(Q1) ∪ Attr(Q2);
7          if Q1 ≠ Q2 then
8              Attr(Q) ← Attr(Q) ∪ attributes in Pred(Q1);
9      else if Q2 ⊏ Q1 then
10         Q ← Q1;
11         Attr(Q) ← Attr(Q1) ∪ Attr(Q2);
12         Attr(Q) ← Attr(Q) ∪ attributes in Pred(Q2) ;
13     else
14         Strm(Q) ← Strm(Q1);
15         Pred(Q) ← Pred(Q1) ∪ Pred(Q2);
16         begin(Q) ← min(begin(Q1), begin(Q2));
17         slide(Q) ← max(slide(Q1), slide(Q2));
18         gcd ← GCD(slide(Q1), slide(Q2));
19         foreach stream s_i ∈ Strm(Q) do
20             diff ← interval_i(Q1) − interval_i(Q2);
21             interval_i(Q) ← max(interval_i(Q1), interval_i(Q2));
22             if interval_i(Q1) > interval(Q2) then
23                 s ← slide(Q1);
24             else if interval_i(Q1) < interval(Q2) then
25                 s ← slide(Q2);
26             else
27                 s ← min(slide(Q2), slide(Q1));
28             slide ← min(s, max(gcd, diff));
29         Attr(Q) ← Attr(Q1) ∪ Attr(Q2);
30         Attr(Q) ← Attr(Q) ∪ attributes in Pred(Q1) or Pred(Q2);
31     return Q;
32 end
```

**Algorithm 4**: Merging two SPJ queries

#### 5.4.2.2 Merging Aggregate Queries

Algorithm 5 is to perform the merging of two aggregate queries. Here only queries with the same input stream, the same Group By attributes are considered for merging and the same window intervals. Again, this algorithm is based on Theorem 5.4.2.

```
1  MergeAgg(Q1, Q2)
2  begin
3      if Strm(Q1) ≠ Strm(Q2)∨
       Groupby(Q1) ≠ Groupby(Q1) ∨ interval(Q1) ≠ interval(Q2) then
       return error;
4      Strm(Q) ← Strm(Q1);
5      Groupby(Q) ← Groupby(Q1);
6      Having(Q) ← Having(Q1) ∪ Having(Q2);
7      Agg(Q) ← Agg(Q1) ∪ Agg(Q2);
8      Attr(Q) ← Attr(Q1) ∪ Attr(Q2);
9      interval(Q) ← interval(Q1);
10     slide(Q) ← GCD(slide(Q1), slide(Q2));
11     return Q;
12 end
```

**Algorithm 5**: Merging two aggregate queries

### 5.4.3 Subscription Generation

As the result streams of only the representative queries will be delivered over a DPSS, users have to register subscriptions to the DPSS to retrieve the results that are of interest to them. This subsection presents how to generate such subscriptions.

Suppose $Q$ is the representative query for a query group and $Q_i$ is one of the members of this group. Algorithm 6 generates a subscription for the user to fetch result of $Q_i$ from the DPSS. The algorithm first initializes the subscription with $Q$'s result stream name, the selected attribute list of $Q_i$ and predicates of $Q_i$ (line 2). Then at line 8, filters are added to check whether the result tuple satisfies the window predicate defined by $Q_i$. These filters are generated based on Lemma 5.4.1.

```
1  SubGen(Q, Q_i)
2  begin
3      Sub.𝒮 ← {Result(Q)};
4      Sub.𝒫 ← Attr(Q_i);
5      Sub.ℱ ← Pred(Q_i);
6      n ← the number of stream involved in Q_i;
7      bg ← begin(Q_i);
8      sl ← slide(Q_i);
9      for j = 1; j < n; j++ do
10         for k = j+1; k <= n; k++ do
11             Sub.ℱ ← Sub.ℱ ∧
12             (−1 · inv[j] ≤ ts_j − ts_k ≤ inv[k]) ∧
13             ts_j > ⌈(ts_k − bg)/sl⌉ ∗ sl − inv[j] ∧
14             ts_k > ⌈(ts_j − bg)/sl⌉ ∗ sl − inv[k];
15     return Sub;
16 end
```

**Algorithm 6**: Subscription Generation

## 5.5 Query Grouping

With the above algorithms, one can merge a query group and efficiently deliver the query results to the individual users. This section investigates how to partition the queries into multiple groups. In particular, we study the optimization algorithms and maintenance mechanisms of query grouping under the assumption that queries are frequently inserted and terminated.

### 5.5.1 Benefit Estimation

The first problem of optimizing query grouping is how to estimate the benefit of merging a query group. The benefit considered in this work is the amount of data communication overhead that can be saved. A common cost metric is adopted here: $\sum_i l_i \cdot c_i$, where $l_i$ is the transmission latency of the $i$th link in the overlay network of the DPSS and $c_i$ is the communication traffic per unit time on $l_i$.

To accurately estimate the benefit of merging a query group, one can count the data transfer rate on each overlay link if we know the exact data dissemination tree. Unfortunately, in a large scale network, it is hard to maintain information in such a detail.

As the problem of how to maintain network structure knowledge in a scalable way is out of the scope of this work, we only adopt a cost model assuming little

knowledge of the network. Since the actual cost is considered in the experimental results, this actually biases against our method. Moreover, if more network knowledge is available, the cost model can be replaced with a more accurate one without much change to our algorithms.

More specifically, the stream processing server only keeps track of the next hop of the delivery path of the result streams. The benefit of the query merging is estimated as $(\sum_i C(Q_i) - C(Q)) \cdot l$, where $C(Q)$ is the data rate (bits/sec) of the representative query $Q$'s result stream, while $C(Q_i)$ is the data rate of the member query $Q_i$'s result stream. Furthermore $l$ is the latency of the common first hop of all the member query $Q_i$. This implicitly says that only queries with a common first hop would be merged, which is intuitive.

The estimation of the result stream rate is a common task required by most query optimizers. Therefore, existing techniques in stream query optimization [23] can be used for this purpose. Furthermore, the data statistics required by our cost model can be shared with the query optimizer and hence little extra overhead will be incurred to maintain the statistics.

### 5.5.2 Query Groups Maintenance

There are a few challenges of the query grouping and our system addresses them in the following ways:

1. Achieve high benefit. The benefit estimation function discussed above is used to estimate the benefit of a grouping. Heuristics are required to derive a good grouping.

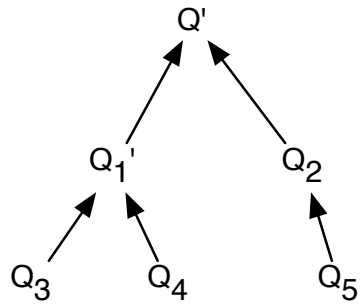



**Figure 5.5.** An example query tree. This tree represents a group of query: $Q_2, Q_3, Q_4$ and $Q_5$. $Q'_1$ is a derived query constructed by merging $Q_3$ and $Q_4$. $Q_5$ is contained by $Q_2$. $Q'$ is the representative query of the whole query group which is derived by merging $Q'_1$ and $Q_2$.

2. Efficient grouping maintenance. We consider the situation that queries would be inserted and terminated anytime. Hence, it is necessary to have an efficient data structure to maintain the query grouping and to facilitate the decision making of when and how to re-merge queries. A multi-tree data structure is adopted to serve this purpose. In this structure, a query tree is built for each query group. The root of the tree is the representative query of the query group. Within the tree, a query $Q_i$ is an ancestor of another one $Q_j$ only if $Q_i$ contains $Q_j$. Figure 5.5 shows an example of such a tree.
As we will see later, such a query tree is helpful in quickly determining whether the merging of the queries under each subtree is beneficial and hence whether re-optimization is required. Furthermore, if some queries in a subtree are terminated, the synthetic queries along the path from the terminated queries to the root might be rewritten, while the other subtrees need not be modified.

3. Re-placing queries into different groups would change the user subscriptions and hence may incur many message exchanges among the nodes in the network in order to modify the routing table in the DPSS. Hence, we try to avoid frequent migration of a query from one group to another. Query grouping is maintained periodically so that the frequency of the changes can be regulated by the length of the period. Furthermore, a query will be retained in the same group as long as it is still beneficial to do so.

More details of the algorithms are presented in the rest of this section.

#### 5.5.2.1 New query insertion.

When a query is first submitted to the system, Algorithm 7 is run to insert it into a query group. The algorithm estimates the benefit of merging the new query with each existing query group and selects the query group with the highest benefit (line 3). If no merging has positive benefit, then a new group is generated.

