Efficiently Processing Complex Queries in Sensor Networks

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der Fakultät für Informatik

des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Mirco Stern

aus Celle

Tag der mündlichen Prüfung:	01. Februar 2010
Erster Gutachter:	Prof. Dr.-Ing. Klemens Böhm
Zweiter Gutachter:	Prof. Dr. rer. nat. Thomas Seidl

Acknowledgments

There is a number of people who contributed to this dissertation. To start with, I want to acknowledge their contributions. But more importantly, I am very grateful to them and, hopefully, the following lines accomplish to express my appreciation.

This dissertation would not exist without Prof. Dr. Klemens Böhm, my advisor. He taught me the peculiarities of doing research and publishing papers. Whenever there was an argument that was not totally convincing, Klemens found it. At the same time, his pace and motivation is impressive. Whenever I sent him something to edit or comment on, I received it back within hours - was it 8 am or pm, Monday or Sunday. Klemens pushed me to publish on well-established conferences, which vastly improved the quality of this work. Finally, I very much appreciated his integrity which made it simple to work under his supervision.

I am also especially grateful to Dr. Erik Buchmann who co-advised this work. Whenever I wrote something up, he was the first one to read it – a tough task. His effort really made this a better dissertation and working with him has been a pleasure.

I am much obliged to Prof. Dr. Dr. Lockemann who never hesitated to give advice whenever I was in need. Also, I would like to thank Prof. Dr. Jörg Sander for sharing some of his insights on how to publish a sensor network paper on a database conference.

I am greatly indebted to my students who did most of the coding as well as the experiments. It certainly needed a lot of patience and motivation to iterate over the code again and again, whenever I had a new idea or felt that the performance might slightly improve by some changes here and there. In particular, Camillo Scandura implemented SENS-Join, Lev Povalahev realized CJF, and Björn Reuber contributed by realizing SNAP.

I have been fortunate to work in an environment where I always felt comfortable. This is primarily the accomplishment of my fellows and colleagues from the database group at the University of Karlsruhe. They really made this a place worth being. In particular, I would like to thank my officemate, Frank Eichinger. I really enjoyed our intensive discussions on just anything. In particular, whatever topic there was, I could be absolutely sure that he would take opposition. But whenever we worked on the same think (such as our lecture), I knew that we would be absolutely on the same line. He is the best officemate I could have asked for. Jutta Mülle is probably the one in our group who I bothered most often with weird questions. It's simple. If there is any organizational problem, ask Jutta and it is half way solved. Life at our chair would be much harder without her. Finally, thanks goes to Markus Bestehorn, Mr.

SunSPOT in our lab. Whenever I had trouble in getting our algorithms to run on a 'real' sensor network, Markus was there to help me out.

As always, the greatest debt one owes is to one's family and friends. They provided me with a life outside of computer science. In particular, I would like to thank my parents for their love and support throughout the years. The final credit is dedicated to Annemarie. I appreciate her understanding and patience, waiting for me whenever I was busy getting some stuff done at the costs of our weekend, her thick skin when I was nervous, waiting for a notification, and her reassurance whenever a notification was frustrating. Annemarie, you're wonderful.

Effiziente Bearbeitung komplexer Anfragen in Sensornetzen

Drahtlose Sensornetze stellen eine neuartige Mess- und Überwachungstechnologie dar, die zunehmend im industriellen und im wissenschaftlichen Bereich eingesetzt wird. Diese Technologie ermöglicht es, bei hoher räumlicher Auflösung Informationen über eine Umgebung/ein Objekt zu gewinnen. Das zugrunde liegende Beschaffen der Daten ist allerdings komplex und hemmt den praktischen Einsatz von Sensornetzen. Deklarative Anfragen als Schnittstelle, wie sie für Datenbanken üblich ist, sind ein vielversprechender Ansatz, die Komplexität der Datenbeschaffung vor der Anwendung zu verbergen. Dabei spezifiziert die Anwendung mit Hilfe SQL-artiger Anfragen, welche Daten sie benötigt. Im Fokus dieser Dissertation steht die anschließende Anfragebearbeitung, deren Ziel es ist, die angefragten Daten möglichst effizient zu beschaffen. "Effizient" meint in diesem Kontext die Minimierung der Kommunikationskosten.

Wesentlicher Beitrag dieser Arbeit sind einerseits die Verfahren SENS-Join und CJF zur Bearbeitung einmalig auszuführender bzw. kontinuierlicher Verbundanfragen, sowie SNAP zur approximativen Anfragebearbeitung andererseits. Diese Zweiteilung ist durch unterschiedliche Arten der Verwendung von Anfragen motiviert. Es gibt Anwendungen, bei denen der relevante Ausschnitt der Daten von vornherein bekannt ist und die gezielt diese Daten anfragen. Beispielsweise werden bei der Überwachung im industriellen Bereich Muster in den Daten gesucht, die auf eine Überhitzung von Anlagen hindeuten oder auf einen Druckabfall in Leitungen. Für das gezielte Anfragen relevanter Daten sind SENS-Join und CJF einschlägig. SNAP dagegen unterstützt Anwendungen, die auf eine Vorauswahl verzichten, um die Daten im Nachhinein analysieren zu können. Beispielsweise bestehen in der Industrie teilweise Auflagen, Überwachungsdaten zur Qualitätssicherung vollständig zu protokollieren. Beide Verwendungsarten werden im Folgenden näher betrachtet.

Gezielte Auswahl relevanter Daten. Im Falle der gezielten Auswahl werden die relevanten Daten mit Hilfe der klassischen Anfrageoperatoren Selektion, Projektion und Verbund (Join) spezifiziert. Eine effiziente Bearbeitung wird dadurch erreicht, die Operatoren innerhalb des Netzes auszuführen, um die in Folge einer Auswahl nicht benötigten Daten gar nicht erst zu senden. Das heißt, die zugrunde liegende Idee ist die Nutzung der Selektivität, um eine effiziente Bearbeitung zu er-

reichen. Für den Verbundoperator stellt die Umsetzung dieser Idee zurzeit allerdings ein offenes Problem dar: Der Verbundoperator kann Beziehungen zwischen beliebigen Messdaten des Netzes ausdrücken. Eine Entscheidung, welche der Daten für die Anfrage relevant sind, setzt daher im Allgemeinen einen Vergleich der Daten aller Knoten voraus. Dies ist sehr kommunikationsaufwendig. Bisherige Ansätze umgehen dieses Problem durch Einschränkungen der möglichen Anfragen. Dadurch sind diese Lösungen nur in äußerst speziellen Fällen anwendbar. Die hier vorgestellten Verfahren SENS-Join und CJF erreichen dagegen eine effiziente Bearbeitung allgemeiner Verbundanfragen.

Beide Methoden orientieren sich an einem idealen Vorgehen, welches eine untere Schranke für die Kommunikationskosten der Bearbeitung von Verbundanfragen darstellt. Dabei werden Daten, die nicht zum Ergebnis beitragen, direkt an den Quellen gefiltert. Die verbleibenden Daten werden zur Basisstation (Senke des Netzes) gesendet, an der anschließend das Anfrageergebnis berechnet wird. Um dieses Ideal in ein praktikables Verfahren zu überführen, muss jeder Knoten das Wissen erlangen, ob seine Daten zum Ergebnis beitragen.

SENS-Join löst dieses Problem durch eine Vorberechnung, die jedem Knoten das notwendige Wissen bereitstellt. Besondere Leistung von SENS-Join ist, den Overhead dieser Vorberechnung gering zu halten. SENS-Join erreicht dies durch gezielte Auswahl der zu sendenden Informationen: Durch die Vorberechnung wäre bei einer naiven Umsetzung jeder Knoten mehrfach in eine Anfrage involviert. SENS-Join schafft es, eine Vielzahl von Knoten nur einmalig einzubeziehen. Zum anderen wurde eine spezielle Kompression der in der Vorberechnung benötigten Werte der Join-Attribute entwickelt, die auf räumlichen Indexstrukturen basiert. Insgesamt ermöglicht SENS-Join eine Reduktion der Kommunikationskosten um bis zu 80% im Vergleich zu den vorher besten Verfahren. Bezogen auf einzelne Knoten werden die Kosten der am meisten belasteten zum Teil um mehr als eine Größenordnung reduziert.

Eine Vorberechnung stellt eine gute Lösung für einmalig auszuführende Anfragen dar. Für kontinuierliche Verbundanfragen ist sie dagegen ungünstig: Da Messwerte zeitlich korreliert sind, ändert sich die Tatsache, ob ein Knoten zum Ergebnis beiträgt, nicht bei jeder Ausführung. Für kontinuierliche Verbundanfragen wird ein Ansatz namens CJF vorgestellt, der Filter verwaltet, anstatt sie wiederholt neu zu verteilen. Filter sind dabei Intervalle, für die gilt, dass ein Knoten seine Daten nicht sendet, solange diese innerhalb seines Intervalls liegen. Wesentlicher Beitrag ist die fortlaufend optimale Berechnung der Filter, d. h. der Intervallgrößen, die die erwarteten Kommunikationskosten minimieren. Ein zweites Problem bei der Filterverwaltung ist deren Aktualisierung. Hier berücksichtigt CJF die Kosten einer Aktualisierung, um zu entscheiden, welche der aktuell verwendeten Filter aktualisiert werden, und für welche die Kosten die Einsparungen einer besseren Filterung übersteigen. Durch den Wegfall der konstanten Kosten einer Vorberechnung liegen die von CJF benötigten Kommunikationskosten nahe an der unteren Schranke.

Bearbeitung nicht-selektiver Anfragen. Die Nutzung der Selektivität zur effizienten Bearbeitung hat das offensichtliche Problem, dass sie nur einschlägig ist, falls Anfragen auch selektiv sind. Dies ist beispielsweise für Anfragen, die auf eine Vorauswahl verzichten, nicht der Fall. Zur effizienten Bearbeitung nicht-selektiver Anfragen wird ein Verfahren namens SNAP vorgestellt, das approximative Anfrageergebnisse ermöglicht. Die Idee einer approximativen Anfragebearbeitung ist, kontrolliert die Präzision des Ergebnisses zu senken, um eine effizientere Datenbeschaffung zu ermöglichen. Dazu spezifiziert die Anwendung die erforderliche Güte der Antwort durch einen maximalen Fehler.

Die meisten Ansätze zur approximativen Anfragebearbeitung in Sensornetzen verwenden mathematische Modelle, um aktuelle Messdaten der Knoten zu schätzen, anstatt diese zu kommunizieren. Diese Ansätze funktionieren gut, solange die Präzisionsanforderungen der Anwendung gering sind. Beispielsweise lässt sich die Temperatur $\pm$ 0,5°C in vielen Umgebungen gut schätzen. Für präzisere Ergebnisse sind modellbasierte Schätzungen jedoch ungeeignet. SNAP setzt daher Synopsen (Zusammenfassungen relationaler Daten) im Zuge der Datenbeschaffung ein. Insbesondere basiert SNAP auf Waveletsynopsen, die gute Eigenschaften bezüglich Datenreduktion und Genauigkeit haben. Der wesentliche Beitrag ist hier die verteilte Erstellung von Waveletsynopsen unter Berücksichtigung von Fehlergarantien. Hierzu wird aufgezeigt, wie man den Datenfluss der Wavelettransformation in das Routing des Sensornetzes integrieren kann, ohne dabei das Datenvolumen zu erhöhen oder von optimalen Kommunikationspfaden abzuweichen. Zum anderen wird ein Verfahren zur Schätzung der Häufigkeiten von Waveletkoeffizienten vorgestellt, das die Grundlage einer verteilten Datenreduktion darstellt. Insgesamt ermöglicht SNAP Fehlergarantien, die eine Größenordnung besser sind als die von modellbasierten Ansätzen. Gleichzeitig wird eine Reduktion des Datenvolumens auf ein Fünftel erreicht. Dies zieht eine entsprechende Reduktion der Kommunikationskosten nach sich.

Fazit. Diese Arbeit erweitert den Stand der Forschung um die effiziente Bearbeitung des Verbundoperators in Sensornetzen, der zusammen mit Selektion und Projektion das gezielte Anfragen relevanter Daten ermöglicht. Darüber hinaus wird ein Verfahren zur approximativen Beantwortung von Anfragen vorgestellt. Grundsätzlich ist die approximative Anfragebearbeitung orthogonal dazu, ob eine Anfrage eine Vorauswahl der Daten trifft und dadurch selektiv ist. So könnte man die nach einer Auswahl verbleibenden Daten approximativ sammeln, um weitere Kosten zu sparen. Da mit einer approximativen Methode aber eine Reduktion der Präzision einhergeht, bietet sich ihr Einsatz vor allem für Anfragen an, die auf eine Vorauswahl größtenteils verzichten und daher ein besonders hohes Datenvolumen haben.

