Synopsis join query processing in sensor networks

Yu, Hai

2007

Yu, H. (2007). Synopsis join query processing in sensor networks. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/35742>

<https://doi.org/10.32657/10356/35742>

Synopsis Join Query Processing in Sensor Networks

Yu Hai

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Master of Engineering

2007

Abstract

Recent advances in hardware and wireless technologies have led to the developments of small, wireless, battery-powered sensors with limited processing power. Sensor networks consisting of hundreds or even thousands of such sensors can be deployed to monitor physical environments with little human intervention. Various constraints of sensor network, such as unreliable wireless connections, limited bandwidth and finite energy, have posed the needs for flexible, reliable, and cost-effective techniques for processing data collected by the sensors. A sensor network can be treated as a database. Collecting and processing the data collected and stored in the sensor network can be achieved using queries. In this thesis, we focus on join queries in sensor networks, which are used to integrate data from the raw sensor readings. We present several possible approaches for evaluating simple equi-join queries in sensor networks, and analyze their performances. Based on the analysis, we propose a synopsis join strategy for evaluating simple equi-join queries, assuming rectangular data source regions are known. The synopsis join strategy comprises three stages: query dissemination, preliminary join, and final join. We make use of synopses in order to reduce the communication cost. Experiments to verify our cost analysis have been conducted and it has been shown that the synopsis join strategy outperforms other strategies when join selectivity is small. We have further generalised the join strategy to work for arbitrary data source regions.

Acknowledgement

I wish to express my deepest gratitude to my supervisor, Professor Lim Ee-Peng. I am thankful to him for providing me the opportunity for research. He has contributed endless hours of advice and guidance regarding my work. His guidance and encouragement have made my research a fulfilling and enriching experience. He spent much time discussing with me even my most absurd ideas. His tireless editorial effort vastly improved the quality of the report. I could have never completed the report without his valuable ideas and suggestions. He has also helped me get the financial support for a conference trip.

Special thanks to Professor Zhang Jun and Professor Tang Xue Yan, for generating ideas and sharing their views of research. Debates with them have generated new ideas for my research directions. My presentation skills have also been improved through weekly discussions with them.

Yao Yuxia and Zhao Qiankun, two students at CAIS, deserve my thanks too for providing help on my report writing. Yuxia has shared her ideas of sensor network research with me. Discussions with her have cleared many of my doubts of research.

I would like to thanks the Director, Professor Ng Wee Keong, for making CAIS an efficient and coveted place to work in. I am especially grateful to the laboratory technicians, Mdm Tan Lay Choo and Mr Lai Chee Keong for their efforts in administration and technical support, and generously expanding my disk quotas.

Finally, I wish to thank my family to whom I owe the greatest debt. I could not have come this far without their love and support. Their sincerity, uprightness and sense of purpose have greatly influenced my life abroad and will continue to do so in the future.

Contents

Abstract	i
Acknowledgement	iii
1 Introduction	1
1.1 Sensor Networks	1
1.1.1 A Sensor Network Architecture	1
1.1.2 Applications of Sensor Networks	3
1.1.3 Characteristics of Sensor Networks	3
1.2 The Synopsis Join Strategy for In-network Join Query Processing	5
1.3 Contributions	6
1.4 Thesis Organization	7
2 Literature Review	9
2.1 Routing Protocols	9
2.1.1 Proactive Routing Protocols	10
2.1.2 Reactive Routing Protocols	12
2.1.3 Hybrid Protocols	14
2.1.4 Geographical Routing Protocols	15
2.2 Storage Schemes	17
2.2.1 External Storage	18
2.2.2 Local Storage	19
2.2.3 Data Centric Storage	22
2.3 Query Processing	24

2.3.1	Overview	24
2.3.2	Aggregate Queries	28
2.3.3	Approximate Queries	33
2.3.4	Range Queries	35
2.3.5	Join Queries	38
2.4	Summary	44
3	In-Network Join Strategies	47
3.1	Motivation	47
3.1.1	Ammunition Supply Monitoring System	50
3.1.2	Traffic Monitoring System	52
3.2	In-Network Binary Equi-Join Processing	53
3.3	General Strategies for Evaluating BEJ Queries	57
3.3.1	Naive Join	59
3.3.2	Sequential Join	59
3.3.3	Centroid Join	60
3.4	Synopsis Join Strategy	61
3.4.1	Query Dissemination	62
3.4.2	Preliminary Join	62
3.4.3	Notification Transmission	68
3.4.4	Final Join	69
3.5	Experiments	70
3.5.1	Experiment Setup	70
3.5.2	Performance Results	71
3.6	Summary	76
4	An Implementation of Synopsis Join Strategy	77
4.1	Query Dissemination	77
4.1.1	Synopsis Generation	78
4.2	Preliminary Join	79
4.3	Final Join	82
4.4	Experiments	83

CONTENTS	vii
4.4.1 Impact of Join Selectivity	83
4.4.2 Impact of Histogram Accuracy	86
4.4.3 Hotspots	87
4.4.4 Impact of Message Losses	89
5 Generalised Synopsis Join Processing	91
5.1 Overview of Generalised Synopsis Join	91
5.2 Revised Preliminary Join	94
5.2.1 Cache Updating Rules	95
5.2.2 Hashing	98
5.3 Example	99
5.4 Proof of Correctness	100
5.5 Experiment Results	102
5.5.1 Experiment Setup	103
5.5.2 Impact of Selectivity	103
5.5.3 Impact of Source Region Size	105
5.6 Summary	105
6 Conclusion and Future Work	113
6.1 Conclusion	113
6.2 Future Work	114
Appendices	116
A Proof of Equation 3.4	117
B Publications	119

List of Tables

1.1	Hardware Specification of Mica Motes [Mad03]	4
2.1	A Taxonomy of Queries in Sensor Networks	28
2.2	Energy requirements of Crossbow MTS400 Sensorboard [Mad03]	34
3.1	Symbols for Performance Comparison	58
3.2	Symbols for Synopsis Join Strategy	65
4.1	Summary of parameters	83

List of Figures

1.1	A Sensor Network System	2
2.1	GPSR forwarding in Greedy mode	16
2.2	GPSR forwarding in Perimeter mode	17
2.3	A Direct Diffusion Example [IGE00]	20
2.4	Three Stages of Query Processing in Sensor Networks	25
2.5	A Semantic Routing Tree [MFHH03]	32
2.6	Voltage and temperature readings from two sensors showing correlations in a sensor network [DGM ⁺ 04]	35
2.7	An Illustration of DIFS Hierarchy	37
2.8	A Network with Zones and a Zone Tree	38
2.9	Operator placements [BB03]	40
2.10	Neighbor exploration [BB03]	41
2.11	Path Join in Sensor Networks [CG05]	42
2.12	Optimal join-region for distribute-broadcast join [CG05]	42
2.13	Sensor groups in REED [AML05]	44
3.1	Events in Sensor Networks	49
3.2	Schemas of Two Relations in Ammunitions Supply Monitoring System . . .	50
3.3	Schemas of Two Relations in Traffic Monitoring System	52
3.4	A Vehicle Surveillance System	55
3.5	Naive Approach	60
3.6	Sequential Approach	60
3.7	Centroid Approach	61

3.8	Query dissemination	63
3.9	Synopsis Join Strategy	64
3.10	An example of synopsis	64
3.11	An example of Preliminary Join	66
3.12	An example of a merge function	67
3.13	Impact of Selectivity	72
3.14	Impact of Network Density	73
3.15	Impact of Memory Capacity	74
3.16	Impact of Synopsis Size	75
3.17	Optimizing Semi-Join Approach with Histograms	75
4.1	Routing tree construction	77
4.2	Preliminary join sensors selection	80
4.3	Cost vs. join selectivity (0.0001 - 0.001)	84
4.4	Cost vs. join selectivity (0.001 - 0.01)	84
4.5	Detailed cost breakdown for SNJ (0.0001 - 0.001)	85
4.6	Detailed cost breakdown for SNJ (0.001 - 0.01)	85
4.7	Impact of histogram accuracy	87
4.8	Hotspots	88
4.9	Impact of message losses	88
5.1	Source Regions of arbitrary shapes and locations	92
5.2	Overview of Generalised SNJ strategy	93
5.3	Hashing Algorithm in Revised Preliminary Join	100
5.4	Determining A_1 , A_2 , and A_3	101
5.5	Ranked Cache	102
5.6	Overlapping region 0%	104
5.7	Overlapping region 50%	106
5.8	Overlapping region 100%	107
5.9	All experiment results for overlapping regions	108
5.10	Detailed cost breakdown for overlapping region 0%	109
5.11	Region all	109

LIST OF FIGURES xiii

5.12 Detailed cost breakdown for different sized source regions 110

Chapter 1

Introduction

The emergence of wireless and digital technologies has enabled the development of tiny, low-power, wireless devices with limited computation and storage capabilities [GW00]. *Sensor nodes* are one kind of such devices capable of sensing the surroundings, and processing the data collected. These wireless sensor nodes can work collaboratively to form a *sensor network*. Therefore a sensor network can be deployed in an environment where the sensors are distributed and they collaboratively perform sensing and monitoring tasks with little human intervention. We briefly overview sensor networks and address the characteristics of sensor networks in this chapter. We present contributions of our research and describe the organization of the thesis at the end of this chapter.

1.1 Sensor Networks

Because of their flexibility and cost-effectiveness, sensor networks have been deployed in many detection, surveillance, and measurement applications. However, the unreliable wireless connections among the sensors and the limited battery power have contributed to difficulties in designing accurate and reliable sensor networks. We begin with a sensor network model and present various factors that affect the performance of sensor networks.

1.1.1 A Sensor Network Architecture

Simply speaking, a sensor network consists of a large amount of sensors connected wirelessly. The wireless connections among the sensors are short-ranged and bandwidth-limited.

Figure 1.1: A Sensor Network System

Figure 1.1 shows a typical setup of a sensor network.

In this model, a sensor network is composed of a large number of sensors. Sensors in the sensor network are responsible for monitoring the area where the sensor network is deployed, and collect data such as temperature or humidity readings. These battery-powered sensors are wirelessly connected and installed at fixed locations or on moving objects (e.g., vehicles, people, etc.). As the communication range of the sensors is limited, multi-hop routing protocols are employed for communication among the sensors. Users control the sensor network at a centralized server which is connected to the sensor network via basestations. In general, users specify sensing tasks or queries at the server. Based on knowledge of the sensor network, e.g., sensors distribution or types of sensors, the server can determine sensing or query plans, which are subsequently directed to the basestations. The basestations disseminate the plans to all sensor nodes in the sensor network. According to the plans, sensors collaboratively perform sensing, data collection, data exchange, and data processing operations, and so on. Relevant data are transmitted back to the server via basestations, and are reported to the users. In addition, with wireless communication devices such as handphones or laptops, mobile users moving in the area covered by the sensor network are able to communicate with nearby sensors, and inject sensing tasks or queries directly to the sensor network.

1.1. SENSOR NETWORKS

3

It is worth noting that that from the application perspective, the sensor network is modeled as a sensor database consisting of data collected from the sensors represented as relations [BGS01, Mad03], whether users are communicating with a centralized server, or a particular sensor. Applications control and interact with the sensor network by specifying queries to the sensor database. The operational details of the sensor network such as the maintenance of sensors, query dissemination, data collection, etc., are all kept at the lower system levels and are transparent to users. This level of abstraction provides a simple interface for applications to interact with the sensor network, and various optimization techniques can be applied to enhance the performance of the network in terms of both cost and accuracy, without programming efforts within the applications.

1.1.2 Applications of Sensor Networks

Sensor networks have been adopted in various scientific and commercial applications using sensors capable of sensing temperature, humidity, pressure, etc.. In [ASSC02], Akyildiz *et al.* have classified sensor network applications into military, environmental, health, home and other commercial applications. One example that demonstrates the effectiveness of sensor networks is the habitat monitoring application deployed in Great Duck Island [MCP⁺02, GDI02], in which wireless sensors are deployed to monitor the microclimates that the Leach's Storm Petrel lives in. In such systems, sensor networks facilitate non-intrusive and non-disruptive monitoring of sensitive wildlife, with little deployment and maintenance costs [GDI02]. Many other wireless sensor network systems have been developed as well, some of which are listed in [ASSC02].

1.1.3 Characteristics of Sensor Networks

Due to the limited capabilities of the sensors, sensor networks pose many technical challenges [GW00, ASSC02, VdSJJ⁺03]. We describe some of the constraints that have to be considered in designing a sensor network system.

Energy Sensors are battery-powered. Therefore energy consumption is the most important factor that affects the lifetime of the sensor network. A sensor may only operate for a few days with a pair of AA batteries if the energy is used unwisely, whereas with careful

Feature	Mica	Mica2
CPU Type	Atmel 128	Atmel 128
CPU Frequency	4 MHz	8 MHz
CPU Data Memory	4 KB	4 KB
CPU Program Memory	128 KB	128 KB
Radio Type	RFM TR1000	ChipCon
Radio Frequency	917 MHz	917 MHz, 433 MHz
Radio Throughput	40 KHz	39.2 KHz
Flash Type	Atmel AT45DB041	Atmel AT45DB041
Flash Capacity	512 KB	512 KB
Flash Write Time	10 ms/256 bytes	10 ms/256 bytes
Flash Read Time	10 μ s/256 bytes	10 μ s/256 bytes

Table 1.1: Hardware Specification of Mica Motes [Mad03]

management of the energy, its lifetime can be extended to 6 months.

The power consumption is dominated by message transmissions [Rag02]. The energy required for message transmission can be of a few orders of magnitude than that used for sensing and data processing. Therefore the goal shared by sensor network applications is to reduce message transmissions among sensors so as to conserve energy and hence to prolong the lifetime of the sensors.

Computation Power Sensors typically possess rather limited computation power. Due to the constraints on the energy resource, the frequency of the microprocessor on sensors is in the range of a few mega hertz. Moreover, the available memory size is often small, compared to that in modern personal computer, or even Personal Digital Assistant (PDA). Table 1.1 shows the hardware specifications of Mica motes [Cro05, Mad03] developed at UC Berkeley. The slow microprocessor speed, and the small sizes of the physical memories limit the complexity of the algorithms that can be implemented for the sensors.

Connection The wireless connection among the sensors are short-ranged, bandwidth-limited, and unreliable. For example the Mica motes used in TinyOS [HSW⁺00] have a data transmission rate in the range 40 ~ 76.8Kbps only. Packet congestions and losses are common in such bandwidth-limited communication channels. In addition, the short-range wireless connections among the sensors can become unavailable in the presence of spatial obstacles.

1.2. THE SYNOPSIS JOIN STRATEGY FOR IN-NETWORK JOIN QUERY PROCESSING⁵

Accuracy The information provided by the sensors can be inaccurate for a few reasons. Firstly, sensor data is subject to noises such as external environmental noises or random hardware noises [EN03]. Secondly, sensors acquire data in a discrete manner. Therefore the data acquired are only samples of the physical world phenomena, which are incomplete and provide only approximations to the real world [DGM⁺04]. Thirdly, sensors may become faulty or even fail due to changes in the surrounding conditions or running out of battery power. The faulty or failed sensors may introduce errors in the sensor readings. Finally, due to the unreliable wireless connections, packet losses and corruptions are inevitable, degrading the accuracy further. All these factors have to be taken into account in designing techniques for collecting and processing data in sensor networks.

1.2 The Synopsis Join Strategy for In-network Join Query Processing

The goal of our research is to design and implement efficient algorithms for query processing in sensor networks. In particular, we focus on join query processing in sensor networks. The objective is to support simple equi-join queries with as little communication cost as possible.

Our work is motivated by the fact that queries are useful for obtaining information from sensor networks. Since it is costly to transmit all data from the sensors to the query sink, we focus on *in-network query processing*, in which data are processed progressively as they flow through the network, so that irrelevant and redundant data can be eliminated in the early stage of the process to reduce communication cost.

Join queries are of particular interest to us. As data are distributed among the sensors, it is desirable to enable users to correlate relevant sensor data in an energy-efficient manner. Join queries provide a possible means to collect data from a set of sensors, and produce combined information based on the conditions specified in queries. Despite the usefulness of join queries, processing sensor network join queries could incur high communication cost due to transmission of large volumes of data.

We believe that in-network query processing techniques can help reduce the communication cost for evaluating join queries. In particular, we have developed a synopsis join strategy that prunes irrelevant data in the early stage of join processing, dramatically reduc-

ing the communication cost. Moreover, the synopsis join strategy places few restrictions on sensors knowing the network topology and data statistics. The distributed algorithms used in synopsis join enable practical and simple implementation of joins. Further optimization and extension are also possible with the synopsis join strategy.

In this thesis, we first look at various approaches for processing join queries in sensor networks. We perform cost analysis comparisons on each join approach. Furthermore, we investigate various techniques for distributed join processing in traditional distributed databases, and identify the key differences that render traditional query processing techniques inapplicable for joins in sensor networks. Based on the pros and cons of various techniques, we propose a synopsis join strategy for processing sensor network join queries in a communication-effective manner. We evaluate the effectiveness of our design through extensive experiments. Simulations are carried out to prove that our design is a practical solution to the join processing problem by addressing various practical issues.

1.3 Contributions

The major contributions of the thesis are listed below:

1. We provide a thorough discussion on possible approaches to join processing in sensor networks. We analyze the performance of each approach and compare their pros and cons.
2. We show that the synopsis join strategy is effective in reducing communication cost and is a feasible and practical solution to sensor network join queries. We verify the effectiveness of synopsis join through cost analysis. We also carry out experiments to validate our analytical results.
3. We show that supporting more generalised join queries is possible by extending the synopsis join strategy. We provide an implementation of the generalised synopsis join to handle more practical joins.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 provides a thorough survey on various aspects related to query processing in sensor networks, including routing protocols, storage schemes, and query processing techniques. We describe our proposed synopsis join strategy in Chapter 3. We focus on the design considerations and present the initial cost analysis and experiment results to verify the effectiveness of our design. We also discuss various issues related to practical implementation of the synopsis join strategy on a network simulator. In Chapter 5, we design an extension to synopsis join strategy to support more general join queries. Finally we conclude the thesis in Chapter 6 with remarks on future works.

Chapter 2

Literature Review

In this chapter, we review the sensor network literature, particularly in perspectives related to query processing. We start with routing protocols that data transmission in sensor networks relies on. We also present various storage schemes in sensor networks for storing sensor readings. We focus on query processing techniques in the last section and provide a taxonomy of various kinds of queries.

2.1 Routing Protocols

In wireless sensor networks, ad-hoc routing protocols are required for delivering data from one node to another. Traditional client-server protocols are infeasible because nodes may be constantly moving and it is hard to keep a connection between any two distant nodes. Therefore multi-hop routing protocols are required in wireless networks to deliver packets in a hop-by-hop manner. The challenge for designing a multi-hop routing protocol is to reduce control traffic overhead while maintaining the connectivity of the network

Wireless routing protocols can be *topology-based* or *geographic-based*. Topology-based protocols for ad-hoc network can be classified into three categories, *proactive*, *reactive*, and *hybrid*. Proactive protocols maintain routing information in each node and update the routing information whenever the network topology changes. In other words, when a message is to be sent to a node, the route can be immediately determined. However, in wireless networks where the nodes may be continuously moving and the network topology changes frequently, proactive routing protocols consume a large portion of the network

bandwidth to keep the routing information up to date, which is undesirable in bandwidth constrained wireless networks. Some examples of proactive protocols include DSDV [PB94], OLSR [JMQ98], and WRP [MGLA96].

Reactive routing protocol, on the other hand, only determines the routes when needed. When a packet is to be forwarded, the protocol tries to find a route first, which induces a delay before data transmission. The discovery of the routes is normally achieved by sending queries to the network, and determining the suitable routes based on the replies from other nodes in the network. As the route information is determined in an on-demand manner, less communication overhead is required for maintaining the routes in reactive routing protocols. The drawback of this approach is that the queries are usually flooded into the network, resulting in considerable volume of control traffic. In addition, the on-demand discovery of routes introduces delay prior to data transmission. TORA [PC97], DSR [JM96], AODV [PR98, PR99] all fall under this category.

Hybrid protocols like ZRP [Haa97] are combinations of the proactive and reactive protocols, which carry the advantages of both types of protocols, while trying to avoid their disadvantages.

Topology-based routing protocols are not ideal for networks of moving nodes whose topology may change dramatically with the movements of the nodes. To overcome this problem, with the assumption that wireless nodes are equipped with location devices such as GPS (Global Positioning System), geographical routing protocols were proposed to handle changing network topology by utilizing location information of the nodes.

2.1.1 Proactive Routing Protocols

We introduce two well-known proactive routing protocols in this section, the Destination-Sequenced Distance-Vector (DSDV) routing protocol [PB94] and the Wireless Routing Protocol (WRP) [MGLA96].

DSDV is a proactive routing protocol in which each node acts as a router and periodically broadcasts its view of the network topology to other nodes. DSDV is a variant of the traditional distance-vector based algorithms.

Each node in DSDV maintains a routing table storing all available destinations and the number of hops to each destination. Each entry in the routing table is tagged with

2.1. ROUTING PROTOCOLS

11

a sequence number assigned by the destination. The nodes in the network periodically broadcast their own routing tables. When a node receives a routing table, it compares the refreshness of entries in the received routing table against entries in its own routing table, by comparing the sequence numbers of entries with the same destination in the two routing tables. If an entry of the receiving routing table has a higher sequence number than the entry with the same destination in the local routing table, the node will update the local entry, and specify the originating node of the received routing table as the next hop to that destination. Other information such as distances is also updated.

When a link failure occurs, the node detecting the failure updates its routing table by setting the distances to destination nodes via the failed link to infinity. The updates are then propagated through the network. These destination nodes are thus disconnected from the network until new routes to them are found and the routing tables of the nodes are updated accordingly.

Incremental updates can be applied to reduce the routing traffic. Some important changes in network topology, such as link failures and addition of new nodes, are advertised immediately. Other less critical changes like change of distances can be delayed.

DSDV suffers from an implicit latency that a node has to wait for the updates from the destination in order to update its routing table for that destination. In addition, the broadcast nature of DSDV leads to excessive routing overhead. The maintenance of the complete list of routes is also an unnecessary burden for most nodes that do not send messages.

WRP [MGLA96] is another proactive routing protocol. In WRP, each node is assumed to be a router and maintains some routing information, consisting of a *routing table*, a *link-cost table*, and a *retransmission list*. The routing table specifies the distance to a known destination node, and the predecessor and successor nodes of the chosen shortest path to the destination. The link-cost table stores information about the costs of sending a packet through the neighbor nodes. The cost of a failed link is set to infinity. The retransmission list maintains information about messages to be retransmitted.

Whenever there is a change in the routing table of a node (either by detecting a failed/new neighbor node, or by receiving an update message from the neighbors), the node will broadcast an *update message* to all its immediate neighbors. The neighbors who

receive the updates respond by sending *acknowledgments*. Periodical empty updates are required in cases where there is no message transmission for a certain interval. Such periodical updates do not require acknowledgements. In this manner, the routing information can be propagated through the entire network. To transmit a packet, a path-finding algorithm is invoked to find a route from the sender to the destination.