The new query is added to the corresponding query tree structure of the selected query group or the newly generated group (line 10). If the existing representative query of the group and the new query do not contain one another, the algorithm will generate a new representative query (line 18) for this query group.

As a side note, query merging may incur query processing overhead. To avoid getting arbitrary high overhead, a threshold can be used to restrict how much overhead can be accepted to trade for the communication efficiency. Here, we use a threshold parameter $\alpha$, which is the maximum percentage of processing overhead that can be accepted. For example, if $\alpha = 0.1$, then the system can tolerate 10%

of processing overhead incurred by query merging. If the merging incurs overhead higher than this threshold, then it will not be considered. This is implemented by line 6. In the estimation of the processing cost, again existing stream query optimization techniques can be used, such as [23].

```
1  Insert(newQuery)
2  begin
3      max ← 0; toMerge ← null;
4      foreach rootQ ∈ trees do
5          bf ← benefit of merging rootQ with newQuery;
6          oh ← processing overhead incurred by merging rootQ with
           newQuery + the current overhead of the query group of rootQ;
7          if bf > max & oh < α then
8              max ← bf;
9              toMerge ← rootQ;
10     if toMerge = null then
11         trees.addRoot(newQuery);
12     else if newQuery ⊑ toMerge then
13         AddChild(toMerge, newQuery);
14     else if toMerge ⊑ newQuery then
15         AddChild(newQuery, toMerge);
16         replace toMerge with newQuery in trees
17     else
18         newRoot ← MergeQ(toMerge, newQuery);
19         add both toMerge and newQuery to newRoot.childlist;
20         replace toMerge with newRoot in trees;
21 end
22 AddChild(parent, newChild)
23 begin
24     foreach child ∈ parent.childlist do
25         if newQuery ⊑ child then
26             AddChild(child, newQuery);
27             return;
28     add newChild to parent.childlist;
29 end
```

**Algorithm 7**: Query Insertion

#### 5.5.2.2 Query termination.

When a query terminates, Algorithm 8 modifies the query trees to reflect the changes. Two types of queries are distinguished in the algorithm: (1) original queries: those queries submitted by the users; (2) derived queries: those queries derived from query merging.

First, the to-be-terminated query is removed from the tree and then transfer its children to its parent or generate a new root if the to-be-terminated query is a root itself.

Second, if the to-be-terminated query's parent is a derived query, the merging algorithm will be run to rewrite the parent query to reflect the change. Rewriting will be propagated up in the tree till the node which is not required to be rewritten. Note that the rewriting of these synthetic queries will not incur changes on the network side (i.e. the subscriptions of the users can remain unchanged). Instead, the rewriting can reduce the communication cost by the possible "tightening" of the representative queries. Hence, we choose to perform this eagerly.

On the other hand, if a query in a query group is terminated, then it might not be beneficial for other queries to be placed in this query group any more. For example, a query is placed into this group because of its overlap with the terminated query. Now, it might not be beneficial to keep it in this group. The grouping could be re-optimized here. However, as we have discussed, moving the query from one group to another has to change the subscription of the user which will incur changes on the overlay network, i.e. the change of user subscriptions and hence the routing tables.

Therefore, a lazy approach is adopted for the re-optimization of grouping. The re-optimization algorithm is run periodically, which will be presented below. As shown in line 10 of Algorithm 8, the re-written derived queries are marked to be re-optimized, which will be done at the next re-optimization round.

#### 5.5.2.3 Query group re-optimization

Periodically, Algorithm 9 will be run to re-optimize the query grouping. In line 4, the algorithm traverses the query trees and re-optimizes them one by one. After that, it gets a list of queries that should be considered to be replaced into different groups. Then line 5 uses the query insertion algorithm (Algorithm 7) to replace the query one by one.

```
1  Terminate(q)
2  begin
3      remove q from its query tree;
4      if q.childlist ≠ null then
5          if q.parent ≠ null then
6              q.parent.childlist.add(q.childlist);
7          else
8              newQuery ← merge all the child queries of q;
9              newQuery.isDerived ← true;
10             newQuery.to_reoptimize ← true;
11     if q.parent ≠ null & q.parent.isDerived then
12         Rewrite (q.parent);
13         q.parent.to_reoptimize ← true;
14 end
15 Rewrite (q)
16 begin
17     Merge all the child queries of q to a new query newQ;
18     if q is not semantically equivalent to newQ then
19         q ← newQ;
20         if q.parent ≠ null & q.parent.isDerived then
21             Rewrite(q.parent);
22 end
```

**Algorithm 8**: Query Termination

```
1  ReopitmizeGroups()
2  begin
3      toReplace ← ∅;
4      foreach root ∈ trees do Reoptmize(root, toReplace);
5      foreach query ∈ toReplace do Insert(query);
6  end
7  Reoptmize(queryNode, toReplace)
8  begin
9      newChildlist ← ∅;
10     foreach child ∈ queryNode.childlist do
11         Reoptmize(child, newChildlist);
12     if queryNode.to_reoptimize then
13         b ← benefit of merging all queries in
           {queryNode.childlist ∪ newChildlist};
14         if b ≤ 0 then
15             toReplace.add(queryNode.childlist);
16             toReplace.add(newChildlist);
17             remove q from the trees;
18         else
19             queryNode ← merge all queries in
               {queryNode.childlist ∪ newChildlist};
20         if toReplace ≠ ∅ & queryNode.parent.isDerived then
21             queryNode.parent.to_reoptimize ← true;
22 end
```

**Algorithm 9**: Query Group Re-Optimization

The re-optimization algorithm for each query group is shown in Line 7. This algorithm takes as inputs *queryNode* (a node in a query tree) and inserts into *toReplace* the queries that are currently in the substree rooted at *queryNode* but are no longer beneficial to be grouped with other queries in the subtree. It is done by traversing the query tree in depth-first order and recursively calling the algorithm on each node in the tree.

For each node, after calling the algorithm recursively on all the child nodes, the algorithm gets a list of query nodes, *newChildlist*, which are the queries that are no longer be beneficial to be placed in the subtrees of the individual child nodes. However, within these queries, those extracted from the subtree of one child node may still have overlap with the queries in the subtree of another child node. Hence, line 12 check whether it is beneficial to merge the queries in *newChildlist* together with its current children. If so, then it simply performs the merging and add all the nodes in *newChildlist* to the current nodes childlist. Otherwise, it returns all these nodes to its parent node.

Note that to minimize the overhead of revising the user subscriptions that would be incurred by re-grouping, queries would be considered for re-grouping only when the current grouping has negative benefit (line 14). Using a threshold here might be able to get a better trade-off. Unfortunately, as our experiments show, this cannot achieve any significant benefit. Therefore, such a threshold is not considered in this work.

## 5.6 Performance Study

In this section, the result of a performance study is reported. We first present the configuration of the experiments and then the detail experimental results.

**Implementation.** The algorithms in this work are implemented in a middleware on top of our stream processing system GSN (Global Sensor Network, http://gsn.sourceforge.net/) [11], which is tailored for efficient processing of sensor data and managing the connections with various heterogeneous sensor networks. The system is implemented mainly in Java. The experiment is conducted in a Linux server with 2 Dual-Core 2.66GHz Intel CPU and 4G memory.

**Data set.** We use the sensor data set collected by our SensorScope project (http://sensorscope.epfl.ch), which measures key environmental data such as air temperature and humidity, surface temperature, incoming solar radiation, wind speed and direction, precipitation, and soil moisture and pressure. The data from

each sensor are treated as one data stream. In the experiments, we use 63 streams as our data set and emulate the streaming scenario by using their timestamp information.