Contents

1 Introduction **1**
- 1.1 Declarative Data Collection in Sensor Networks . . . 2
- 1.2 Selective vs. Non-Selective Queries . . . 3
- 1.3 Exploiting Selectivity for Efficiently Collecting Data . . . 5
- 1.4 Consolidating Sensor Relations . . . 7
- 1.5 Contributions of this Dissertation . . . 8
- 1.6 Outline of this Dissertation . . . 9

2 Background **11**
- 2.1 Network Architecture . . . 11
  - 2.1.1 Sensor Network Platforms . . . 11
  - 2.1.2 Impact of the Platform on Data Management . . . 12
- 2.2 Query Processing in Sensor Networks . . . 14
  - 2.2.1 Declarative Queries . . . 15
  - 2.2.2 A Query Processing Architecture . . . 16
- 2.3 Discussion . . . 21

3 Join Processing in Sensor Networks **23**
- 3.1 The Problem: "Join Processing in Sensor Networks" . . . 23
  - 3.1.1 The Difficulty of Processing Joins . . . 24
  - 3.1.2 Problem Statement . . . 25
  - 3.1.3 IDEAL Join Processing . . . 26
- 3.2 Related Work on Join Processing . . . 27
  - 3.2.1 Distributed Join Processing . . . 27
  - 3.2.2 Join Processing in Sensor Networks . . . 29
- 3.3 Summary . . . 32

4 Efficiently Processing General-Purpose Joins **35**
- 4.1 Overview of SENS-Join . . . 35
- 4.2 The SENS-Join Approach . . . 37
  - 4.2.1 Collecting Join-Attribute Tuples . . . 39
  - 4.2.2 Disseminating the Join Filter . . . 41
  - 4.2.3 Final Result Computation . . . 42
  - 4.2.4 Design Considerations . . . 43

4.2.5 Design for Error Tolerance 44
4.3 Compactly Representing Join-Attribute Tuples 44
4.3.1 Key Ideas of the Compact Representation 45
4.3.2 Quantization 45
4.3.3 Representing Points Using a Spatial Index 48
4.3.4 Computing Low-Level Primitives 51
4.4 Experimental Study 52
4.4.1 Efficiency of SENS-Join 54
4.4.2 Costs of SENS-Join – Breakdown 57
4.5 Tradeoffs 59
4.6 Summary 59

5 Processing Continuous Join Queries: a Filtering Approach 61
5.1 The CJF Approach 61
5.2 Related Work on Filtering Data Streams 64
5.3 Filtering for Join Processing 66
5.3.1 Filters in CJF 66
5.3.2 A Join-Filtering Framework 67
5.3.3 Maintaining Filters 68
5.4 Computing Optimal Filters 69
5.4.1 The Optimization Problem 69
5.4.2 Continuously Optimizing Filters 73
5.5 Efficiently Updating Filters 82
5.6 Prototype Design 84
5.7 Evaluation 85
5.7.1 Experimental Setup 86
5.7.2 Comparative Experiments 87
5.7.3 Analysis of CJF 89
5.8 Summary 91

6 Approximate Query Processing in Sensor Networks 93
6.1 Problem Statement: Processing Non-Selective Queries 93
6.2 Constructing Wavelet Synopses 95
6.3 Related Work on Approximate Query Processing in Sensor Networks 97
6.3.1 Model-based Approaches 97
6.3.2 Wavelet-Based Approaches 100
6.3.3 Further Approaches 103
6.4 Discussion 103

7 Efficiently Approximating Sensor Relations with Quality Guarantees 105
7.1 The SNAP Approach 105

7.2 Distributing Wavelet Transforms . . . 106
7.2.1 Design Space for Distributing Wavelet Transforms . . . 107
7.2.2 Integration Approach . . . 108
7.2.3 Finding the Optimal Structure Tree . . . 109
7.3 Constructing Synopses . . . 111
7.3.1 Thresholding in a Distributed Setting . . . 112
7.3.2 Foundations for Compact Synopses . . . 113
7.3.3 Distribution by Estimating Frequencies . . . 117
7.4 Sending Approximations . . . 122
7.5 Evaluation . . . 124
7.5.1 Experimental Setup . . . 124
7.5.2 Comparative Experiments . . . 125
7.5.3 Analysis of SNAP . . . 127
7.6 Summary . . . 130

8 Conclusions and Future Directions 131
8.1 Summary . . . 131
8.2 Future Work . . . 133

Appendix 141

A Optimal Locations for Join Processing 141
A.1 Preliminaries . . . 141
A.1.1 Terminology . . . 141
A.1.2 Network Model . . . 143
A.1.3 Cost Model . . . 145
A.2 Centralized Join Processing . . . 147
A.2.1 Cost Model for Centralized Strategies . . . 147
A.2.2 Method . . . 148
A.2.3 Monotonicity Assumptions . . . 149
A.2.4 Gain of Centralized Strategies . . . 151
A.3 IDEAL Join Processing . . . 151
A.3.1 Optimal Number of Processing Sites . . . 153
A.3.2 Location of Optimal Site per Group . . . 155
A.4 Conclusions . . . 156

B Primitives for Constructing the Quadtree Datastructure 157
B.1 Constructing Quadtrees . . . 157
B.2 Merging Quadtrees . . . 161
B.3 Summary . . . 162

C CJF: Generalization of the Optimization Problem for Asymmetric Join Conditions 167

List of Figures

2.1 SunSPOT 12
2.2 Routing tree as an overlay over the connectivity graph of the network 18

4.1 SENS-Join, at each node 38
4.2 ForwardJoinAttrValues 40
4.3 ForwardJoinFilter 42
4.4 ForwardCompleteTuples 43
4.5 Distribution of values for 3 join attributes 45
4.6 Distribution of values for 2 join attributes 46
4.7 Z-ordering 46
4.8 EncodeTuple 47
4.9 Construction of the quadtree 49
4.10 Encoding of a quadtree 50
4.11 Overall savings of SENS-Join 54
4.12 Per node savings of SENS-Join 55
4.13 Influence of the ratio of $\frac{\text{3 join attributes}}{x \text{ attributes overall}}$ 55
4.14 Influence of the ratio of $\frac{\text{1 join attribute}}{x \text{ attributes overall}}$ 56
4.15 Influence of the network size 56
4.16 Costs in the different steps of SENS-Join 57
4.17 Influence of quadtree representation 58

5.1 Slack between filters in dimension "temperature" 72
5.2 Continuously differentiable function $slack^i_{jh}$ for $|A.att_i - B.att_i| < d_i^{max}$ 72
5.3 Optimization Method 76
5.4 Neighbor Method 77
5.5 Optimizing $commCost_j(s_j)$ (= 'cC_j' in this figure) 79
5.6 Costs of join approaches for different types of queries 87
5.7 Considering dependencies among the nodes 88
5.8 Using uniform filter settings 89
5.9 Performance of filtering of CJF 89
5.10 Initialization of filters and convergence 90
5.11 Influence of different data sets on CJF 91

6.1 Structure graph for the Haar example 96

7.1 Integration approach 108
7.2 Heuristical algorithm for computing the structure tree 110
7.3 DP for computing the optimal structure tree – a benchmark for the heuristical approach 112
7.4 Combining two partial solutions into one 113
7.5 Format of Wavelets (Encoding) 116
7.6 Estimating frequencies 118
7.7 Comparison of integration approaches (real data) 126
7.8 Comparison of integration approaches (synth. data) 126
7.9 Comparison to Ken (real data) 127
7.10 Scalability of SNAP (synthetic data) 128
7.11 Influence of the number of attributes (synth. data) 128
7.12 SNAP: heuristical vs. optimal integration (real data) 129
7.13 SNAP: performance on extreme data (synth. data) 129

A.1 Scenario 144
A.2 (a) Influence of $\angle \mathcal{A}_c \mathcal{R} \mathcal{B}_c$ & distribution; (b) Influence of the result's size 149
A.3 Example of two join locations 153
A.4 (a) Gain: optimal number vs. single site; (b) Gain: optimal location vs. root node 155

B.1 $\text{Insert}_{\text{Join\_Atts}}$ 158
B.2 Representing points with a quadtree 159
B.3 Method *Split* 159
B.4 Method *InsertPoint* 163
B.5 Method *GetPositionOfSubtree* 164
B.6 Method *Union* 165

C.1 Continuously differentiable functions $slack_{jh}^{Ai}$ and $slack_{jh}^{Bi}$ for the join condition $A.att_i - B.att_i > d_i^{min}$ 170

1 Introduction

Wireless sensor networks are a novel measuring technology that is increasingly used in industrial and scientific settings. Such networks consist of a large number of nodes that are equipped with sensors, allowing for observations at a high spatial resolution. However, the underlying data acquisition is complex and is in the way of an easy deployment of sensor networks.

Declarative queries, as used in commercial database management systems, are a promising approach for hiding the intricacies of data collection from the application. In particular, applications specify the data they want to collect by means of ordinary SQL queries. A query processor then takes care of efficiently collecting the data from the network. In the context of wireless sensor networks, "efficiently" means using a minimum amount of energy.

Efficient query processing is in the focus of this dissertation. Prior work has studied processing simple queries, especially query operators such as selection, projection and aggregation. In this work, we are concerned with processing join queries as well as processing non-selective queries.

Join processing in wireless sensor networks is difficult: As the tuples can be arbitrarily distributed within the network, matching pairs of tuples is communication-intensive and thus costly in terms of energy. While join processing has received a lot of attention recently, current solutions build on strict assumptions that are frequently not met. We present two join methods that overcome this problem: SENS-Join for efficiently processing one-time queries and CJF for processing continuous join queries.

The idea of efficiently processing selection, projection, and join operators is to avoid shipping data that is irrelevant to the result. That is, the idea is to exploit the selectivity in the query. However, if most of the data is relevant to the result, i.e., if the selectivity is low, the volume of the data cannot be significantly reduced by discarding irrelevant portions. Thus, a simple data collection is again communication-intensive. Most notably, non-selective queries are the rule in a number of monitoring applications. To this end, we propose SNAP, an approach for approximate query processing. SNAP can efficiently consolidate entire sensor relations.

The following section motivates data collection in sensor networks along with the use of a query interface. Section 1.2 introduces a conceptual separation of processing selective and processing non-selective queries. We discuss the problems in exploiting selectivity for an efficient join processing in Section 1.3. In Section 1.4, we introduce the challenges of processing non-selective queries. We point out the contributions of this dissertation in Section 1.5, before outlining the remainder of the text.

1.1 Declarative Data Collection in Sensor Networks

Today, wireless sensor networks are commercially available [Sena] and are used in industrial settings for purposes such as monitoring and surveillance. As a concrete example, consider a drug manufacturer who has to comply with legal requirements for documenting the conditions in the production process [Pha]. To do so, the manufacturer has installed 130 wireless sensor nodes on an existing production line. Wireless sensor networks are ideally suited for such settings. The fact that they come without wiring lowers installation costs by 80% and installation time by 90% and enables simple reconfigurations [Wer].

In fact, there is a broad range of industrial applications of sensor networks. For instance, they are used in manufacturing plants, temperature controlled storage warehouses, or computer server rooms to monitor the diverse pieces of equipment. Staff needs the sensor readings to dispatch repair teams or to shutdown problematic equipment in localized areas where temperature spikes or other faults occur.

Beyond industry, there are numerous deployments of wireless sensor networks for scientific purposes. The most popular among them are probably the early ones such as the Great Duck Island deployment [MCP+02, CEH+01] or the Berkeley Botanical Garden deployment [Daw98]. The Great Duck Island deployment was used to monitor the nesting and breeding behavior of sea-birds. The researchers placed sensor nodes in a number of burrows during the nesting season in 2002, and recorded sensor data as birds came and went. They captured information such as light readings, ambient temperature, surface temperature, humidity and air pressure. The deployment in the Berkeley Botanical Garden was used in June and July 2003 to monitor environmental conditions in and around coastal redwood trees. The botanists at UC Berkeley were interested in the role the trees play in regulating and controlling their environment, especially how they affect the humidity and temperature of the forest floor. An example of a number of recent scientific deployments is the Sensorscope project [Senb] which focuses on environmental monitoring. Some of the experiments in this dissertation are actually based on traces of sensor data that stem from this latter project.

Abstracting from these concrete deployments, wireless sensor networks (or "sensor networks" hereafter) consist of many nodes which are equipped with sensors, have constrained communication and computation capabilities and are battery operated. They allow for monitoring at a high spatial resolution, and can even penetrate the phenomena of interest. At the same time, they are non-intrusive, which is important for applications such as wildlife monitoring. Finally, sensor networks can cover large regions and can monitor over long periods of time.

To obtain longevity, energy-efficiency is mandatory. Poor energy consumption can be dramatic in practice: For example, [CDHH06] reports that a software bug in the

Berkeley Botanical Garden deployment caused a third of the nodes to constantly keep their radios active; they exhausted their batteries in only a few days.

The need to make careful use of resources, especially of energy, makes engineering sensor network applications difficult. Making matters worse, sensor nodes usually provide low-level programming abstractions which further complicate sensor network programming.

To overcome the problems and thus to simplify data collection, query interfaces such as SQL have proven to be attractive. Chapter 2 highlights the advantages of using a query interface as opposed to programming the data collection from scratch. In particular, declarative queries are high-level statements that specify the data of interest. They do not describe the actual algorithms to collect the answer set. Thus, the application developer is unburdened from dealing with the constraint resources and the low-level programming abstractions.

To facilitate declarative queries, the idea is to abstract the network into a relation. There is one tuple per node of the network. The attributes represent the sensors of the nodes. Again, we provide the details in Chapter 2.

Then, an application can specify the data of interest by means of a SQL query.

Example 1.1: The following query is a simple example that periodically collects the node_ID, temperature and humidity of those sensor nodes that observe a temperature of more than 24.0°C[1]:

```
SELECT ID, temp, humidity
FROM Sensors
WHERE temp > 24.0°C
SAMPLE PERIOD 30s
```

■

A query processor takes care of collecting the requested data, using a minimum amount of energy. In particular, on typical sensor nodes, message transmission expends orders of magnitude more energy than CPU computations over an equivalent length of time (cf. Chapter 2). Therefore, minimizing the communication costs is the optimization goal of query processing in sensor networks.