2.1.2 Reactive Routing Protocols

The Dynamic Source Routing (DSR) protocol [JM96] is one of the earliest reactive routing protocols proposed for ad-hoc network. Instead of using the *distance vector* or *link state* routing algorithms that are commonly used in wired networks, DSR adopts the *dynamic source routing* approach in which the sender of a packet determines the sequence of nodes to be routed, and stores the hop information into the packet to be sent. The route is determined based on the route information stored in a local *route cache*. If the route cache does not contain a route to the destination, a *route recovery* process is invoked to discover one, and the discovered route is sent back to the sender in a route reply message. In route recovery, the sender broadcasts a *route request*, which is re-broadcast by intermediate nodes that are not the destination node. When the destination node receives the route request message, it determines the route from the source to the destination, based on the node ID's inserted into the route broadcast packet by the intermediate node during re-broadcast. A *route reply* message is then sent back to the sender.

Route maintenance is performed while a route is in use. In route maintenance, any node participating in the routing will notify the sender when a packet transmission failure occurs, in which case the route recovery protocol has to be invoked for discovering a new route. There is no periodical refresh message required in route maintenance, which is desirable for reducing control traffic.

Various optimization techniques are applied in DSR by making use of the topological information of the networks. Readers are referred to [JM96] for details of these optimizations.

DSR supports only single path routing, in which only one route is selected for routing a message. Single path routing has the drawback that if any node along the route fails, the message is lost. Multi-path routing in DSR can be supported with some modification. The

2.1. ROUTING PROTOCOLS

13

drawback of DSR is that it does not scale well with the size of the network. If a routing path is too long, the size of the packets will become large because all hops along the routing path are stored in the packets.

Perkins and Royer proposed the Ad-hoc On-Demand Distance Vector routing (AODV) protocol [PR98, PR99]. As a reactive routing protocol, AODV obtains routes on-demand, without resorting to periodical advertisements. Like DSR, AODV also has two stages: route discovery and route maintenance.

The route discovery process is initiated whenever a source node needs to send packets to another node for which it has no routing information. In route discovery, the source node broadcasts a *route request* (RREQ) packet to all nodes the network. When an intermediate node receives the RREQ packet, a *reverse path* towards the originator of the RREQ packet is recorded, which can be used for forwarding a packet to the originator in future.

When the destination node receives the RREQ packet, a *route reply* (RREP) message is composed and sent back to the source node using the reverse path. If an intermediate node has an up-to-date route to the destination, it can reply on behalf of the destination node. As the RREP packet is routed back to the source, each intermediate node along the path creates a *forward path*. Once the source node receives a RREP message, it can start data transmission to the destination using the latest routing information learnt from the route discovery process. Routing loops are avoided with the use of a destination-generated sequence number that ensures the freshness of the routes.

Upon a node detecting a link failure by using either periodical *hello* messages or *link layer acknowledgements*, the route maintenance process is initiated. The node sends a special route reply packet to its neighbors notifying the failure of the link, and the packet is propagated through the network. Each source that is transmitting data via the failed link is notified, and as a result, re-initiates the route discovery process.

AODV has also been extended to support multicast and broadcast operations [RP99].

Park and Corson proposed a temporally-ordered routing algorithm (TORA) [PC97] based on earlier link-reversal algorithms. The route discovery process in TORA is as follows. When a source node has no route to a destination, it constructs a directed acyclic graph (DAG) from the source node to the destination node for hop-by-hop routing. Specifically, the source node broadcasts a route request message into the network. The intermediate

nodes that are not the destination node rebroadcast the packet until the destination receives it. Upon receiving the route request, the destination node sets itself to zero height with respect to itself, indicating it is at the lowest level of the DAG. Subsequently, it broadcasts an update message as the response to the route request. An intermediate nodes receiving the update message increments the height stored in the update message by one while setting it to be its own height. The modified update message is rebroadcast by the intermediate node until it reaches the source node. After completion this process, the destination node becomes the root node of the DAG, and messages from the source node can flow along the directed edges of the DAG to reach the destination node.

Route maintenance is carried out if a link failure causes a node to lose its last downstream link, which means there is no route from the node to any destination node. In this case the node sets a new height, and updates its neighbors in order to propagate the failure. Invalid routes are discarded in response to the failure. The route discovery process may be re-initiated to find new routes to the destination nodes.

2.1.3 Hybrid Protocols

As mentioned earlier, both proactive and reactive routing protocols have some drawbacks. Proactive protocols tend to incur high control traffic overhead, while reactive protocols have considerable delay prior to data transmission when routing information is not available. Therefore ZRP [Haa97, HP00] is proposed as a hybrid protocol which tries to reduce the control traffic overhead by using an on-demand route discovery, and in the mean time, limit the searching time. Each node in ZRP applies a proactive protocol within a local neighborhood, and a reactive protocol for finding routes to destinations located farther away.

In ZRP, each node is associated with a *routing zone*, which is defined as a collection of nodes that is within n -hop distance. The local topology in a routing zone is maintained via a proactive *intra-zone routing protocol* (IARP), which is a modified link state protocol making use of periodical broadcasts.

To route to a destination node outside the routing zone, a node applies a reactive *inter-zone routing protocol* (IERP) to discover routes on-demand. Instead of using the normal flooding algorithms, ZRP initiates a *bordercasting* process, in which a source node sends

packets to nodes on the border of the routing zone (i.e., n -hops away). When a route is to be discovered, the source bordercasts a route request packet to its border nodes. The nodes receiving the route request packet check its local routing information to locate the destination. If the destination is not found, the route request packet is re-bordercast until it reaches a node whose routing zone contains the destination node. An *accumulated route* can be formed in this manner from the source routing zone to the destination zone, the nodes in which contain intra-zone route information that can be utilized for routing between routing zones.

Bordercasting introduces more overhead than traditional flooding based techniques. Neighboring routing zones may overlap, resulting in more route request messages than flooding. In addition, each bordercast message needs to travel n hops. ZRP proposed several optimization techniques to remedy these problems [HP98]. Redundant queries can be detected using *query detection*, and removed using *early termination* or *loopback termination*. Selective bordercasting can also be applied based on local routing zone information.

Link failure can be detected using IARP. Instead of propagating the failure to the entire network, a node tries to repair the failure locally. If necessary, the source can be notified to re-initiate the route discovery process.

2.1.4 Geographical Routing Protocols

The protocols mentioned above are all based on topology of the network. With the aid of location-aware devices such as GPS, it is feasible to implement routing protocols based on the physical locations of the mobile nodes. One novel protocol, GPSR (Greedy Perimeter Stateless Routing) [KK00] is an example of geographic routing protocol and has been widely adopted in many sensor network systems. Instead of using node identifiers, it makes use of the geographic locations obtained from location devices to make routing decisions.

In GPSR, each node is assumed to have the knowledge of its neighbors only. Each node is aware of its neighbors' IDs and geographic locations. There are two modes for a node to forward a packet, *greedy mode* and *perimeter mode*.

When forwarding a packet, a node checks the destination (a geographic location) of the packet, and compares it with the locations of all its neighbors. If there exists one neighbor that is nearer to the destination than the current node is, the node enters greedy mode and

Figure 2.1: GPSR forwarding in Greedy mode

forwards the packet to that neighbor. Figure 2.1 shows an example of packet forwarding in greedy mode. Node n_s is the node that is currently forwarding a packet towards the destination n_d . The nodes $(n_0, \dots, n_4)$ within the circle centered at n_s are neighboring nodes of n_s who are in the direct communication range of n_s . When forwarding the packet, n_s finds among its neighbors the nearest node to n_d , which is n_4 , and forwards the packet to n_4 . Subsequently, n_4 forwards the packet to a neighbor node that is nearer to the destination (n_5 in this case) if such a neighbor node exists.

If the current node is already the nearest node to the destination among all its neighbors, as shown in Figure 2.2, the node will forward the packet in perimeter mode. In the perimeter mode, the packet is routed temporarily to nodes that are farther from the destination location before it could finally reach the destination. Once entering perimeter mode, a packet is routed according to the well-known right-hand rule to traverse a number of nodes that are farther from the destination location than the current node. However, as soon as a nearer node is found, the packet will be set back to greedy mode to move to a node that is nearer to the destination location. Consider the example shown in Figure 2.2, a packet p is forwarded from n_0 to n_s in greedy mode since the distance $D(n_s, n_d)$ is less than both $D(n_0, n_d)$ and $D(n_3, n_d)$. After receiving the packet p , n_s discovers that none of its neighbors, i.e., n_0 and n_1 , is nearer to n_d compared to itself. Thus n_s will forward p in perimeter mode. According to the right-hand rule, p is forwarded to the next hop n_1 in perimeter mode, shown as a dashed arrow in the figure. n_1 looks among its neighbors for a near node to the destination other than n_0 . However such a node does not exist. Therefore

Figure 2.2: GPSR forwarding in Perimeter mode

the packet is forwarded again in perimeter mode to node n_2 . Now that n_2 has a neighbor n_5 that is nearer to the destination node n_d , the packet is switched back to greedy mode and forwarded to n_5 . The forwarding continues until the packet p reaches the destination n_d .

If a node notices that a packet has gone through the entire perimeter of the destination location, it means that the packet has already reached a node that is closest to the destination, and proper actions should be taken to handle the packet: either to drop it (as in GPSR), or to pass the packet to the application layer for processing (as in GHT [RKL⁺02] discussed in Section 2.2.3).

2.2 Storage Schemes

A sensor network can be viewed as a distributed database, where each sensor stores some data or measurements [BGS01]. There are mainly three approaches to storage in sensor networks, namely *external storage*, *local storage* and *data centric storage* [SRK⁺03]. It is worth noting that a sensor network can implement these three approaches at the same time without any conflict, each one serving different types of queries or applications. In this section, we briefly discuss these storage approaches and the related work. Throughout the section, N denotes the number of sensors in a sensor network.

2.2.1 External Storage

In the external storage scheme, data is stored in a central server that is located outside the sensor network, yet has a connection with the network through one or more basestations. Data collected by the sensors is transmitted to the central server where queries are processed.

The transmission of data in this storage approach is usually done via a routing tree rooted at the basestation, which is constructed during the deployment of the network. Since all data has to be sent to the central server, the existence of the routing tree facilitates reliable routing, and avoids the cost of discovering the routing path everytime a packet is sent. The tradeoff is that the routing tree has to be maintained properly, especially when the sensor nodes move frequently. The transmission cost of sending one message from a sensor to the basestation is $O(\sqrt{N})$.

User queries are processed at the central server. If the query is posed directly to the server, there is no transmission cost. However, if the query is injected at a particular node of the network, the query has to be routed to the central server for processing, which requires a cost of $O(\sqrt{N})$, and a cost of $O(\sqrt{N})$ for sending the query answers.

The external storage approach suffers from high volume of data transmission, since every sensor has to send its raw readings to the server. To reduce the cost, some intuitive measures can be taken. For example, instead of sending all readings collected, the sensors can send portions of the readings that potentially satisfy a given query. If the amount of readings being queried is much less than the amount of all readings, the communication cost can be reduced significantly. In addition to the high transmission cost, the routing tree based routing results in hotspots at the higher levels of the tree. The nodes near the root deliver more data packets, and hence their energy is drained more quickly than that of those nodes at the lower level of the routing tree, shortening the lifetime of the battery-powered network. This also explains why very few sensor network systems adopt the external storage approach.

In [DGM⁺04] the model-based querier for sensor networks (MQSN) adopts the external storage approach. A central server is responsible for generating a probabilistic model of the sensor network, based on which a subset of the sensor nodes are selected for answering a given query. With the partial query results collected from this subset of sensors, the central server is able to approximate the entire set of the query results by making use of

2.2. STORAGE SCHEMES

19

the probabilistic model, which is discussed in detail in Chapter 3. The external storage is used because the probabilistic model facilitates the reduction of the data transmission cost, and a central server is necessary for generating and maintaining the probabilistic model.

2.2.2 Local Storage

Unlike the external storage scheme that suffers from high cost of transmitting raw readings to the central server, the local storage scheme is in the other extreme where all data is stored locally at the sensors where data is generated.

In the local storage scheme, each sensor stores the data that is generated locally. Only relevant data is sent to the central server upon request. When the basestation requests for data by injecting a query into the sensor network, the sensor establishes a route to the basestation, along which the data is transmitted. In other words, the data is collected by the basestation in an on-demand manner. The advantage is that not all data is transmitted to the basestation or the central server, greatly saving the limited bandwidth. Note that in the local storage scheme, a query can be posed from any node within the network for data collection and processing without depending on the central server. The cost of sending one message to the query sink is $O(\sqrt{N})$.

Additional cost incurs for transmitting query in the local storage scheme as data is scattered in the network, and the locations storing the answers for a query are unknown. Therefore a query has to be flooded into the entire network, at a cost of $O(N)$, in order to ensure completeness in the query answers. The cost of transmitting answers to the sink (either a sensor node or the basestation) is $O(\sqrt{N})$.

Local storage scheme is commonly used in many sensor network systems because of its simplicity and effectiveness. We briefly introduce some of them here.

Directed diffusion [IGE00] is one typical system using the local storage scheme. Data collected by sensor is represented in the form of attribute-value pairs, which effectively describe the nature of the data. For example, the sighting of a four-legged animal can be described using a combination of the animal detected, detection time, location and other parameters.

Queries are expressed as *interests* using a similar naming scheme as the data. A interest is periodically broadcast by the sink to its neighbors. Upon receiving the interest, the

Figure 2.3: A Direct Diffusion Example [IGE00]

neighbors record the interest if this interest has not been received before. The interest is then passed downward by the neighbor nodes to their own neighbors, propagating the interest throughout the network. Based on the information stored in the interest packet, the nodes receiving the interest determine their rates of sending data back to the sink, which are named *gradients*. This interest propagation process filters out the sensors that do not possess answers to the query, and in the mean time, constructs routes (specified by gradients) from the data sources to the query sink. Given that interests are normally long-lasting, the cost of this query flooding process is negligible.

When an event is detected, a sensor node will send the readings of the event back to the sink, possibly using multiple routing paths based on the gradients established in the query flooding stage, if the event satisfies any interest stored previously. Upon starting to receive data from the source nodes, the sink sends updated interests to choose one or more sources with higher quality data, so that the original multi-path routes can be turned into single-path routes. Data caching and aggregations are employed to handle packet losses and to reduce message size for saving communication cost.

Figure 2.3 illustrates an example of interest dissemination and data collection in directed diffusion. Figure 2.3(a) shows the interest flooding process. The sink broadcasts the interest to its neighbors, and the neighbors rebroadcast the interest until all data sources receive the interest. During the process of interest flooding, the gradients are set up in the nodes, as depicted in Figure 2.3(b). There exist multiple routes from the source to the sink according to the gradient. In the last step shown in Figure 2.3(c), the sink selects one route, indicated as arrows in the figure, for the source to deliver the data to the sink.

As in most local storage systems, the drawback of directed diffusion is that it requires

2.2. STORAGE SCHEMES

21

a initial query flooding process to look for sensors with available data, only after which the communication between the sink and the sources can be saved by doing a refinement on the routing paths and selection of source nodes. The refresh of interests is an additional overhead too.

TinyDB [Mad03] and Cougar systems [YG03] adopt the local storage scheme. In these systems, sensors store readings locally according to a predefined schema. Queries are flooded to the network in a hop-by-hop manner. During the query flooding process, a routing tree structure rooted at the query sink is constructed and readings from the data sources are routed to the sink. These two systems focus more on data aggregation where partial query results collected from the sensors can be aggregated on the way to the sink. More details on query processing in these two systems will be covered in Chapter 3.

In addition to storing raw readings locally, TinyDB supports *storage points* for storing local events. A storage point can be viewed as a materialized view of the data stored in a sensor, which stores a bounded streaming view of the data. A storage point has a schema and a name. An example storage point is defined as follows [Mad03]:

Query 2.1.

```
CREATE
  STORAGE POINT recentLight
  SIZE 10 seconds
  (nodeid uint 16, light uint16)
```

A user is able to issue a query that selects some of the local readings and stores them into the storage point. For example, the following query can be issued to insert readings with light level > 10:

Query 2.2.

```
SELECT nodeid, light
  WHERE light > 10
  SAMPLE PERIOD 2s
  INTO recentLight
```

Since a storage point is size limited, when new data is inserted, old data has to be discarded. Users can specify queries that operate on storage points. Data stored in a

storage point can also be joined with data stored in other storage points, or other nodes.

2.2.3 Data Centric Storage

The data centric storage approach is a compromise of the external and local storage approaches. In data centric storage, a subset of sensor nodes of the network are responsible for storing some special data called *events* [SRK⁺03].

An event can be viewed as a composition of certain readings that satisfy some conditions. For example, one can define a temperature reading greater than 50° and a light level reading greater than 10 as a “fire” event. The number of events detected in a sensor network can be much less than the amount of readings collected by the sensors. Events are named, where each event name indicates a type of events. Events of the same type are stored together, either in a single sensor node, or in a cluster of nearby sensor nodes. Queries therefore operate on events instead of raw readings.

Data centric storage aims to direct queries to only those nodes containing the answers to the queries. To achieve this, a mapping is defined between the event types and the sensor nodes, e.g., a geographic hash function [RKL⁺02], given which the query processor can limit the set of nodes participating in the query to only those that have the potential results, hence, reducing the query cost. The cost of transmitting an event to the storage location is $O(\sqrt{N})$, and the costs of sending the query and sending the results back to the sink are both $O(\sqrt{N})$.

Analysis shows that data centric storage is preferable to the external and local storage schemes in certain cases, where the network size is large, and the number of events queried is much less than the number of events detected and the total amount of readings collected [SRK⁺03].

The Geographic Hash Table (GHT) is one of the early systems that implement the data centric storage concept [RKL⁺02]. In GHT, every event is assigned an event name (or event type). GHT hashes events to a geographic location based on the event name using a geographic hashing algorithm. Specifically, an event is represented as a $\langle key, value \rangle$ pair, where the key is the event name, and the value is the readings of the event. Based on the key, an event is hashed to a geographic location, which is not necessarily the location where it is detected. After this, the event is routed towards the geographic location using the

2.2. STORAGE SCHEMES

23

GPSR routing protocol [KK00] mentioned earlier in this chapter.

Note that at the geographic location it is likely that there is no sensor at all because the originating sensor does not have the knowledge of all sensors. Therefore GPSR will locate a sensor that is nearest to the hashed geographic location as the node for storing the event. This node is called the *home node* which is responsible for all operations (storing and retrieving events) on the same event name. The consistent selection of the home node is ensured by the perimeter mode of GPSR.

GHT employs the *perimeter refresh protocol* (PRP) to handle node failures and movements. Specifically, every node on the perimeter of the home node stores a replica of the data in the home node, and they are called replica nodes. The home node periodically sends refresh messages to these replica node to update their status. When a replica node receives the refresh message, it checks its own location against that of the home node. If it is nearer to the hashed location, it forwards an updated refresh message informing other nodes on the perimeter, so that it can be turned into a home node if no other node on the perimeter is nearer to the hashed location compared to itself, so that the consistency of the selection of the home node is preserved. Otherwise it simply forwards the refresh message. In case the home node fails, the replica node that first detects the missing of refresh messages from the home node (triggered by a predefined timeout) initiates the refresh message transmission process. As the refresh message travels along the perimeter, all replica nodes are notified about the failure of the home node, and a new home node is elected.

PRP increases the local traffic around the perimeter of the home node. However, the overhead can be kept low in dense networks where the perimeters are relatively short. Another drawback of the GHT is that it creates hotspots. If there are many events with the same key, the home node of the key will become a hotspot. To address this problem, GHT uses a *structured replication* (SR) scheme to balance the load in such cases, at the cost of increasing the cost of routing queries and responses. In SR, each node has a number of mirrors, where events of the same key are distributed among the mirror nodes. An event is stored at the nearest mirror node to the location where it is detected. To retrieve the event, all mirrors are searched and only those that have the query results respond to the query. SR is mainly useful for frequently occurring events which are hard to handle using a single sensor.

GHT only supports queries based on event names. To answer queries based on event values, other techniques have to be employed. We introduce them in Section 2.3.

2.3 Query Processing

Queries are commonly used to retrieve data stored in sensor networks. Constraints can be specified in queries to retrieve only the data that users are interested in. In this section, we firstly provide an overview of sensor network queries. We discuss some common sensor network query processing strategies in the rest of this section.

2.3.1 Overview

Query processing in sensor networks is very different from that in traditional centralized and distributed database systems. We address the unique differences in this section.

Unlike traditional centralized database query processing, queries in sensor network are distributed to a large number of sensors in the network. Traditional centralized algorithms are therefore infeasible in sensor networks. Though more similar to traditional distributed databases, sensor network query processing is still different in that a sensor node has much less resources compared to a node in a distributed database system. This effectively renders the query processing algorithms in distributed database inapplicable in sensor networks.

Sensor network queries can be long-running. Sensor networks are often deployed for measurement, detection, surveillance applications, where the sensors collaboratively monitor the area they are deployed in. Queries in these sensor networks are designed to retrieve streams of data continuously produced by the sensors. Hence the query results are continuously changing while new data is being produced. Such long-running queries require new query processing techniques.

Query processing in sensor network often has to cope with inaccurate and incomplete data. As we mentioned earlier, sensors may become faulty, and the data they generate may contain noises. Furthermore, loss of packets between nodes also result in incomplete data. This is very different from traditional systems where data are assumed to be complete and accurate. Therefore approximation techniques have to be utilized in sensor networks for query processing.

2.3. QUERY PROCESSING

Figure 2.4: Three Stages of Query Processing in Sensor Networks

One of the most important differences between queries in sensor networks and queries in traditional databases is the cost model for query processing. In traditional centralized databases the query cost model is normally measured in terms of the number of disk block accesses because disk accesses are known to incur much processing time overhead. The optimization goal is therefore to reduce the disk accesses of the queries. Similarly in traditional distributed database, the most costly operation is the transmission of data among the database nodes, leading to different distributed query processing approaches to reduce communication cost. However in sensor networks, the optimization goal is to conserve the battery power of the sensors by minimizing the communication cost that dominates energy consumption. Such differences in the optimization goal give rise to different query processing strategies in sensor networks.

The goal of query processing is to facilitate data collection in sensor networks in an cost-effective manner. The ultimate challenge is to reduce the communication cost as much as possible to prolong the life time of the sensors. In addition, due to unreliable wireless connections, message losses have to be handled properly during the evaluation of the queries.

Generally, sensor network query processing comprises three stages: *query dissemination*, *query execution*, and *result collection*. Figure 2.4 depicts the three stages.