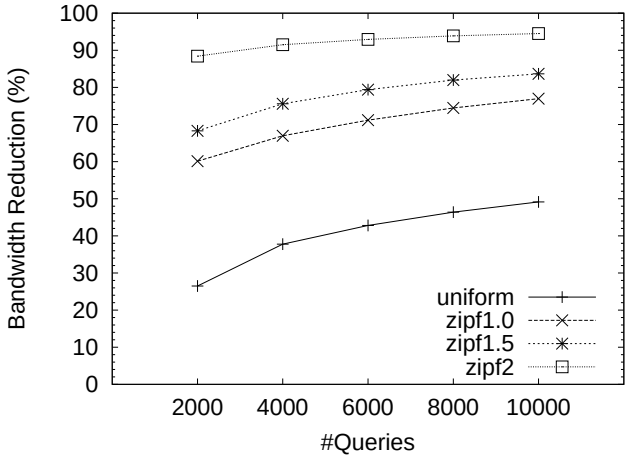
**Query generation.** Each query in the experiment is generated randomly in the way described as follows. First, a few streams (a random number from one to five) are selected randomly to be involved in the query. Then a few predicates are generated based on the column information of the streams (such as the column names, the maximum/min values etc.). In the experiments, we vary the distribution used to select the streams and the portion of data selected by the predicates. Both uniform and zipfian distribution are used. Furthermore, the window predicates are generated with random parameters (time intervals, sliding steps and starting times). Finally, the projection attributes and aggregate functions are generated randomly. All the experiments are repeated 20 times with different random queries and the average results are reported.

**DPSS.** The DPSS which is used to disseminate the query results is simulated in the experiments. The topology generator BRITE (http://www.cs.bu.edu/brite/) is used to generate a power law network topology with 1000 nodes. Then a minimum spanning tree is constructed as the dissemination tree. One of the nodes is selected as the stream query processor and, for each query, a random node is selected as the origin of the query, which should be the destination of the query result.

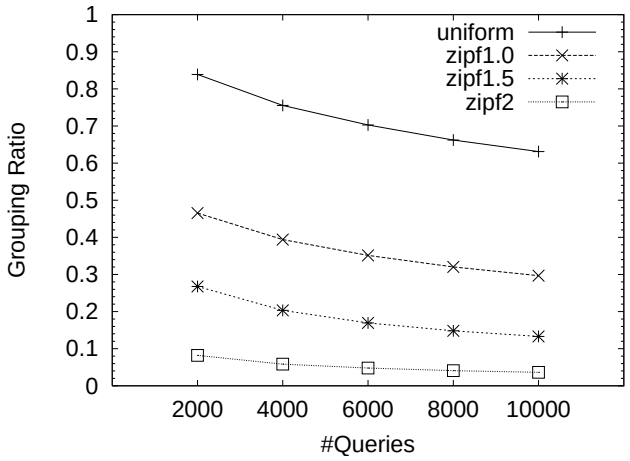
### 5.6.1 Query Insertion

In this subsection, we examine the performance of query insertion. In the experiments, queries arrive at the system one by one. Our query insertion algorithm is run to optimize the query grouping incrementally.

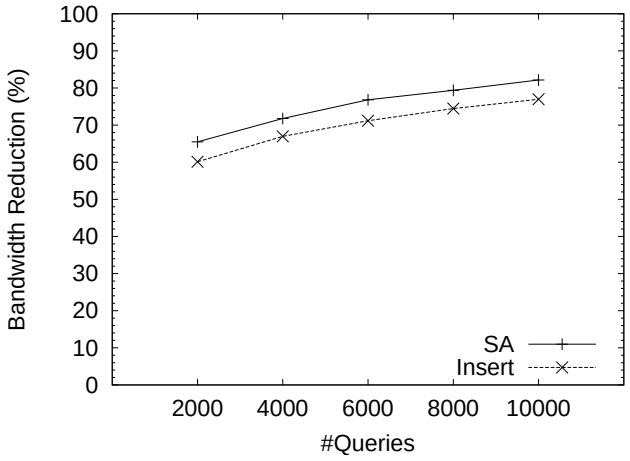
In the first experiment, we set the overhead threshold $\alpha$ to a relatively high value (0.5) to see how much benefit we can get without worrying too much about the processing overhead. In Figure 5.6(a), we present the *bandwidth reduction* at each time instance when a certain number of queries are inserted. *Bandwidth reduction* is computed as the percentage of the sum of the bandwidth consumption of each overlay link (weighted by the latency of each link as discussed in Section 5.5.1) that is reduced by the query merging in comparison to the case without merging. A few interesting points can be derived from the figure. First, with a higher number of queries added to the system, there are more opportunities for the query merging approach to explore the sharing of communication and hence




(a) Bandwidth Reduction




(b) Grouping Ratio




(c) Compare with Simulated Annealing (zipf1.0)

**Figure 5.6.** Query Insertion

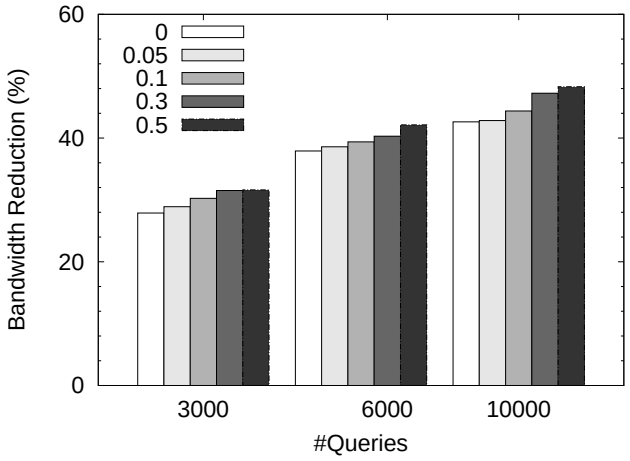
a larger bandwidth reduction can be achieved. Another interesting point is that query merging is more beneficial with a skewed query distribution. The reason is obvious. With more queries interested in the same subset of data, the probability that we can merge the queries would be higher. Figure 5.6(b) provides another perspective on the experimental results. The *grouping ratio* is the ratio of the number of query groups to the total number of queries. Generally, the lower the grouping ratio, the higher the bandwidth reduction could be.

To examine the optimality of the query grouping, we also compare with the Simulated Annealing (SA) algorithm [61], which has been shown very effective in solving many NP-Hard problems. The parameters of SA are tuned to achieve the best performance as we could. For clarity, only the results with zipfian distribution ($\theta$=1.0) are reported. The other results have similar trends. As shown in Figure 5.6(c), the insertion algorithm performs slightly worse than the SA algorithm does. However, SA runs more than 100 times slower than the insertion algorithm in all our experiments. As queries often come and leave frequently in reality, the insertion algorithm is more favorable in a real deployment.

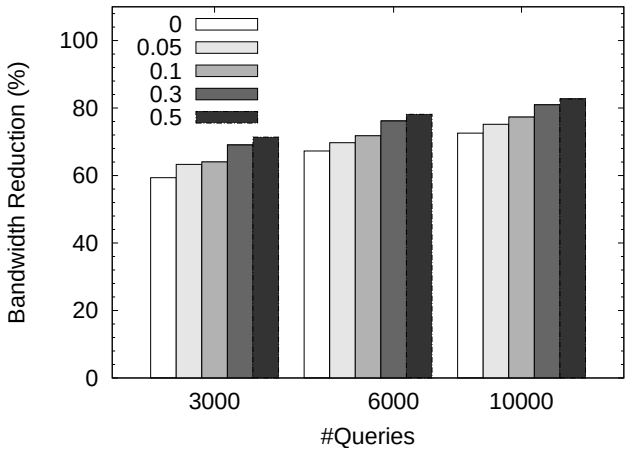
Another experiment is to investigate the sensitivity of our algorithms to the processing overhead threshold $\alpha$. Figure 5.7 shows the result of the experiments. The value of $\alpha$ (in Algorithm 7 under Section 5.5.2.1) is varied from 0 to 0.5. The general trend is, a higher $\alpha$ value provides more opportunities for query merging and hence the outcome bandwidth reduction is higher. Note that the difference is not very significant. The reason could be, for the randomly generated query set in this experimental study, only a small number of merging could incur high processing overhead. We have varied the query generation parameters but could not find a set of parameters that can make this difference more significant. Hence in such query set, a very low $\alpha$ is desirable, as it can significantly reduce the communication cost without incurring much processing overhead. In reality, there could exist some query set that could be more sensitive to the value of $\alpha$. The tuning of an optimal $\alpha$ value could get a better trade-off between communication cost and processing cost.