1.2 Selective vs. Non-Selective Queries

In this dissertation, we classify queries into (a) selective and (b) non-selective queries. Before motivating and justifying this distinction, we briefly clarify our understanding of 'selective queries': Intuitively, the "selectivity factor" of an operator indicates the

[1] Of course, the physical units are absent in actual queries – queries correspond to ordinary SQL queries with extensions for temporal aspects of data collection (cf. Chapter 2). However, we sometimes include the units in our examples as it makes the reading more intuitive.

cardinality of its result related to the cardinality of the input. For instance, the selectivity factor of a selection predicate $p(\cdot)$ over a Relation R is defined as $\sigma = \frac{card(p(R))}{card(R)}$. The selectivity factor of a join operator is defined $\sigma = \frac{card(A \bowtie B)}{card(A) \cdot card(B)}$. We call an operator "selective", if the selectivity factor is small, i.e., if the cardinality of the result is small.

Terminology. We say that *a query is selective* if the cardinality of the query result is small.

The distinction of selective and non-selective queries is a conceptual one, i.e., there is no clear cut borderline for queries being selective or not. However, the distinction is very useful for the following reasons.

- **Development of Algorithms.** With respect to query processing, different techniques apply. In particular, the selectivity of queries can be exploited for obtaining efficiency – prior work has done so extensively. Thus, the distinction indicates a way to obtain efficiency and is important for designing algorithms for query processing. However, the resulting techniques for processing selective and non-selective queries are actually orthogonal. Thus, while it is convenient to think of queries in terms of being selective or not for designing algorithms, the resulting techniques can be combined to further reduce the energy costs. We will return to this point in Chapter 8 after presenting the solutions.

- **Typical Usage of Queries.** In a way, 'selective' and 'non-selective' queries are the extremes in the spectrum of possible selectivity factors. As indicated, the corresponding algorithms can be combined in a query processor if the query execution based on only one of them is too costly. However, the extremes are common cases in the usage of sensor networks:

  (a) There are applications that have a well-separated interest, i.e., only specific parts of the measurement data are relevant. As examples, think of monitoring technical installations or structural works. Here, the goal is to detect certain deficiencies, such as temperature spikes or fine cracks in the construction. Such deficiencies are usually detected based on characteristic patterns that they generate in the sensor readings. Measurements that do not correspond to these patterns are irrelevant. For such applications, queries are not only used to get the current sensor readings out of the network. As in traditional databases, queries are also used to discard irrelevant data from the query result. Thus, the queries issued by such applications are usually selective.

  (b) In contrast, sometimes applications simply collect the current sensor readings without prior selections. Consider the scientific applications that we introduced in the preceding section as an example. As the researchers were interested in a later analysis of the data, the applications collected

the entire measurements and stored them outside the network. As an example of an industrial application, consider the drug manufacturer. In particular, law requires the manufacturer to log the entire data [Pha]. For such applications, queries have a low selectivity.

In the following, we discuss the problems as well as the query processing for both classes.

1.3 Exploiting Selectivity for Efficiently Collecting Data

Given a query interface, applications specify the data of interest by means of classical Select-Project-Join (SPJ) statements. Among these operators, efficiently processing selections and projections is well understood (cf. Chapter 2). To reduce communication, the idea is to exploit the selectivity of queries by pushing data-reducing operators into the network. Then, only the data that qualifies for the result is sent to the base station (sink of the network).

In contrast, efficiently processing join queries in sensor networks is an open problem that has received considerable attention recently [ZGT09, YLOT07, AML05, CN07, CNS07]. The join is important as it allows for combining data from different nodes. To illustrate its application, think of monitoring (a) similar conditions at different locations or (b) dissimilar conditions. For instance, we might be interested in noting exceptional conditions in an industrial setting, such as a loss of pressure within a pipe, or a machine that radiates a (substantially) higher temperature than others.

Example 1.2: The following is a simple example of a join query which we will use as an illustration later in the text. It acquires data from nodes that observe similar temperature and humidity conditions.

```
SELECT A.*, B.*
FROM Sensors A, Sensors B
WHERE |A.hum - B.hum| < 2%
AND   |A.temp - B.temp| < 0.3°C
AND    A.id != B.id
SAMPLE PERIOD 30s
```

■

The problem with exploiting the selectivity for an efficient join processing is that a node does not know if its tuple qualifies for the result. Thus, a node cannot easily discard its tuple. In general, finding out if a tuple joins requires matching it to the tuples of the other relation in the query. However, in sensor networks tuples which

have to be joined can be arbitrarily distributed. Thus, the matching is communication-intensive.

Prior join approaches avoid the problem of matching arbitrarily distributed tuples by specializing to specific types of queries. For instance, [YLOT07] requires one of the relations to contain a few (well less than ten) tuples only. As a consequence, it is possible to distribute one of the relations within the network at little costs. However, such specific requirements restrict applicability. For instance, in the simple query in Example 1.2, none of the relations is restricted to containing one or two tuples only. In fact, the requirements imposed by prior approaches are strict and are frequently not met (cf. Chapter 3).

In contrast, we present two join methods, SENS-Join and Continuous Join Filtering (CJF), that allow for efficiently processing join queries without imposing restrictions on the number or kind of join conditions. Among them, SENS-Join targets one-time queries. For continuous join queries, CJF can further reduce the communication costs. In the remainder of this section, we briefly introduce both approaches.

As a foundation of designing join approaches, we derive the following concept which lower bounds the communication costs of join processing: Firstly, each node discards its tuple if it does not join, i.e., non-joining tuples are not at all sent. The second step says how to proceed with the joining tuples: The remaining tuples after discarding are sent to the base station where the join is computed. However, this approach is infeasible in practice: Each node would have to know if its tuple joins. This depends on the data of the other nodes and is actually the result of the matching that we discussed above.

The lower-bound concept guides the design of our join algorithms. In particular, both approaches discard non-joining tuples at the sources to avoid sending them and compute the result at the base station. The problem in turning this concept into a practical approach is to supply each node with the knowledge if its tuple joins. To do so, SENS-Join introduces a *precomputation* step which identifies the tuples that join. Subsequently, SENS-Join can actually implement the lower-bound approach which sends the joining tuples to the base station. At a high-level, the main contribution of SENS-Join is the design of the precomputation: It is highly efficient in order not to exhaust the savings due to discarding non-joining tuples.

It is possible to use such a precomputation-based approach for answering one-time queries as well as continuous join queries. However, for continuous queries, this incurs unnecessary costs: The precomputation is repeated for each execution, leaving aside temporal correlations in the data. To take advantage of temporal correlations for efficiently processing continuous join queries, we present CJF, a filtering-based approach. The idea of CJF is that the base station installs filters on the nodes. The filters are used to discard non-joining tuples. In particular, we keep the filters for subsequent executions and maintain them. This avoids the repeated precomputation. The problems are for the base station to determine optimal filters and to decide which filters to update. With respect to processing continuous join queries, our contribution

is to solve these problems.

1.4 Consolidating Sensor Relations

In the preceding section, the idea for obtaining efficiency was to exploit the selectivity of queries: The algorithms discard data that is irrelevant for the result as early as possible in the processing. This cuts communication costs since we refrain from sending unnecessary data.

However, there is an obvious problem in relying on selectivity for obtaining efficiency: Queries might not be selective. As motivated in Section 1.2, this case is actually relevant in practice. The problem is that if selectivity is low, collecting the data is communication-intensive due to its large volume.

A common approach for processing non-selective queries is approximate query processing. The idea is to trade the accuracy in the result for communication costs. In particular, applications differ in their accuracy requirements [DGM+04, CDHH06]. Such applications can take advantage of a mechanism for approximate query processing, if the application is given control over the accuracy. In particular, it is critical for any approximate query processing scheme to provide error guarantees.

Prior work on approximate query processing in sensor networks is mostly model-based. Here, a model is a compact representation ("synopsis") of the measurements, and queries are processed against it. Model-based query processing saves communication costs compared to a straightforward data collection, if the accuracy requirements are loose, e.g., if the temperature is required within $\pm 0.5°$C. For more accuracy, the models need frequent updates, and the communication costs quickly increase. In addition, sophisticated models incur substantial training costs.

To overcome these problems, we propose SNAP ("SNapshot APproximation"), a query-processing scheme based on wavelet synopses. Briefly, the wavelet transform is a mathematical tool that transforms the data into a set of wavelet coefficients. In particular, these coefficients are not equally important: If we apply the inverse transform, some coefficients are details and can be ignored while introducing only little errors in the reconstructed data. The idea for obtaining a synopsis from the transformed data is to discard as many details as possible without violating the error bounds. This step is called "thresholding".

To exploit the data reduction properties of wavelet transforms for an efficient data collection, SNAP has to incrementally construct a wavelet synopsis during the collection. Here, our main contributions are to show how to distribute the transform as well as the thresholding.

1.5 Contributions of this Dissertation

This dissertation makes the following contributions:

SENS-Join. We present SENS-Join, the first join method for sensor networks that does not rely on restrictive requirements for obtaining efficiency: SENS-Join can efficiently handle any number of join conditions and arbitrary distributions of the nodes involved.

As a foundation of the design of our join algorithms, we derive a concept that lower bounds the communication costs. Its cornerstones are to discard non-joining tuples at the sources and to send joining tuples to the base station.

SENS-Join provides the knowledge which tuples join by means of a precomputation. The main contribution of SENS-Join is the design of the precomputation: It is highly efficient. In more detail, the contributions are:

- **Information Flow.** If the precomputation is realized naively, each node is involved multiple times in processing a join query. SENS-Join comprises mechanisms that achieve to involve most of the nodes only once – this significantly reduces the number of overall transmissions. In particular, these ideas are not specific to join processing. They apply to designing efficient precomputations in sensor networks.

- **Data Representation.** We propose a new compact representation of the precomputation data. Our mechanism is based on a spatial index to remove redundancy. In contrast to compression algorithms, this achieves compactness even for small amounts of input data, which are the rule for data collection in sensor networks.

- **Performance.** We extensively evaluate the performance of SENS-Join. Our results indicate that SENS-Join can reduce the overall energy consumption by more than 80% compared to the state-of-the-art. It can reduce the per node energy consumption of the most loaded nodes by more than an order of magnitude.

Continuous Join Filtering (CJF). For continuous join queries, we propose CJF, a filtering-based approach. At a high level, the main contribution of CJF is to maintain filters in an optimal way. In more detail, the contributions are:

- **Optimizing Filters.** The base station continuously computes filters that minimize communication costs. In particular, if a filter is too small, it is likely that the measurement of a node falls outside of it and the node sends its data, which the filter was supposed to avoid. In contrast, large filters cause costs related to

guaranteeing a correct result, as we will explain in Chapter 5. We show how to map join queries to a mathematical optimization problem for optimizing filters and how to continuously solve it.

- **Optimizing Updating Costs.** For each execution, the base station has to decide which of the current filters to update. This also incurs costs. However, due to temporal correlations, filters do not change by much in between subsequent query executions and the costs of updating might not pay off. We propose a scheme to decide which of the filters to actually update.

- **Performance.** We evaluate the performance of CJF. In particular, CJF comes close to the lower-bound concept. For continuous queries, CJF consistently outperforms other join approaches.

SNAP. We present SNAP, the first distributed wavelet compaction in sensor networks that provides error guarantees. In detail, our contributions are:

- **Distributing the Transform.** To distribute the wavelet transform, we need to map the data flow of the wavelet transform onto the routing structure of the network. We show how to integrate the transform with the routing without sacrificing optimal routes.

- **Distributing the Compaction.** We compactly encode coefficients, instead of discarding them ('thresholding'). This is key to distributing the construction of synopses. To encode them, it is necessary to know the frequencies of the coefficients in the overall synopsis. We show how to accurately estimate these frequencies in a way that is mathematically sound as a basis of a distributed encoding.

- **Performance.** Our experimental results indicate that SNAP reduces the communication costs by more than a factor of five compared to state-of-the-art approaches. SNAP improves the limit in the error guarantees for which data can be efficiently consolidated by more than an order of magnitude.

1.6 Outline of this Dissertation

The remainder of this dissertation presents our solutions in detail. As a foundation, we provide some background on sensor networks and especially on query processing in Chapter 2. In Chapter 3, we turn to the first integral part of this dissertation, join processing. We introduce the intricacies and discuss prior solutions. Then, in Chapter 4, we present SENS-Join, our solution to join processing. While SENS-Join can be used to answer one-time as well as continuous join queries, it is especially

suited to one-time queries. For continuous join queries, we present our method CJF in Chapter 5. All algorithms on processing particular query operators exploit their selectivity. In a second integral part of this dissertation, we address non-selective queries. In Chapter 6, we point out why approximate query processing approaches make sense for dealing with non-selective queries. In addition, we review related work on approximate query processing in sensor networks. In Chapter 7, we then present SNAP, our approximate query processing scheme. Finally, in Chapter 8, we summarize our results and provide an outlook on interesting problems with respect to declarative data acquisition in sensor networks that we did not cover in this dissertation.

Portions of this work were originally published in shortened form in [SBB08] (lower bound), [SBB09b] (SENS-Join), [SBB10] (CJF) and [SBB09a] (SNAP).

2 Background

The notion of a "sensor network" is by no means well-defined. Frequently, the application influences the hardware deployed and the network architecture. This section provides the context of our work. The network architecture is described in Section 2.1. Section 2.2 then provides some background on query processing in sensor networks.