Query dissemination refers to the process of transmitting a query to the sensors that are able to respond to the query with relevant data. A query is firstly posed to a basestation connected with the sensor network, or is injected into a sensor in the network. The sensor or the basestation that is responsible for collecting the results is called *sink*, while

the sensors that respond to the query are called *sources*. Upon receiving a user query, the query processor at the sink determines the query dissemination plan based on its knowledge of the network conditions and topology information, and disseminates the query according to the plan. During the query dissemination process, certain routing structures could be constructed, as in TinyDB [Mad03]. The same routing structures can route query results back to the query sink. When a sensor receives the query, it preprocesses the query. The preprocessing includes verification of the query, allocation of resources, and scheduling operations based on the query plan. The query has to be verified at the sensor because the sink may not know all information of the sensor, e.g., the tables in the sensor. Storage and other resources such as specialized data structures have to be allocated. In addition, the sensor needs to schedule observations according to the query plan. Synchronization of time with other sensors may be required as well. After all these steps are completed, the sensor is ready for query execution.

During query execution, sensors periodically monitor their surroundings, and generate the raw sensor readings. The period of monitoring is decided based on the query plan during the query dissemination process. For certain queries such as historical queries and approximate queries, it is possible for a sensor to use old readings taken before, avoiding the expensive sensing cost. The sensors then check readings against the query conditions, and eliminate the invalid readings. In queries requiring collaborations among the sensors, sensors exchange their data with each other in the query execution phase.

When query results are ready, they can be collected. The result collection process can use the routing structure established during the query dissemination process to avoid the overhead of discovering routes from the sources to the sink. Note that the query execution and result collection phases can take place simultaneously. A sensor can report the partial query result as soon as it is available, while continuing the query execution to generate more results. Each sensor completes the processing of the query when it has no more query results to deliver, or the query expires or is aborted. Sensors are switched to sleep mode as they wait for the next query or task.

A Taxonomy of Queries

There is a wide range of queries that are useful in sensor network applications. In this section we attempt to provide a taxonomy of sensor network queries.

Depending on the number of times it is evaluated, a sensor network query can be *one-shot*, or *continuous*. An one-shot query is evaluated only once. The query execution at a sensor ends as soon as it generates the result for the query. An example of one-shot queries is to retrieve the number of people currently in LT8. One-shot queries are usually used to retrieve a snapshot of the status of the network. Continuous queries, on the other hand, are evaluated multiple times. As it is infeasible to evaluate a query continuously, a continuous query is often specified with an execution rate, and is evaluated periodically. An example continuous query is to return the temperature of a particular area every 10 seconds for a duration of one hour.

Queries can be either *exact* queries or *approximate* queries in terms of the precision of query results. Exact queries refer to queries based on the actual readings of the sensors, such as the queries in TinyDB and Cougar, whereas approximate queries only requires estimated sensor readings. Confidence levels and error bounds can be specified in approximate queries to control the accuracy of the query results. MQSN [DGM⁺04] is a system providing approximate query support. The advantage of approximate query is that approximation reduces the amount of data to be transmitted. Some costly observations can be avoided since not-so-accurate and slightly outdated readings may be acceptable for the an approximate query. An example of approximate queries is provided below.

```
SELECT nodeid, temperature  $\pm$ 0.1, confidence(0.95)
WHERE nodeid IN 1..8
```

The query retrieves the temperature with a confidence of 0.95, and an error bound of 0.1°C . If the confidence level is 1 and error bound is 0, the query becomes an exact query.

We can also categorize a query based on the query operators it uses. Sensor network queries can be classified as *select* queries, *aggregate* queries, *join* queries, etc.. Aggregate queries have received much attention in the literature [MFHH03, MFHH02, HSI⁺01, IEGH02, IGE00], because there are many sensor network applications requiring summarized information, instead of individual raw sensor readings which are hard to interpret.

Categorization Criteria	Query Categories
number of evaluations	one short, continuous
result precision	accurate, approximate
query operator	select, aggregate, join, ...

Table 2.1: A Taxonomy of Queries in Sensor Networks

Simple select queries have not been addressed extensively because it is relatively easy to process a select query by scheduling the participating sensors to report their readings. The problem of processing select queries is therefore more on the efficient delivery of data from sources to the sink. Special techniques have been developed to handle select queries specific to data centric storage [RKL⁺02, GEG⁺03, LKGH03]. Selection of events based on event names is directly supported by data centric storage system. However, *range* queries, which are queries on values of events, cannot be handled directly. Since event data can be single-dimensional or multi-dimensional, the query constraint on the value range can also be single-dimensional or multi-dimensional. DIFS [GEG⁺03] and DIM [LKGH03] are two techniques for solving single-dimensional and multi-dimensional range queries, respectively.

We can also classify sensor network queries based on the types of the query conditions. Some conditions are spatial, and some may be temporal. Therefore queries can be classified as normal queries, spatial queries, temporal queries, and spatio-temporal queries. We do not discuss them in detail here.

Table 2.1 summarizes the taxonomy of queries in sensor networks. In the rest of this chapter we survey the research efforts in sensor network query processing.

2.3.2 Aggregate Queries

Data aggregation is one of the important queries used in sensor networks, which has been addressed in [MFHH02, HSI⁺01, IEGH02, IGE00].

Aggregates of sensor readings can be of greater interests to users compared with individual raw sensor readings. There are a few reasons. Firstly, aggregates are generally easier for applications and users to consume. Individual readings normally contain very raw information. For example, a temperature sensor may generate a raw temperature reading of 23.5° at time 10 : 55 : 23 pm for location l_1 . Such information may be transient and is subject to noise. In contrast, the summarized information obtained by aggregating the sensor readings

2.3. QUERY PROCESSING

may be more meaningful.

Secondly, as stated earlier, sensors may malfunction and generate garbage readings. The dynamic environment where the sensors are deployed and volatile wireless connections among the sensors may also contribute to inaccurate data. All these affect the accuracy of the readings collected by the sensors. Hence, aggregates of data collected from a group of sensors in vicinity are more accurate since the possibility of most sensors in the group generating faulty data is relatively small.

Finally, data aggregation saves communication costs. Since the sum of the sizes of each individual readings is normally greater than the size of the aggregated data which can be as small as one message, we can save a large portion of the communication cost if the aggregation is performed in the network as early as possible.

TAG [MFHH02] provides a distributed in-network aggregation service. The term 'in-network' means that the aggregation is performed while data flows through the network with irrelevant and redundant data removed as early as possible to reduce communication cost. TAG provides a generic interface for querying and aggregating data in the sensor network. A declarative SQL query language is proposed in TAG for expressing aggregate queries. The query syntax is in the following form:

```
SELECT          {agg(expr), attrs}
FROM            sensors
WHERE           {select_predicates}
GROUP BY        {attrs}
HAVING          {having_predicates}
EPOCH DURATION i
```

In TAG, the entire sensor network is treated as a database containing a single table called `sensors`. The `sensors` table can be viewed as a combination of all local tables stored in the sensors. Each local table consists of a set of attributes that is part of the `sensors` table schema. For example, in a network where there are both light and temperature sensors, both `light` and `temperature` attributes are present in the `sensors` table schema. However, for light sensors, the readings they collect do not contain temperature attribute. In cases where attribute values are missing, null values are assigned when the data tuple is inserted in the `sensors` table.

Unlike SQL queries, the aggregate queries in TAG generates a stream of tuples, instead

of a table or a set of tuples. This is very useful in monitoring applications where sensors continuously monitor the environment and generate readings or samples relating to the status of the environment, and users check the conditions of the environment being monitored by looking at the stream of readings reflecting the changes in the environment. In such monitoring applications individual readings are less important because they are hard to interpret. For continuous queries, the EPOCH DURATION clause is introduced in TAG, specifying the time interval between two successive samples.

TAG proposes the *aggregate decomposition* method to evaluate aggregate queries. The idea is fairly simple. A *routing tree*¹ is firstly constructed in the sensor network. The root of the routing tree is the query sink. Sensors participating in query processing organize themselves into a tree during the query dissemination phase. The query is flooded into the sensor network. Each sensor may receive multiple query messages. A sensor selects and notifies one sensor as its parent that is the source of the query message it receives, and retransmits the query message to other nodes. The parent node maintains a list of child node IDs obtained from the notification message sent from the child nodes. Each sensor reports its own readings to its parent, and the parent node will aggregate all readings it receives. The aggregated result is then delivered to the grandparent nodes for aggregation, and the process repeats till the root node completes the final aggregation. In this way, the aggregate query is decomposed into a number of aggregate queries which can be further decomposed. There are three categories of aggregates [HK01], among which only *distributive*² and *algebraic*³ aggregates can be processed in this manner. For *holistic*⁴ aggregates, no partial aggregates can be used. All readings have to be forwarded to the root for processing. The processing of aggregate queries in TAG consists of two phases, a *distribution* phase and a *collection* phase. In the distribution phase, a aggregate query is disseminated into the

¹Though we call it a tree, a routing tree can be a graph or any other structure as long as it provides delivery of queries to all sensor nodes and messages from sensors to the sink without duplication.

²A distributive aggregate can be computed by partitioning the data into n partitions. The aggregate value is computed for each partition, and the final result is obtained by aggregating the n aggregate values. SUM and COUNT are two examples of distributive aggregates.

³A algebraic aggregate is computed using an algebraic function with some arguments. Each argument is a distributive aggregate. For example, AVERAGE is an algebraic aggregate with two arguments: SUM and COUNT. Both arguments are distributive aggregates. The average can be derived by computing SUM/COUNT.

⁴A *holistic* aggregate is an aggregate that cannot be calculated or described using some subaggregates. Examples of such aggregates are MEAN, RANK, etc..

2.3. QUERY PROCESSING

31

network, and a routing tree is constructed. In the collection phase, sensors report their readings to their parents. Each sensor is assigned a time interval during which a message containing the partial aggregate value has to be sent to the parent. The time interval is determined by the parent by dividing the EPOCH DURATION into a number of small durations which allow enough time for children to report their values, and ensure that the result can be passed back to the root before the EPOCH DURATION ends. In addition, there is a lower-bound on EPOCH DURATION since collisions may occur if EPOCH DURATION is so short that too many sensors send data almost concurrently causing congestion in the communication channel.

TAG has the disadvantage that it requires a query to be flooded to all nodes in the network in spite of some nodes not participating in the query. To remedy this problem, a *Semantic Routing Tree* (SRT) is suggested in [MFHH03].

An SRT can be viewed as an index on some *constant attribute* in the sensor network. A constant attribute is an attribute whose value does not change, such as the ID of a node and the location of a static sensor node. Each node in the SRT stores an interval⁵ which covers all attribute values beneath its children. When a query specifying a range of the indexed attribute arrives at a sensor node, the node checks if its interval overlaps with the range specified in the query. If there is an overlap, the node will forward the query to its children, and collect results from them, based on which a partial aggregate is computed. The node stops disseminating query if there is no overlap detected since no node in its subtree is able to contribute relevant data. The communication costs are therefore saved.

Figure 2.5 depicts a SRT example defined on the constant attribute latitude attribute x . Node 1 maintains a interval $[5, 10]$, which covers all values of the children of the node 3. When a query specifying a value range of $[3, 7]$ arrives at node 1, node 1 checks it intervals, and finds that node 3 may contain the relevant data, but not node 2 because there is no overlap between node 2's interval and the query range. The query is then forwarded to node 3, and the same steps are repeated. Therefore in this example, only nodes 1, 3 and 4 are accessed. The other nodes are excluded from the query processing. Note that node 3 has to participate in the query because it has to forward data sent from node 4 to node 1, although it does not contain any query result.

⁵SRT can be extended easily to support multiple intervals.

Figure 2.5: A Semantic Routing Tree [MFHH03]

SRT can be applied to constant attributes only, although with some form of maintenance it is possible to index slowly changing attributes as well. In addition, for moving sensors, it is hard to construct and maintain such a tree. These drawbacks limit the usefulness of SRT.

The Cougar system [YG02, YG03] from Cornell University is another system supporting in-network aggregation queries. Like the TAG approach, sensor network is treated as a database in Cougar and a similar declarative query language is designed, supporting periodical and long running queries as well. At each sensor node, there is a *query proxy* for handling queries. The query proxy is a layer between the network layer and the application layer. It hides low-level message routing and query processing from higher-level applications. Queries are posed to the query proxies by applications to collect data from the sensor network.

When a query is injected into the network, a query optimizer generates a query plan based on the current work load, sensor placement, and other relevant status information about the network. The query plan deploys a few *flow blocks* to collect data from relevant sensors, and to coordinate the computation of the query. Each flow block corresponds to a set of sensor nodes known as a *cluster*. Each cluster has a *leader node*. A flow block in a cluster is used to collect data from the sensors in the cluster, and deliver them to the destination or internal nodes for partial aggregation. A few flow blocks are able to collect the partial aggregates to produce the query results. To maintain the clusters in case of node failure (including leader node), heartbeat messages and leader election have

to be employed. Since the flow blocks consumes bandwidth, and induces communication overhead, query optimizations are required to determine the appropriate number of flow blocks and their distribution, and to reuse the existing clusters in the network as much as possible.

2.3.3 Approximate Queries

As we have mentioned, sensors suffer from unreliable and bandwidth-limited wireless connection, as well as environmental noises and sensor malfunctions. Due to these constraints, answers obtained using exact queries can be biased and inaccurate. Motivated by these facts, researchers have proposed solutions using approximate queries in situations where acquisition of exact answers is impossible or unnecessary, since approximate queries are able to provide estimated answers based on partial data from the network, which has the potential of saving communication cost.

Considine *et al.* proposed *sketch*-based techniques for approximate aggregation [CLKB04] to address the limitation of TAG and Cougar in message duplication. Duplicate messages degrade the accuracy of the aggregation results, for duplicate sensitive aggregates such as COUNT and AVG. However, duplication is necessary in sensor networks where message losses are common and removing duplicate messages requires sophisticated sensor collaborations and routing schemes. Therefore in the paper the authors proposed techniques for computing duplicate sensitive aggregates using the well-known duplicate insensitive sketches [FM85]. Unlike in TinyDB, a directed acyclic graph (DAG) is used as the routing structure where each node has multiple routing paths to the root, so that if a node fails, its children can still route the messages to the root using other parent nodes. Instead of using partial aggregates, sketches are used for computing the aggregate values so that the accuracy of the query results is not affected by the duplicate values caused by the multi-path routing.

Recently Deshpande *et al.* proposed a model-based approximate query answering scheme [DGM⁺04] by exploiting the correlation among sensor readings. Figure 2.6 shows the two-day trace of voltage and temperature readings taken from two sensors. We can see that the temperature readings of the two sensors are closely related. Both temperature readings are low around time 500 and 2500 (at night), and are high around time 1000 and 3000 (at noon). This is expected since the two sensors are monitoring the same area that is cold

Sensor	Energy/Sample (3V), mJ
Solar Radiation	0.525
Barometric Pressure	0.003
Humidity and Temperature	0.5
Voltage	0.00009

Table 2.2: Energy requirements of Crossbow MTS400 Sensorboard [Mad03]

at night and hot at noon and hence the temperature readings of the two sensors should be similar. Such correlations can also be found in the voltage readings. Therefore if the readings of one sensor can be obtained, the readings of the other sensor can be estimated. Moreover, a correlation also exists between the temperature and the voltage readings: when temperature goes high, the voltage goes high as well, though the magnitude is different. There is a simple explanation that as the temperature changes, the batteries of the sensors become heat or cool, which in turn variates the voltages. For some types of sensors, e.g., Berkeley Motes [Cro05], the energy required for temperature sampling is much more than that for voltage sampling (see Table 2.2). For this reason, instead of sampling temperature, a query processor could just sample the voltage and approximate the temperature based on the correlation between voltage and temperature so that the battery power can be conserved.

Based on the above observations, Deshpande *et al.* built an approximate query engine called MQSN (Model-based Querier for Sensor Networks). In MQSN, the user specifies an SQL-like query with an error bound and confidence value. The query processor determines the optimal observation plan based on the probabilistic model. The sensors then collect data according to the plan and return results to the user. The probabilistic model is a *probability density function* (pdf):

$$p(X_1, X_2, \dots, X_n)$$

where X_i is an attribute on a sensor, for example, the temperature on sensor 1 and the voltage on sensor 6. The pdf gives the joint probability for each possible combination of the attribute values $(x_1, x_2, \dots, x_n)$. The initial model has to be learned from historical data, and it can adapt to changing data afterwards. Although the paper uses a probabilistic model based on the time-varying multivariate Gaussians, it is notable that other models can be used as well.

Figure 2.6: Voltage and temperature readings from two sensors showing correlations in a sensor network [DGM⁺04]

MQSN supports three types of queries, namely, range queries, value queries, and aggregate queries, all of which can be solved by using the pdf. New observations are required if the current pdf does not provide high confidence. Using the pdf, an observation plan can be generated to select a set of attributes for observation.

To optimize observation plans, MQSN selects those attributes that meet the confidence requirements of the query results at minimal cost, where the cost includes both acquisition cost and transmission cost. Given a set of observations, the expected benefit R on a set of attributes $\mathcal{O}$ can be computed. A greedy algorithm is proposed, which starts from an empty set $\mathcal{O} = \Phi$, and incrementally adds an attribute i to $\mathcal{O}$ that provides high expected benefits R_i , with lowest total cost. The algorithm terminates when the confidence value exceeds the desired confidence level.

2.3.4 Range Queries

DIFS [GEG⁺03] is proposed to handle range queries on top of GHT which only supports queries on event names. In DIFS, a hierarchical index with multiple roots is used to index

events with different ranges. Conceptually, nodes at the higher level of the index store information about events with smaller ranges detected in larger geographic regions. Nodes at the lower level of the index store information about events with larger ranges detected in smaller geographic regions. Each node except the roots has a number of parent nodes, each of which stores one portion of the event range stored in the child node. For example, if a node stores events with values in the range $[0, 7]$, and it has two parents, the two parent nodes should store values in the range $[0, 3]$ and $[4, 7]$, respectively. The parent nodes of the node storing $[0, 3]$ further partition the range and store $[0, 1]$ and $[2, 3]$ respectively. The root nodes in this example store single values $0, 1, \dots, 7$. The selection of parent nodes is done using a bounded geographic hash function.

When an event is detected, the event is hashed to a geographic location, and a home node will be selected for storing this event. Mean-time, the sensor node that is closest to the location where the event is occurs will store the hashed geographic location, as well as the value of the event, and the sensor node will be the leaf node of the index. Each parent node of the leaf node stores a portion of the entire event range, and each grandparent node stores a portion of the range maintained by the parent node.

Refer to Figure 2.7. Assuming the event value has a domain of $[1, 8]$. Suppose an event with value 2 is detected at position l . The event will be hashed to a location for storage according to GHT. In addition, the node nearest to l , which is node n_3 , will store a pointer to this event, and become a leaf node of the index, as shown on the right of the diagram. Node n_3 covers an area of size 1, and stores a range $[1, 8]$. Each parent of n_3 stores half the range of n_3 , and covers an area that is four time that of n_3 . In this case, the pointer will be stored at a parent node n_2 of n_3 covering the value range $[1, 4]$, that is half of the range of n_3 , and at a grandparent n_1 of n_3 with a value range of $[1, 2]$. The pointer is propagated upward until the root node n_0 which covers only one value 2. The area covered by n_2 is four times that of n_3 . And the area covered by n_1 is four times that of n_2 . The root covers the entire network that is four time that of n_1 .

To evaluate a range query, a minimum set of index nodes which exactly maintain the range specified in the query will be searched to locate the events satisfying the query. For example, given the DIFS index in Figure 2.7, consider a query on events with values in the range $[2, 6]$. 3 nodes will be accessed for this query: node n_0 covering the value 2, a node

Figure 2.7: An Illustration of DIFS Hierarchy

covering the range $[3, 4]$, and a node covering the range $[5, 6]$. Queries specifying space or distribution constraints can also be answered using the index.

DIFS only works for events with single value. [LKGH03] proposed a *distributed index for multi-dimensional data* (DIM) to support range queries on events with multiple attributes. The rationale of DIM is to store events with similar values into the same or nearby nodes.

In DIM, the area covered by a sensor network is recursively divided into smaller areas entitled *zones*. The zones are organized into a binary tree, and each is assigned a unique *zone code* consisting of 0's and 1's. Each zone is also given an address which is the centroid of the zone. A sensor node *owns* a zone if the zone is the largest covering the sensor node and no other node. Similar to DIFS, GPSR [KK00] is used to resolve this relationship between sensor nodes and zones dynamically and distributively, at the time when a query is posed or an event is detected and stored.

Figure 2.8 depicts a network area partitioned into zones, and the corresponding zone tree. The circles denote the sensors, and the 0 and 1 digits denote the zone codes for the sensors. The assignment of the zone code is done by recursively partitioning an area into two parts, each one's zone code is appended 0 or 1.

When an event is detected, it is encoded with an *event code* that has the same length of the zone code of the node that detects it. It is then routed to a destination, which is the centroid of the zone whose zone code matches with the event node, using the GPSR protocol. During the process of routing, the event node is progressively refined until it

Figure 2.8: A Network with Zones and a Zone Tree

reaches a destination zone and stored in the zone owner. Undetermined zone boundaries can also be determined during this progress.

In query evaluation, a initial zone is selected so that it contains the range specified in the query and none of its children does. The query is routed to this zone in the first step and the zone node of this zone divides the query into sub ranges and sends them to the corresponding children nodes. The process repeats until the range maintained by a zone node covers the query range completely. Node failures and packet losses have also been considered in DIM.

2.3.5 Join Queries

TinyDB [Mad03] supports join queries on the same sensor node, i.e, joining tuples stored locally at the same sensor node, or joining tuples from a local sensor with the global `sensors` table. Join operations across arbitrary pairs of sensor nodes are not supported. TinyDB makes use of *storage points* to facilitate blocking operations like join. As mentioned in Section 2.2, a storage point is a materialization point that contains a buffer of data from the sensor data streams. Users can specify queries for creating and dropping a storage point. Join operation can be performed between two storage points on the same node, or between a storage point and the `sensors` relation. For example, given the storage point `recentLight` created in Query 2.1, the following join query can be defined [Mad03]:

Query 2.3.

2.3. QUERY PROCESSING

39

```

SELECT COUNT(*)
  FROM sensors AS s, recentLigh AS rl
 WHERE rl.nodeid = s.nodeid AND s.light < rl.light
SAMPLE PERIOD 10s

```

Query 2.3 computes the number of recent light readings that is brighter than the current light reading. The equality of the attribute `nodeid` indicates that the join is performed locally on sensors where the storage point `recentLight` is defined. Without storage point, such kind of blocking join operation cannot be performed directly on the `sensors` relation since `sensors` is an unbound and continuous data stream.

Cougar [YG03] handles queries using flow blocks. A flow block collects data from a set of sensors, and an operator is specified in the flow block for computing data collected. Join queries can be evaluated in Cougar by assigning join operators to flow blocks. While a join query is processed by flow blocks, the result size may either increase or decrease depending on the selectivity of the join. If the result size increases, a centralized processing is more effective, in which sensor readings are transmitted to the sink for join. If the result size decreases, the join is performed in-network, where sensors send data to some leader nodes, and the leader nodes produce the join results and send them back to the sink. Statistics of the data distribution have to be maintained in order to estimate the selectivity of the join, and choose an appropriate query plan.