### 5.6.2 Query Grouping Re-optimization

In this section, we examine the performance of the query re-optimization algorithms. In the experiment, $10,000$ queries are first inserted into the system and then we terminate half of the queries. It is compared with two cases: (1) without



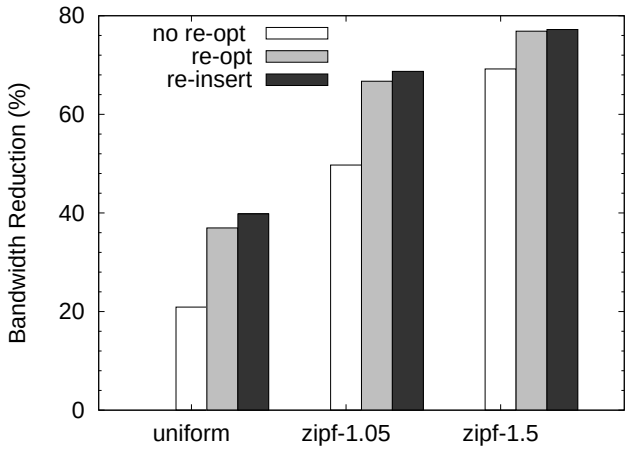

(a) Uniform



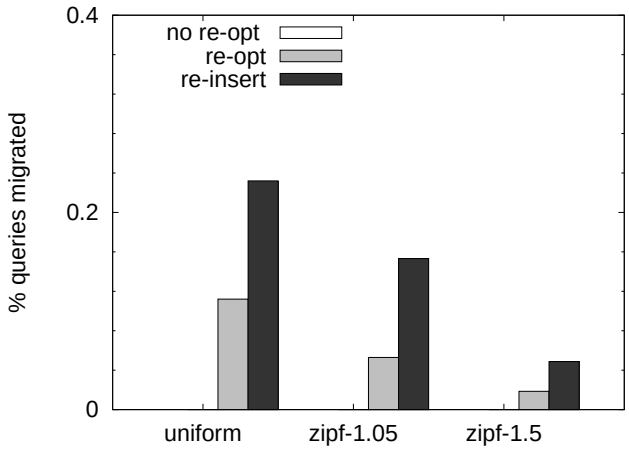

(b) Zipf 1.5

**Figure 5.7.** Sensitivity to $\alpha$

running the re-optimization algorithm ("no re-opt") and (2) running the insertion algorithm on all the remaining queries from scratch ("re-insert").




(a) Bandwidth Reduction




(b) % of queries migrated to a different group

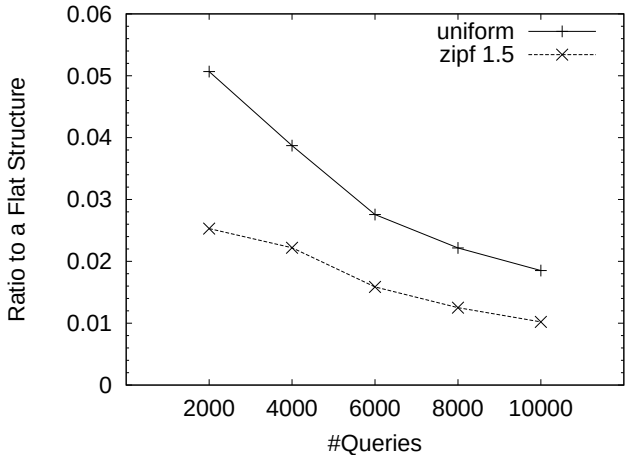
**Figure 5.8.** Query Grouping Re-optimization

Figure 5.8(a) shows the comparison of the bandwidth reduction among the three cases and Figure 5.8(b) presents the percentage of queries that have been migrated to another query group. As one can see, "Re-opt" can achieve much larger bandwidth reduction than the case without re-optimization. Furthermore, "re-insert" works slightly better than "re-opt". This is because "re-insert" can explore a larger solution space than "re-opt". However, as shown in Figure 5.8(b), "re-insert" incurs much more migrations than "re-opt", which would result in higher overhead

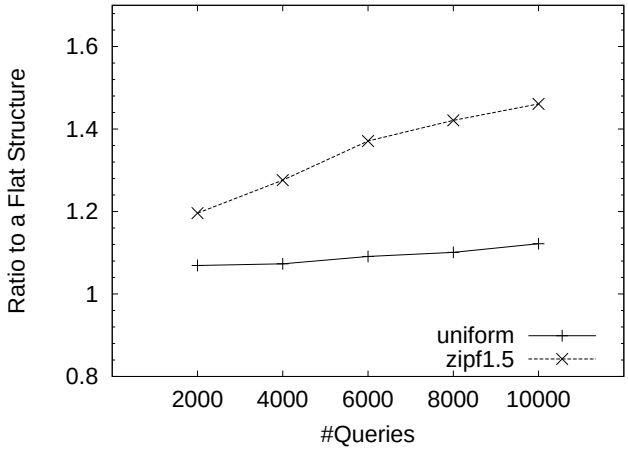
over the network.

Another interesting point that can be observed is, with a more skewed query distribution, less benefit can be achieved by re-optimizing the query grouping. This is because more queries have overlap relationships and hence less queries need to be migrated to another query group. It is also reflected in Figure 5.8(b). Both "re-opt" and "re-insert" migrate less queries for a more skewed query distribution.

### 5.6.3 Efficiency of the Query Tree




(a) Time to Terminate a Query




(b) Time to Insert a Query

**Figure 5.9.** Tree Structure vs. Flat Structure

This experiment is to examine whether the query tree is effective to enhance the query grouping maintenance efficiency. We compare it with a flat structure, where queries within each group are kept in a flat list. We compare the maximum query termination and query insertion time between the two approaches. In the experiment, we first insert a certain number of queries into the system and then try to insert (or remove) a query to (or from) the most popular query group. This is expected to be the maximum insertion (or termination) time. One can see from Figure 5.9(a), the tree approach is much more efficient in query termination than the flat one. That is because it avoids the running of unnecessary query merging while a flat structure cannot exploit this opportunity. The difference is more obvious with a skewed query distribution and a larger query population where query groups tend to have more members. On the other hand, it can be seen from Figure 5.9(b) that the tree approach works slightly worse than the flat structure for query insertion. This is due to the fact that, in the tree approach, a new query has to travel a few levels down the query tree before it is settled in a place within the tree. But this is not necessary in a flat structure. In summary, for systems with similar query termination and insertion rates, the query tree approach is much more efficient than the flat one.

## 5.7 Conclusion

This chapter addresses an important stream processing issue that we faced during our deployments, query result stream delivery, which is often overlooked by existing stream processing systems, and proposes an easy-to-implement yet effective solution. To enhance the system's scalability, DPSS, a scalable and efficient communication paradigm, is employed to deliver query result streams. To fully exploit the message delivery sharing capability of a DPSS, we propose a query grouping and merging approach. To realize this approach, stream query containment theorems are first studied, based on which, query merging and query grouping algorithms are proposed. To deal with the frequent arrival and removal of queries, a multi-tree structure is used to facilitate efficient maintenance of query grouping. Furthermore, adaptive re-optimization algorithms are proposed to continuously adapt the query grouping to the change of the query set and meanwhile keep the query migration overhead to be low. The experiments conducted show that this approach is very efficient and effective , especially with a large number of queries or a skewed query distribution.

# Chapter 6

# Sensor Data Sharing and Visualization

## Overview

In this chapter, we present a sensor network data gathering and visualization infrastructure, comprising of Global Sensor Network (GSN) middleware and Microsoft SensorMap. We provide the use-cases involving the process of monitoring real-world deployments in which scientists can inspect measured data in the form of contour plots overlayed onto a high resolution map and a digital topographic model. Scientists can go back in time virtually to search for interesting events or simply to visualize the temporal dependencies of the data. The system presented is not only interesting and visually enticing for non-expert users but brings substantial benefits to environmental scientists. The easily installed data acquisition component as well as the powerful data sharing and visualization platform opens up new ground in collaborative data gathering and interpretation in the spirit of Web 2.0 applications.

## 6.1 Introduction

In science, the requirement to develop an acquisition, databasing and querying infrastructure for each application adds a layer of expense and a requirement for skills which may not be present within a research team. A generic infrastructure which addresses all of these issues whilst remaining open and flexible enough to allow the scientist to carry out any data processing required, allows costs to be reduced and allows more "science" to take place. Such a generic infrastructure to support environmental science projects is presented here, focusing on environmental monitoring inside the Swiss Experiment (SwissEx)[1] project.

SwissEx is a collaboration of environmental science and technology research projects. These projects cover a range of environmental hazards from sustainable land use, to earthquakes and avalanches. In these projects, there is a large potential for synergy of overlap where they may benefit from sharing data, particularly if experiments can be arranged to take place on common sites. Measurements such as meteorological parameters, soil temperature/conductivity/humidity and hydrological parameters are common across many projects and some projects even have synergies on much larger scales.