2.1 Network Architecture

Our work is based on a network architecture consisting of hundreds to thousands of stationary sensor nodes. Each one is equipped with several sensors, a processor, a small RAM, a wireless radio, and it is battery operated. A powered base station serves as an access point. Each node is aware of the nodes within its wireless range, which form its neighborhood. It communicates with nodes other than its neighbors using multi-hop routing.

In the following, we introduce two popular hardware platforms to illustrate the architecture, SunSPOTs [Mic] and Mica2 motes [Tec]. We then abstract from the concrete hardware and identify the characteristics that influence data management. These have to be considered in the design of our solutions.

2.1.1 Sensor Network Platforms

The sensor nodes which we use in our lab are SunSPOTs as shown in Figure 2.1. The size of a SunSPOT is about 70 x 41 x 23 mm. At a high level, the nodes have a modular architecture, consisting of a sensor board, a processor board, and a battery.

By default, SunSPOTs are equipped with sensor boards that contain a light sensor, a temperature sensor, and an accelerometer. A default board makes sense since SunSPOTs are a development platform. In real deployments, the sensors are usually customized to the needs of the application. For SunSPOTs, it is possible to add extension boards and custom sensors like humidity sensors, barometric pressure, sound, magnetic field, vibration sensors, etc.

SunSPOTs incorporate a processor with a 32 bit ARM920T core which executes at 180 MHz maximum clock speed. The nodes are equipped with 512 KB RAM. In addition to the RAM, the nodes come with a 4 MB flash memory. The wireless network uses an integrated radio transceiver, the TI CC2420 (formerly ChipCon)

Figure 2.1: SunSPOT

which is IEEE 802.15.4 compliant ("ZigBee"). It is capable of transferring up to 250 Kbps. The nodes are powered with a 3.7 V rechargeable 750 mAh lithium-ion battery.

For battery conservation, a SunSPOT has available different states: run, idle and deep-sleep. Run means that all processors and the radio are running. In idle mode, the clocks and the radio are off. This mode is generally used to sleep for short periods of time (less than 2 seconds). For longer periods, the nodes enter deep-sleep. Memory contents are maintained as long as power is supplied, even during periods of deep-sleep.

SunSPOTs are programmable in Java. The Java VM ("Squawk") runs directly on the processor without any other operating system.

As a second example of sensor network platforms, we introduce Mica2 motes. They are sold by Crossbow Corporation. Mica2 motes incorporate a 7 Mhz processor, a 38.6 Kbps radio, 4 KB of RAM and 512 KB flash. The motes run on common AA batteries which for the most part determine the size of a mote. Commonly available sensors include light, temperature, humidity, vibration, acceleration, and position (via GPS or ultrasound).

Motes run an operating system called TinyOS [HSW+00]. TinyOS consists of a set of components for managing and accessing the mote hardware, and a programming language called nesC. In particular, the system provides a radio stack, which sends and receives packets and functions to read sensor values. In addition, it allows an application to put the device into a low power sleep mode.

2.1.2 Impact of the Platform on Data Management

Our solutions are not tailored to a specific hardware platform. We now abstract from concrete platforms and highlight those properties that affect data management. In particular, sensor nodes have constrained communication and computation capabili-

ties and are battery-operated.

Power Consumption. Because sensor nodes are battery powered, energy-efficiency is of utmost importance to obtain longevity of the network. Power is consumed by a number of factors such as sensing, communication, computation and accessing RAM. Writing to flash incurs non-negligible energy costs as well. However, our algorithms do not write to flash memory.

Typically, sensing and communication dominate the power consumption by orders of magnitude, compared to computation and accessing RAM [YG03, MFHH03]. In the context of query processing, sensing costs are important for evaluating selection predicates, as discussed in the next section. For operators other than selections, each algorithm will pay the same costs for running the sensors [AML05]. Therefore, in the context of this dissertation, minimizing the communication is key to obtain energy-efficiency. We will use communication as a proxy for energy consumption in the performance metrics in this dissertation.

In [Mad03], Madden argues that the capacity of batteries is not expected to increase by more than a small constant factor in the next ten years. Therefore, power will remain to be the primary limitation in the near future. Moreover, the ratio between the energy costs of communication and computation will even widen: While processors will become more integrated, the physical costs of pushing radio waves over the air are likely to remain constant [MFHH02]. Thus, communication will continue to dominate power consumption in the foreseeable future.

Communication. Communication is difficult in wireless sensor networks. Radio communication is lossy. For instance, [Mad03] mentions loss rates of 20% to 30% at a range of about 10 meters for Mica2 motes. Similar numbers are given in [WTC03]. Thus, retransmissions are critical in order not to loose large parts of the data. Retransmissions in sensor networks build upon link-level acknowledgments, indicating whether or not a message was received by the intended neighbor node.

Retransmissions allow to reliably communicate between neighboring nodes. However, common low-power radios enable communication ranges of only up to a few hundred feet [Mad03]. Therefore, communication in sensor networks is usually multi-hop. In particular, a node will often need to use neighboring nodes to relay data to the base station. This leads to the problem of routing, i.e., of choosing a node to do the forwarding. A particular problem in sensor networks is that signal strengths between devices vary frequently. Thus, connectivities (links) are changing on a regular basis. Additionally, sometimes nodes fail. Due to these problems, routes in sensor networks must continually be reinforced.

To accommodate the difficulties of communication in sensor networks, all of the algorithms presented in this dissertation build upon state-of-the-art communication protocols for sensor networks, e.g., [WTC03, KK00]. Among them, tree-based rout-

ing is the most fundamental for data collection. We will introduce it in the next section.

The networking community has made major efforts to cope with the aforementioned problems, i.e., a lot of work has gone into maintaining routes in light of changing connectivities and node failures and into providing reliability. To take advantage of their results, we clearly separate the networking from data management and do not tamper with the routes given by the networking layer. This is key to handling most of the errors that can occur. We will discuss error handling throughout the dissertation whenever mechanisms like retransmission and route maintenance do not suffice to obtain correct query results.

It is interesting to note that multi-hop routing will likely continue to be used in low-power networks in the foreseeable future, despite advances in communication hardware. This is because the energy to transmit goes up by a factor between the square and the cube of the distance between the sender and receiver, depending on the environment. Thus, even if long range communication was possible, a multi-hop topology is fundamentally lower power than a single hop topology.

Computation and Memory Constraints. Current generation sensor node platforms feature very limited processors. This will likely change – Moore's law suggests that the energy cost per CPU cycle will continue to fall as transistors become smaller and require less voltage. However, in the context of this dissertation, nodes perform only simple computations.

Finally, the available RAM is very limited. The development of the size of the memory is not that clear. Abadi et al. [AML05] argue that using large quantities of RAM is problematic as it would significantly increase the power consumption. However, they also admit that future generations of devices will certainly have somewhat more RAM than Mica2 motes. For instance, SunSPOTs substantiate this trend. The state that is maintained by our algorithms is several orders of magnitude less than, e.g., the 512 KB of RAM, that is provided by SunSPOTs.

2.2 Query Processing in Sensor Networks

This section introduces the fundamentals of query processing in sensor networks. After motivating the need of a declarative interface, we establish the abstraction of a sensor network as a relation (Section 2.2.1). It defines the semantics of queries. In addition, we sketch the process of answering queries, before we turn to efficiently processing the most important query operators in more detail (Section 2.2.2). Most of the basics that are presented in this section are results from the TinyDB [Tin] and Cougar [Cou] project. These projects were the first to build prototype query processors for sensor networks.

2.2.1 Declarative Queries

Engineering applications that collect data from sensor networks is challenging. The programming itself is difficult as the programming abstractions are low-level and debugging is usually done via a few LEDs on the device. But even worse, if data collection is implemented naively, the energy resources of the network are depleted in a few days [MFHH03]. To obtain efficiency, the algorithms have to be highly optimized and have to carefully manage energy and radio bandwidth. This requires major engineering efforts.

Query interfaces are attractive to overcome these problems [DGM+04, YG03]. As in traditional database systems, declarative queries describe a logical set of data that an application is interested in, but do not describe the actual algorithms or operators to collect this set. A query processor then takes care of efficiently collecting and processing requested data within the sensor network.

Beyond hiding the process of how the relevant data is actually retrieved from the application, declarative queries also have a performance advantage: The argument is the same as in traditional databases. Since queries do not specify how data has to be acquired, the system can choose a plan when it comes to evaluating a query. This is an advantage as, in some cases, it depends on the current data (measurements) which plan is the best. Thus, any hard coded algorithm to retrieve the data is suboptimal from time to time. Chapter 4 will illustrate this situation for join queries. There, the choice of the join algorithm depends on the current data.

To facilitate declarative queries, the network is seen as a *(sensor) relation*. For homogeneous networks, i.e., networks of identical nodes, there is one relation. One can perceive it as a relation with one attribute per sensor (e.g., temperature, humidity) and one tuple per node. For heterogeneous networks, groups of nodes form different relations. As the attributes reflect the sensors of a node, sensor relations typically have less than ten attributes.

Terminology. We say that *a node belongs to a sensor relation* R if it contributes a Tuple t to R.

Note that sensor relations are virtual relations. The term "virtual" means that the relations are not actually materialized. The systems only generate the attributes and rows referenced in active queries. The sole purpose of the relations is to settle the semantics of queries.

Queries can be one-time or continuous: A *one-time query* asks for the current snapshot, i.e., the virtual sensor relation at the particular instance of time – a snapshot contains one tuple per node. A typical usage of one-time queries is an interactive exploration. A *continuous query* reports snapshots periodically. Continuous queries are prevalent, e.g., in monitoring and surveillance applications.

The prototypes of query processors, TinyDB and Cougar, consider queries conforming to the following structure[1]:

[1]The prototypes provide some further, non-SQL constructs, e.g., to store data within the network.

```
SELECT Att_1, ..., Att_n, agg(Att_n+1), ..., agg(Att_m)
FROM Relation
{WHERE predicate(Att_i) AND ... AND predicate(Att_i')}
{GROUP BY Att_1, ..., Att_n}
{HAVING predicate(Att_j)}
ONCE | SAMPLE PERIOD x
```

The semantics is the standard SQL semantics with extensions for temporal aspects of sensor data: The `SELECT`-clause specifies required attributes and aggregates from the relation which is given in the `FROM`-clause. Optionally, the `WHERE`-clauses can narrow down the scope of the query. The `GROUP BY`-clause classifies tuples into different partitions according to some attributes, and the `HAVING`-clause eliminates groups by a predicate.

`ONCE` and `SAMPLE PERIOD` support one-time and continuous queries, respectively: `ONCE` computes the result based on the current snapshot. Thus, `SELECT temp FROM Sensors ONCE` returns one tuple from each node that belongs to `Sensors`. `SAMPLE PERIOD` yields a continuous monitoring. It defines the time interval between *independent* executions of the query. The query is executed every x seconds on the most recent snapshot.

Finally, for continuous queries, it is necessary to indicate when to stop their execution. In TinyDB and Cougar, this can be done either at the time the query is issued or at runtime [MFHH03]: Setting queries to run for a specific time period can be done via a special clause (`FOR`-clause). Alternatively, when an application issues a query, the system assigns the query an id, which is returned. Using the id, the application can stop a query via a `STOP QUERY` command at any time.

2.2.2 A Query Processing Architecture

At a high level, query processing involves the following phases:

1. Dissemination of the query within the network.
2. Efficient collection of (preprocessed) data.
3. Computation of the final result at the base station.

In sensor networks, the base station cooperates with the nodes to compute the result, i.e., parts of the processing is done inside the network (Step 2, 'preprocessed' data). The base station computes the final result (Step 3). In particular, the sensor nodes and the base station have different responsibilities. We now discuss query processing in more detail.

They are very specific and knowing them is not needed to follow this dissertation. We refer the interested reader to [Tin, Cou], in particular to [MFHH03].

A query is input at the base station where it is parsed just as in traditional database systems. It is then optimized: As declarative queries include no specification of how the required data is to be collected, the system can choose among different algorithms (query plans) that have the same logical behavior. It chooses the one which is expected to offer the best performance in terms of communication costs.

Example 2.1: Consider the following, simple query which collects the current sensor readings from all nodes that observed a temperature of more than 24°C:

```
SELECT *
FROM Sensors
WHERE temp > 24.0°C
ONCE
```

One way to process this query is to collect the data from each node at the base station and to evaluate the query there. Alternatively, the nodes can perform the selection and send their data to the base station, if and only if the predicate evaluates to true. Obviously, the latter plan results in lower communication costs and would be chosen. ■

In general, pushing selections and projections to the data sources is the best choice in sensor networks. This is due to the optimization goal of minimizing communication. However, as mentioned above, for other operators the situation is not that clear. Sometimes it depends on the current sensor readings how to process a query. This is why query optimization is important in sensor networks as well. At a very high level, query optimizers work by enumerating a set of possible plans, assigning a cost to each plan based on estimated costs of each of the operators, and choosing the lowest-cost plan. Note that query optimization occurs as much as possible at the base station, as it can be computationally intensive.

After choosing a plan, the system disseminates the query within the network. Subsequently, query results are propagated to the base station. In sensor networks, the basic primitive for data dissemination and collection is a routing tree [WC01]. It is a spanning tree rooted at the base station over the radio-connectivity graph of the network.

Before we discuss the processing of the most important query operators within this architecture, we describe the routing structure in more detail as it is fundamental for this work.