Bonfils and Bonnet proposed techniques addressing the problem of optimal operator placement scheme in a sensor network [BB03]. Their work aims to find the operator placement that minimizes the data transmission in a network. Join queries can be handled by defining correlation operators that correlate data streams from different areas in the sensor network. Figure 2.9 depicts two scenarios when correlation operators are defined. In Figure 2.9(a), the correlation operator is *reductive*, which produces much less data compared to the input data from data sources A and B. In this case, the operator is placed near region A where more data are being generated. When the correlation operator produces more data, the optimal placement is intuitively near the sink as shown in Figure 2.9(b), so that the overall cost of data transmission can be minimized.

To handle continuous queries, the operator placement has to be adaptive to the changing data rate which measures the amount of data a sensor node is transmitting. When

Figure 2.9: Operator placements [BB03]

data rate changes, the operator is adaptively placed in a new location which is optimal in terms of communication cost. A *neighbor exploration* strategy is used to locate possible sensor nodes to reduce the cost, in which *active operators* reside on *active nodes* where the correlation of data takes place, while *tentative operators* reside on *tentative nodes*, and estimate communication cost if the tentative sensor nodes were to conduct the correlation (see Figure 2.10). Specifically, each active operator computes its cost based on the data rates. The active operator communicate its data rate and cost with its immediate neighbors called tentative nodes where a tentative operator is placed. The tentative operator estimates its own cost. If the estimated cost of a tentative operator is lower than the cost of the active operator, the active parent operator that is the parent of the active operator initiates a *node switching* process, in which the tentative operator is switched to be the new active operator and the active operator stops its operation. In the node switching process, the active parent operator *deactivates* the active operator, informing the cease of data flow. It also *activates* the tentative operator, informing the start of data flow. The query operation can therefore be resumed. The process is shown in Figure 2.10.

The operator placement scheme has the restriction that each operator is only carried out at a single sensor. Since a sensor has only a small amount of memory, this requirement restricts the memory usage of operators. The optimal operator placement cannot handle join operators that require a large amount of memory for storing temporary tables produced during join processing. In addition, the exchange of data rate and cost information

Figure 2.10: Neighbor exploration [BB03]

introduces high communication overhead when data rates vary constantly.

Chowdhary *et al.* [CG05] proposed a *path-join* algorithm to find an optimal set of sensors to carry out the join operation. The basic idea is to distribute an operand table along a path in the network. Other operand tables are routed through the path so that join operators can be performed during the path traversal.

The path-join algorithm assumes two static tables R and S , stored in two small regions $\mathcal{R}$ and $\mathcal{S}$ in the sensor network, respectively, as depicted in Figure 2.11. Each sensor in $\mathcal{R}$ and $\mathcal{S}$ stores a horizontal partition of R and S , respectively. The authors focused on optimal *distribute-broadcast* join algorithm in which the table R is distributed to a *join region* P other than the region $\mathcal{R}$, and table S is broadcast to sensors in P . Each sensor in P is therefore able to join S with the portion of R it has received, and send the join results to the query sink Q . The path-join algorithm is a special case of distribute-broadcast join, which finds the optimal set of sensors for performing the join and these sensors lie on a path P (see Figure 2.11), i.e., the join region, which minimizes the total communication cost of distributing R to P , routing S through P , and sending the resultant tuples to Q . The authors proved that in a distribute-broadcast join algorithm, the shape of the join region P that incurs minimum communication cost is a path as depicted in

Figure 2.11: Path Join in Sensor Networks [CG05]

Figure 2.12: Optimal join-region for distribute-broadcast join [CG05]

Figure 2.12(a) or Figure 2.12(b). Based on the proof, an optimal P can be derived by finding the combination of C_r , C_s , and C_q that minimizes the communication cost. In addition, to reduce the complexity of the optimal algorithm, a suboptimal algorithm was designed based on heuristics.

The path-join algorithm requires knowing locations of all sensors in the network before computing the join region, since it has to consider all possible sensors during the computation. Thus, it has to be carried out at the basestation which is assumed to have the knowledge of the entire network topology. The accurate computation of the join region is also dependent on the statistics about the source data R and S that are difficult to maintain. These shortcomings therefore restricts the feasibility of path-join.

2.3. QUERY PROCESSING

43

In [AML05], Abadi *et al.* proposed an extension to TinyDB, REED, to support event detection in sensor networks using joins. In REED, a user specifies *conditions* for events to be reported by sensors. The conditions are distributed into the sensor network. A sensor joins its local sensor readings with the event conditions. The join results are events that are detected and are to be reported.

Event conditions are specified in an external table referred to as *predicate table*. The predicate table is transmitted into the sensor network. Due to limited storage, a sensor may only store a portion of the predicate table, in which case, it has to collaborate with other sensors in order to join local sensor readings with other portions of the predicate table that are not stored locally. A sample join query [AML05] below is defined to detect events using REED:

Query 2.4.

```
SELECT s.nodeid, a.condition_type
FROM sensors AS s, alert_table AS a
WHERE s.temp > a.temp_thresh
      AND s.humidity > a.humid_thresh
      AND s.time = a.time
SAMPLE PERIOD 1s
```

In Query 2.4, the `alert_table` table is the predicate table. The sensor readings in the `sensors` table are joined with `alert_table` to produce tuples that exceed the temperature and humidity thresholds.

REED employs a variation of the *nested-loop* join algorithm, in which the join result is the union of the individual results of joining local sensor readings with a partition of the predicate table, where the partitions are disjoint. Specifically, if the predicate table fits in the storage space of the sensors, it is transmitted to every sensor in the network, and joined with sensor readings as they are produced. If the predicate table is too large to fit on one sensor, it has to be horizontally partitioned. A *group* of nearby sensors share the predicate table, each storing one partition. The sensors in one group are in the broadcasting range of each other, so that communications within one group are one-hop. For example, consider the query tree depicted in Figure 2.13. In the figure, the root is the basestation collecting the join results, and each sensor maintains a list of neighbors within its broadcast

Figure 2.13: Sensor groups in REED [AML05]

range, indicated by numbers enclosed in brackets. The predicate table is distributed into the network. For the sensor group $\{5, 6, 7\}$, each sensor stores a horizontal partition of the predicate table. When sensor 7 produces a sensor reading, it is broadcast to sensors 5 and 6. If the sensor reading received by 5 joins successfully when the partition of the predicate table stored in 5, the join result is sent to the basestation via sensor 2. If the predicate table is too large to store in a sensor group, the basestation only transmits a portion of the predicate table to the sensor network, by choosing low selectivity predicates based on historical statistics.

REED provides a framework for joining sensor readings with external tables. However, it does not enable joining sensor readings scattered in the sensor network. Data stored in different regions of the sensor network cannot be correlated using REED.

2.4 Summary

In this chapter, we surveyed the sensor network literature and presented various concepts and techniques for routing protocols, storage schemes, and query processing in sensor networks.

The wireless routing protocols used in sensor networks can be classified into two general categories: *topology-based* and *geographic-based* protocols. The topology-based routing protocols can be further divided into three sub-categories: proactive routing, reactive routing, and hybrid routing. Proactive routing protocols actively maintain routing information and adapt quickly in response to network topology changes. Reactive routing protocols

2.4. SUMMARY

45

determine routing information only when needed. Hybrid routing protocols combines the benefits of the previous two categories of routing protocols, and adapt to network changes with little overhead. Geographic-based routing protocols make use of geographic information obtained from the sensors and are suitable for highly dynamic sensor networks whose topologies change frequently.

Storage schemes in sensor network provide means to store sensor readings. Based on where sensor readings are stored, there are three types of storage schemes, namely, *external storage*, *local storage*, and *data centric storage*. External storage schemes store data outside of the sensor network, providing easy data access, however, with huge data update cost. Local storage schemes, on the other hand, store sensor readings at the place where they are generated, resulting in low data update cost, but high data access cost. The data centric storage schemes facilitate fast access to named data.

Query processing in sensor networks have been the focus of the database community working on sensor networks. Declarative queries are commonly employed to process sensor network data. There are numerous types of queries for sensor networks. This chapter reviews selected work on aggregate queries, approximate queries, range queries, and join queries. The common performance measure of sensor network query processing is the energy consumption.

Chapter 3

In-Network Join Strategies

In this chapter, we discuss a few strategies for processing join queries in sensor networks and analyze their performance. We then propose a synopsis join strategy for solving one-shot simple equi-join queries in sensor networks. We verify our analysis through experiments.

3.1 Motivation

We begin by introducing a SQL-like query language for describing queries used in this chapter. Most sensor network systems adopt SQL-like declarative query language because of its expressiveness and popularity [Mad03, YG03, DGM⁺04].

In general, a sensor network query uses the following template.

```
SELECT attributes, functions
FROM sensor-tables
WHERE select-predicates
GROUP BY attributes
HAVING having-predicates
SAMPLE PERIOD period
DURATION duration
```

In the query template, *attributes* is a list of attributes of the query result. *functions* refers to one or more aggregation on some attribute(s), e.g., `AVG()`, `COUNT()`, `MIDPOINT()`¹. *sensor-tables* is a list of tables that the data is drawn from. Some systems, e.g., TinyDB, have only one table in the network, and hence the *sensor-tables* field involves a single

¹MIDPOINT() is a spatial function that returns the midpoint of the line connecting two locations.

table. In our work, we assume the sensors are heterogeneous and there can be multiple tables defined on different types of sensor readings, e.g., temperature, humidity, location. *select-predicates* and *having-predicates* specify the query conditions on the tuples (sensor readings) and grouped tuples, respectively. The `SAMPLE PERIOD` and `DURATION` clauses are required for long running queries only. `SAMPLE PERIOD` specifies a *period* between two successive evaluations of a query. `DURATION` specifies the time duration that the query is evaluated before termination. An example is given below.

Query 3.1.

```
SELECT T.loc, AVG(T.temperature)
FROM temperature AS T
GROUP BY T.loc
HAVING AVG(T.temperature) > 10°C
SAMPLE PERIOD 1s
DURATION 20s
```

The above query selects the average temperature in different locations, if it is greater than 10°C. The query operates on the *temperature* table. The sensors readings are temperatures in different locations, which are grouped and averaged based on the location attribute *temperature*. The query is evaluated every 1 second, for a duration of 20 seconds.

A join query, in simple terms, relates two or more data tuples together based on some common attributes. To some extent, it can be viewed as an operation to combine simple information into a more complex form. Due to the ability of generating data tuples with more information, join queries can be used for *event generation* in sensor networks. In addition, join queries facilitate correlating information as in traditional database systems.

Event generation refers to the process of generating *simple events* from raw sensor readings, or generating *complex events* from simple events, based on the user defined requirements and constraints. The concept of events has been introduced in Data-Centric Storage schemes [SRK⁺03, IGE00]. Events are changes in the environment being monitored by the sensors that are meaningful to sensor applications and users. However, details on event generation mechanism had not been addressed fully by the existing literature [AML05].

Events can be generated in two ways, either at the system level or application level. At the system level, an event can be defined as an external interrupt at a sensor node.

Figure 3.1: Events in Sensor Networks

When an event occurs, a signal is placed on the interrupt pin of the microprocessor of the sensor. A specially designed routine is invoked to handle the event, possibly by waking up the sensing devices and collecting the readings.

The other way to generate events is using rules at the application level. Events have been mentioned in workflow research and active database research [GD93]. It is usually specified as a query condition. Join is one of the operations that can generate events by summarizing sensor readings. An example for generating events using joins is in [AML05]. In such cases, sensors continuously evaluate queries including joins to generate events based on predefined conditions.

TinyDB [Mad03] supports both ways of event generation. Events in TinyDB are generated locally. The collaborative generation of events involving multiple sensors has not been addressed in TinyDB. Two approaches for collaborative event generation exist: *local collaborative* and *wide-area collaborative* [SRK⁺03], as shown in Figure 3.1. In the local collaborative approach, results from a small local neighborhood of sensors are synthesized. An example of using it for object tracking applications is described in [YHR⁺98]. Wide-area collaborative approach is akin to the local collaborative approach, except that the nodes participating in the collaboration may not be local. One such event generation method has been proposed also for object tracking applications in [ZSR02]. The methods proposed in the above two papers, however, are not generic enough to be extended to other application domains.

Join queries can be a useful tool for collaborative event generation. By representing data stored in each sensor as a table, event generation is essentially a join of these tables, with certain constraints specified as the join condition. Join query can therefore provide a generic event generation capability for different sorts of applications. In the local collabo-

Soldier-ID	Ammunition-Type	Shortfalls
(a) Soldier Relation		
Supplier-ID	Ammunition-Type	Quantity
(b) Supplier Relation		

Figure 3.2: Schemas of Two Relations in Ammunitions Supply Monitoring System

rative approach, join is limited to a small number of nearby nodes. And in the wide-area collaborative approach, there is no restriction on the nodes participating in the join operation.

In addition to event generation, join queries facilitate processing of complex queries in sensor networks. Join queries can retrieve correlated information in a sensor network. Such a capability is partially supported in TinyDB and Cougar. To the best of our knowledge, no query optimization have been investigated for processing join queries in these systems.

In the following section, we provide two scenarios to illustrate the usefulness of join queries in sensor networks.

3.1.1 Ammunition Supply Monitoring System

Consider an Ammunition Supply Monitoring System that monitors the ammunition usage in battle units and reports the information to the supply team and other departments in charge.

There are three kinds of users in general: soldiers, suppliers, and commanders. Each soldier is attached a sensor which monitors the ammunitions he carries. The suppliers are equipped with sensors monitoring the ammunitions that they can supply. The information monitored includes the ammunition types, and the quantity of ammunitions. We assume that the sensors are connected wirelessly. The commanders are able to access the sensor information from a central server that is connected to the sensor network via base stations.

There are two virtual relations in the system as shown in Figure 3.2. Each virtual relation contains tuples maintained at the sensor nodes.

Suppose commanders, in charge of the supply management, are interested to know the weapon usage information using two example queries listed below.

Query 3.2. *Is there any soldier who cannot find a supplier with enough ammunitions to satisfy his demand?*

3.1. MOTIVATION

51

Query 3.3. *Is there any supplier who is unable to supply any soldier with the ammunitions he carries?*

Query 3.2 can be used to trigger an event to notify the commander that more supply is needed. Query 3.3 enables the commander to recall suppliers for replenishment.

To answer Query 3.2, a simple approach is that the sensor attached to a soldier, say *s*, issues a select query to list the suppliers who are able to supply the weapons that the soldier demands. The query can be specified at a sensor *s* attached to a soldier as follows.

Query 3.4.

```
SELECT supplier-ID
FROM suppliers, s
WHERE suppliers.quantity > s.shortfalls
AND suppliers.ammunition-type = s.ammunition-type
```

If the query result is null, an event is then triggered to notify the commander that there is a soldier who could not find any supplier with enough ammunition for him. However, considering that there are numerous soldiers in the battlefield, the communication cost will be very high if all soldiers query all suppliers.

For Query 3.3, similar queries can be issued from the suppliers instead of soldiers. Sensors on the soldiers are responsible for answering the queries. Again, the high communication cost issue exists.

The commanders can issue a complex query at the central server, involving a join followed by a select query. For example, Query 3.2 can be defined as follows.

```
SELECT soldier-ID
FROM soldiers A
WHERE NOT EXISTS (
  SELECT soldier-ID, supplier-ID
  FROM suppliers, soldiers B
  WHERE suppliers.quantity >= B.shortfalls
  AND suppliers.ammunition-type = B.ammunition-type
  AND A.soldier-ID = B.soldier-ID)
```

The join query pairs the tuples from the soldiers and suppliers, and the select query selects those soldier-IDs that do not have matching supplier-IDs, indicating an event where

GPS-location	Road-Segment-Name	Car-Count
--------------	-------------------	-----------

(a) Car Count

GPS-location	Road-Segment-Name	Car-Speed
--------------	-------------------	-----------

(b) Speed

Figure 3.3: Schemas of Two Relations in Traffic Monitoring System

the corresponding soldier is not able to find a supplier. Similar queries can be defined for Query 3.3. With a good strategy of processing the join query, we can potentially save the communication cost to a large extent.

3.1.2 Traffic Monitoring System

Consider a traffic monitoring system with a set of sensors deployed at each road segment. We assume that there are two types of sensors, speed sensors and car counting sensors. Speed sensors monitor the speeds of the cars passing by, whereas car counting sensors monitor the number of cars nearby. We assume that all sensors know their own geographic locations via GPS. The traffic monitoring system continuously monitors the traffic condition of all roads. For example, it reports the number of vehicles and the average speed on each road. Assuming each road is divided into multiple road segments, and each road segment is assigned a name, we have the following two tables in the system as shown in Figure 3.3.

In order to monitor the traffic condition, events defined on combinations of the speed and car number are required. An event can be defined to report the traffic condition periodically. Naming the event as “traffic event”, we can use Query 3.5 to generate the events.

Query 3.5. *Join the readings from the speed and car counting sensors if the distance between the two sensors is less than 10 m.*

```
SELECT MIDPOINT(s.GPS-location, c.GPS-location),
       s.road-segment-name, s.speed, c.car-count
FROM speed AS s, count AS c
WHERE DISTANCE(s.GPS-location, c.GPS-location) < 10m
```

Query 3.5 is a *spatial join* on the location attributes of the sensor readings. The query produces “traffic events” in the form $\langle \text{time}, \text{location}^2, \text{road segment name}, \text{speed}, \text{car}$

²The location attribute is the midpoint of a line connecting two sensors. This is more reasonable because

3.2. IN-NETWORK BINARY EQUI-JOIN PROCESSING

53

number}. A “traffic event” is a simple event since it is the combination of low-level raw readings.

Complex events can also be obtained from simple events. For example, we define the following two simple events.

- **low speed event:** if the average speed on a road segment is less than 10 km/h, a low speed event is triggered. A low speed event has the form $\langle \text{time}, \text{road-segment-name}, \text{speed} \rangle$.
- **crowd event:** if the number of cars detected on a road segment exceeds 20, a crowd event is generated, which has the form $\langle \text{time}, \text{road-segment-name}, \text{car-count} \rangle$.

Both the “low speed events” and the “crowd events” can be defined using simple aggregate queries which are evaluated periodically by sensors. The generated events are stored at the sensors evaluating the aggregate queries. If the average speed on a road segment is low, and the number of cars on same road segment is high, it indicates a possible traffic jam. Therefore, assuming “low speed events” are stored in a virtual table `LOW_SPEED_EVENTS`, and “crowd events” in a virtual table `CROWD_EVENTS`, a “traffic jam event” can be defined using Query 3.6.

Query 3.6. *Natural join on the road segment name attribute of low speed events and crowd events if they have the same timestamp.*

```
SELECT time, road-segment-name, l.speed, c.car-count
FROM LOW_SPEED_EVENTS AS l, CROWD_EVENTS AS c
WHERE l.time = c.time AND l.road-segment-name = c.road-segment-name
```

Query 3.6 defines a natural join operation on the road segment name. The traffic jam event is a complex event obtained using simple events.

3.2 In-Network Binary Equi-Join Processing

We describe the system model and problem definition in this section. We assume a wireless sensor network with mobile sensors. Each sensor is equipped with a GPS device capable of an event should only take place at one location.

providing location information. Sensors have limited processing power and memory capacity. They continuously monitor the surroundings and generate *sensor readings* containing multiple attributes, such as temperature, humidity. Sensor readings are stored locally at each sensor. If a sensor moves, the sensor readings it stores should be passed to a nearest neighbor sensor³. In this way, a sensor reading is always stored at the location where it is detected. Since moving sensors have little impact on the data distribution in a dense sensor network, in this thesis, we assume that sensors are static for simplicity.

Sensor readings stored at a sensor s_i are collectively called the local table of s_i , denoted by T_i . Therefore, a sensor reading is also called a *tuple*. Hereafter, unless explicitly stated, we use sensor reading and tuple exchangeably. A local table consists of two *mandatory attributes*, `timestamp` and `sensorID`, where `timestamp` refers to the time at which a sensor reading is generated, and `sensorID` is the identifier of s_i where the sensor reading is taken. A local table also consists of other *data attributes*, e.g., temperature, light level, etc., depending on the sensing capabilities of s_i . These mandatory and data attributes define the relational schema of T_i . Sensor s_i has a limited amount of memory, the size of which is denoted by m . Therefore, the size of the local table, $|T_i|$, must be smaller than m .

A *distributed table* T can be defined over a set of sensors $S = \{s_1, s_2, \dots, s_n\}$, where $T = \bigcup_i T_i$, $i = (1, 2, \dots, n)$. We assume all sensors in S are homogeneous in their sensing capabilities and hence the schemas of all T_i 's are identical. T is said to be stored in S . The creation of a distributed table can be done by issuing a table creation query to select the set of sensors S , and mark the local table of each sensor to be part of T [Mad03].

We are interested in the evaluation of static one-shot *binary equi-join* (BEJ) queries in sensor networks. A BEJ query Q for sensor networks is defined as follows.

Definition 3.1. Given two sensor tables $T_E(X_1, X_2, \dots, X_n)$ and $T_L(Y_1, Y_2, \dots, Y_m)$, a *binary equi-join* (BEJ) Q is

$$Q : T_E \bowtie_{\mathbf{X}=\mathbf{Y}} T_L,$$

where $\mathbf{X}$ and $\mathbf{Y}$ are attributes of T_L and T_E , respectively, which have the same domain.

The source tables T_E and T_L are two distributed tables in the sensor network with

³The handover of all local sensor readings to the nearest neighbor node may be prohibitive in terms of communication cost. Alternative ways such as setting up forwarding pointers at the neighboring nodes can be applied.

3.2. IN-NETWORK BINARY EQUI-JOIN PROCESSING

55

Figure 3.4: A Vehicle Surveillance System

different schemas. We limit our join queries to binary join for simplicity. Multi-way joins can be handled by extending our techniques.

Consider the example of a vehicle tracking system shown in Figure 3.4. The transportation department is interested in vehicles that entered one end of the road during time interval r_1 and exited another end of the road during time interval r_2 in order to monitor the traffic volume and speed. The data correlation problem can naturally be expressed using BEJ queries, as shown in Query 3.7. As these BEJ queries are ad-hoc, it is not necessary for sensors to report their readings to the base station continuously. Instead, they can store the readings locally and report answers only when user queries are given.

A naive way to answer an BEJ query is to move the sensor readings back to the base station, and perform the join at the base station. For example, as shown in Figure 3.4, sensors are deployed at $\mathcal{R}_L$ and $\mathcal{R}_E$ to detect vehicles entering and leaving the main road. Suppose a BEJ query is issued at the base station. Using the naive strategy, the sensors send their readings to the base station which joins these readings. This approach may incur high communication cost because all sensor readings have to be transmitted even if they do not contribute to the join results. The high volume of data transmission will quickly drain the limited energy of battery-powered sensors and hence is undesirable in wireless sensor networks. A better approach is to transmit only those readings that are likely to contribute to the join results. Our proposed synopsis join strategy is therefore designed based on this idea.