Scientific projects have in the past been very isolated, data has seldom been reused within departments, opportunities for data sharing within institutions are missed and collaboration across institutions has generally only taken place when the expertise did not exist in-house. E-science is changing this and Swiss-Experiment is one such e-science project. The SwissEx collaboration encourages data sharing and preservation of knowledge across projects and institutions through the use of a common, state-of-the-art database and data processing infrastructure. The addition of a spatially aware interface, combined with advanced querying tools is aimed at making scientists aware of what data exists and encouraging them to re-use data and/or collaborate on data acquisition. Through the re-use of data across projects, SwissEx aims to bridge the traditional scientific domains, broadening scientific knowledge on the interdisciplinary process interactions with the aim of eventually exploiting these links in large scale sensor deployments to improve environmental hazard forecasting and warning.

The same visual interface, utilizing common tools such as spatial interpolation, is aimed at allowing scientists to easily try out various techniques on their data. Visualization of the results on a map/digital topographic model allows scientists to

[1] http://www.swiss-experiment.ch

better understand the relationship between the 2D results and the real processes that are occurring. This interface can also be used in publishing scientific results in an interactive electronic form, providing greater public interest and hence awareness of environmental research and the processes occurring in the environment around them.

## 6.2 Application Scenarios

The infrastructure is aimed at assisting throughout the life cycle of environmental monitoring. The work presented in this chapter is addressing the following partially fictive application scenarios:

**1. Planning:** *Marc, a renowned hydrologist, is in his office and wishes to review existing datasets that have been captured in the past year at the Le Genepi field deployment of a wireless sensor network in order to plan the deployment for this year's campaign. He wants to better understand the interaction between the rock glacier and the atmosphere, in particular how the wind patterns drive the ventilation of the rock glacier. To do this he retrieves the data on rock and air temperature from the SwissEx data repository and generates a visualization of the temperature differences on the SensorMap[74] interface. He is surprised by the large deviations at some locations and decides to concentrate more stations there. When visualizing the measurements of the rain gauges, he observes that they gave mostly uniform measurements and decides to reduce the number of rain gauge sensors. This year he also received new satellite data on temperature, accessible through Web Services, and uses a visualization of this data to decide on the placement of some stations at a larger scale surrounding the core area of the measurement campaign.*

**2. Monitoring the deployment:** *The sensor stations have been deployed as planned. Over the Web he and his group can at any time observe the current measurements through SensorMap. One evening Marc receives a warning email generated by the underlying data stream processing middleware that the measurements of some sensors are out of the expected ranges. After inspecting some graphs of recent measurements Marc realizes that some sensors are malfunctioning and decides to go to the field next day by helicopter. In the field his team discovers that some of the wind sensors have frozen and they fix the problem. In order to maximize the benefit of the field trip they also visualize model data generated from the*

*real-time measurements through a hand held device and use this information to optimize the placement of some stations. The metadata on the new positions and time of replacement is immediately updated and fed back to the SwissEx data repository so that later models are correctly computed.*

**3. Analyzing the data:** *After the campaign, as more stations have been placed in critical regions Marc can refine the resolution of his energy balance model. Simulating the models that are implemented in Matlab requires several hours of computation on his large workstation. As a result he obtains visualizations of the energy flows that can be overlayed in SensorMap. After looking at the map while sliding back and forth in time, and comparing it to the model results from last year's data, he realizes that he has to revise some assumptions of his models. He annotates the regions exhibiting strange behavior on the map. He will hand over his data and observations to a Postdoc who will be in charge of next years campaign. In the meantime the measurement data, the model data and annotations are archived in the SwissEx repository. Browsing in the repository a PhD student in another research group discovers that she could apply her new risk model for landslides on Marc's energy flow model. Though not perfect data the surrounding communities are highly interested in these risk assessments and so she decides to make them available to selected decision makers through SensorMap.*

## 6.3 System Description

Our system comprises of two components: a sensor middleware component that handles the data acquisition and a data visualization and data sharing component. We will briefly review the fundamental concepts behind these assumptions and then focus on the interactions between them and the challenges that arise in the integration process.

### 6.3.1 Data Acquisition: Global Sensor Network (GSN)

Multiple GSN instances are used to compose the backbone of the acquisition network. A set of wrappers allow live data to be imported into the system. The data streams are processed according to virtual sensor files. We use GSN's virtual sensor concept to create processing chains and repeat live data. The virtual sensors are connected together in order to build the required processing path (cf. Figure

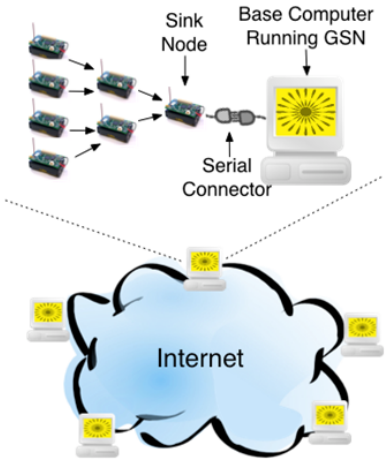



**Figure 6.1.** A standard application of GSN (top) showing GSN gathering data from a sensor network via a serial forwarder that is itself connected to the network's sink node. GSN also offers the functionality of connecting several GSN instances to allow advanced query processing.

6.1). For example, one can imagine an anemometer that would send its data into GSN through a TinyOS wrapper, this data stream could then be sent to an averaging virtual sensor, the output of this virtual sensor could then be split and sent to a database for recording or to a visualization layer for displaying the average measured wind in real time.

GSN obtains the data directly from sensor network deployments and provides the capability of replaying previously measured data, for demonstration or exploration purposes.

### 6.3.2 Data Sharing and Exploration: SenseWeb/SensorMap

Once measurements about the physical world have been collected through GSN, it is advantageous to share the data, allowing multiple projects to share the instrumentation costs and deployment and maintenance effort. Sharing of large volumes of scientific data imposes challenges in data exploration techniques to efficiently discover a subset of data containing phenomena of interest to scientists. To tackle the challenges, an extensible infrastructure for data sharing (called *SenseWeb* [74]) has been designed by Microsoft Research as well as a map-based front-end (called *SensorMap*) to visually explore the shared datasets on geocentric interfaces such as maps and 3D terrain topographies.

The overall infrastructure allows scientists to share their sensor data acquisi-

tion systems over the common, programmable interface supported by SenseWeb, thus making the collected data available for researchers globally. Scientists share the sensors by adding their descriptions to SenseWeb. Such shared sensors then can be discovered based on location, type, or other characteristics. To efficiently support spatial queries of sensor metadata, SenseWeb indexes sensors by using a hierarchical triangular mesh (HTM) indexing scheme [87], which is particularly suitable for geographic queries.

SensorMap further enables scientists to explore the spatio-temporal distributions and correlations of the shared sensor data. SensorMap allows a user to directly specify the area of interest based on a browsable map, by drawing polygons or typing in geonames. Sensors within the specified geographical region are automatically aggregated at an appropriate granularity based on the zoom level of the map. SensorMap directly depicts the sensors on maps as image icons with different color schemes indicating the real-time readings.

Besides the real-time view, a user can explore sensor data streams in historic or spatial views. Via SensorMap, they can select a list of sensors of interest and visualize their temporal distributions in a single comparison chart or in multiple side-by-side time series charts. A third feature of SensorMap is to generate map/image-overlaid contours of selected sensors in view, which can be zoomed or panned together with the underlying map/image.

### 6.3.3 Integration

Figure 6.2 illustrates the architecture of the integrated system. At the bottom of the architecture are multiple GSN instances that acquire data from deployed sensors such as weather stations. Data streams collected by GSN are registered with the SenseWeb infrastructure to share among environmental scientists across multiple deployments. SensorMap accesses the shared data and visualizes their temporal and spatial correlations on top of maps and topological terrains.

SenseWeb is using GeoDB for storing sensor metadata. The metadata includes information about data publisher, sensor name, sensor type, measurement unit, data access API, socket address for retrieving data and access control details. GeoDB uses hierarchical triangular mesh indexing technique [88]. In order to deliver data streams to SenseWeb, data stream producer needs to implement the Datahub API defined in [3]. DataHub acts as a proxy layer between the actual sensors and SenseWeb portal.