Collection Trees. The query processor is responsible for maintaining the routing tree, independent of active queries. 'Maintaining' means to adapt the routing structure to changing connectivities and node failures. A routing tree is maintained in a distributed fashion: Based on a periodic beaconing mechanism, each node maintains a parent that minimizes the distance to the base station. In a state-of-the-art

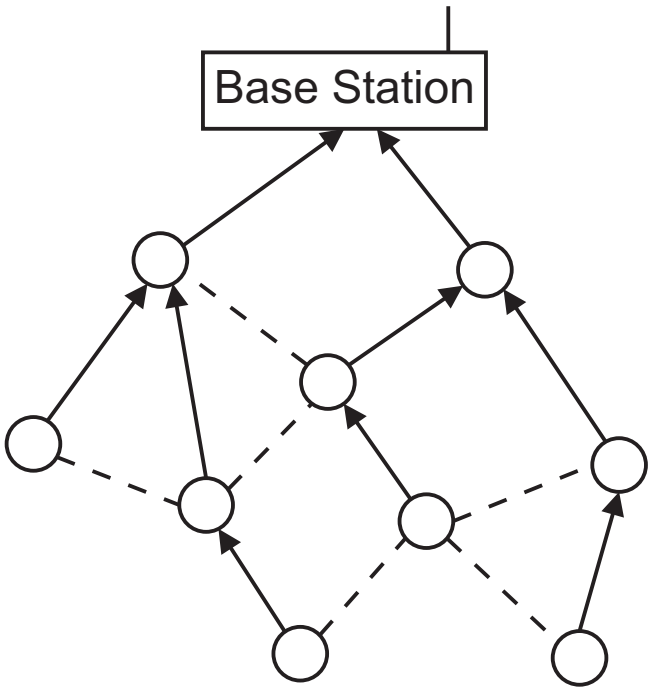


Figure 2.2: Routing tree as an overlay over the connectivity graph of the network

implementation, the distance is measured in 'number of expected transmissions'. Intuitively, this is the hop count weighted with the link quality[2]. This guarantees an important property of routing trees: They are shortest path trees, with respect to the distance metric. Figure 2.2 shows an example sensor network topology and a routing tree. Dotted lines indicate nodes that can hear each other but do not use each other for routing. Note that routing trees are highly irregular, i.e., the degree of the nodes can vary arbitrarily (in practice, a node has about 0 to 15 children), and the tree is by no means balanced.

The details of maintaining routing trees are as follows: The base station sends a beacon. All child nodes that hear this beacon forward it on to their children, and so on, until the entire network has heard it. Each beacon contains the distance from the broadcaster to the base station. To determine their own distance to the base station, nodes pick a parent node that minimizes the distance to the base station and add their distance to the chosen parent. Actually implementing this protocol is complicated as it requires to deal with routing loops, asymmetric links, and so on. For a complete discussion of these issues, including the details of the distance metric, see [WTC03].

Routing trees have several properties that make them a good match to sensor net-

[2]More precisely, with the inverse of the link quality. The intuition is that the number of retransmissions increases with decreasing link quality. This "prolongs" a path in terms of the number of expected transmissions.

works. In particular, there is only little state due to routing that has to be maintained by each node. It is basically a list of neighbors along with their current distance to the base station. Among these neighbors, a node chooses its parent. In addition, routing trees enable a simple mechanism to synchronize the communication. In order to conserve energy, it is mandatory for the nodes to sleep for most of the time. The problem then is for each node to know when to wake up to receive data and when to forward data (that is, to know, at what time the parent is listening to its radio). If communication is performed along a routing tree, a node can use its level within the tree to determine when to wake up. Here, the level is simply the hop-count to the base station. Briefly, the idea is to have "intervals" of time of a pre-defined length. Then, a node at level l listens to the radio in the same interval in which a node at level $l+1$ sends data. Nodes at levels other than l and $l+1$ sleep during this interval. In the subsequent interval, nodes at levels $l-1$ and l communicate. [MFHH02] describes the details of synchronizing nodes within a routing tree.

After having described query processing at a high level, we now turn to regarding some operators in detail. In particular, we discuss those operators involved in the queries presented in Section 2.2.1:

- Selection
- Aggregation
- Projection

The principle of achieving efficiency for these operators is to exploit their selectivity. The idea is to push the operators into the network in order to reduce the data volume as close to the sources as possible. For instance, in case of selections and projections, the operators select a subset of the sensor relation to be relevant to the query. By executing these operators on the nodes where the data originated, irrelevant data is not at all sent.

Selection. Processing selections is straightforward and has already been described in Example 2.1. Each node applies the selection predicates to its current tuple. It discards the tuple if any of the predicates evaluates to false. In this case, the node sends nothing.

Aggregation. For aggregates that can be incrementally maintained in constant space, i.e., for so called distributive and algebraic aggregates, the idea is to compute partial results within the routing tree. For instance, an average can be computed incrementally by maintaining the sum and a count for the data seen so far. Each node receives the pair (sum, count) from all of its children in the routing tree. Then, it aggregates them, i.e., it computes the overall sum and count, including its own

data. The node then forwards this aggregate, covering its entire subtree, to its parent. Finally, the base station computes the overall average.

An idea that is related to distributing the computation of aggregates is packet merging (which some authors in the literature on sensor networks refer to as 'aggregation' as well). Packet merging is based on the observation that sending multiple small packets is more expensive than sending one large packet (considering the costs of reserving the channel and the packet header overhead). Therefore, for collecting data along a routing tree, a node receives the tuples of all of its children and merges them into a large packet. This way, it pays the per-packet overhead only once.

Projection. The term 'projection' is actually a misuse in sensor networks. Recall that sensor relations are virtual relations. In particular, only those attributes are materialized, that are relevant to a query. Therefore, 'acquisition' of the attributes (or 'sampling'/'reading' the sensors) is more appropriate than projection[3]. Sampling the sensors opens up a major potential for energy savings. In particular, reading the sensors can consume a lot of energy, depending on the type of the sensor. For instance, sampling the accelerometer is three orders of magnitude more expensive than obtaining the current temperature on a Mica2 mote [MFHH03]. The following observation is key to optimizing data acquisition on the nodes: Due to the conjunctions in the query, if one of the predicates evaluates to false, then subsequent predicates need not examine the tuple – and hence the expense of sampling any attributes referenced in those subsequent predicates can be avoided. [MFHH03] discusses how to optimally order the evaluation of predicates (and thus, the sampling of the sensors). Basically, the idea is to weigh off the costs due to the type of the sensors involved and the selectivity of the predicate. With respect to the costs of sampling, it is preferable to start with those predicates that refer to cheap sensors. With respect to the selectivity, it is better to first evaluate highly selective predicates, as in traditional databases. The details of ordering the sampling of sensors are not required in the remainder of this dissertation.

As a last mechanism which we briefly introduce in the context of query processing, [MFHH03] proposes so called "semantic routing trees" (SRT). Their purpose is to avoid distributing a query to nodes with non-satisfiable predicates over constant-valued attributes. As an example, in our case of stationary sensor nodes, the location attributes are constant-valued, i.e., the coordinates of a node. In principle, an SRT is an index. It allows each node in the routing tree to decide which attribute values occur in its subtree. Based on this knowledge, the node can refrain from forwarding a query, if all of its descendants are known to not satisfy a predicate.

[3] As a remark, note that for obtaining the sensor readings that are relevant to a query, the system needs to assure that each node reads its sensors at (approximately) the same instance of time. This is due to the snapshot semantics of queries.

2.3 Discussion

In this chapter, we introduced our notion of a sensor network. In particular, we discussed how the hardware affects data collection in sensor networks: We derived the optimization goal of minimizing energy consumption. In addition, the subtleties of wireless communication affect our work – choosing routes is very complex and has received substantial attention by the networking community. Our algorithms avoid modifying the routes that are given by the routing layer.

We also introduced the fundamentals of query processing in sensor networks. The results are based on two projects, TinyDB and Cougar, that were the first to develop prototypes of query processors. In particular, we presented the semantics of queries in sensor networks, which is based on an abstraction of the network as a virtual relation. We presented routing trees which are the basic primitive for data dissemination and collection. Finally, we discussed the idea of pushing operators into the network in order to reduce the data that is involved in the query as early as possible. This idea has been illustrated for selections, projections and aggregations.

In contrast, existing systems do not support join operations well. However, the join is important as it allows for combining data from different nodes, as we have motivated in Chapter 1. Cougar does not feature an in-network implementation of joins. The authors argue that an in-network join can sometimes be beneficial [YG03]. However, no details are provided. TinyDB allows to join tuples that are located on the same node only. Each node can materialize its sensor data for a specified interval. TinyDB provides a join operation between two materializations or a materialization and the current data.

In addition, the idea of discarding tuples as early as possible does only result in savings if some of the tuples actually are irrelevant to a query. However, as we have motivated in Chapter 1 as well, there are important scenarios in which applications simply consolidate the sensor relations without any a priory selection. For such non-selective queries, the data volume is high and thus, query processing is expensive.

In the following, we address these issues, starting with computing joins in the next chapter.

3 Join Processing in Sensor Networks

We now turn our attention to the first integral part of this dissertation: We consider the situation that applications have a well-separated interest, i.e., only specific parts of the measurement data are relevant. Thus, the queries issued by such applications are usually selective. In Chapter 1, we used event detection in industrial settings as an example. There, certain patterns in the data indicate faults that require human intervention.

For such applications, queries are not only used to get the current sensor readings out of the network. As in traditional databases, queries are also used to search for the relevant data within the massive amounts of sensor readings, and to discard irrelevant data from the query result. This yields selectivity.

To acquire relevant data, the application specifies the data of interest by means of a classical Select-Project-Join (SPJ) statement. Among these operators, processing selections and projections is well understood. Efficient implementations exist in data management systems for sensor networks such as TinyDB and Cougar, as discussed in Chapter 2. In contrast, efficiently processing join queries in sensor networks is an open problem. As motivated in Chapter 1, the join is needed to relate sensor readings from different nodes to each other. This chapter provides the foundations of our work on join processing. We first state the problem and its intricacy in detail (Section 3.1). We then review related approaches and illustrate why they are not adequate as a solution to the join-processing problem (Section 3.2).

3.1 The Problem: "Join Processing in Sensor Networks"

In this section, we explain why join processing is more difficult than handling projections and selections. We then formally state the join-processing problem, that our algorithms in Chapters 4 and 5 address. Finally, we derive lower bounds on the communication costs for this problem. Knowing the lower bounds will be important for understanding a number of design decisions of our algorithms.

3.1.1 The Difficulty of Processing Joins

For query operators such as selection and projection, communication can be minimized based on the following key property: A node can decide *locally* that some data is not required for the result and can refrain from sending it. For the join, the picture is different: A node does not know if there exists a join partner for a tuple somewhere in the network. Most notably, tuples which have to be joined can be arbitrarily distributed. Thus, matching tuples is communication intensive.

As an illustration, consider the following example, which is based on one of the queries from Chapter 1 – for convenience, we have included the query in the example:

Example 3.1: The following query acquires data from nodes that observe similar temperature and humidity conditions:

```
SELECT A.*, B.*
FROM Sensors A, Sensors B
WHERE |A.hum - B.hum| < 2%
AND   |A.temp - B.temp| < 0.3°C
AND    A.id != B.id
SAMPLE PERIOD 30s
```

From the point of view of a single node, its join partners can be *anywhere* in the network. Consider the alternative: If the query contained an additional join condition such as `distance(A.x, A.y, B.x, B.y) < 40m`, the location of potential join partners of a node would be restricted. Most notably, potential join partners are known to be located on *nearby* nodes. This can obviously be exploited by a node to limit the search for join partners, and thus to reduce the communication costs of determining if its tuple joins. In contrast, computing the join in the general case requires a global matching of tuples, i.e., it requires matching all pairs of tuples. ■

Note that the global matching is inherent in the problem – it stems from the semantics of the join operator. This makes efficient join processing difficult in sensor networks. Prior join approaches avoid the costs involved in a global matching by restrictions in the queries or in the locations of the input tuples within the network. For instance, REED [AML05] supports the comparison of sensor data to pre-defined patterns, given as a static, external relation. [YLOT07] presents a join method for tracking rare events, i.e., one of the relations must contain a few tuples only. In addition, there are methods that restrict the placement of the tuples involved. For instance, some require the tuples to be located in small regions that are close to each other, e.g., [BB04, CNS07, YLZ06] (cf. Section 3.2). Such specific requirements restrict applicability. In particular, no join method we are aware of can efficiently process queries in the style of Example 3.1. Our focus in turn is on efficient *general-purpose* join methods, as introduced subsequently.

3.1.2 Problem Statement

We call a join method "general-purpose" if it fulfills the following requirements:

Requirement 3.1 (Join Conditions)
A general-purpose join method must be able to handle any number and any kind of θ-join conditions.

This is because join queries can easily have more than one join condition. Any attribute can appear in a join condition. In particular, the θ join is the most general join operator that is defined in SQL[1]. We consider standard θ-join conditions, involving $\{\leq, <, >, \geq, =, \neq\}$, without any further restrictions. For instance, as in Example 3.1, the conditions may contain absolute value operators to realize similarity joins.

Requirement 3.2 (Tuple Distribution)
A general-purpose join method must be able to efficiently handle queries with arbitrary placements of the tuples involved.

Due to Requirement 3.1, a general-purpose join method has to support a global matching. Requirement 3.2 captures that the join method should still provide efficiency, even if the query demands this global matching.