Query 3.7. *Find vehicles that entered one end of the road during time interval r_1 and exited another end of the road during time interval r_2*

```

SELECT  $T_L$ .vehId,  $T_L$ .time,  $T_E$ .time FROM  $T_L$ ,  $T_E$ 
WHERE  $T_L$ .vehId =  $T_E$ .vehId
      AND  $T_L$ .time IN  $r_1$ 
      AND  $T_E$ .time IN  $r_2$ 

```

Several sensor network characteristics affect the design of BEJ query evaluation strategies. As message sending and receiving dominate energy consumption, we aim to reduce the amount of messages to be transmitted during BEJ query evaluation. Ideally, only messages carrying tuples involved in the join result should be transmitted. However without a careful design, a BEJ query may require transmission of all data tuples in the network, e.g., sending all data to the sink for join, which is prohibitively costly.

Another challenge to be considered is the storage overflow problem. A BEJ query may produce more data than the original source relations. In a traditional database system, this is not a major issue since the storage space is assumed to be unlimited. Sensors, however, only have limited storage space. An overloaded sensor may suffer from insufficient storage space during processing of join queries. The solution to this problem is to distribute the workload evenly to a set of sensors. Instead of using one sensor for the join operation, a set of sensors collaborate to work on the join operation. The additional benefit is the reduction of hotspots.

We also notice that join queries in sensor networks are similar to that in distributed databases. Both involve a set of distributed tables, and tuples from them are to be combined. Moreover, join strategies for distributed databases often aim to reduce communications among the nodes, which is similar to message reduction in sensor networks. However there still exist some differences.

1. Distributed databases assume wide bandwidth among the processors, normally in the range of 10Mbps to 1Gbps with modern internet technologies, which is much greater than that of the wireless connections in sensor networks, which are typically 1 – 100Kbps [GW00]. The availability of the bandwidth enables transmission of large amount of data.
2. Message loss is not an important issue in distributed databases with reliable communication channels. However, due to unstable wireless connections, sensor network

suffers from message losses. To simplify our problem, we leave handling of message loss for future work.

3. Each node in a distributed database system usually has significant amount of resources (energy, storage, etc.), as well as computation power. Sophisticated algorithms can therefore be implemented. In sensor networks, the limited computation power restricts the complexity of the algorithms that can be utilized.
4. A sensor network can involve thousands of sensors, as opposed to a few nodes in a distributed database system. With limited number of nodes, nodes in a distributed database system can jointly maintain some statistics about their tables and some indexes to local data. It is infeasible for sensors to obtain such statistics and to locate required data. A sensor has to locate the data first before processing it. In the worst case, locating the data requires flooding of the entire network, which is undesirable.

Our objective is to minimize the total communication cost for processing a given BEJ query in order to prolong the sensor network lifetime. In addition, the join scheme has to ensure that the memory space needed by the join operation on each join sensor does not exceed the available memory space. In Section 3.3, we present several strategies for performing in-network binary equi-joins. In Section 3.4 we describe a *synopsis join strategy* in which unnecessary data transmission is reduced by an additional synopsis join phase.

3.3 General Strategies for Evaluating BEJ Queries

In this section we summarize general strategies for solving BEJ queries. We also analyse their performances in terms of communication cost, measured by the total number of messages incurred during the query processing. An application layer message can be broken down into multiple routing layer packets. Control packets of the routing protocol for route discovery and maintenance are excluded from the cost analysis of different approaches, because the control traffic overhead is negligible compared to data communication. Hereafter, unless explicitly stated, a message refers to a routing layer packet.

In computing the communication cost of each strategy, we use the asymptotic cost of $O(N)$ messages for flooding and $O(\sqrt{N})$ for point-to-point routing [RKL⁺02]. We denote

Symbol	Meaning
$ S $	total number of sensors in the sensor network
T_L, T_E	two distributed tables
T_L^f, T_E^f	horizontal partitions of T_L and T_E
S_L	the set of sensors storing T_L
S_E	the set of sensors storing T_E
S_F	the set of join sensors
$ S_L $	number of sensors in S_L
$ S_E $	number of sensors in S_E
$ S_F $	number of sensors in S_F
$ T_L $	number of tuples in T_L
$ T_E $	number of tuples in T_E
$\mathcal{R}_L$	source region where T_L is stored
$\mathcal{R}_E$	source region where T_E is stored
k	number of results = $ T_L \bowtie T_E $
δ	selectivity of join, $\delta = k/ T_L \cdot T_E $

Table 3.1: Symbols for Performance Comparison

$|S|$ as the number of sensors in the sensor collection S . Two distributed tables T_L and T_E are stored in two sets of sensors S_L and S_E , respectively.

Suppose that a BEJ query Q joins T_L and T_E and produces k number of result tuples. The selectivity δ of the query Q is defined as:

$$\delta = \frac{k}{|T_L| \cdot |T_E|}$$

The notations and symbols for the performance comparison of various BEJ query strategies are summarised in Table 3.1

In general strategies, *join nodes* S_F are selected to join tuples of T_L and T_E , without attempting to firstly filter out tuples that are not involved in the join results (referred to as *non-candidate tuples*).

When a join query is issued, a *join node selection* process is initiated to find a set of join nodes S_F to perform the join. T_L tuples are routed to a *join region* R_F where the join nodes S_F reside in. Each join node $s_f \in S_F$ stores a *horizontal partition* of the table T_L , denoted as T_L^f . T_E tuples are transmitted to and broadcast in R_F . Each join node s_f receives a copy of T_E and processes local join $T_L^f \bowtie T_E$. The query sink obtains the join results by collecting the partial join results at each s_f . Note that though T_E could be too large for s_f to store, the local join $T_L^f \bowtie T_E$ at s_f can be performed in a pipelined manner

3.3. GENERAL STRATEGIES FOR EVALUATING BEJ QUERIES

59

to avoid memory overflow [LC85].

The selection of S_F is critical to the join performance. Join node selection involves selecting the number of nodes in S_F , denoted by $|S_F|$, and the location of the join region R_F . To avoid memory overflow, assuming T_L is evenly distributed in S_F , $|S_F|$ should be at least $|T_L|/m_L$, where $|T_L|$ denotes the number of tuples in T_L and m_L denotes the maximum number of T_L tuples a join node s_f can store.

Depending on the location of the join region, we have at least three join strategies, namely, *naive join*, *sequential join*, and *centroid join*. We discuss these join strategies and analyse their communication costs in terms of number of messages required.

3.3.1 Naive Join

In naive join, sensor nodes around the sink are selected as the join nodes S_F , so that the cost of routing join results to the sink can be minimized (see Figure 3.5). The communication cost involves routing tables T_L and T_E to the join region S_F , broadcasting T_E to the join nodes, and sending the join results from S_F to the sink (shown in Equation 3.1)⁴. Naive join establishes the performance benchmark for all join strategies, since any join strategy should at least perform better than naive join in terms of total communication cost in order to be a reasonable join strategy.

$$C_{\text{naive-join}} = |T_E| \text{dist}(\mathcal{R}_E, \mathcal{R}_F) + |T_E| |S_F| + \sum_{s_f \in S_F} (|T_L^f| \text{dist}(\mathcal{R}_L, s_f) + |T_L^f \bowtie T_E| \text{dist}(s_f, \text{sink})) . \quad (3.1)$$

3.3.2 Sequential Join

Sequential join minimizes the cost of routing and distributing T_L tuples to the join region by selecting the sensors S_L as S_F (see Figure 3.6). In this case, T_L tuples remain in their respective nodes. T_E tuples are routed to the region R_L , and broadcast to all nodes S_L . Each node $s_l \in S_L$ performs the local join $T_l \bowtie T_E$ where T_l is the local table stored at s_l . Join results are delivered to the sink as shown in Figure 3.6. The communication cost of

⁴ $\text{dist}(A, B)$ refers to the hop distance between A and B . If A, B are two nodes, $\text{dist}(A, B)$ is the average hop distance of the routes selected by the routing protocol. If A, B are two regions, $\text{dist}(A, B)$ is the average hop distance between any pair of nodes from A and B . If A is a region and B is a node, $\text{dist}(A, B)$ refers to the average hop distance between B and all nodes in A .

Figure 3.5: Naive Approach

Figure 3.6: Sequential Approach

this strategy is:

$$C_{\text{seq-join}} = |T_E| \cdot \text{dist}(\mathcal{R}_L, \mathcal{R}_E) + |T_E| \cdot |S_L| + \sum_{s_l \in S_L} |T_l \bowtie T_E| \cdot \text{dist}(\mathcal{R}_L, \text{sink}) . \quad (3.2)$$

3.3.3 Centroid Join

Centroid join selects an optimal join region within the triangle formed by T_L , T_E , and the sink, such that the total communication cost is minimized (see Figure 3.7). The communication cost is shown in Equation 3.3. Path-Join [CG05] is an example of this strategy, which finds an optimal join region by minimizing a target cost function. Note that naive

Figure 3.7: Centroid Approach

join and sequential join are special cases of centroid join.

$$C_{\text{cen-join}} = \sum_{s_j \in S_E} |T_E^j| \cdot \text{dist}(s_j, R_F) + |T_E| \cdot |S_F| + \sum_{s_i \in S_L} |T_L^i| \cdot \text{dist}(s_i, R_F) + \sum_{s_f \in S_F} |T_E^f \bowtie T_L^f| \cdot \text{dist}(s_f, \text{sink}) . \quad (3.3)$$

The above three strategies can be further optimized for BEJ queries. A hash-based join can be applied in which both T_L and T_E are partitioned into a number of disjoint sub-tables, each with a join attribute value range. Each node s_f in S_F ⁵ is dedicated to join two subsets of $T_L(v)$ and $T_E(v)$ with the same join value range v . In this way, tuples with the same join attribute value are always joined at the same join node, and the broadcasting of T_E in S_F can be avoided.

The major problem associated with the three general strategies is the communication overhead for transmitting non-candidate tuples in T_L and T_E , especially for queries with low join selectivity.

3.4 Synopsis Join Strategy

Based on the common drawbacks of the general join strategies, we propose the synopsis join strategy (SNJ) that can prune non-candidate tuples and only candidate tuples are trans-

⁵A subset of S_F is needed if one node does not have enough memory space for handling the join.

mitted to the final join sensors. The key to the pruning process is to keep the cost overhead as low as possible. The synopsis join strategy comprises four phases, *query dissemination*, *preliminary join*, *notification transmission* and *final join*.

3.4.1 Query Dissemination

In this stage, the join query is disseminated to the sensors that participate in the query. The simplest way is to flood the query to all sensor nodes in the network. The flooding can be achieved by broadcasting the query to the neighboring nodes at the sink. Upon receiving the query, each node rebroadcasts the query if it is the first time that the query arrives at the node. The process is depicted in Figure 3.8(a). The arrows denote the query messages. The cost of flooding the query in this case is $O(N)$.

Alternatively, if a data-centric storage scheme is adopted, the query can be delivered to only those nodes that store T_L or T_E tuples. The cost is therefore $(|S_L| + |S_E|) \cdot O(\sqrt{|S|})$. In this approach, a location-based routing protocol such as GPSR [KK00] is required to route the query to the destination sensors.

A hybrid approach for disseminating the query is that the sink sends a BEJ query to one of the S_L (and S_E) sensors using a location-based routing protocol, as shown in Figure 3.8(b). Once the first S_L (and S_E) sensor receives the query, the sensor broadcasts the query among the S_L (S_E) sensors. The query dissemination cost in this case is therefore $O(\sqrt{|S| + |S_L| + |S_E|})$.

3.4.2 Preliminary Join

The preliminary join phase performs an inexpensive *synopsis join*, aiming at reducing the number of T_L and T_E tuples to be transmitted for final join. The preliminary join phase comprises two steps: *synopsis generation*, *synopsis join*.

Synopsis Generation

A *synopsis* is a digest of a relation that is able to represent the relation to perform some operation, such as aggregation or join. We denote $H(T)$ as the synopsis of a table T . A synopsis can be in any form such as histograms, wavelets, etc., which is generally smaller than the size of the corresponding table. In this thesis, we adopt simple histograms as

(a) Broadcast approach

(b) Hybrid approach

Figure 3.8: Query dissemination

Figure 3.9: Synopsis Join Strategy

Timestamp	Vehicle-Type	Speed (km/h)
10:23:12	car	82
10:25:29	bus	69
10:30:48	car	85
10:31:31	lorry	70
10:36:07	lorry	62
10:36:40	car	78

(a) Original Table $\mathcal{R}$

Tuple-ID	Vehicle-Type	Count
t1	car	3
t2	bus	1
t4	lorry	2

(b) Synopsis $S(\mathcal{R})$

Figure 3.10: An example of synopsis

synopses where a synopsis is represented by the join attribute values of a table and the number of tuples for each join attribute value, i.e., the frequencies of the join attribute values. For example, assume a sensor table T shown in Figure 3.10(a). Let the join attribute be **Vehicle-type**. The corresponding synopsis $H(T)$ consists of two attributes, the join attribute value, whose domain is all possible values of **Vehicle-Type**, and the number of tuples for each **Vehicle-Type**, as shown in Figure 3.10(b).

In the synopsis generation step, each sensor generates a synopsis of its local table. Consider a distributed table T distributed among S sensor nodes. Each node $s_i \in S$ stores a local table T_i that is part of T . s_i generates a *local synopsis* $H(T_i)$ of T_i by extracting the

Symbol	Meaning
T	a distributed table
$H(T)$	the synopsis of a distributed table T
T_i	a local table stored at sensor s_i
$H(T_i)$	the synopsis of a local table T_i
$\mathbf{X}$	the join column of T
$ \mathbf{X} $	the distinct values of the join column $\mathbf{X}$
$H_i(T_l)$	the synopsis of tuples sent to the preliminary join sensor s_i from by sensor s_l storing a local table T_l
$H_i^v(T_l)$	a partition of $H_i(T_l)$ with join attribute values in v sent to a sensor s_i maintaining the range v

Table 3.2: Symbols for Synopsis Join Strategy

join column $\mathbf{X}$ of T_i , and computing the frequencies of the distinct values in $\mathbf{X}$. Assuming uniform data distribution, we can derive $|H(T_i)|$ as:

$$|H(T_i)| = |\mathbf{X}| \left(1 - \left(1 - \frac{1}{|\mathbf{S}|} \right)^{|T|/|\mathbf{X}|} \right). \quad (3.4)$$

where $|\mathbf{X}|$ denotes the number of distinct values for the join column $\mathbf{X}$. The proof is provided in Appendix A. To ease the understanding of the synopsis join strategy, a summary of symbols used is provided in Table 3.2.

Synopsis Join

In this stage, a set of *preliminary join sensors* S_I in the *preliminary join region* $\mathcal{R}_I$ is selected to join the synopses of T_L and T_E to determine the candidate tuples in T_L and T_E , respectively (see Figure 3.9). Once S_I sensors are determined, the local synopses are routed to them for preliminary join. For BEJ queries, each preliminary join sensor $s_i \in S_I$ is assigned a range v of join attribute values by using a geographic hash function such as GHT [RKL⁺02], so that only synopses with the join attribute value in the range v are transmitted to s_i for preliminary join. For a node $s_l \in S_L$, the local synopsis $H(T_l)$ is divided into $|S_I|$ partitions. A partition $H_i^v(T_l)$ containing a synopsis of tuples with join attribute values in v is sent to the preliminary join sensor s_i maintaining the corresponding range v . The process is illustrated in Figure 3.11. In this figure, s_l is a sensor whose local synopsis is $H(T_l)$. $H_{i1}(T_l)$ and $H_{i2}(T_l)$ are two partitions of $H(T_l)$. The partition $H_{i1}(T_l)$ is sent to the preliminary join sensor s_{i1} which is responsible for joining synopses containing

Figure 3.11: An example of Preliminary Join

values 3 and 5. Similarly, $H_{i2}(T_l)$ is sent to s_{i2} responsible for joining synopses containing values 7 and 9.

Consider the example shown in Figure 3.10. Suppose there are two preliminary join nodes s_{i1} and s_{i2} . s_{i1} is dedicated to handle join attribute values `car`, while s_{i2} handles `bus` and `lorry`. When a sensor s_l generates a local synopses as the one in Figure 3.10(b), it divides the synopses into two partitions, one partition $H_{i1}(T_l)$ contains tuples $t1$ and $t3$, whose join attribute values are `car`, and the other partition $H_{i2}(T_l)$ contains tuples $t2$ and $t4$ whose join attribute values are `bus` and `lorry`. Therefore $H_{i1}(T_l)$ and $H_{i2}(T_l)$ are sent to s_{i1} and s_{i2} for preliminary join, respectively.

The preliminary join sensors perform preliminary join as local synopses from S_L and S_E nodes arrive. We denote a local synopsis from a node $s_l \in S_L$ received by a preliminary join sensor s_i as $H_i(T_l)$. A preliminary join operation performed at s_i is defined as follows.

$$\biguplus_{s_l \in S_L} H_i(T_l) \bowtie \biguplus_{s_e \in S_E} H_i(T_e). \quad (3.5)$$

The $\biguplus$ operator is a merge function which takes multiple local synopses as inputs and produces a new synopsis. In particular, for our histogram synopsis, $\biguplus$ is defined as a function that accumulates the frequency values if two input tuples are of the same join attribute value. The output of $\biguplus$ is therefore the accumulation of the input histograms. An example of the $\biguplus$ operator is shown in Figure 3.12. In the figure, $H_0(T_1)$ and $H_0(T_2)$ are two local synopses received by a preliminary join sensor s_0 . Both synopses are histograms

3.4. SYNOPSIS JOIN STRATEGY

67

Value	Frequency		Value	Frequency		Value	Frequency
3	4	$\uplus$	3	2	=	3	6
7	2		9	1		7	2
10	7		10	1		9	1
						10	8
(a) Local Synopsis $H_0(T_1)$			(b) Local Synopsis $H_0(T_2)$			(c) $H'_0 = H_0(T_1) \uplus H_0(T_2)$	

Figure 3.12: An example of a merge function

with 3 distinct values. The $\uplus$ operator produces a histogram H'_0 by adding the frequencies of each distinct value from $H_0(T_1)$ and $H_0(T_2)$. For example, for value 3, the new frequency in H'_0 is $4 + 2 = 6$, whereas value 9 only appears in $H_0(T_2)$, therefore, its frequency in H'_0 is still 1, the same as in $H_0(T_2)$.

Preliminary Join Sensors Selection The number of preliminary join sensors is determined by the sizes of the local synopses that N_L nodes receive. Specifically, suppose a sensor's memory space is m_s (number of synopsis tuples that can fit into a sensor), the number of preliminary join sensors is determined as follows.

$$|S_I| = \frac{1}{m_s} \sum_{s_l \in S_L} |H(T_l)|. \quad (3.6)$$

The locations of S_I nodes are selected so that the communication cost for routing the local synopses is minimized. The communication cost of sending local synopses from S_L and S_E nodes to S_I nodes can be expressed as:

$$\sum_{s_i \in S_I} \sum_{s_l \in S_L} |H_i(T_l)| \cdot \text{dist}(s_i, s_l) + \sum_{s_i \in S_I} \sum_{s_e \in S_E} |H_i(T_e)| \cdot \text{dist}(s_i, s_e). \quad (3.7)$$

Assuming the preliminary join region $\mathcal{R}_I$ is small compared to $\text{dist}(s_i, s_l)$ and $\text{dist}(s_i, s_e)$, we can simplify the above equation as:

$$C_{\text{synopsis-routing}} = |P_L| \sum_{s_l \in S_L} |H(T_l)| + |P_E| \sum_{s_e \in S_E} |H(T_e)|, \quad (3.8)$$

where $|P_L|$ (or $|P_E|$) is $\text{dist}(\mathcal{R}_L, \mathcal{R}_I)$ (or $\text{dist}(\mathcal{R}_E, \mathcal{R}_I)$)⁶. Given the above equation, the optimal set of preliminary join sensors that minimize $C_{\text{syno-join}}$ are located on the line

⁶ P_L (or P_E) is the path connecting the centers of $\mathcal{R}_L$ (or $\mathcal{R}_E$) and $\mathcal{R}_I$.

connecting $\mathcal{R}_L$ and $\mathcal{R}_E$. Therefore we have:

$$|P_L| + |P_E| = \text{dist}(\mathcal{R}_L, \mathcal{R}_E) . \quad (3.9)$$

From the above two equations, assuming $\sum_{s_l \in S_L} |H(T_l)| > \sum_{s_e \in S_E} |H(T_e)|$, it is obvious that $C_{\text{synopsis-routing}}$ is minimized when $|P_L|$ is zero, and $|P_E|$ is $\text{dist}(\mathcal{R}_L, \mathcal{R}_E)$. The minimum $C_{\text{synopsis-routing}}$ is hence:

$$\min(C_{\text{synopsis-routing}}) = \text{dist}(\mathcal{R}_L, \mathcal{R}_E) \sum_{s_e \in S_E} |H(T_e)| . \quad (3.10)$$

Therefore, the optimal set of preliminary join sensors N_L are chosen from sensors in S_L that are nearest to S_E , assuming the size of the region $\mathcal{R}_L$ is small compared to the distance between $\mathcal{R}_L$ and $\mathcal{R}_E$. Optimal selection of S_I for arbitrary $\mathcal{R}_L$ and $\mathcal{R}_E$ regions are part of the future work.

3.4.3 Notification Transmission

Each sensor node in S_L and S_E needs to be notified of which are the candidate tuples. To achieve this, a preliminary join sensor s_i stores the ID of the sensor a local synopsis originates from. For each join attribute value v , it identifies two sets of sensors S_L^v and S_E^v storing tuples with join attribute value v , and selects a *final join sensor* s_f to join these tuples from S_L^v and S_E^v , such that the communication cost of sending data tuples with join attribute value v from S_L^v and S_E^v to s_f , and sending the result tuples from s_f to the sink is minimized. Therefore s_f is the sensor that minimizes the cost function in Formula 3.11.

$$\begin{aligned} \sum_{s_l \in S_L^v} |T_l^v| \cdot \text{dist}(s_l, s_f) + \sum_{s_e \in S_E^v} |T_e^v| \cdot \text{dist}(s_e, s_f) + \\ |T_L^v \bowtie T_E^v| \cdot \text{dist}(s_f, \text{sink}) , \end{aligned} \quad (3.11)$$

where $|T_l^v|$ and $|T_e^v|$ denote the number of T_L tuples in s_l and T_E tuples in s_e with the join attribute value v , respectively. $|T_L^v \bowtie T_E^v|$ denotes the size of the result of jointing all T_l^v tuples with T_e^v tuples at the final join sensor s_f .

In order to simplify the problem, the weighted centers of sensors in S_L^v and S_E^v are derived, respectively. The weighted center c of a set of sensors S storing a distributed table

3.4 SYNOPSIS JOIN STRATEGY

69

T are defined in Equation 3.12, where T_i refers to the table stored in node s_i , and $\text{loc}(s)$ refers to the location of a node s .

$$\text{loc}(s) = \frac{1}{\sum_{s_i \in S} |T_i|} \cdot \sum_{s_i \in S} |T_i| \cdot \text{loc}(s_i) . \quad (3.12)$$

With Formula 3.12, the weighted centers c_l and c_e for sensors in S_L^v and S_E^v can be computed respectively. Since $\sum_{s_l \in S_L^v} |T_l^v| = |T_L^v|$ and $\sum_{s_e \in S_E^v} |T_e^v| = |T_E^v|$, we can rewrite Formula 3.11 as:

$$|T_L^v| \cdot \text{dist}(c_l, s_f) + |T_E^v| \cdot \text{dist}(c_e, s_f) + |T_L^v \bowtie T_E^v| \cdot \text{dist}(s_f, \text{sink}) . \quad (3.13)$$

Formula 3.13 is minimum when s_f is the *generalized Fermat's point* [GR65] of the triangle formed by c_l, c_e , and the sink. Note that there may not exist a sensor located at the derived generalized Fermat's point g . GPSR is used to select a node that is nearest to g as the final join node s_f .