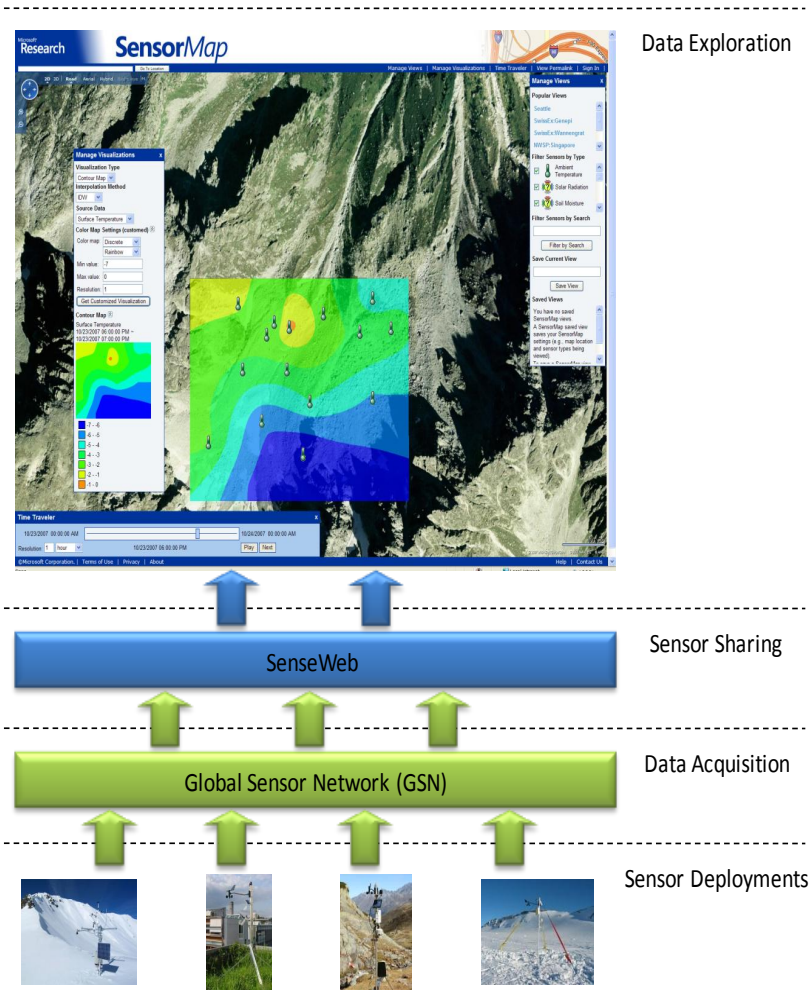



**Figure 6.2.** Architecture of the integrated system

One of the challenges in the integration process is to design a suitable communication protocol between GSN and SensorMap, on which we can pose the following requirements:

- It has to be pull based: many sensors have the capability of producing large data volumes and we want to minimize the communication between the GSN instances and the SensorMap servers. The protocol should be able to retrieve the data on demand when it is requested by a user (e.g., when somebody zooms-in on a specific point on the map).

- It has to offer support for aggregation queries: being able to get aggregated values greatly reduces the communication traffic. SensorMap shows the high

level picture of the data (e.g., aggregated every 6 or 12 hours) and once a user decides that they require a greater temporal resolution of data, SensorMap contacts the responsible GSN instance and asks for the high resolution data set.

- It should be location aware (GPS latitude and longitude)
- Its output should be machine parseable and preferably also human readable.
- It has to be simple enough to get adopted by a sufficiently large user community.

In order to address these requirements, the GeoRSS[2] standard was initially selected. GeoRSS is a geographically coded RSS output generated by GSN. One can specify the aggregation parameters in a simple REST request and retrieve the desired data stream. The output of GeoRSS is in XML format, making it convenient for other softwares to parse the output and produce their own visual interfaces over the GSN instances. This approach is what was used in version 2 of SensorMap.

The major shortfalls associated with the GeoRSS based solution have been the following:

- Extendibility: requests had to be modified to provide new parameters such as measurement types, sensor output rate, etc which are not part of GeoRSS.
- Interface: the solution has a push like interface for users who are interested in having a real-time view of a low traffic sensors (a value every few seconds).

In order to address these two issues, we decided to use a Web Services interface which is flexible enough to handle both set of requirements. Version 3 of SensorMap is designed to use the new interface and GSN was also adopted accordingly. Using the new interface, the parameters are passed as method arguments (e.g., aggregating period, time range, etc) and since one can perform multiple calls over one Web services connection, the servers for SensorMap may maintain an open connection to their desired GSN instances in order to reduce the latency time. The push behavior can also be implemented using a simple call back interface.

## 6.4 GSN/SensorMap In Practise

The SensorMap server is hosted at Microsoft Research in Redmond, both GSN instances are located in Switzerland, at EPFL in Lausanne and at SLF in Davos.

[2] http://www.georss.org/

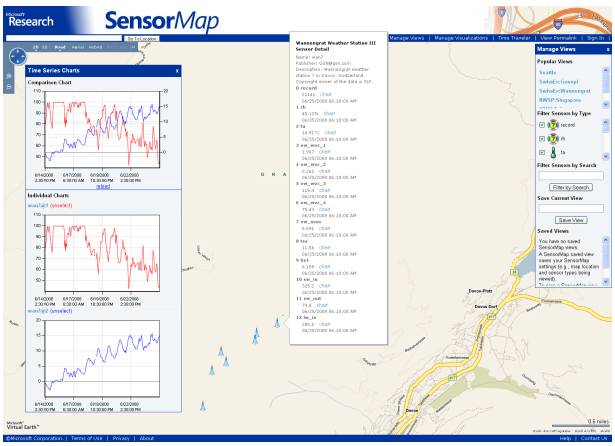



**Figure 6.3.** Real-time and historic views of sensor data streams from the Wannengrat Deployment. Time series charts clearly illustrate the temporal correlations between humidity (red curve) and air temperature (blue curve).

To better demonstrate the applicability of the work, we present two real-world sensor network deployments in the Swiss alps.

**The Le Genepi Deployment:** The "Le Genepi" field deployment of a wireless sensor network was a campaign held between August and September 2007 conducted by the SensorScope team at EPFL. SensorScope [25] provides low cost, wireless and reliable sensor network systems for environmental monitoring to a wide community. It improves present data collection techniques with the latest technology, while meeting the requirements of the environmental scientists. The "Le Genepi" experiment was deployed on a glacier in the canton Valais (Switzerland) close to Martigny. In the three week experiment, 16 weather stations were deployed, measuring *air temperature, surface temperature, air humidity, wind direction, wind speed, precipitation and solar radiation*.

**The Wannengrat Deployment:** Above the town of Davos, Switzerland, at the Wannengrat alpine observatory, seven sensor stations have been installed for studying environmental processes involving snow. The project is maintained by environmental engineers from SLF in Davos. We use this installation as a valuable permanent test scenario for Global Sensor Network (GSN). This scenario is significantly different from the SensorScope scenario, for instance the Genepi glacier experiment(as described above), since the stream data is inserted into the system as a periodic bulk import.

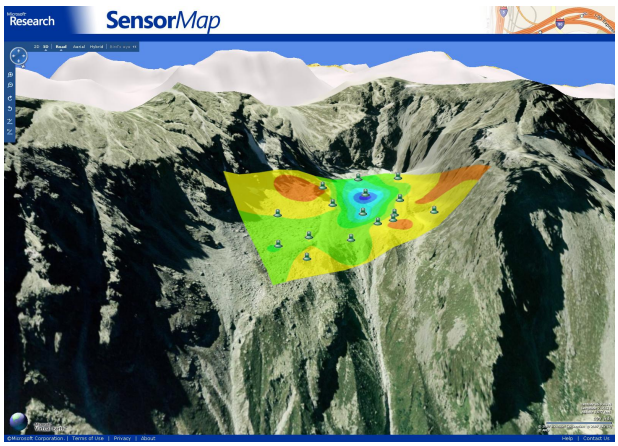



**Figure 6.4.** Spatial visualization of ambient temperature measurements from the Le Genepi deployment. SensorMap generates this type of contour map on request and overlays it on top of 3D high resolution maps. Users clearly see the interaction between ambient temperature and the terrain.

The installation of GSN is at SLF in Davos and is connected to a GSN installation in Lausanne. The real-time and historic views of two aforementioned real-world sensor streams are depicted in Figure 6.3.