A general-purpose join method complements the existing, specialized ones which are geared towards a certain scenario. While their performance is very good when they are applicable, the underlying assumptions are strict and are frequently not met.

We now state the structure of general-purpose join queries formally:

```
SELECT A.attrs, B.attrs
FROM Relation_1 A, Relation_2 B
WHERE predicates(A) AND predicates(B)
AND join-expr(A.join-atts, B.join-atts)
AND ... AND join-expr(A.join-atts, B.join-atts)
SAMPLE PERIOD x | ONCE
```

We require the join conditions to be arbitrary θ-join expressions over the join attributes. That is, we do not restrict the kind or number of join conditions. In the special case of a self-join, the `FROM` clause contains the same relation twice. The queries may contain `WHERE`-clauses to narrow down the scope of the query. As described in Chapter 2, the corresponding selections are executed locally at the nodes and thus prior to the join. After executing the query, the join result is available at the base station.

Currently, the only join method that supports arbitrary join queries is the "external join":

[1] SQL allows for user-defined functions to extend the set of operators [EN07]. These can be used in join conditions as well and can thus extend the set of possible join operators. By considering θ-joins, we focus on the set of *pre-defined* join operators.

Definition 3.1 (External Join)
The external join sends the tuples from the entire input relations to the base station where the result is computed. It uses packet merging to aggregate different tuples as they move up the routing tree – this reduces the number of packets. Finally, it performs projections and selections as early as possible.

The problem with the external join is that it requires sending many tuples that do not contribute to the result, and efficiency is low. Example 3.1 serves as an illustration. If there are only a few tuples with similar temperature and humidity values, the external join sends a large number of tuples that are not needed.

Intuitively, an optimal join algorithm, i.e., a join algorithm that minimizes communication costs, would refrain from sending irrelevant data. This observation motivates the following section: We will derive lower bounds on the communication costs of processing join queries. The corresponding discussion includes some insights which will guide the design of our join algorithms.

3.1.3 IDEAL Join Processing

The following concept ("IDEAL") lower bounds the communication costs of a join: Firstly, each node discards its tuple if it does not join. Then, the remaining tuples are sent to the base station where the join is computed.

IDEAL is infeasible in practice: Each node would have to know if its tuple joins. This depends on the data of the other nodes and is actually the result of the global matching that we discussed earlier. Thus, IDEAL really is just a lower bound.

In the following, we formally define IDEAL and provide an intuition why IDEAL lower bounds the communication costs of join processing. A thorough theoretical justification is given in Appendix A. However, the intuitions are sufficient to understand the design of our algorithms which we present in Chapters 4 and 5.

Definition 3.2 (IDEAL)
IDEAL refers to the following process:

1. *Discard non-joining tuples at the sources.*
2. *Collect the remaining tuples at the base station.*
3. *Join the received tuples at the base station to compute the result.*

Step 1 is based on the following simple observation: In theory, we can divide each input relation into the tuples that join and those that do not. The non-joining tuples do not influence the result in any way. Thus, it is cost-efficient to not send them.

A subsequent question is where to join the remaining input tuples. *After discarding tuples that do not join*, it is optimal to perform the join at the base station. Although

surprising at first glance, this is actually intuitive: After discarding non-joining tuples, the join increases the cardinality, i.e., there are more tuples in the result than in the input, except for pathological cases. This leads to Steps 2 and 3. Note that Steps 2 and 3 correspond to the external join, except that they involve only joining tuples.

Having introduced the join-processing problem along with some insights on join processing, we subsequently discuss related works. In particular, we justify the statement that their applicability is limited by pointing out in which scenarios they are efficient.

3.2 Related Work on Join Processing

There is extensive work on join processing in the database literature. In particular, our problem is to process a join in a distributed environment where the relations are horizontally fragmented. We therefore review join processing in distributed environments in Section 3.2.1. It turns out that traditional approaches do not apply to sensor networks. As a consequence, join processing in sensor networks has evolved as a topic of its own. We survey approaches specific to sensor networks in detail in Section 3.2.2.

We leave aside approaches for join processing that do not focus on the costs of data access. These consider different problems. For instance, there is work on parallel join processing, e.g., [RLM87, LST91]. The question there is how distribute the processing to multiple processors. In particular, the problem is how to split up the computation while guaranteeing not to miss a tuple in the join result. The solutions usually are to hash or to sort the relations. Also, work on joins over data streams studies a different problem, e.g., [DGR03, HAE08, KNV03]. The focus is on designing non-blocking operators that can process infinite streams.

3.2.1 Distributed Join Processing

Traditional techniques from distributed databases assume each of the relations to be fragmented across a few site(s). In contrast, sensor relations are highly fragmented – each node has a single tuple. Therefore, the corresponding join methods do not apply to sensor networks. Subsequently, we will illustrate this point based on the solution from INGRES and System R*. Other systems such as SDD-1 [BGW$^+$81] do not consider fragmentation at all. To use their join methods, each of the relations would have to be consolidated at a single site. However, this is expensive in sensor networks and is what the corresponding approaches actually try to avoid.

INGRES [ESW78] can handle fragments. For joining fragmented relations, INGRES checks whether it is advantageous to use more than one processing site. In more detail, INGRES first restricts the set of potential processing locations to those sites that store fragments of the larger relation in the join. Among these sites, the

location with the largest amount of data is the default join location. To determine if any other site within the set should be used, INGRES applies a heuristic that is based on the following observation: If a site is chosen as a processing site, it has to receive the entire other (smaller) relation. Further, if the site stores a fragment of the smaller relation itself, it also has to send this fragment to the other processing sites. In contrast, if the site is not chosen, it does not need to receive any data. As before, it has to send its fragment of the smaller relation to all processing sites. Additionally, it has to send its fragment of the larger relation to one of the processing sites. The heuristic is: If the amount of data that the site must receive is larger than the additional amount of data it would have to send if it were not a processing site, it is chosen.

This heuristic makes sense given that the relations are fragmented among a small number of sites and thus, the sites have a substantial part of the relations. In contrast, in sensor networks, each node has a single tuple. For this setting, the heuristic devises a single processing location. Note that there is no obvious extension to our scenario: The whole idea of comparing the data to send/receive does not apply if each node has exactly one tuple. As a final remark, observe that the heuristic ignores the costs of transmitting the data to the result site (base station in our case).

The implemented version of R* does not support fragmentation [OV99]. However, the algorithm that has originally been described in [SA80] does discuss the handling of fragments. The idea is that a query on a table T which is horizontally fragmented into n fragments can be transformed into a union of that query on each of the fragments ($Q(T) \rightarrow UNION(Q(T_1), ..., Q(T_n))$). Then, the optimizer considers each of the subqueries separately.

This idea assumes that the costs of the subqueries are independent of each other. We briefly illustrate why the independence assumption is violated for joins. Assume that we apply the preceding idea in our context, which is a join on two highly fragmented relations. This effectively means to consider the join of each pair of nodes separately. In particular, for a network of n nodes, we would devise on the order of n^2 (optimal) join locations. Then, each tuple is sent to up to $(n-1)$ sites which is prohibitively expensive. The problem is that there is the potential of sharing tuples for multiple subqueries, if a number of tuples is sent to the same site. This is why we cannot consider the join locations in isolation, i.e., why the subqueries are not independent of each other.

We remark that [SA80] did not propose to integrate this idea with processing joins. In particular, the paper does not say how to handle joins if the relations are fragmented.

The preceding papers reflect early work on distributed query processing. Handling fragmentation naturally was not the first issue to address. However, a few years later, it was realized that communication is not the bottleneck of the processing time – in contrast to what early work assumed. Thus, fragmentation was not an issue anymore. Simply consolidating the relations is an option in this context.

3.2.2 Join Processing in Sensor Networks

We already discussed in Chapter 2 that both TinyDB and Cougar initially did not support join operations. However, Abadi et al. extended TinyDB with the ability to support joins between sensor data and static tables built outside the sensor network [AML05]. Their system, REED (Robust and Efficient Event Detection), targets the detection of events that are specified as tuples of a static external relation. The basic idea of REED is to distribute the static relation within the network such that each node can access it at little costs. As further contributions, the authors describe a number of interesting optimizations. Most notably, if the relation is too large to fit into the memory of a single node, REED allows to store the relation among a group of nodes that are within communication distance to each other. While this latter requirement is important for accessing the static relation at little costs, it restricts the size of a group. If even a group cannot store the relation, REED proposes to distribute so called 'empty range descriptions' (ERDs). The idea is to capture parts of the join-attribute space that do not join. This allows to discard non-joining tuples within the network. In particular, ERDs are distributed based on a caching mechanism. This way, the nodes can access the most useful ERDs at little costs. This design exploits that one of the relations is static. In contrast to REED, our problem is to compute joins over sensor relations.

The difficulty of joins over sensor relations stems from the semantics of the operator. The join relates tuples from arbitrary locations to each other. This makes the processing communication-intensive. The following approaches avoid a global matching by specializing to specific types of queries – at the expense of applicability.

Join methods for specific types of queries. Yang et al. [YLOT07] investigate the tracking of rare events. Due to the tracking aspect, the tuples to be joined stem from different points in time, i.e., the query is a window self-join. The algorithm involves two phases: In the first phase, one of the relations is materialized at the base station. Then, similarly to REED, it is distributed to serve as a filter. As a main contribution, the authors show how to suppress distributing the filter relation to achieve an efficient continuous execution. This is possible if it is already contained (in part) in an earlier filter.

An important requirement for this approach to be efficient is that one of the relations is very small. In the evaluation, it consisted of a single tuple. This is because it is materialized at the base station as a first step. In contrast, consider the query in Example 3.1. It does not restrict any of the relations. Thus, the first step would consolidate the entire relation and already incurs the costs of the external join. In contrast, [YLOT07] considers tracking rare events. Thus, one of the relations is supposed to express this rare event, and the assumption that it is small is reasonable. Beyond that, their approach is restricted to one join attribute besides the mandatory temporal one. Our goal is not to impose such constraints.

Yiu et al. [YMB07] consider spatial joins. Their approach joins tuples from neighboring nodes, i.e., the join condition is $distance(A, B) \leq d$ where d is less than the communication range. The idea is that each node of Relation A broadcasts its tuple. Each node of Relation B performs the join and sends the result to the base station. In addition, the authors propose a number of extensions. For instance, they propose a protocol for a distributed join processing if the join condition does not restrict the distance to one hop but to a small number of hops. However, the evaluation indicates that their solutions can reduce the communication costs mainly in the case where the distance is less than the communication range. In contrast, our goal is to allow for arbitrary join conditions and tuple distributions.

Solutions that do not restrict the queries actually have to perform the global matching that the preceding approaches avoid by restricting the queries. The following approaches do so by computing the join at a central location inside the network.

Centralized Join Approaches. We first introduce the centralized approaches before pointing out their disadvantages. Bonfils et al. [BB04] study long-running join queries. They reduce the problem to a task-assignment problem. The approach starts by arbitrarily mapping the operators from the query onto the nodes in the sensor network. Then, the idea is to stepwise adapt the placement to minimize the communication costs. In more detail, the authors distinguish 'active' nodes from 'tentative' nodes: Active nodes are those that currently execute an operator. Tentative nodes are their neighbors. The tentative nodes estimate the costs of executing the operator and periodically compare their estimates to the costs incurred at the active node. The operator is relocated if it is cheaper to perform it on one of the tentative nodes.

Chowdhary et al. [CG05] propose an approach to process sliding-window joins inside the network. The authors restrict the set of join locations to not contain the base station. Under these circumstances, they present an approach for executing the join within an central region inside the network. In particular, their approaches collect the tuples of two relations, R and S, within a region around the centroid of the relations. Again, we need a region, i.e., more than one node, due to memory constraints. Whenever a tuple of R arrives, it is compared to all of the tuples in S (and vice versa). Due to the sliding-window semantics, the tuples can be expired after some time. As a main contribution, the authors reason about the optimal shape of the region. [PG06] optimizes the approach for range join and equi-joins. While in [CG05], a new tuple has to be sent to all nodes in the region to compute the join result, the idea in this follow-on work is to organize the tuples within the region by means of a hash or index data structure. Then, a new tuple has to be sent to one or a few nodes only as identified by the data structure.

Coman et al. [CN07] address the data flow for a centralized processing in order to meet the memory constraints of the nodes. In detail, the authors consider computing a join between two regions of the network. The algorithm first elects a coor-

dinator within each region and establishes a routing tree with the coordinator as the root. Then, a join region is established. In particular, the join region also identifies a coordinator. The coordinators are responsible for exchanging data between different regions. First, the tuples of the larger region are stored within the join region. Afterwards, the coordinator of the join region pulls the tuples from the other relation/region. In particular, the coordinator pulls each of the packets separately as the processing within the join region incurs substantial communication. This is key to assuring that a packet is processed before the next one arrives.

These centralized approaches are restricted in their applicability to very specific distributions of the input tuples in the network. In Appendix A, we study theoretically in which scenarios a centralized approach is more efficient than the external join. The results are that the input relations have to stem from two small regions. This corresponds to the setting that [CN07] considers. In addition, the regions need to be close to each other, compared to their distance to the base station, and the selectivity has to be very high. In particular, for queries such as Example 3.1, which do not restrict the input relations, the external join is much more efficient than any of the centralized variants.