The same operation is performed for all join attribute values handled by s_i . When synopsis join is completed, s_i obtains for each sensor node s_l a set of $\langle v, s_f \rangle$ pairs, which means tuples stored in s_l with the join attribute value v are to be sent to s_f for final join. The set of pairs are sent to s_l in a *notification message*. The notification message can be broken up into multiple ones if it cannot fit into one network packet. The communication cost for the notification transmission is similar to Equation 3.10.

$$C_{\text{notification}} = \text{dist}(\mathcal{R}_L, \mathcal{R}_E) \sum_{s_k \in S_L \cup S_E} |d_k| , \quad (3.14)$$

where $|d_k|$ denotes the total size of the notification messages sensor s_k receives.

3.4.4 Final Join

Upon receiving a notification message from a preliminary join sensor, each sensor in S_L or S_E sends the candidate tuples whose join attribute values are specified in the notification message to a final join node s_f . In the final join stage, a group of final join nodes S_F are selected to join the candidate tuples sent from sensors of S_L and S_E , as shown in Figure 3.9. A final join node s_f performs the join $T_L^v \bowtie T_E^v$, and sends the join results to the query

sink. If s_f does not have enough memory space, it requests its neighbors to help in the join operation.

3.5 Experiments

In this section, we evaluate the performance of synopsis join strategy and compare it with other general join strategies, i.e., naive join, sequential join, centroid join, and optimal join which is discussed later in this section. Throughout the experiments, performance was measured by the total number of messages incurred. The control messages for synchronization and coordination among the sensors are negligible compared to the heavy data traffic caused by large tables.

We varied the following parameters: join selectivity, network density, node memory capacity and synopsis size. Join selectivity δ is defined as $|T_L \bowtie T_E| / (|T_L| \cdot |T_E|)$. The join attribute values are uniformly distributed within the domain of the attribute. Network density affects the number of neighboring nodes within the communication range of a sensor node. We varied the communication radius of the sensors to achieve different network densities. The size of the synopsis is determined by the data width of join attribute. If the synopsis size is small, the number of messages needed for routing the synopses to the preliminary join sensors becomes small. If it is large, we expect a high communication overhead incurred due to the transmission of synopses.

3.5.1 Experiment Setup

We created a simulation environment with 10,000 sensor nodes uniformly placed in a 100×100 grid. Each grid contains one sensor node located at the center of the grid. The sink is located at the right-top corner of the area, with coordinates (0.5, 0.5). The regions $\mathcal{R}_L$ and $\mathcal{R}_E$ are located at the bottom-right and bottom-left corners of the network region, respectively, each covering 870 sensor nodes. The distributed table T_L consists of 2000 tuples, while T_E consists of 1000 tuples. T_L and T_E tuples are uniformly distributed in $\mathcal{R}_L$ and $\mathcal{R}_E$, respectively.

We assumed a dense network with GPSR as the routing protocol. The number of hops required to route a message from a source node to a destination node is approximated using

the distance between the two sensors and the communication radius. The simplification enables analysis of network traffic under ideal conditions where there is no message loss. In addition, the overhead of GPSR perimeter mode is avoided with the assumption of dense network. Simulations and experiments under real conditions using GPSR are presented in Chapter 4. We set a message size of 40 bytes, which is equal to the size of a data tuple. A tuple in the join result is 80 bytes since it is a concatenation of two data tuples.

Join Strategies

We evaluated and compared the performances of five different join strategies, namely, naive join, centroid join, sequential join, optimal join, and synopsis join.

The optimal join provides a lower bound on the total communication cost involved in the join operation. It assumes that the query sink has a complete knowledge about the distribution of T_L and T_E . Hence, unlike centroid join, only candidate tuples are transmitted for the final join at S_F . Similar to the final join phase of the synopsis join strategy, for each join attribute value v , an optimal node s_f is selected such that the total cost of routing T_L^v and T_E^v to s_f and routing the result $T_L^v \bowtie T_E^v$ is minimized. The cost is expressed as in Equation 3.15. Since for any join strategy, the transmissions of candidate tuples and the join results cannot be avoided, the optimal join provides a lower bound on the total number of messages. Note that the assumption is impractical in real environment.

$$C_{\text{optimal-join}} = \sum_{s_f \in S_F} (|T_L^v| \cdot \text{dist}(\mathcal{R}_L, s_f) + |T_E^v| \cdot \text{dist}(\mathcal{R}_E, s_f) + |T_L^v \bowtie T_E^v| \cdot \text{dist}(s_f, \text{sink})) . \quad (3.15)$$

3.5.2 Performance Results

Performance vs. Join Selectivity

Figure 3.13 shows the total communication cost for different join selectivities while keeping the memory capacity, communication radius and synopsis size fixed at 250×40 bytes, 2 units and 10 bytes respectively. As shown in the figure, sequential join performs worse than all others due to the high cost of broadcasting T_E to all sensors in S_L . Therefore we exclude it in subsequent experiments. The optimal join outperforms all other strategies for all

Figure 3.13: Impact of Selectivity

selectivities as expected. When selectivity is lower than 0.001, SNJ outperforms naive join and centroid join. This is because most of the non-candidate tuples can be determined in the preliminary join stage, and only a small portion of data are transmitted during the final join. On the other hand, when the selectivity is high, almost all data tuples are involved in the result. With large join result sizes, the final join sensors are centered around the sink. This explains why the naive, centroid and optimal joins have the same communication cost. However, synopsis join incurs unnecessary communication sending the synopses, making it less desirable for high selectivity joins.

Although there is an overhead of using preliminary join when selectivity is high, it accounts for a small portion of the total communication cost. The overhead when selectivity is 0.1 is only 7%. Only when the selectivity is 0.005 and 0.01, the preliminary join overhead accounts for a significant portion (20% ~ 30%) of the total cost. In addition, many BEJ queries have small selectivities where SNJ is more suitable. For high selectivity queries, specialised join processing techniques should be adopted for superior performance.

Figure 3.14: Impact of Network Density

Impact of Network Density

Figure 3.14 shows the scalability of the join strategies with varied network density. In this experiment, sensors have a memory capacity of 250×40 bytes. The join selectivity and synopsis size is 0.0001 and 10 bytes, respectively. As the network becomes denser, the total communication costs for all strategies decrease, too. This is expected because with a larger communication range, fewer hops are needed to send a message across the network.

Impact of Memory Capacity

Figure 3.15 shows the total communication cost with different memory capacities. In this experiment, the communication radius is 2. The synopsis size is 10 bytes, and the selectivity is 0.0001. It is shown that the communication costs of all strategies do not change much when the memory capacity increases. The change in the memory capacity only affects the number of final join sensors (and the number of preliminary join sensors for SNJ). When the memory capacity is larger, there are fewer final join sensors selected (8 final join sensors reduced to 1 in our experiment setup), and fewer messages are required for sending the result tuples to the sink. There is little reduction in the communication cost of sending data from T_L and T_E to the final join sensors. Therefore we cannot see much reduction in

Figure 3.15: Impact of Memory Capacity

the total communication cost.

Impact of Synopsis Size

Figure 3.16 shows the total communication cost with varied synopsis sizes and join selectivities. The memory capacity is 250×40 bytes, and the communication radius is 2 units. As shown in Figure 3.16, with the experiment setup, the smaller the synopsis size, the better the performance of the SNJ. Small synopses result in lower communication overhead during the preliminary join stage. Therefore, it is beneficial for synopses with small join attribute width compared to the data tuple size. We also observe that SNJ performs slightly worse than the centroid join when the synopsis size is greater than 30 bytes, indicating that the overhead of sending the synopsis is greater than the cost savings in data tuple transmission in this case.

We use an example to illustrate the impact of histogram accuracy on the communication cost. A histogram is constructed on the join attributes, which splits the domain of join attribute values into a number of *bins* and measures the frequencies of distinct join attribute values in each bin. Generally, the size of the histogram is dependent on the number of bins. If the number of bins decreases, the size of the histograms decreases as well.

Figure 3.16: Impact of Synopsis Size

Figure 3.17: Optimizing Semi-Join Approach with Histograms

Consider the example shown in Figure 3.17. In Figure 3.17(a), a preliminary-join is performed using histograms with bin size equal to 1. The preliminary join sensor s joins the histograms and for each value v , it compares f_a and f_b . If $f_a < f_b$, a sensor A is notified to send its tuples with the join attribute value v to sensor B . Sensor B is notified otherwise. For example, when $v = 1$, since $f_a = 3 > f_b = 2$, sensor B should be notified to send its 2 tuples with $v = 1$ to sensor A . When histograms are used as depicted in Figure 3.17(b), the preliminary join sensor joins histograms with bin size equal to 2, as opposed to bin size 1 in the previous case. Obviously the number of messages required for transmitting the histograms is reduced, because the number of entries in each histogram is smaller compared with that in the corresponding summary in Figure 3.17(a). However, consider the bin $v = \{1, 2\}$. f_a is now 3 and f_b is 7, hence the sensor A needs to send its local tuples with value 1 or 2 to sensor B , which requires 3 messages. However sensor B has less number of tuples (2 events for join attribute value $v = 1$) which will be involved in the join results. As a result, 1 more message is required for transmitting data tuples when using histograms in this case. Such additional costs caused by inaccurate approximate has to be considered as well if other types of synopses are employed.

3.6 Summary

In this chapter, we have presented a few strategies for join processing in sensor networks including naive join, centroid join, sequential join. We have described our novel in-network join strategy, SNJ, which reduces communication cost by eliminating non-candidate tuples in the early stage of the join processing using synopses. We implemented SNJ in a simulation environment. Our experiment showed that SNJ outperforms other join strategies and is an ideal solution to efficiently correlate data collected by sensors. SNJ outperforms the centroid join strategy when join selectivity is small. The effect of synopsis accuracy was also measured. In future work, we plan to develop methods for pre-computing synopsis accuracy for optimal join performance. Processing complex join queries is another direction to be pursued.

Chapter 4

An Implementation of Synopsis Join Strategy

In the previous chapter, we have described the general strategies for evaluating join queries in sensor networks and presented a synopsis join strategy for reducing communication cost. In this chapter, we present an implementation of SNJ in a simulation environment.

4.1 Query Dissemination

In the query dissemination phase, the query is disseminated to the regions $\mathcal{R}_L$ and $\mathcal{R}_E$, where T_L and T_E are stored, respectively. During the query dissemination, a *routing tree* for each region is constructed for routing query to all sensors of T_L or T_E and synchronizing message transmissions among them. In the following discussion, we focus on sensors in $\mathcal{R}_L$ and the distributed table T_L . Sensors in $\mathcal{R}_E$ carry out similar operations. The process is depicted in Figure 4.1.

Figure 4.1: Routing tree construction

A join query Q specified by a user is injected into the network at a sensor $\hat{s}$ referred to as the *query sink*. Q contains the geographical information of $\mathcal{R}_L$ and $\mathcal{R}_E$ (e.g., the shape and geographical coordinates), from which $\hat{s}$ computes two locations p_l and p_e for $\mathcal{R}_L$ and $\mathcal{R}_E$, respectively. The sensor nearest to p_l is the root of the routing tree in $\mathcal{R}_L$, denoted as $root(\mathcal{R}_L)$. Similarly, $root(\mathcal{R}_E)$ is the sensor nearest to p_e .

We choose the centers of $\mathcal{R}_L$ and $\mathcal{R}_E$ to be p_l and p_e respectively although other points could be used as well. In a dense network, this allows us to find a sensor that is nearest to p_l (or p_e), and reachable by other sensors in the corresponding query region $\mathcal{R}_L$ (or $\mathcal{R}_E$).

Once p_l and p_e are derived, the sink sends two query messages to p_l and p_e , respectively, using a location routing protocol GPSR [KK00, RKL⁺02], which ensures that the query messages will arrive at $root(\mathcal{R}_L)$ and $root(\mathcal{R}_E)$. For $root(\mathcal{R}_L)$, upon receiving the query message, it sets itself to be the root of the routing tree in $\mathcal{R}_L$, and starts broadcasting the query message to its neighboring sensors. When a sensor receives the query message Q , it rebroadcasts Q if it is located in $\mathcal{R}_L$, which ensures that all sensors in $\mathcal{R}_L$ will receive Q . Note that we assume a dense network where all sensors S_L in $\mathcal{R}_L$ are able to receive the broadcast query message.

A sensor s receiving a the broadcast query Q originated from another sensor s_p responds by sending an acknowledgement message to s_p , registering itself to be one of s_p 's child nodes. s also stores a pointer to the parent node s_p for Q . If multiple query messages are received (due to rebroadcasting) before an acknowledgement is sent, s can select any of the originators as the parent node based on different criteria, e.g., energy, physical distance. In this paper, s responds to the first query message it receives since it indicates a short path to the root of the routing tree. On receiving the acknowledgement message from s , s_p adds s to its child list. At the end of broadcasting process, a routing tree is constructed in region $\mathcal{R}_L$.

4.1.1 Synopsis Generation

Each sensor generates a *local synopsis* upon receiving the query Q . A *synopsis* $H(T)$ is a small digest of a table T . We denote $H(T_l)$ as the local synopsis of a sensor s_l storing a local table T_l , where s_l denotes a sensor located in $\mathcal{R}_L$. The advantage of a synopsis is that it is so compact that it can represent a large table to perform some expensive operations

at low costs. For example, a histogram of a table can be used for counting queries. The cost of reading the entire table can be avoided by accessing the statistical information in the histogram. A synopsis can be in many forms such as histograms, bloomfilters, wavelets, etc.. In this paper, we simply use histograms but our technique can be easily extended to other forms of synopses.

4.2 Preliminary Join

The *preliminary join* stage eliminates non-candidate tuples by joining local synopses generated in the synopsis generation stage. A set of sensors are designated to perform preliminary join and are referred to as *preliminary join sensors*.

Selecting preliminary join sensors We select a set of preliminary join sensors S_I from all sensors in the network except the source sensors (i.e., S_L and S_E) and the query sink. Ideally, S_I should be the set of sensors that minimize the cost of sending local synopses to S_I . However, finding the optimal S_I is a hard problem without knowing the exact data distribution and network topology.

To simplify the selection process, we select a square region $\mathcal{R}_I$ which lies on the line connecting the centers of $\mathcal{R}_L$ and $\mathcal{R}_E$ (see Figure 4.2). Sensors in $\mathcal{R}_I$ are then selected as the preliminary sensors S_I . We need to determine the center of $\mathcal{R}_I$ and its side length.

Let $area(\mathcal{R})$ denote the area of a given region $\mathcal{R}$, and $|T|$ denote the number of tuples in a given table T . The center of $\mathcal{R}_I$ is first determined using Equation 4.1.

$$center(\mathcal{R}_I) = p_l \cdot (1 - f) + p_e \cdot f, \text{ where} \quad (4.1)$$

$$f = \frac{area(\mathcal{R}_E) \sum_e |H(T_e)|}{area(\mathcal{R}_E) \sum_e |H(T_e)| + area(\mathcal{R}_L) \sum_l |H(T_l)|}$$

The total size of the local synopses from $\mathcal{R}_L$, $\sum_l |H(T_l)|$, is gathered by $root(\mathcal{R}_L)$. Each leaf sensor of the tree sends the size of its local synopsis to the parent sensor. The parent sensor sums up the sizes received from the child sensors and its own local synopsis size, and sends the number to its own parent. The process repeats until $root(\mathcal{R}_L)$ collects and computes the total size of the local synopses in region $\mathcal{R}_L$. Once $\sum_l |H(T_l)|$ is determined, $root(\mathcal{R}_L)$ transmits it to $root(\mathcal{R}_E)$. Similarly, the total size of local synopses in region $\mathcal{R}_E$

Figure 4.2: Preliminary join sensors selection

is computed and transmitted to $root(\mathcal{R}_L)$.

Note that Equation 4.1 is chosen to minimize not only the transmission cost of local synopses, but also the likelihood of using the source sensors for synopsis join. It is determined by having roughly equal costs for $\mathcal{R}_L$ and $\mathcal{R}_E$. Otherwise, the sensors selected for preliminary join may turn out to be the source sensors, increasing their traffic and depleting their energy at a faster rate.

Next, we determine the side length of $\mathcal{R}_I$ by estimating $area(\mathcal{R}_I)$ based on network density. Given $\sum_l |H(T_l)|$ and $\sum_e |H(T_e)|$, the number of preliminary join sensors $|S_I|$ is approximately $(\sum_l |H(T_l)| + \sum_e |H(T_e)|)/M$, where M is the average free memory space per sensor. $area(\mathcal{R}_I)$ is hence $|S_I|/d$, where d is the network density. The side length of $\mathcal{R}_I$ is $\sqrt{area(\mathcal{R}_I)}$.

$\mathcal{R}_I$ is therefore defined as the square with edge length $\sqrt{area(\mathcal{R}_I)}$ centered at $center(\mathcal{R}_I)$. The sensor that is nearest to $center(\mathcal{R}_I)$ is designated as the *preliminary join coordinator*, denoted as $\tilde{s}$.

Once $root(\mathcal{R}_L)$ and $root(\mathcal{R}_E)$ computes the center and side length of $\mathcal{R}_I$, the information is distributed to all sensors in $\mathcal{R}_L$ and $\mathcal{R}_E$ using the two routing trees.

Transmitting local synopses Upon receiving the geographical information of $\mathcal{R}_I$, the source sensors transmit the local synopses to sensors in $\mathcal{R}_I$. To balance the processing loads of sensors in S_I , we hash the local synopses into $|S_I|$ grids¹ based on the join attribute value. Hence, for a sensor $s_l \in S_L$ ($s_e \in S_E$), $|S_I|$ partitions of $H(T_l)$ ($H(T_e)$) are constructed, one for each grid. We denoted the partition of $H(T_l)$ ($H(T_e)$) created for grid g by $H(T_l^g)$ ($H(T_e^g)$). Each $H(T_l^g)$ ($H(T_e^g)$) is transmitted from s_l (s_e) directly to the center of g

¹Since the exact value of $|S_I|$ is unknown, the source sensors approximate the value by $(\sum_l |H(T_l)| + \sum_e |H(T_e)|)/M$

4.2. PRELIMINARY JOIN

81

using GPSR. The sensor that receives local synopsis partition(s) is then designated as a preliminary join sensor and thus belongs to S_I .

Sensors in S_I are required to know when they have received all local synopses from $\mathcal{R}_L$ and $\mathcal{R}_E$. To achieve this, we make use of routing trees in $\mathcal{R}_L$ and $\mathcal{R}_E$. For $\mathcal{R}_L$, when a sensor s_l in $\mathcal{R}_L$ finishes sending local synopsis to its parent sensor s_p , it also sends an *end of synopsis transmission* (EST) message to s_p . s_p in turn sends an EST message to its own parent sensor if it has finished sending out its local synopsis and received EST messages from all its child sensors, indicating that all sensors in its sub-tree have finished transmitting their local synopses. The EST message is therefore propagated to the root sensor $root(\mathcal{R}_L)$, which in turn notifies the preliminary join coordinator $\bar{s}$ with an EST message. The procedure is also carried out by sensors in $\mathcal{R}_E$ and $\bar{s}$ finally receives EST messages from both $root(\mathcal{R}_L)$ and $root(\mathcal{R}_E)$. It then broadcasts an EST message to all sensors in $\mathcal{R}_I$. Sensors in S_I can now start joining the local synopses they have received earlier.

Note that it may happen that $\bar{s}$ receives the EST messages from $root(\mathcal{R}_L)$ and $root(\mathcal{R}_E)$ before all sensors in S_I receive all local synopses, due to network delay or message loss. To tackle this problem, an EST message sent by the root node include the expected number of tuples each sensor in S_I should receive. Therefore if a sensor in S_I received less tuples, it can either request for re-transmission, or simply wait until timeout.

Joining local synopses Local synopses are joined in preliminary join sensors S_I to determine for each sensor in $\mathcal{R}_L$ and $\mathcal{R}_E$ the candidate tuples.

A preliminary join sensor $s_i \in S_I$ combines and stores all local synopses received in two tables, H_i^L and H_i^E , for those synopses from $\mathcal{R}_L$ and $\mathcal{R}_E$, respectively. When s_i receives an EST message, it joins H_i^L and H_i^E on the join attribute. The resultant table is denoted as $\hat{H}_i$. A tuple in $\hat{H}_i$ is a triple $\langle v, c_l, c_e \rangle$, where c_l and c_e are non-zero numbers of tuples from T_L and T_E with join attribute equals v , respectively. Tuples of T_L and T_E with join attribute value v are candidate tuples.

For each triple $\langle v, c_l, c_e \rangle$, a *final join sensor* $s_f(v)$ is determined such that the total cost of transmitting all candidate tuples to $s_f(v)$ and the join results from s_f to the query sink $\hat{s}$ is minimized. Specifically, given v , s_i determines all sensors in S_L that have tuples with

join attribute value v , denoted as $S_L(v)$. $S_E(v)$ is obtained similarly. s_i then computes the location of the final join sensor $s_f(v)$ for the value v , which is denoted as $p_{s_f(v)}$.

$p_{s_f(v)}$ is approximately the *generalized fermat point* [GR65] of the triangle formed by centers of $S_L(v)$ and $S_E(v)$, $c(S_L(v))$, $c(S_E(v))$, and $\hat{s}$, which minimizes the cost function² shown in Equation 4.2.

$$c_l \cdot \text{dist}(c(S_L^v), p_{s_f(v)}) + c_e \cdot \text{dist}(c(S_E^v), p_{s_f(v)}) + c_l \cdot c_e \cdot \text{dist}(p_{\hat{s}}, p_{s_f(v)}) \quad (4.2)$$

Sending notifications Once the preliminary join sensors S_I have determined the common join attribute values, as well as the location of the final join sensor for each join value, they pass the information back to the source sensors S_L and S_E using *notification* (NT) messages. Each NT message contains a list of value-location pairs. When a sensor s_l (or s_e) receives an NT message, for each pair $\langle v, p_{s_f(v)} \rangle$ in the message, s_l (or s_e) sends all tuples in its local table T_l having the join attribute value v to the location $p_{s_f(v)}$, where $p_{s_f(v)}$ is the location of the final join sensor $s_f(v)$. The NT message is tailored for each individual sensor, i.e., each sensor s_l receives NT messages containing exactly the join attribute values of the candidate tuples stored in s_l .