Scientists can inspect real-time data as well as virtually go back in time to search for interesting events or analyze the temporal dependencies of the data. The scientists can explore the contour visualizations of snapshot data of any selected time point, which helps in understanding spatial correlations among dispersed measurements. Figure 6.4 illustrates one such contour plot overlaid over 3D terrain maps. Moreover, one is able to request an animation of contour plots for customized time durations and resolutions, through which a preliminary understanding of the spatio-temporal characteristics of selected data streams is obtained.

# Chapter 7

# Conclusion and Future Work

In this thesis, we presented the design and implementation of the Global Sensor Network platform. Full potential of sensor technology will be unleashed through large-scale (up to global scale) data-oriented integration of sensor networks. To realize such a vision of a "Sensor Internet", we presented Global Sensor Network (GSN) middleware. GSN enables fast and flexible deployment and interconnection of sensor networks. Its virtual sensor concept can abstract from arbitrary stream data sources. Virtual sensor's powerful declarative specification and query tools, provides simple and uniform access to heterogeneous technologies. GSN offers zero-programming deployment and data-oriented integration of sensor networks and supports dynamic configuration and adaptation at runtime. Experimental evaluation of GSN demonstrates that the architecture is highly efficient, offers very good performance and throughput even under high loads and scales gracefully in the number of nodes, queries, and query complexity.

Moreover, in order to effectively tackle the performance issues we faced in our deployments in the field of environmental monitoring, we presented a set of algorithms and techniques to deal with the management of window-based continuous queries in stream processing systems for wireless sensor networks. Our proposed algorithms can play a critical role in the large-scale data stream processing systems in which there exist a large number of users registered to hundreds of high rate data streams. Thanks to batch sliding and sliding graph concepts, processing time of continuous queries with sliding predicates can be significantly reduced.

In addition to aforementioned contributions, we presented solutions for efficient query result delivery, which is often overlooked by existing stream processing systems. To fully exploit the message delivery sharing capability of a distributed stream processing engine, we proposed a query grouping and merging approach.

The effectiveness of our proposed query merging and query grouping approach is supported by a number of stream query containment theorems.

In order to deal with the frequent arrival and removal of queries, a multi-tree structure is introduced to facilitate efficient maintenance of query grouping. Furthermore, adaptive re-optimization algorithms are proposed to continuously adapt query grouping to the change of the query set, while keeping query migration overhead low.

In order to present how this thesis would be used in real deployments by the environmental scientists, we present a sensor network data gathering and visualization infrastructure, comprising of GSN and Microsoft SenseWeb. We provide use-cases involving the process of monitoring real-world deployments in which scientists can inspect measured data in the form of contour plots overlayed onto a high resolution map and a digital topographic model.

As future work, there are several interesting problems to explore. If the number of queries is large and data rate is very high, a single processing stream processor system might become overloaded. One interesting solution to this problem could be the use of approximate sliding window management. In this approach, sliding windows which have close slide values could be grouped in the same sliding group to further reduce the processing time. As another possible future direction, in the local-time-based sliding windows, the number of timers was reduced as a result of using a sliding graph. However, it is possible to enhance the batch sliding approach either by merging similar timers which are used for different data streams, or by using a few number of global timers in the system for all data streams.

Also, in the context of efficient query result delivery for stream processing engines, our approach merges all the queries in a query group into one representative query. However, sometimes this would increase the processing cost. Another possible approach would be to generate one or more representative queries for each query group and keep the processing overhead as low as possible. Users then should subscribe to multiple result streams instead of one.

Finally, in a distributed stream processing system, query operators can be allocated to multiple processing servers. It would be interesting to study how the operator allocation and the query grouping and merging will interact with each other.

# Bibliography


[1] Bt-node. Website, 2009. http://www.btnode.ethz.ch/.

[2] Mica2 mote. Website, 2009. http://blog.xbow.com/xblog/mica2_mote/.

[3] Microsoft sensor map. Website, 2009. http://atom.research.microsoft.com/sensewebv3/sensormap.

[4] Sensor data lab. Website, 2009. http://www.sensordatalab.org.

[5] Sensor middleware unit, digital enterprise research institute (deri), ireland. Website, 2009. http://www.deri.ie.

[6] Swiss experiment. Website, 2009. http://www.swissexperiment.ch.

[7] Swiss federal institute for forest, snow and landscape research wsl. Website, 2009. http://www.slf.ch.

[8] Tmote. Website, 2009. http://www.moteiv.com/.

[9] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[10] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[11] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *MDM*, pages 198–205, 2007.

[12] Charu C. Aggarwal. An intuitive framework for understanding changes in evolving data streams. In *ICDE*, page 261. IEEE Computer Society, 2002.

[13] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *VLDB*, pages 81–92, 2003.

[14] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for projected clustering of high dimensional data streams. In *VLDB*, pages 852–863. Morgan Kaufmann, 2004.

[15] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for on-demand classification of evolving data streams. *IEEE Trans. Knowl. Data Eng.*, 18(5):577–589, 2006.

[16] Marcos Kawazoe Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar Deepak Chandra. Matching events in a content-based subscription system. In *PODC*, pages 53–61, 1999.

[17] Yanif Ahmad and Ugur Çetintemel. Networked query processing for distributed stream-based applications. In *VLDB*, pages 456–467. Morgan Kaufmann, 2004.

[18] Noga Alon, Phillip B. Gibbons, Yossi Matias, and Mario Szegedy. Tracking join and self-join sizes in limited storage. In *PODS*, pages 10–20. ACM Press, 1999.

[19] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom. *Data-Stream Management: Processing High-Speed Data Streams*, chapter STREAM: The Stanford Data Stream Management System. Springer, 2006.

[20] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager. In *SIGMOD Conference*, page 665, 2003.

[21] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[22] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *PODS*, pages 286–296. ACM, 2004.

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

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

[25] Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, Martin Vetterli, Olivier Couach, and Marc Parlange. Sensorscope: Out-of-the-box environmental monitoring. In *IPSN*, pages 332–343, 2008.

[26] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *PODS*, pages 149–158. ACM Press, 1998.

[27] Feng Cao, Martin Ester, Weining Qian, and Aoying Zhou. Density-based clustering over an evolving data stream with noise. In *SDM*. SIAM, 2006.

[28] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[29] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90. ACM, 1977.

[30] Badrish Chandramouli, Junyi Xie, and Jun Yang. On the database/network interface in large-scale publish/subscribe systems. In *SIGMOD Conference*, pages 587–598. SIGMOD Conference, 2006.

[31] Badrish Chandramouli and Jun Yang. End-to-end support for joins in large-scale publish/subscribe systems. *PVLDB*, 1(1):434–450, 2008.

[32] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[33] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *CIDR*, 2003.

[34] Moses Charikar, Liadan O'Callaghan, and Rina Panigrahy. Better streaming algorithms for clustering problems. In *STOC*, pages 30–39. ACM, 2003.

[35] Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. ACM, 2006.

[36] Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *KDD*, pages 133–142. ACM, 2007.

[37] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable Distributed Stream Processing. In *CIDR*, 2003.

[38] Yun Chi, Haixun Wang, and Philip S. Yu. Loadstar: Load shedding in data stream mining. In *VLDB*, pages 1303–1305. ACM, 2005.

[39] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Containment of aggregate queries. In *ICDT*, volume 2572 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2003.

[40] Graham Cormode and Minos N. Garofalakis. Sketching probabilistic data streams. In *SIGMOD Conference*, pages 281–292. ACM, 2007.

[41] Graham Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SDM*, 2005.

[42] Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD Conference*, pages 647–651, 2003.

[43] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *SIGMOD Conference*, pages 40–51. ACM, 2003.

[44] Luping Ding and Elke A. Rundensteiner. Evaluating window joins over punctuated streams. In *CIKM*, pages 98–107. ACM, 2004.

[45] Alin Dobra, Minos N. Garofalakis, Johannes Gehrke, and Rajeev Rastogi. Processing complex aggregate queries over data streams. In *SIGMOD Conference*, pages 61–72. ACM, 2002.

[46] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *KDD*, pages 71–80, 2000.

[47] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN*, pages 455–462. IEEE Computer Society, 2004.

[48] M. Franklin, S. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong. Design Considerations for High Fan-in Systems: The HiFi Approach. In *CIDR*, 2005.

[49] David Gay, Philip Levis, and David E. Culler. Software design patterns for tinyos. *ACM Trans. Embedded Comput. Syst.*, 6(4), 2007.