Decentralized Join Approaches. The centralized approaches compute the result inside the network. We know of only one approach that computes the result inside the network but not at a central location: Zhu et al. [ZGT09] decentralize the processing. To do so, one of the relations is distributed along a line of nodes. Then, the other relation is sent along a perpendicular line such that each pair of tuples meets. This idea requires a grid structure of the network. As their main contribution, Zhu et al. discuss how to establish such a grid structure as an overlay, if the topology is not a grid. Finally, the authors present an optimization that restricts the distribution of tuples and the probing, if the join condition is $distance(A, B) \leq d$. The most important motivation of this work is to balance the load of the nodes: For tree-based data collection, the nodes close to the root are the most loaded. The distributed approach by Zhu et al. overcomes this problem, if the selectivity is very high, i.e., if the result is almost empty. Otherwise, the costs of sending the result to the base station dominate the processing costs. Also, the overall costs are higher than those of the external join because the routes are longer and the distributed approach cannot exploit packet merging. This is true even if the result is empty. All in all, even if the load-balancing is worth an increase in the overall costs, this approach achieves a load distribution only if the cardinality of the result is extremely small. Thus, the applicability of this approach is severely limited.

Finally, some approaches share the idea of IDEAL which is to discard tuples to achieve efficiency. They all build upon the semi-join idea and use the join-attribute values to find out which tuples join.

Semi-Join Based Filtering. With respect to join processing, the filtering idea has been explored originally in the context of distributed databases. The semi-join filters one of the relations based on the join-attribute values of the other relation. This idea has been extended to filtering all of the relations ("N-way semi-join") [RK91]. We already discussed join processing in distributed databases. The algorithms are not applicable in sensor networks due to the fragmentation.

For sensor networks, there are two approaches that apply a filtering. In [CNS07], Coman et al. present a semi-join approach where the join-attribute values of one of the relations are broadcast over the nodes of the other relation to discard non-joining tuples. Then, the join-attribute values of the remaining tuples are used to discard non-joining tuples in the first relation. Finally, the remaining tuples of both relations are sent to the base station where the result is computed. However, simply collecting the tuples of one relation and broadcasting them over the entire nodes of the other relation is communication-intensive – except if both relations correspond to very small regions. In particular, this approach suffers from the same restrictions as the centralized ones. As a further contribution, Coman et al. [CNS07] compare this approach to a number of others, in particular to the external join. In addition, as none of their methods outperforms the others in all scenarios, they present a cost-based model to select among different approaches.

Yu et al. propose an approach that also focuses on joining two regions [YLZ06]. In both regions, a histogram synopsis is constructed. The synopses are then sent to a central location where a semi-join is computed. In addition, for each joining tuple, the authors propose to compute the optimal join location(s). However, as discussed in Appendix A, this location often is the base station. The problem is that computing optimal join locations requires collecting location information if the coordinates are not part of the join attributes. This is a huge overhead. In addition, the optimal locations also have to be disseminated. Finally, [YLZ06] does not say how to construct a compact histogram to compute a multi-dimensional join. All in all, the approach only yields energy savings if two small regions are joined and if the query is highly selective.

To conclude, there currently is no efficient general-purpose join method. All prior approaches require restrictions with respect to the type of queries or the distribution of the input tuples. In particular, we highlighted in this section why for a query as given in Example 3.1, the external join outperforms all of the prior join approaches and is state-of-the-art.

3.3 Summary

In this chapter, we illustrated the difficulty in join processing – it stems from the semantics of the operator: The join relates tuples from arbitrary locations to each other. Therefore, in contrast to projections and selections, a node cannot decide locally if

its tuple contributes to the join result.

Current join approaches avoid a global matching by specializing to specific types of queries or placements of the input tuples. This comes at the expense of applicability and motivates our focus on general-purpose join methods. The latter avoid such specializations. Finally, we introduced IDEAL, a concept that lower bounds the communication costs for the join-processing problem. IDEAL will justify some of the design decisions in the subsequent chapters.

We now focus on our solutions to the join-processing problem. In particular, in Chapter 4, we introduce SENS-Join, a general-purpose join method. While SENS-Join can be applied to one-time queries as well as to continuous join queries, it actually targets one-time queries. For continuous join queries, we show in Chapter 5 how to further reduce the costs, coming close to the lower bound.

4 Efficiently Processing General-Purpose Joins

In this chapter, we present SENS-Join, an efficient general-purpose join method for sensor networks. SENS-Join can process any kind of join query (e.g., equi-joins, similarity-joins, or theta-joins) over any number of join conditions.

We start the description of SENS-Join with an overview in the following section. It highlights the contributions of SENS-Join and outlines the structure of the remainder of this chapter. Up front, we briefly introduce two simple join queries, which will serve as examples throughout our presentation of SENS-Join:

Example 4.1: The first query returns the minimal distance between two nodes with a temperature difference of more than ten degrees:

```
SELECT MIN(distance(A.x, A.y, B.x, B.y))
FROM Sensors A, Sensors B                          (Q1)
WHERE A.temp - B.temp > 10.0
ONCE
```

■

Example 4.2: The second query checks the similarity of the humidity and pressure readings of two nodes that observe a similar temperature. As a further join condition, the corresponding nodes have to have a minimum distance of 100 m (e.g., the application wants to exclude the influence of spatial correlation on the result):

```
SELECT |A.hum - B.hum|, |A.pres - B.pres|
FROM Sensors A, Sensors B
WHERE |A.temp - B.temp| < 0.3                      (Q2)
AND distance(A.x, A.y, B.x, B.y) > 100
ONCE
```

■

4.1 Overview of SENS-Join

To obtain efficiency, SENS-Join incorporates the idea of IDEAL: It discards non-joining tuples at the sources, i.e., SENS-Join avoids shipping tuples that do not join. As discussed in Section 3.1.3, the problem in turning this concept into a practical

approach is to supply each node with the knowledge if its tuple joins. SENS-Join solves this problem by introducing a *precomputation* which identifies the tuples that join. In a subsequent *final-result-computation* step, SENS-Join sends joining tuples to the base station and computes the join. Most notably, this final result computation actually corresponds to IDEAL. At a high-level, the main contribution of SENS-Join is the design of the precomputation: It is highly efficient in order not to exhaust the savings due to discarding non-joining tuples.

To accomplish an efficient precomputation, we base it on a well known idea from distributed databases, the (N-way) semi-join [RK91]: A relation is filtered based on the join-attribute values of the other relations. However, semi-join algorithms as proposed earlier [OV99] are not applicable here: Relations in sensor networks are highly distributed. Applying a conventional semi-join algorithm requires consolidating the relations at one or a few (2, 3) site(s). In sensor networks, this is prohibitively expensive. Prior attempts to deploy the semi-join idea in sensor networks [CNS07, YLZ06] have not resulted in a general-purpose solution (cf. Section 3.2). So far, it is unclear how to realize a semi-join approach in sensor networks.

SENS-Join addresses this problem. At a high level, the precomputation consists of two steps: (a) The join-attribute values of both relations are sent to the base station where they are joined in order to create a join filter. The join filter specifies which values will be in the result. (b) The join filter is disseminated in the network. This precomputation is followed by the final result computation in which all tuples that match the join filter are sent to the base station where the result is computed.

Our main innovations are the following features, which are key to an efficient precomputation, despite the need to match each pair of join-attribute values:

- **Information Flow:** If the precomputation is realized naively, each node is involved multiple times in processing a join query. SENS-Join comprises two mechanisms that achieve to involve most of the nodes only once – this significantly reduces the number of overall transmissions: "Treecut" sometimes sends complete tuples instead of join-attribute values to avoid their transmission in a later step. "Selective Filter Forwarding" is pruning the filter progressively during its dissemination, depending on the data distribution.

- **Quadtree Representation of Precomputation Data:** We propose a new compact representation of the join-attribute values. These are sent during the precomputation. In SENS-Join, the data used in the precomputation is independent of the tuples used for the final result computation. We can thus reduce their accuracy without sacrificing correctness of the final result. In addition, our mechanism exploits spatially correlated sensor readings by encoding join-attribute values using a spatial index. Compact representations like Bloom Filters cannot be applied here since they do not allow for evaluating arbitrary join conditions.

We extensively evaluate the performance of SENS-Join. Our results indicate that SENS-Join can reduce the overall energy consumption by more than 80% compared to the state-of-the-art. It can reduce the per node energy consumption of the most loaded nodes by more than an order of magnitude.

Chapter Outline. In Section 4.2, we describe SENS-Join in a top-down manner. In particular, we present the mechanisms Treecut and Selective Filter Forwarding, which achieve an efficient information flow. In Section 4.3, we describe the quadtree representation which yields a compact encoding of the join-attribute values. Finally, we present an experimental study in Section 4.4 that highlights the efficiency of SENS-Join.

4.2 The SENS-Join Approach

To enable a precise description of SENS-Join, we start with defining "join-attribute tuples". A join-attribute tuple is the data of a node that is involved in a semi-join:

Definition 4.1 (Join-Attribute Tuple)
Let Q be a join query. A join-attribute tuple T' is a tuple that results from the projection of a tuple T on the join attributes of Q, i.e., $T' = \pi_{JoinAttr}(T)$.

Example 4.3: For Query Q2, a join-attribute tuple contains the X- and Y-coordinates and the temperature. ■

We now present SENS-Join in detail. To achieve efficiency, SENS-Join incorporates IDEAL as a second step. In the following, we first present our approach before substantiating the design in Section 4.2.4. To separate concerns, we present SENS-Join presuming a robust operation of the network. We address node failures and network-related problems that cannot be solved by the routing layer in Section 4.2.5.

Suppose that the query has already been distributed. Then, at a high level, SENS-Join comprises the following steps:

1. Precomputation:

a. Join-Attribute-Collection: Collect the join-attribute tuples of all relations at the base station and join them.

b. Filter-Dissemination: Distribute a join filter specifying which tuples join.

2. Final-Result-Computation:
The nodes in question send their complete tuples to the base station where the final result is computed.

Finding out which nodes contribute to the result requires the join-attribute tuple of each node. The base station collects them and joins them (Step 1a). The join-attribute tuples that have a partner form the "*join filter*", i.e., the filter is a set of join-attribute tuples. It lets a node decide if it contributes to the result by checking whether the filter contains its join-attribute tuple. Therefore, we need to disseminate the filter in the network (Step 1b). Finally, the base station collects those tuples that actually join and computes the result.

As our main contribution, SENS-Join features several mechanisms that reduce the communication costs of the precomputation significantly: Treecut (cf. Section 4.2.1), Selective Filter Forwarding (cf. Section 4.2.2) and a compact representation of the join-attribute tuples (cf. Section 4.3) used in Steps 1a and b.

We now provide a top-down description of SENS-Join. SENS-Join is a distributed process. Its implementation is event-driven, as explained below. Figure 4.1 presents SENS-Join from the point of view of a single node.

```
1  SENS-Join
2
3  //At the end of the query's dissemination:
4  sleepUntilNextStep(); //wait for beginning of SENS−Join
5
6  Join-Attribute-Collection:
7      ReceivedData = collectMessagesFromChildren();
8      T = constructTupleFromLocalSensorData();
9          //returns T = NULL if (T ∉ A) and (T ∉ B)
10     ForwardJoinAttrValues(ReceivedData, T); //cf. 4.2.1
11     sleepUntilNextStep();
12
13 Filter-Dissemination:
14     JoinFilter = receiveFromParent();
15     ForwardJoinFilter(JoinFilter); //cf. 4.2.2
16     sleepUntilNextStep();
17
18 Final-Result-Computation:
19     ReceivedData = collectMessagesFromChildren();
20     ForwardCompleteTuples(ReceivedData, T); //cf. 4.2.3
```

Figure 4.1: SENS-Join, at each node

The node wakes up three times altogether, at the beginning of each step. As described in Chapter 2, a node knows when its children will send their data for the Join-Attribute-Collection – this is due to the synchronization in tree-based data collection. The node sets the wakeup-time accordingly and goes to sleep (Lines 3, 4). When waking up the first time (Line 6), a node receives the join-attribute tuples of

its children (Line 7). It then uses its own sensor readings to generate the Tuple T (Line 8). The procedure returns NULL if the node does not belong to either of the Relations A and B, or if T does not meet the selection predicates in the `WHERE`-clauses. Finally, the node forwards the join-attribute tuples received along with its own join-attribute tuple to its parent (Line 10, cf. Section 4.2.1). This requires projecting its tuple T onto the join attributes. The projection is part of the forwarding procedure and is not shown in Figure 4.1. A node then sleeps until the beginning of the Filter-Dissemination step (Line 13). Now the join filter is disseminated in the network along the routing tree. A node simply receives (Line 14) and forwards the filter (Line 15, cf. Section 4.2.2). As a final step (Line 18), the complete tuples are forwarded along the routing tree (Line 20, cf. Section 4.2.3) to the base station. There the query result is computed.

4.2.1 Collecting Join-Attribute Tuples

Which data does a node send in this step? The tuple of a node might belong to either Relation A, B, or none of the relations. Thus, a node contributes a join-attribute tuple or nothing. For self-joins it is also possible that the node belongs to both relations. However, it still sends one join-attribute tuple only, consisting of the join-attribute values from both relations. The reason is that the join attributes usually overlap (e.g., they are identical in Q1 and Q2). So we avoid sending attribute values redundantly. In summary, each node contributes at most one tuple.