4.3 Final Join

In the final join stage, final join sensors joins candidate tuples as they arrive. Specifically, a final join sensor s_f maintains two tables for the value v , $T_l^f(v)$ and $T_e^f(v)$, where $T_l^f(v)$ stores the candidate tuples from T_L and $T_e^f(v)$ stores candidate tuples from T_E with join attribute value v . When a candidate tuple t_l^v (or t_e^v) is received, it is joined with $T_e^f(v)$ (or $T_l^f(v)$), and the join results are transmitted to the query sink.

The final join sensors stop the join operation if the number of result tuples they have produced matches with the expected number, which is collected by the preliminary join coordinator and transmitted to all final join nodes.

² $\text{dist}(a, b)$ is the distance between two points a and b

Parameter	Value
network size	$500 \times 500m$
communication radius	$40m$
data tuple size	40 bytes
synopsis tuple size	10 bytes
join selectivity	$0.0001 - 0.01$
histogram bin size	$1 - 5$
table size ($ T_L $ and $ T_E $)	both 1000 tuples
join attribute range	$1 - 100$

Table 4.1: Summary of parameters

4.4 Experiments

To evaluate the performance of the in-network SNJ strategy, we implemented it in a network simulator *J-Sim* [jsi]. In the simulation-based experiments, we compared SNJ with a centroid join (CNJ) scheme. In CNJ, we assume the query sink $\hat{s}$ is able to estimate a selectivity value of the join query, given some pre-computed statistics about the data stored in the network. [CG05]. Therefore, based on the selectivity, a group of final join sensors can be determined, whose center lies on the generalized Fermat’s point of the triangle formed by $\hat{s}$ and the centers of $\mathcal{R}_L$ and $\mathcal{R}_E$. In other words, the final join sensors in CNJ are good approximations of the optimal sensors that minimize the total transmission costs of routing the tuples from T_L and T_E to the final join sensors, and routing the join results from the final join sensors to the query sink.

We created a sensor network consisting of 400 sensors, uniformly placed in a $500m \times 500m$ area. The communication radius is $40m$. The query sink is located near the top center of the network area. The area $\mathcal{R}_L$ is a $60m \times 60m$ square, centered at the bottom-left corner of the network, while $\mathcal{R}_E$ is of the same size located at the bottom-right corner. The distance between the centers of $\mathcal{R}_L$ and $\mathcal{R}_E$ is $320m$. A summary of the query and network parameters used in our experiments is shown in Table 4.1.

4.4.1 Impact of Join Selectivity

We varied the join selectivity to observe the performance of SNJ. We generated data sets with join selectivity ranging from 0.0001 to 0.01. Join selectivity is defined as $|T_L \bowtie T_E|/(|T_L| \cdot |T_E|)$. For each join selectivity, 5 datasets were generated and the average performance was measured. The experiment results are depicted in Figures 4.3 and 4.4,

Figure 4.3: Cost vs. join selectivity (0.0001 - 0.001)

Figure 4.4: Cost vs. join selectivity (0.001 - 0.01)

4.4. EXPERIMENTS

Figure 4.5: Detailed cost breakdown for SNJ (0.0001 - 0.001)

Figure 4.6: Detailed cost breakdown for SNJ (0.001 - 0.01)

where the x axis shows the join selectivity, while the y axis shows the total number of messages transmitted and received by sensors. As depicted in Figure 4.3, SNJ consistently outperforms CNJ for selectivities ranging from 0.0001 to 0.001. CNJ has to route every data tuple in $\mathcal{R}_L$ and $\mathcal{R}_E$ to the final join sensors, while in SNJ only candidate tuples are transmitted. Figure 4.5 shows the detailed cost breakdown of different types of network messages from SNJ. The cost of transmitting local synopses is almost constant because their total size does not vary with the join selectivity. It is only affected by the size of the tables (1000 tuples each), and the join attribute range ($[0, 99]$). It is clear that when selectivity is low, local synopses account for a significant portion of the total cost. However, this overhead is compensated by the reduction of transmissions of data tuples, therefore, the total cost is small compared to CNJ. For all selectivities, the overheads of control messages and notification routing are negligible.

When the join selectivity grows to the range $[0.001, 0.01]$, the cost of SNJ quickly approaches the centroid join (see Figure 4.4). The reason is that the cost of transmitting the join results now starts dominating the total communication cost. In addition, since most of the tuples are now candidate tuples, the savings due to preliminary join is little as opposed to its overhead of local synopses transmissions. When selectivity is very high (0.009 and 0.01), SNJ performs worse than CNJ since both strategies select final join sensors around the query sink, and transmit almost the same amount of data tuples. Preliminary join in SNJ therefore becomes a cost overhead.

Note that when selectivity is greater than 0.001, the size of join results is $1000 \times 1000 \times 0.001 = 1000$, which is fairly large compared to the original table. In real-world sensor network applications, this is undesirable due to the extreme high volume of data transmission. Instead, some means should be taken to reduce the data size, for example, using top- k join. Therefore, in the rest of experiments, we only show results for selectivity in $[0.0001, 0.001]$.

4.4.2 Impact of Histogram Accuracy

We varied the accuracy of histograms to observe its impact on the performance of SNJ. The size of histograms is inversely proportional to bin size. If bin size is small, a histogram is large in size, and vice versa. The accuracy of the histogram however degrades when bin size increases. We changed the bin size of the histograms in the experiments and the

Figure 4.7: Impact of histogram accuracy

results are shown in Figure 4.7. When bin size equals to 1, SNJ incurs overheads due to more local synopses messages. When bin size is large (4 and 5), the histogram accuracy degrades causing an increase of candidate tuples that are not part of join results. They are transmitted for final join contributing to additional messaging overheads. The best performance was observed when bin size was 3 which gives a good balance between the histogram size and its accuracy. The determination of a good histogram bin size for the synopses is therefore an interesting problem to be investigated in our future work.

4.4.3 Hotspots

We evaluated the hotspots usage in the network when processing the join query. The result is shown in Figure 4.8. We measured the number of messages transmitted and received by each sensor in the network. The message counts are then normalized and the standard deviation is computed, which measures the variation of the message counts of each sensor from the average message counts. A large standard deviation value indicates that the sensor network suffers from more hotspots. From the figure we can see that SNJ outperforms CNJ consistently. For instance, when selectivity is 0.0004, for SNJ, there are 19 sensors contributing 50% of total messages, while for CNJ, there are 13 sensors contributing 50%

Figure 4.8: Hotspots

Figure 4.9: Impact of message losses

of total messages. With more sensors contributing fewer messages compared with CNJ, sensors using the SNJ strategy can have a long lifetime. Notice that when selectivity is low, the standard deviation value is high. This is because most of the sensors are transmitting small number of messages while only the preliminary join sensors experience heavier traffic. However, as the selectivity increases, more sensors are involved in transmitting the increasing data tuples and join results. Therefore the load of the sensors becomes more and more balanced, resulting smaller standard deviation values.

4.4.4 Impact of Message Losses

Figure 4.9 depicts the impact of message losses on join result accuracy under a simple loss model in which each message is dropped with a probability of 0.01. The more hops a message travels, the high probability that it is lost. To reduce the effect of losing a local synopsis message, SNJ transmits multiple copies of a local synopsis message. The number of copies is determined by $\lceil 1/(1 - P)^k \rceil$, where P is message loss probability and k is the estimated hop distance a message may travel³. SNJ produces smaller error than CNJ when selectivity is low (0.0001 and 0.0002). In these cases, final join sensors are far from the sink and near the source nodes. Therefore the probability of a source tuple loss is lower than that of a result tuple loss, which in turn produces more result tuples because the benefit of generating more join results using more complete source tuples is greater than the effect of result tuple losses. On the other hand, when selectivity is high, the final join sensors are around the sink, the losses of source tuples result in less join results, therefore degrade the result accuracy of SNJ. Join sensors in CNJ are located around the center of the triangle formed by the source regions and the sink, and the change of selectivity has less impact on their locations compared to the case of SNJ. Therefore, its errors are relatively stable.

³We observed that this formula is adequate for getting satisfactory join results, while keeping the total cost small compared to CNJ.

Chapter 5

Generalised Synopsis Join Processing

In this chapter, we generalise the synopsis join strategy to support join queries over arbitrary source regions. The synopsis join strategy presented in the previous chapter only works for rectangle source regions that are far apart from each other. This limitation restricts the applicability of synopsis join in some real-world scenarios. We therefore revise the preliminary join step of synopsis join to select preliminary join sensors based on the locations of source regions and memory capacities of sensors. We carry out experiments to evaluate the performance of generalized synopsis join strategy.

5.1 Overview of Generalised Synopsis Join

The synopsis join strategy introduced in Chapter 3 assumes two rectangular source regions that are far apart from each other. In real-world scenarios, this assumption may not be appropriate. The shapes of source regions may not be rectangle. In addition, the source regions may be very close or even overlapping. In an extreme case, two source tables are stored in the entire sensor network, i.e., the two source regions cover the entire sensor network area and overlap with each other. This extreme case requires a generalised strategy to work without prior source region knowledge.

We aim to support source regions of arbitrary shapes and locations. In addition, we drop the requirement that the two source regions must be disjoint. The areas of the source

Figure 5.1: Source Regions of arbitrary shapes and locations

regions are not restricted. Figure 5.1 depicts an example scenario. In the following, we firstly analyze the reasons why previous synopsis join strategy cannot handle arbitrary source regions.

The arbitrary shapes and locations of source regions affect mainly the preliminary join phase of our previous simple SNJ strategy. In the simple SNJ strategy, we assume the source sensors do not participate in the preliminary join. Therefore, the preliminary join sensors are selected from sensors that reside between the two source regions. This implicitly requires the two source regions are non-overlapping. If they overlap, no preliminary join sensors could be selected according to the selection criteria. In addition, the distance between the two source regions is assumed to be large enough so that there are sufficient non-source sensors between the two regions to be selected as the candidates for preliminary join sensors. The shapes of the source regions are restricted to rectangles, so that it is easier to calculate the areas of the source regions when determining the locations of the preliminary join sensors.

The abandoning of the non-overlapping assumption suggests that some source sensors must also be used as preliminary join sensors. Previously, the number of preliminary join sensors required is computed based on a known parameter m denoting the memory capacity of each sensor. If source sensors are allowed to carry out preliminary join, the memory space available for preliminary join will vary among source sensors since they may store local tables

5.1. OVERVIEW OF GENERALISED SYNOPSIS JOIN

93

Figure 5.2: Overview of Generalised SNJ strategy

of different sizes. Therefore, an adaptive algorithm is required for selecting preliminary join sensors according to their available memory space. Once the preliminary join sensors are determined, the remaining steps of the synopsis join strategy can be performed as described in Chapter 3.

Figure 5.2 depicts the generalised SNJ strategy to cope with the new source region assumption. The generalised SNJ strategy remains the basic steps but some changes are required for the query dissemination and preliminary join phases. The source regions can be described using geographical coordinators, from which the sink is able to determine a center for each source location ¹. In the query dissemination phase, the centers are disseminated together with the join query to the two source regions $\mathcal{R}_L$ and $\mathcal{R}_E$ which can be overlapping. The sink also determines a preliminary join coordinator which is responsible for redirecting local synopses to preliminary join sensors.

During the preliminary join phase, local synopses from the source region are routed towards the preliminary join coordinator, which redirects the local synopses to preliminary join sensors that are selected adaptively based on memory conditions of these sensors.

We also develop a *ranked caching* scheme to balance the load of the preliminary join sensors. The ranked cache enables early redirection of local synopses to intended preliminary join sensors without involving preliminary join coordinator. The assumption is that caching is always possible for every sensor since a cache is very small compared to data tuples stored by sensors. We assume a chunk of memory space is pre-allocated for cache and the cache size never exceeds the pre-allocated space, so that when calculating the available memory space of a sensor, we do not have to consider the changing size of its cache, under the assumption that the cache is usually small and only occupies a small portion of a sensor's

¹For source regions with irregular shapes, it can be difficult to compute their centers. A simple approach is to compute the center of the bounding box of the corresponding source region.

memory.

5.2 Revised Preliminary Join

As stated earlier, we need to revise the preliminary join step in order to support join queries over arbitrary join regions by dynamically selecting preliminary join sensors given their available memory capacity and locations. Our approach is to find a preliminary join coordinator which coordinates the selection of preliminary join sensors and directing local synopses to their corresponding preliminary join sensor where preliminary join is to be carried out.

The preliminary join coordinator is the sensor that is nearest to the midpoint of the centers of the two source regions. Specifically, given the two source regions $\mathcal{R}_{\mathcal{L}}$ and $\mathcal{R}_{\mathcal{E}}$, the preliminary join coordinator is $s(p)$, where p is the midpoint of $center(\mathcal{R}_{\mathcal{L}})$ and $center(\mathcal{R}_{\mathcal{E}})$, and $s(p)$ is the sensor that is nearest to p . The location p is determined by the sink upon insertion of the join query and disseminated to all source sensors together with the join query. As shown in the previous chapter, the preliminary join step incurs low communication cost. Therefore we can afford a non-optimal preliminary join coordinator. The potential performance degradation is the trade-off to pay for the benefit of flexibility.

In the revised preliminary join, the source sensors send their local synopses towards the location p using GPSR. When the preliminary join coordinator receives local synopses, it tries to select sensors in its close proximity as preliminary join sensors if there exist sensors with free memory space. Each preliminary join sensor is dedicated to handle local synopses with a particular range of values, so that the preliminary join coordinator can redirect the local synopses to corresponding preliminary join sensors. We allow the preliminary join coordinator to act as a preliminary join sensor, i.e., it is also able to store some local synopses. The consequence is that the preliminary join coordinator will only start redirecting local synopses when the number of distinct join attribute values of local synopses it has received exceeds its memory constraint ². In this scheme, the preliminary join coordinator has to receive all local synopses, which heavily overloads the preliminary join coordinator.

²The memory constraint is defined by the number of distinct join attribute values of local synopses. This simplifies our problem by allowing synopses tuples with the same join attribute value are always joined at the same preliminary join sensor.

We adopt a *ranked caching* scheme to reduce the load of preliminary join coordinator.

The preliminary join starts by source sensors sending their local synopses to the preliminary join coordinator upon they receiving the join query. When the preliminary join coordinator $s(p)$ receives a local synopsis, for each value-count pair $\langle v, c \rangle$, if $s(p)$ is not full, it stores $\langle v, c \rangle$ for processing. Otherwise, $s(p)$ hashes v to one of its neighbors s' , and redirects $\langle v, c \rangle$ to s' . Each neighbor of $s(p)$ therefore is responsible for receiving synopses with a set of values. After redirecting $\langle v, c \rangle$ to s' , $s(p)$ stores a value-sensor pair $\langle v, s' \rangle$ in its cache with a rank 1, so that if a subsequent pair $\langle v, c' \rangle$ is received, it is forwarded to s' . The rank indicates s' is one hop away from $s(p)$.

When all neighbors of $s(p)$ are full, $s(p)$ chooses a neighbor s'' that is responsible for redirecting $\langle v, c \rangle$ to neighbors of s'' . After a new sensor s''' is selected, $s(p)$ set up a cache entry $\langle v, s'' \rangle$ with rank 1, and s'' set up a cache entry $\langle v, s''' \rangle$ with rank 2. When a new pair $h(v)$ with value v arrives at $s(p)$, it follows the cache entries in $s(p)$ and s'' and is redirected to s''' . If it happens that on the way to $s(p)$, $h(v)$ is received by s'' , s'' will check its own cache and redirect $h(v)$ to s''' instead of routing $h(v)$ to $s(p)$. Therefore, the load of $s(p)$ can be reduced. The whole process is described in Algorithms 5.1 and 5.2.

5.2.1 Cache Updating Rules

In Algorithms 5.1 and 5.2, a cache exists for each sensor that has received a local synopsis message. The purpose of the cache is to enable early redirection of local synopses so as to reduce the load of preliminary join coordinator and its neighbors.

Each cache entry is associated with a rank which is useful when removing obsolete entries during cache updates. The rank of a cache entry $\langle v, s \rangle$ denotes for the value v the number of hops that the destination s is away from the preliminary join coordinator $s(p)$. A local cache is updated when

1. a new pair is redirected to a neighbor, or
2. a new cache entry is received from a neighboring sensor, or
3. a cache entry is received with a higher rank than the one currently in the local cache

Algorithm 5.1 *recvSynopsisMsg(msg, src, dst, rank(dst), lasthop)*

s_p: preliminary join coordinator*s*: sensor performing this function*src*: the source sensor of the synopsis message*dst*: the destination of the synopsis message*rank(dst)*: the rank of *dst*, initially 0*lasthop*: last hop of the synopsis message

```

1: for all pair  $\langle v, c \rangle$  in msg do
2:   if v is in cache then
3:     sk = cache.get(v)
4:     if rank(sk) > rank(dst) then
5:       forward  $\langle v, c, src, s_k, rank(s_k) \rangle$  to sk
6:       update neighbors cache with  $\langle v, (s_k, rank(s_k)) \rangle$  //NCU rule 2
7:     else if rank(sk) == rank(dst) then
8:       forward  $\langle v, c, src, s_k, rank(s_k) \rangle$  to sk
9:     else //outdated self cache
10:      if dst is s then
11:        store(v, c, src, dst, rank(dst), lasthop)
12:      else
13:        forward  $\langle v, c, src, dst, rank(dst) \rangle$  to dst
14:        update self cache with  $\langle v, (dst, rank(dst)) \rangle$  //LCU rule 3
15:      end if
16:    end if
17:  else if dst is sp then
18:    forward  $\langle v, c \rangle$  to sp
19:  else if dst is s then
20:    store(v, c, src, dst, rank(dst), lasthop)
21:  else
22:    forward  $\langle v, c, src, dst, rank(dst) \rangle$  to dst
23:    add  $\langle v, (dst, rank(dst)) \rangle$  to cache //LCU rule 2
24:  end if
25: end for

```

5.2. REVISED PRELIMINARY JOIN

97

Algorithm 5.2 $\text{store}(v, c, \text{src}, \text{dst}, \text{rank}(\text{dst}), \text{lasthop})$

s_p : preliminary join coordinator
 s : sensor performing this function
 src : the source sensor of the synopsis message
 dst : the destination of the synopsis message
 $\text{rank}(\text{dst})$: the rank of dst
 lasthop : last hop of the synopsis message

```

1: if  $s$  is full then
2:   find new location  $s_k = \text{hash}(v, \text{lasthop})$ 
3:    $\text{rank}(s_k) = \text{rank}(\text{dst}) + 1$ 
4:   add  $\langle v, (s_k, \text{rank}(s_k)) \rangle$  to cache //LCU rule 1
5:   update neighbors cache with  $\langle v, (s_k, \text{rank}(s_k)) \rangle$  //NCU rule 1
6: else
7:   store  $\langle v, c \rangle$  locally
8: end if

```

These two rules are referred to as *local cache update rules* (LCU rules). A sensor may update its neighboring sensors' caches if

1. a cache update occurs due to redirection of a new pair, or
2. a cache entry is received with a lower rank than the one currently in local cache

These two rules are referred to as *neighbor cache update rules* (NCU rules). Both the LCU and NCU rules have been incorporated into our algorithms as shown.

Explicit cache update using dedicated messages is a huge communication overhead which should be avoided. A simple solution is to embed the cache update messages in GPSR beacon messages, so that the cache update is carried out implicitly when sensors update each other, reducing the communication overhead. The drawback of this approach is that the caches are not updated immediately as they supposed to be because beacon messages are periodical. However, late cache updates do not affect the correctness of our cache update algorithm since the synopses pairs always follow paths that lead to the correct destination albeit late cache updates may introduce longer paths. Given the low data rate of sensors, the communication overhead caused by longer paths is negligible since once a cache is updated, all subsequent synopses pairs will follow the correct path.

5.2.2 Hashing

We have mentioned in previous sections that if a sensor s is full, s hashes a new pair $h(v)$ $\langle v, c \rangle$ to a neighboring node of s . The hashing algorithm is depicted in Algorithms 5.3 and 5.4.

Algorithm 5.3 $\text{hash}(v, s')$

v : value
 s' : incoming sensor
 s : sensor that received v
 $S = s_1, \dots, s_n - \{s'\}$: neighboring sensors of s

- 1: $S' = \text{nonfull}(S)$
- 2: **if** $S' \neq \phi$ **then**
- 3: $s_k = \text{choose}(s, s_p, S')$
- 4: **else**
- 5: $s_k = \text{choose}(s, s_p, S)$
- 6: **end if**
- 7: **return** s_k

Algorithm 5.4 $\text{choose}(s, s_p, S)$

s : local sensor
 s_p : preliminary join coordinator
 S : a set of candidate sensors

- 1: $S' = \{s' \in S \mid \text{dist}(s', s_p) > \text{dist}(s, s_p)\}$
- 2: choose $s_k \in S'$ with probability of $\text{area}(s_k) / \sum_{s_i \in S'} \text{area}(s_i)$
- 3: **return** s_k

The hash algorithm locates a neighboring sensor of sensor s to handle a value v . The resultant sensor must not be the sensor s' from which the pair $h(v)$ is received. Otherwise $h(v)$ may bounce between s and s' , and never reach its destination.

The hash algorithm firstly selects a non-full sensor based on the value of v . All non-full neighbors of s are possible candidates. The hash algorithm simply applies a hash function on v and determines a non-full neighboring sensor.

In case all neighboring nodes are full, the hash algorithm tries to the neighboring node that expands the preliminary join region with a high probability. Simply speaking, it finds the sensor that is further than s' from the preliminary join coordinator. This ensures that the preliminary join region keeps expanding so that a sensor can always be found to store new incoming local synopses.

5.3. EXAMPLE

99

There are however cases when the above approach does not work. We illustrate this by example. Consider the sensor network shown in Figure 5.3, the preliminary join coordinator is located near the network boundary. Now sensor s has 3 candidates, s_1, s_2, s_3 . Any one of the three sensors can be selected by the hash algorithm. However, if s_3 is selected, we can observe that it is possible that the sensors in region A_3 would be quickly used up by preliminary join sensors, in which case it is not possible to find preliminary join sensors for other incoming local synopses if all preliminary join sensors in A_3 are full. Therefore, it is desirable to select s_1 with a higher probability, and s_3 with a lower probability, assuming that $area(A_1) > area(A_2) > area(A_3)$. Therefore, we choose to select s_1 with probability $area(A_1)/(area(A_1) + area(A_2) + area(A_3))$.

We now examine how the areas A_1, A_2 and A_3 can be determined. We firstly determine a region A which sensor s is responsible for (see Figure 5.4(a)). Given all non-candidate sensors, we find s_4 and s_5 that form the smallest angles a and b with s , respectively. A region A with angle $(a + b)/2$ is chosen to be the *responsible region* of s , as shown in Figure 5.4(b). If there is only one non-candidate sensor (which is the last hop of the synopsis message), the angle $a + b$ is set to 360° .

For candidate sensors s_1, s_2 and s_3 , we need to determine a responsible region for each sensor as shown in Figure 5.4(c). For non-boundary sensor s_2 , the boundaries of its responsible region A_2 are lines that partition $\angle s_1 s_p s_2$ and $\angle s_2 s_p s_3$ to two equal halves, respectively. Therefore the angle at s_p of region A_2 is $a_1/2 + a_2/2$, where $a_1 = \angle s_1 s_p s_2$ and $a_3 = \angle s_2 s_p s_3$.