[50] Bugra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. Grubjoin: An adaptive, multi-way, windowed stream join with time correlation-aware cpu load shedding. *IEEE Trans. Knowl. Data Eng.*, 19(10):1363–1380, 2007.

[51] Johannes Gehrke, Flip Korn, and Divesh Srivastava. On computing correlated aggregates over continual data streams. In *SIGMOD Conference*, pages 13–24, 2001.

[52] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An Architecture for a World-Wide Sensor Web. *IEEE Pervasive Computing*, 2(4), 2003.

[53] Phillip B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, pages 541–550. Morgan Kaufmann, 2001.

[54] Phillip B. Gibbons and Yossi Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD Conference*, pages 331–342. ACM Press, 1998.

[55] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

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

[57] A. J. G. Gray and W. Nutt. A Data Stream Publish/Subscribe Architecture with Self-adapting Queries. In *International Conference on Cooperative Information Systems (CoopIS)*, 2005.

[58] Sudipto Guha. Tight results for clustering and summarizing data streams. In *ICDT*, volume 361 of *ACM International Conference Proceeding Series*, pages 268–275. ACM, 2009.

[59] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[60] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *KDD*, pages 97–106, 2001.

[61] Yannis E. Ioannidis and Eugene Wong. Query optimization by simulated annealing. In *SIGMOD Conference*, pages 9–22. ACM Press, 1987.

[62] Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting change in data streams. In *VLDB*, pages 180–191. Morgan Kaufmann, 2004.

[63] Phokion G. Kolaitis, David L. Martin, and Madhukar N. Thakur. On the complexity of the containment problem for conjunctive queries with built-in predicates. In *PODS*, pages 197–204. ACM Press, 1998.

[64] Alberto Lerner and Dennis Shasha. Aquery: Query language for ordered data, optimization techniques, and experiments. In *VLDB*, pages 345–356, 2003.

[65] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM.

[66] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record*, 34(1):39–44, 2005.

[67] Liqian Luo, Aman Kansal, Suman Nath, and Feng Zhao. Sharing and exploring sensor streams over geocentric interfaces. In *GIS*, page 3. ACM, 2008.

[68] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[69] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[70] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, pages 49–60. ACM, 2002.

[71] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB*, pages 346–357. Morgan Kaufmann, 2002.

[72] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys*, pages 250–262. ACM, 2004.

[73] Suman Nath, Jie Liu, Jessica Miller, Feng Zhao, and André Santanche. Sensormap: a web site for sensors world-wide. In *SenSys*, pages 373–374. ACM, 2006.

[74] Suman Nath, Jie Liu, and Feng Zhao. Sensormap for wide-area sensor webs. *IEEE Computer*, 40(7):90–93, 2008.

[75] Werner Nutt, Yehoshua Sagiv, and Sara Shurin. Deciding equivalences among aggregate queries. In *PODS*, pages 214–223. ACM Press, 1998.

[76] Liadan O'Callaghan, Adam Meyerson, Rajeev Motwani, Nina Mishra, and Sudipto Guha. Streaming-data algorithms for high-quality clustering. In *ICDE*. IEEE Computer Society, 2002.

[77] Kivanc M. Ozonat. An information-theoretic approach to detecting performance anomalies and changes for large-scale distributed web services. In *DSN*, pages 522–531. IEEE Computer Society, 2008.

[78] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, page 49. IEEE Computer Society, 2006.

[79] Sean Rooney, Daniel Bauer, and Paolo Scotton. Techniques for Integrating Sensors into the Enterprise Network. *IEEE eTransactions on Network and Service Management*, 2(1), 2006.

[80] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[81] Ali Salehi. Nextick. Website, 2008. http://nextick.org.

[82] M. Sgroi, A. Wolisz, A. Sangiovanni-Vincentelli, and J. M. Rabaey. A service-based universal application interface for ad hoc wireless sensor and actuator networks. In *Ambient Intelligence*. Springer Verlag, 2005.

[83] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical Report TR-21-04, Harvard University, EECS, 2004. http://www.eecs.harvard.edu/~syrah/hourglass/papers/tr2104.pdf.

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

[85] Utkarsh Srivastava and Jennifer Widom. Flexible time management in data stream systems. In *PODS*, pages 263–274, 2004.

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

[87] Alex Szalay, Jim Gray, Gyorgy Fekete, Peter Kunszt, Peter Kukol, and Ani Thakar. Indexing the sphere with the hierarchical triangular mesh. In *MSR-TR-2005-123*, September 2005.

[88] Alexander S. Szalay, Jim Gray, George Fekete, Peter Z. Kunszt, Peter Kukol, and Ani Thakar. Indexing the sphere with the hierarchical triangular mesh. *CoRR*, abs/cs/0701164, 2007.

[89] 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.

[90] Yan Xia, Keith E. Bettinger, Lin Shen, and Allan L. Reiss. Automatic segmentation of the caudate nucleus from human brain mr images. *IEEE Trans. Med. Imaging*, 26(4):509–517, 2007.

[91] Ying Xing, Stanley B. Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802. IEEE Computer Society, 2005.

[92] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.

[93] Yong Yao and Johannes Gehrke. Query Processing in Sensor Networks. In *CIDR*, 2003.

[94] Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Cetintemel, Magdalena Balazinska, and Hari Balakrishnan. The Aurora and Medusa Projects. *Bulletin of the Technical Committe on Data Engineering, IEEE Computer Society*, 2003.

[95] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *OTM Conferences (1)*, volume 4275 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 2006.

[96] Yongluan Zhou, Kian-Lee Tan, and Feng Yu. Leveraging distributed publish/-subscribe systems for scalable stream query processing. In *BIRTE*, volume 4365 of *Lecture Notes in Computer Science*, pages 20–33. Springer, 2006.

| | |
|---|---|
| Name: | Ali SALEHI |
| Nationality: | Iran |
| Languages: | English [fluent, C1] |
| Email: | ali.salehi@epfl.ch |
| Birth: | 1982 |
| Citizenship | Iran |


***Objective***: challenging position involving development, design, architecture and leadership.

## *Profile and Management Skills*

- Lead developer and manager in more than half a dozen enterprise software projects.
- Managed all phases of complex software projects including requirements, architecture, development, testing and releasing.
- Coordinated and supervised development team of 5-10 engineers on large software projects.
- Strong background in Stream Processing, Data Management Systems and Software Architectures.

## *Education*

2004 – 2010 **PhD** of **Computer Science** in **Stream Data Management Systems**, *Ecole Polytechnique Fédérale de Lausanne (**EPFL**)*, 2010, Switzerland.

2000 – 2004 **Software Engineering**, Bachelor degree, *Isfahan University*.

## *Software Experiences*

### ***Lead Software Architect***, GSN Project, *http://gsn.sf.net*, EPFL, 2004-2010

- Stream data management and data integration infrastructure for wireless sensor networks.
- ***Coordinated and supervised development team of 10 engineers.***
- ***Project planning*** and ***prioritized*** the deliverables to the end users.
- Coordinated with industrial users, ***Microsoft Research Redmond***, Digital Enterprise Research Institute in Ireland and ***Swiss Federal Snow Research. Swisscom, EDF, …***
- Used as the ***core technology*** in over ***10 EU/Swiss funded*** research projects.
- Real-time patient monitoring demo, built using GSN, to be aired on Nov 2008 in RTE (Irish national TV; Investigators Program). Live radio interview of the project on Dublin City FM (17th July, 2008) and an article on the Irish Times (15th July, 2008) about the GSN platform and Patient Monitoring use case.

### ***Software Architect***, NexTick Project, *http://nextick.org*, EPFL, 2007-2010

- Tracks, analyzes and visualizes stock ticks from **NYSE** and **NASDAQ** in real time.
- Helps investors spot attractive securities, used by over ***500 investors***.
- Provides ***Technical analysis*** and candle stick pattern identification in real time.
- ***Downloaded over 100 times*** in the ***first week*** of its release.

## *PhD Research, EPFL*

Multiple publications in international conferences and international workshops including: VLDB, ICDE, MDM

## *Background Knowledge*

**Technical***:* C++, Ruby on Rails, Java, Scala, AJAX, Enterprise Java Beans (EJB), Web services, Struts, JSP,Swing, Linux, Cygwin, MySQL, Microsoft SQL Server, JavaScript, J2EE,JUnit, JQuery, Prototype.