The boundary edge E_1 of A_1 is the line that satisfies $\angle E_1 s_p E_3 = a_1$. Similarly, E_2 is the line that satisfies $\angle E_4 s_p E_3 = a_3$ (see Figure 5.4(d)). We then choose two edges from the boundary edges E_1, E_2, E_5, E_6 to be the final boundary edges of A_1 and A_3 , which are the two edges that form the largest angle. In this case, E_1 and E_2 are selected as the final boundary edges for A_1 and A_3 , respectively.

5.3 Example

Consider the example shown in Figure 5.5. In the figure, the black dot denotes the preliminary join coordinator s_p . When it receives a pair $h(3) = \langle 3, 4 \rangle$, it hashes $h(3)$ to s_2 and redirects it to sensor s_2 , since it is full. In addition, a cache entry $\langle 3, (s_2, 1) \rangle$ is inserted

Figure 5.3: Hashing Algorithm in Revised Preliminary Join

into the cache of s_p based on LCU rule 1, indicating that all pairs with value 3 should be routed to s_2 , which is of 1 hop away from s_p . According to NCU rule 1, it also updates its neighbor s_1 with the same cache entry, so that s_1 is able to redirect any pair with value 3 to s_2 .

When s_2 receives the pair $h(3)$, it finds itself full. Therefore, s_2 hashes $h(3)$ to s_4 and redirects $h(3)$ to s_4 , and add a cache entry $\langle 3, (s_4, 2) \rangle$ (LCU rule 1), indicating that s_4 , the destination of all pairs with value 3, is 2 hops away from s_p . The neighbors of s_2 , including s_p and s_3 , are updated with the cache entry (NCU rule 1).

Suppose that now s_1 receives a new pair $h'(3) = \langle 3, 6 \rangle$, whose destination is s_p . s_1 intercepts the message and tries to redirect the pair if there is a cache hit, so that the message can bypass s_p , reducing the load of s_p . According to the cache of s_1 , $h'(3)$ should be forwarded to s_2 as shown in Figure 5.5. $h'(3)$ is therefore routed via s_3 towards s_2 using GPSR. However, when s_3 receives $h'(3)$, it finds there is a cache entry $\langle 3, (s_4, 2) \rangle$ whose rank is greater than rank 1 in the cache entry of previous hop s_1 . According to NCU rule 2, s_3 updates its neighbors s_1 , s_2 and s_4 , informing them to update the cache entry with value 3 to $\langle 3, (s_4, 2) \rangle$. When s_1 receives the cache update, it updates its local cache according to LCU rule 3. Thus, subsequent pairs with value 3 are redirected to s_4 .

5.4 Proof of Correctness

We prove the correctness of the preliminary join algorithm by contradiction.

Suppose a pair $h(v) = \langle v, c \rangle$ does not arrive at its desired destination s , instead, it arrives at s' .

5.4. PROOF OF CORRECTNESS

101

Figure 5.4: Determining A_1 , A_2 , and A_3

Since s is the desired destination, if $h(v)$ arrives at s' , there must be a cache entry $\langle v, (s', \text{rank}(s')) \rangle$ pointing to s' , following which $h(v)$ is routed. Since s is the desired destination, there must also be a cache entry $\langle v, (s, \text{rank}(s)) \rangle$ pointing to s . In addition, $\text{rank}(s')$ must be lower than $\text{rank}(s)$, since s is the desired destination and it should have the highest rank according to the definition of rank.

Along the path to s' , $h(v)$ may follow either of the two cache entries, or both of them. We discuss the three possibilities below:

1. $h(v)$ followed $\langle v, (s, \text{rank}(s)) \rangle$, but it did not encounter $\langle v, (s', \text{rank}(s')) \rangle$. This will result in $h(v)$ arriving at s , which contradicts our assumption.
2. $h(v)$ followed $\langle v, (s', \text{rank}(s')) \rangle$, but it did not encounter $\langle v, (s, \text{rank}(s)) \rangle$. According to the cache update rules, if there is a cache entry $\langle v, (s', \text{rank}(s')) \rangle$, s' must be full and is not able to store pairs with value v . Therefore there must exist a cache entry $\langle v, (s'', \text{rank}(s'')) \rangle$ at sensor s' , where $\text{rank}(s'') > \text{rank}(s')$, indicating that s' is not able to store any pairs with value v and those pairs are passed to s'' . Therefore, once $h(v)$ arrives at s' , it will be routed to s'' . Since it is not impossible that $s'' = s'$, s' is

Figure 5.5: Ranked Cache

not the final destination.

3. $h(v)$ followed both $\langle v, (s, rank(s)) \rangle$ and $\langle v, (s', rank(s')) \rangle$. In this case $h(v)$ must arrive at s since the rank of $rank(s)$ is the highest.

All of the three possibilities lead to a contradiction with our assumption. Therefore, $h(v)$ must arrive at the desired destination s under our routing and caching schemes.

5.5 Experiment Results

In this section, we present the performance of synopsis join strategy with revised preliminary join. We measure the total number of messages resulted from synopsis join, excluding the cache update messages since they are embedded in beacon messages, and compare the performance with centroid join. We do not run naive join in this experiment because naive

join is a special case of centroid join and always performs no better than centroid join.

5.5.1 Experiment Setup

We use the same sensor network as in previous chapter, where there are 400 sensors distributed in a $500m \times 500m$ area. The communication radius is set to 40m. The sink is located near the top center of the network area. Two source regions $\mathcal{R}_L$ and $\mathcal{R}_E$ are specified in a query. $\mathcal{R}_L$ and $\mathcal{R}_E$ are still rectangles, but we did not use any property that is specific to rectangles, therefore other shapes for source regions can be used as well.

We varied the locations of the source regions to change the overlapping areas of the two source regions. The performance is measured by the total number of messages incurred during the join processing. The experiment results are averages over 5 datasets for each selectivity.

5.5.2 Impact of Selectivity

In this section, we measure the performance of the generalised synopsis join strategy under different join selectivities. We also varied the locations of the source regions to observe the effect of overlapping regions. Figure 5.6 shows the communication costs when the two source regions are adjacent to each other without overlap. The generalised synopsis join outperformed centroid join for all selectivities. The cost of generalised synopsis join is about half of that of centroid join when selectivity is 0.0001. It is about 1/3 of the cost of centroid join when selectivity increases to 0.001, showing greater scalability. The performance of generalised synopsis join is slightly better than synopsis join in previous chapter. For example, at selectivity 0.0005, SNJ incurred 8698 messages, while the generalised SNJ only incurred 6719 messages in total. The reason is that the two source regions $\mathcal{R}_L$ and $\mathcal{R}_E$ are now close to each other, reducing the cost of local synopses transmission. Similarly, the centroid join with the new setup performs better than that in previous chapter, due to closer source regions.

Figure 5.7 and Figure 5.8 depict the performance of the generalised synopsis join and centroid join when the two source regions overlap 50% and 100% of each other, respectively. The results are similar to the case when there is no overlapping of the two source regions. These results show that the generalised SNJ strategy copes well with overlapping source

(a)

(b)

Figure 5.6: Overlapping region 0%

regions.

Figure 5.12 shows all cost curves in the same figure. We can observe that as the overlapping area increases, centroid join performs better since the closer the two source regions, the less cost is needed for routing data to join sensors. The performance of the generalised synopsis join changes very little indicating that the cost savings caused by closer source regions are negligible compared to the total cost.

Figure 5.10 shows the detailed cost breakdown when the two source regions do not overlap. When selectivity increases, source tuples gradually dominate the total cost. The cost for routing local synopses remains constant since the join selectivity has little impact on the locations of preliminary join sensors. The overhead incurred during the query processing is negligible compared to the total cost.

5.5.3 Impact of Source Region Size

Figure 5.11 depicts the total communication costs incurred for different source region sizes. In this set of experiments, we varied the size of $\mathcal{R}_L$ so that it is 2, 4, and 6 times of the size of $\mathcal{R}_E$, respectively. For generalised SNJ, the communication cost is lowest when $\mathcal{R}_L$ is 2 times of $\mathcal{R}_E$ in size. When $\mathcal{R}_L$ is 4 and 6 times of $\mathcal{R}_E$, the communication costs are about the same. The reason is that when the size of $\mathcal{R}_L$ increases, the amount of messages required for local synopses routing since the distance they travel is increased. This can be verified by comparing Figure 5.12(a) and Figure 5.12(b). The cost of local synopses increased from 3200 to 5000, approximately. The cost for routing result tuples is reduced, because the final join sensors are closer to the sink when $\mathcal{R}_L = 6 \times \text{area}(\mathcal{R}_E)$. This experiment show that new techniques may have to be designed to further reduce the overhead of routing local synopses in similar cases where the source regions are very large. Otherwise, it may be better just to use the centroid join strategy.

5.6 Summary

In this chapter we presented a generalised version of synopsis join to support join queries over arbitrary source regions. In the generalised synopsis join strategy, we revised the preliminary join stage, so that the selection of preliminary join sensors are adaptive to data

(a)

(b)

Figure 5.7: Overlapping region 50%

(a)

(b)

Figure 5.8: Overlapping region 100%

(a)

(b)

Figure 5.9: All experiment results for overlapping regions

5.6. SUMMARY

Figure 5.10: Detailed cost breakdown for overlapping region 0%

Figure 5.11: Region all

(a) $area(\mathcal{R}_L) = 2 \times area(\mathcal{R}_E)$

(b) $area(\mathcal{R}_L) = 6 \times area(\mathcal{R}_E)$

Figure 5.12: Detailed cost breakdown for different sized source regions

5.6. SUMMARY

111

distribution and sensors' memory states. The new preliminary join stage adopts a ranked caching scheme for efficient redirection of local synopses. With enhanced adaptivity, the generalised synopsis join strategy provides a robust mechanism for locating preliminary join sensors, and transmitting local synopses.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Recent technology developments have facilitated the production of new generation sensors which are small, wireless, battery-powered, but with data processing capabilities. Consisting of numerous such smart sensors, sensor networks can be deployed to sense, collect, and disseminate information in many kinds of applications. Query processing in sensor networks has become an important means to interact with sensor networks in an effective and efficient manner. In this report we have focused on join queries in sensor networks. The main contributions are as follows.

- We have surveyed some of the state-of-art routing protocols and storage schemes widely used in sensor networks.
- We have reviewed various types of queries in sensor networks and techniques that have been proposed in the literature for evaluating the queries. We have mainly focused on aggregate and approximate queries that have drawn great interest of researchers. Moreover, range queries that are specially for data-centric storage schemes were discussed.
- We have provided a definition of binary equi-join (BEJ) query in sensor networks, which we believe is useful both in providing comprehensive information about the environment the sensors are monitoring, and in generating correlating information collected by the sensors.

- We discussed a few possible strategies for evaluating binary equi-join queries in-network, including naive join, sequential join, centroid join. We performed cost analysis on these methods, and compared their advantages and disadvantages.
- We proposed a novel synopsis join strategy for processing binary equi-join queries. Synopsis are adopted to reduce the communication cost by eliminating non-candidate tuples before final join is performed. Excessive message transmission of non-candidate tuples is therefore avoided, at the cost of performing a preliminary-join operation, which is cheap in most circumstances. The cost analysis shows the effectiveness of our proposed method. We evaluated the performance of SNJ through thorough experiments, and the results showed SNJ is promising in terms of communication cost.
- We implemented SNJ in a simulation environment. We defined message transmission protocol which enables routing and processing synopses and data tuples. Experiments carried out in the simulation environment are closer to real-world scenarios and SNJ was proved to be outperforming other join strategies in most cases.
- We presented the generalised SNJ, an extension to SNJ to support arbitrary source regions. The proposed strategy facilitates join queries with overlapping and irregular-shaped source regions. Preliminary join sensors are dynamically selected based on their available memory capacity. Ranked caching further reduces communication cost.

6.2 Future Work

We discuss some of the limitations of our work, and point out directions to future work.

Join Queries We have been focusing on binary equi-join queries in sensor networks. Our method can be extended to support more complex join queries such as multi-way join, non-equi-join queries. This requires a revisit of the preliminary join phase of our query processing: instead of simple histograms, other types of synopses are required for performing non-equi-join. In addition, continuous joins should also be investigated for monitoring applications.

Approximate Join Queries Processing join queries require large amount of resources in collecting, processing, and transmitting data. Approximate queries provide a potential

6.2. FUTURE WORK

115

means to minimize the query cost. We are particularly interested in approximate join queries. The first research issue is the definition of approximate join queries. The design objective is to be able to answer approximate join queries with error bound and confidence levels, with a relatively low communication cost.

Query Plan Optimization For complex queries that involve multiple operators, a query plan is needed. Optimization of the query plan requires cost analysis and comparison. Statistics about the sensor networks are needed during the optimization. The methods for gathering statistical data will therefore be an interesting research topic.

Appendix A

Proof of Equation 3.4

Consider a distributed table T distributed among S sensors. Each node $s_i \in S$ stores a local table T_i . $H(T_i)$ is the local synopsis of T_i , with the join column $\mathbf{X}$ of T_i and the frequencies of each distinct values in $\mathbf{X}$. Assuming uniform distribution, we need to prove the number of distinct values $|H(T_i)|$ of $H(T_i)$ is:

$$|H(T_i)| = |\mathbf{X}| \left(1 - \left(1 - \frac{1}{|S|} \right)^{|T|/|\mathbf{X}|} \right).$$

Proof. Let $|\mathbf{X}|$ be the number of distinct values in the join column $\mathbf{X}$, the frequency of a distinct value $\mathbf{x} \in \mathbf{X}$ is $|T|/|\mathbf{X}|$.

For a particular sensor $s_i \in S$ and a particular tuple t with value $\mathbf{x}$, the probability that a value $\mathbf{x}$ not appearing in s_i is:

$$p(\mathbf{x}, t) = \left(1 - \frac{1}{|S|} \right)$$

As mentioned, the number of tuples with the value $\mathbf{x}$ is $|T|/|\mathbf{X}|$. Therefore, the possibility that the value $\mathbf{x}$ not appearing in a sensor s_i is:

$$p(\mathbf{x}) = \left(1 - \frac{1}{|S|} \right)^{|T|/|\mathbf{X}|}$$

The possibility that $\mathbf{x}$ appearing in a sensor s_i is therefore:

$$1 - p(\mathbf{x}) = 1 - \left(1 - \frac{1}{|S|}\right)^{|T|/|\mathbf{X}|}$$

For $|\mathbf{X}|$ distinct values, the expected average number of distinct values in s_i is therefore

$$|\mathbf{X}|(1 - p(\mathbf{x})) = |\mathbf{X}| \left(1 - \left(1 - \frac{1}{|S|}\right)^{|T|/|\mathbf{X}|}\right)$$

Since number of tuples in $H(T_i)$ is the same as the average number of distinct value in s_i , we have

$$|H(T_i)| = |\mathbf{X}| \left(1 - \left(1 - \frac{1}{|S|}\right)^{|T|/|\mathbf{X}|}\right) .$$

□

Appendix B

Publications

1. Hai Yu, Ee-Peng Lim, and Jun Zhang. In-network join processing for sensor networks. In *Proceedings of the Eighth Asia Pacific Web Conference (APWeb'06)*, pages 263 – 274, Harbin, China, January 2006.
2. Hai Yu, Ee-Peng Lim, and Jun Zhang. On In-network Synopsis Join Processing for Sensor Networks. In *Proceedings of the 7th International Conference on Mobile Data Management (MDM'06)*, page 32, Nara, Japan, May 2006 .

Bibliography

- [AML05] Daniel Abadi, Samuel Madden, and Wolfgang Lindner. Reed: Robust, efficient filtering and event detection in sensor networks. In *Proceedings of VLDB*, 2005.
- [ASSC02] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 38(4):393–422, March 2002.
- [BB03] Boris Jan Bonfils and Philippe Bonnet. Adaptive and Decentralized Operator Placement for In-Network Query Processing. In *Proceedings of IPSN*, 2003.
- [BGS01] Philippe Bonnet, J. E. Gehrke, and Praveen Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, Hong Kong, January 2001.
- [CG05] Vishal Chowdhary and Himanshu Gupta. Communication-Efficient Implementation of Join in Sensor Networks. In *DASFAA*, Beijing, China, April 2005.
- [CLKB04] Jeffrey Considine, Feifei Li, George Kollios, and John Byers. Approximate aggregation techniques for sensor databases. In *Proceedings of International Conference on Data Engineering (ICDE)*, Boston, MA, USA, March 2004.
- [Cro05] I. Crossbow. Wireless sensor networks (mica motes). Website, 2005. http://www.xbow.com/products/wireless_sensor_networks.html.
- [DGM⁺04] Amol Deshpande, Carlos Guestrin, Samuel Madden, Joseph Hellerstein, and Wei Hong. Model driven data acquisition in sensor networks. In *Proceedings of VLDB'04*, Toronto, Canada, April 2004.

- [EN03] Eiman Elnahrawy and Badri Nath. Cleaning and querying noisy sensors. In *Proceedings of WSNA '03*, San Diego, USA, 2003.
- [FM85] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for database applications. *Journal of Computer and System Sciences*, 31, 1985.
- [GD93] S. Gatziau and K.R. Dittrich. Events in an active object-oriented database system. In *Proc. 1st Intl. Workshop on Rules in Database Systems (RIDS)*, Edinburgh, UK, September 1993.
- [GDI02] Habitat monitoring on great duck island. <http://www.greatduckisland.net/>, 2002.
- [GEG⁺03] Benjamin Greenstein, Deborah Estrin, Ramesh Govindan, Sylvia Ratnasamy, , and Scott Shenker. DIFS: A Distributed Index for Features in Sensor Networks. In *Proceedings of 1st IEEE International Workshop on Sensor Network Protocols and Applications (SNPA '03)*, Anchorage, AK, May 2003.
- [GR65] I. Greenberg and R. A. Robertello. The three factory problem. *Mathematics Magazine*, 38(2):67–72, Mar 1965.
- [GW00] G.J.Pottie and W.J.Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, May 2000.
- [Haa97] Zygmunt J. Haas. A routing protocol for the reconfigurable wireless networks. In *Proceedings of IEEE ICUPC'97*, San Diego, CA, October 1997.
- [HK01] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*, chapter 2. Academic Press, 2001.
- [HP98] Zygmunt J. Haas and Marc R. Pearlman. The performance of query control schemes for the zone routing protocol. In *Proceedings of ACM SIGCOMM'98*, Vancouver, BC, August 1998.
- [HP00] Zygmunt J. Haas and Marc R. Pearlman. The zone routing protocol: A hybrid framework for routing in ad hoc networks. In C. E. Perkins, editor, *Ad Hoc Networks*. Addison-Wesley, 2000.

- [HSI⁺01] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building Efficient Wireless Sensor Networks with Low-level Naming. In *SOSP*, October 2001.
- [HSW⁺00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, and David Culler and Kristofer Pister. System architecture directions for networked sensors. In *ASPLOS*, November 2000.
- [IEGH02] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann. Impact of network density on data aggregation in wireless sensor networks. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, June 2002.
- [IGE00] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proceedings of MobiComm'00*, Boston, US, August 2000.
- [JM96] David B. Johnson and David A. Maltz. Dynamic source routing in ad hoc wireless networks. In Tomasz Imielinski and Hank Korth, editors, *Mobile Computing*, chapter 5, pages 153–181. Kluwer Academic Publishers, 1996.
- [JMQ98] P. Jacquet, P. Muhlethaler, and A. Qayyum. Optimized link state routing protocol. IETF MANET: Internet Draft, November 1998.
- [jsi] J-sim. Website. <http://www.j-sim.org>.
- [KK00] Brad Karp and H. T. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of MobiComm'00*, Boston, United States, August 2000.
- [LC85] Hongjun Lu and Michael J. Carey. Some experimental results on distributed join algorithms in a local network. In *Proceedings of 11th International Conference on Very Large Data Bases (VLDB)*, Stockholm, Sweden, August 1985.
- [LKGH03] Xin Li, Young Jin Kim, Ramesh Govindan, and Wei Hong. Multi-Dimensional Range Queries in Sensor Networks. In *Proceedings of the First International*

Conference on Embedded Networked Sensor Systems (SenSys '03), Los Angeles, California, USA, November 2003.

- [Mad03] Samuel Madden. *The Design and Evaluation of a Query Processing Architecture for Sensor Networks*. PhD thesis, UC Berkeley, 2003.
- [MCP⁺02] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of WSNA 2002*, Atlanta, USA, 2002.
- [MFHH02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation Service for Ad-hoc Sensor Networks. In *Proceedings of OSDI '02*, 2002.
- [MFHH03] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491 – 502, San Diego, California, June 2003.
- [MGLA96] S. Murthy and J. J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. *Mobile Networks and Applications*, 1(2):183–197, October 1996.
- [PB94] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. *SIGCOMM'94: Computer Communications Review*, 24(4):234–244, October 1994.
- [PC97] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proceedings of IEEE Conference on Computer Communications*, pages 1405–1413, Kobe, Japan, April 1997.
- [PR98] C. E. Perkins and E. M. Royer. Ad hoc on demand distance vector (aodv) routing. *IETF Internet Draft*, November 1998.
- [PR99] C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, February 1999.

- [Rag02] V. Raghunathan. Energy aware wireless microsensor networks. *Signal Processing Magazine*, 45(2):40–50, 2002.
- [RKL⁺02] Sylvia Ratnasamy, Brad Karp, Yin Li, Fang Yu, Deborah Estrin, Ramesh Govindan, and Scott Shenker. GHT: A geographic hash table for data-centric storage. In *Proceedings of WSNA '03*, 2002.
- [RP99] Elizabeth M. Royer and Charles E. Perkins. Multicast operation of the ad hoc on-demand distance vector routing protocol. In *Proceedings of MobiCom '99*, pages 207–218, Seattle, WA, August 1999.
- [SRK⁺03] Scott Shenker, Sylvia Ratnasamy, Brad Karp, Ramesh Govindan, and Deborah Estrin. Data-Centric Storage in Sensornets. *ACM SIGCOMM Computer Communications Review*, 33(1):137–142, January 2003.
- [VdSJJ⁺03] M.A.M. Vieira, D.C. da Silva Jr., C.N. Coelho Jr., , and J.M. da Mata. Survey on wireless sensor network devices. In *Emerging Technologies and Factory Automation(ETFA03). IEEE Conference*, Sept 2003.
- [YG02] Yong Yao and Johannes. E. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, September 2002.
- [YG03] Yong Yao and Johannes. E. Gehrke. Query Processing for Sensor Networks. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, California, January 2003.
- [YHR⁺98] K. Yao, R. Hudson, C. Reed, D. Chen, and F. Lorenzelli. Blind beamforming on a randomly distributed sensor array. *IEEE Journal on Selected Areas in Communication*, 16(8), Oct 1998.
- [ZSR02] F. Zhao, J. Shin, and J. Reich. Information-driven dynamic sensor collaboration for tracking applications. *IEEE Signal Processing Magazine*, 9(2):61–72, March 2002.