To assess the design of the Join-Attribute-Collection step, suppose that join-attribute tuples are collected in a straightforward way. Each node, starting at the leaves of the routing tree, collects these tuples from its children and forwards them to its parent along with its own tuple. A leaf node solely sends its join-attribute tuple T'. However, the difference to sending the complete tuple is only a few bytes. For instance, the difference of T - T' in Q2 is two attributes. Assuming that each attribute requires two bytes, this corresponds to sending only four bytes less. The important observation is that the energy savings due to sending T' instead of T are negligible[1]. Depending on the number of children, this remains true at the next level of the tree. The problem with these minor savings near the leaves is that at the same time *we risk the costs of sending an additional packet in the Final-Result-Computation phase if a tuple actually contributes to the result*.

Treecut. We avoid this inefficiency as follows: Starting at the leaves of the tree, we send complete tuples for the precomputation as long as the volume of data that has to be sent is less than a predefined threshold D_{max}. This applies near the leaves

[1]For instance, removing about 10 bytes from a packet incurs a saving on the order of 5% for SunSPOTs or MicaZ. The reason is the huge overhead due to networking-related issues like channel acquisition, synchronization ([DGM+04]).

of the tree where the forwarding load is small. We use $D_{max} = 30$ bytes (cf. discussion in Section 4.2.4). If the sum of the data that needs to be sent exceeds D_{max} at some node, this node stores the complete tuples of its subtree and switches to sending join-attribute tuples. In the subsequent steps, the node serves as a proxy for its children, i.e., it handles the Final-Result-Computation without requesting data from the children. Intuitively, Treecut reduces the depth of the tree for the following steps. This improves the efficiency of SENS-Join: We do not have to forward the join filter to those parts of the tree that were "cut off". In addition, if there are tuples that join, we have already forwarded them one or two levels up in the tree. Figure 4.2 presents the forwarding procedure.

```
1  ForwardJoinAttrValues(Set {S_1, ..., S_n}, Tuple T)
2  //S_i: data received from child i
3
4  Set_Of_Full_Tuples FullTuples = ∅;
5  Join_Attr_Structure JoinAttTuples = ∅;
6  for all S_i ∈ {S_1, ..., S_n}
7      if (S_i is Set_Of_Full_Tuples)
8          FullTuples = Union_Full_Tuples(FullTuples, S_i);
9      else
10         JoinAttTuples = Union_Join_Atts(JoinAttTuples, S_i);
11
12 if (Size({S_1, ..., S_n}) + Size(T) ≤ D_max)
13     && (∀S_i ∈ {S_1, ..., S_n}: S_i is Set_Of_Full_Tuples)
14     //use Treecut: hand over data to parent and go to sleep
15     FullTuples = Insert_Full_Tuples(FullTuples, T);
16     send(FullTuples, parent);
17     //query execution is complete:
18     exitQuery();
19 else
20     store FullTuples; //act as proxy for received complete tuples
21     store JoinAttTuples as "SubtreeJoinAtts";
22     ProxyJoinAttTuples = π_JoinAttr(FullTuples);
23     JoinAttTuples = Union_Join_Atts(JoinAttTuples, ProxyJoinAttTuples);
24     T' = π_JoinAttr(T);
25     JoinAttTuples = Insert_Join_Atts(JoinAttTuples, T');
26     send(JoinAttTuples, parent);
27     //sleep until next step - cf. Figure 4.1
```

Figure 4.2: ForwardJoinAttrValues

Due to Treecut, a node either sends complete tuples or join-attribute tuples. Thus, we have to distinguish between two different data structures for transmission. Complete tuples are forwarded as a multiset (`Set_Of_Full_Tuples`). In contrast, we insert the join-attribute tuples into a more elaborate data structure, discussed in Section 4.3 (`Join_Attr_Structure`).

`ForwardJoinAttrValues` starts by merging the data from the children into a single data structure for complete tuples (Line 8) and join-attribute tuples (Line 10), respectively. Then it is determined whether Treecut applies, depending on the amount of data to send (Lines 12, 13). If so, the node adds its tuple to the data received, sends it to its parent, and is done executing the query (Lines 15 - 18). Otherwise, the node stores the complete tuples from its children to act on behalf of them in the Final-Result-Computation step (Line 20). In addition, it stores the join-attribute tuples of its subtree (Line 21). This is used in the Filter-Dissemination step, and Section 4.2.2 will deal with it. Then the node generates the join-attribute tuples of the complete tuples received as well as its own join-attribute tuple T', adds it to the data received, sends it to its parent, and waits for the Filter-Dissemination (Lines 22 - 27).

Memory Capacities. Treecut introduces proxies that store the data of their descendants. The amount of memory needed is limited by D_{max} (30 bytes) multiplied by the number of children of a node. The children are a subset of its communication neighbors. Thus, the number of these neighbors can serve as an upper bound, usually around 6 to 15 [MFHH03, CNS07]. In summary, Treecut requires only a small fraction of the capacities of the node (hundreds of KBs, cf. Section 2.1).

4.2.2 Disseminating the Join Filter

After the base station has received the join-attribute tuples of both relations, it joins them and creates the join filter. SENS-Join now has to disseminate the filter in the network. A simple way to do so would be to forward it along the routing tree. However, based on the following observation, we can significantly reduce the number of packets: *In the Join-Attribute-Collection step, each node gets to know the join-attribute tuples of its descendants.* If a node keeps this knowledge until the Filter Dissemination, it can decide

1. which part of the join filter is relevant for its subtree (which join-attribute tuples appear in it), and
2. whether it is necessary to forward the join filter at all.

Selective Filter Forwarding. The first item means reducing the size of the join filter while being forwarded to the leaves, i.e., the filter is pruned progressively. The second item refers to the situation where none of the tuples from the filter

appears in the subtree of a node. In this case there is no need to forward the filter, i.e., the filter is forwarded exclusively into those regions that contain result tuples. Figure 4.3 shows the forwarding procedure. Since the filter also is a set of join-attribute tuples, the same data structure is used as for the prior collection step (`Join_Attr_Structure`).

```
1 ForwardJoinFilter(Join_Attr_Structure Filter)
2
3 SubtreeFilter = Intersect_Join_Atts(Filter, SubtreeJoinAtts);
4 if (SubtreeFilter ≠ ∅)
5     //send join-attribute tuples of subtree to children
6     broadcast(SubtreeFilter);
7 else
8     //do nothing - the subtree won't be involved in final step
9 //sleep until next step - cf. Figure 4.1
```

Figure 4.3: ForwardJoinFilter

Recall that nodes have stored the necessary knowledge (SubtreeJoinAtts) during the Join-Attribute-Collection step. `ForwardJoinFilter` simply intersects the set of join-attribute tuples that appear in the subtree with the join filter (Line 3). This yields the set of join-attribute tuples that contribute to the result and are located in the subtree. The node forwards the result of the intersection if it is not empty. Note that especially if the sensor readings are spatially correlated, complete regions of the tree might not have to forward the join filter.

Memory Capacities. Selective Filter Forwarding trades memory for transmission costs. The memory requirements for Selective Filter Forwarding are determined by the specifics of the data structure (which we have not yet discussed). Even without knowing the details, observe that it is possible to bound the memory used: A node keeps the join-attribute tuples of its subtree if their size is less than a predefined limit. We use a limit of 500 bytes. To illustrate, this is only a small fraction of the 512 KB of a SunSPOT. Introducing a limit only has a minor influence on the performance of Selective Filter Forwarding since the amount of data exceeds a few hundred bytes close to the root only. However, the mechanism has its main benefit towards the leaves.

4.2.3 Final Result Computation

After the filter has been disseminated, this step collects the complete tuples. Conceptually, this is the simplest step: All it does is forwarding the tuples along the routing tree to the base station. Figure 4.4 shows pseudocode for this final step.

```
1  ForwardCompleteTuples(Set {S_1, ..., S_n}, Tuple T)
2  //S_i: multiset of full tuples received from child i
3
4  Set_Of_Full_Tuples FullTuples = ∅;
5  for all S_i ∈ {S_1, ..., S_n}
6      FullTuples = Union_Full_Tuples(FullTuples, S_i);
7  if (T' ∈ Filter) // T' = π_JoinAttr(T);
8      FullTuples = Insert_Full_Tuples(FullTuples, T);
9  send(FullTuples, parent);
10 //query execution is complete:
11 exitQuery();
```

Figure 4.4: ForwardCompleteTuples

In particular, `ForwardCompleteTuples` is similar to the Join-Attribute-Collection except that it concerns only some of the nodes. Thus, depending on how many tuples actually join, the data volume can be very small. The tuples are simply forwarded along the routing tree to the base station. Finally, the base station computes the result.

Note that the complete tuple needs to be stored in the first step (cf. Figure 4.1, Line 8). It is not possible to re-acquire it from the sensors as the sensor readings could have changed since the Join-Attribute-Collection. As any other join algorithm, SENS-Join reads the sensors exactly once.

4.2.4 Design Considerations

Parameter D_{max}. We have argued that, if the absolute amount of data (D_{max}) is already small, it can hardly be reduced. Thus the energy savings would be small. This argument holds if the number of packets is not affected, leading to an important constraint: $D_{max} <$ *MAX_PACKET_SIZE*. Beyond that, our choice of $D_{max} = 30$ bytes is justified by our experiments: For instance, if the set of tuples exceeds 50 bytes, SENS-Join already achieves a data reduction of about 25 to 30 bytes. This is due to switching to join-attribute tuples as well as to our compact data structure (`Join_Attr_Structure`).

Join Locations. An important design decision is where the join-attribute tuples are joined and where to compute the final result. We perform both computations at the base station. For the final result, the choice is motivated by IDEAL. In fact, the base station is the optimal join location (cf. Section 3.1.3). For the precomputation, in-network approaches are superior to using the base station in specific scenarios only. More precisely, these are exactly those cases that are covered by the specialized

join methods presented in Section 3.2. Thus, performing the semi-join at the base station is superior to other choices in those cases that SENS-Join actually targets.

Discussion. [MFHH03] proposes using an index ("semantic routing tree") on static attributes to reduce the costs of forwarding queries inside the network. Selective Filter Forwarding is different. Our mechanism prunes the join filter which is forwarded *progressively*. In addition, we do not build a dedicated index tree. We exploit the knowledge available at a node anyhow. Thus, we employ a temporary structure which comes at no additional costs. Our mechanism is also applicable for attributes whose value changes frequently, in contrast to [MFHH03].

4.2.5 Design for Error Tolerance

One of the most critical issues for algorithms in sensor networks is coping with changes in the network topology, due to links going down and node failures. SENS-Join builds upon tree-based routing. In particular, the collection tree is capable of adapting to changes in the topology. However, an execution of SENS-Join requires the routing tree to be stable for the duration of a single execution. This is on the order of a few seconds. Beyond a single execution, SENS-Join does not maintain any state. If a link goes down during the execution of a query, we rely upon the tree protocol to re-establish the routing structure. Afterwards, we simply re-execute the query. If data loss needs to be avoided, a more elaborate technique for handling link failures would be required that stores the data during the outage. However, while network disconnection is a problem, it is infrequent given the short execution time of queries.

4.3 Compactly Representing Join-Attribute Tuples

We now describe the data structure used to represent join-attribute tuples in the first two steps. Our mechanism more than halves the costs of the precomputation.

Mechanisms like Bloom Filters [Blo70] cannot serve as a compact representation in our context since they only allow for evaluating equi-joins. Compression algorithms would also be unsuitable: Firstly, they are not targeted towards small data volumes. This results in bad compression ratios for our problem, see Section 4.4.2. Next, a compression algorithm would introduce a huge overhead [SM06]: A node must decompress the data received before adding its own tuple. The data then needs to be re-compressed before forwarding it. Our compact representation avoids this problem by computing the primitives $\texttt{Insert}_{\texttt{Join\_Atts}}$, $\texttt{Union}_{\texttt{Join\_Atts}}$, and $\texttt{Intersect}_{\texttt{Join\_Atts}}$ directly on it.

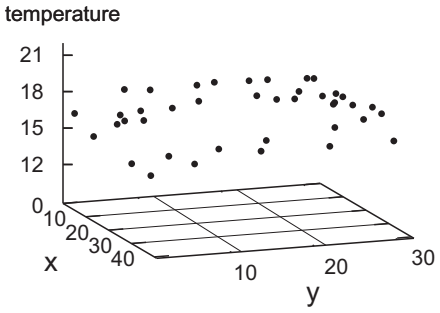


Figure 4.5: Distribution of values for 3 join attributes

4.3.1 Key Ideas of the Compact Representation

Our mechanism pursues two goals:

1. We minimize the number of bits required to represent a **single join-attribute tuple**.

2. We compactly encode a **set of tuples**.

The key idea towards representing single join-attribute tuples is to perform a quantization of the range of each sensor type. This lets us influence the number of bits. Clearly, a quantization reduces the accuracy of the join-attribute tuples. This is not a problem: As the join-attribute tuples are only used for the precomputation, we can reduce their accuracy without sacrificing correctness of the final result.

To compactly encode a set of join-attribute tuples we exploit spatial (auto-) correlation of sensor readings. We briefly provide the intuition: Figure 4.5 shows temperature measurements and their locations, taken from a real-world deployment [Int]. In the presence of spatial correlation, sensor readings from nearby nodes are likely to be similar. As a consequence, a set of join-attribute tuples is highly redundant. Our representation eliminates this redundancy by means of a spatial index.

4.3.2 Quantization

Conceptually, join-attribute tuples are points in an unbounded, continuous, n-dimensional space. Figure 4.6 illustrates this perception: Given a query with two join attributes, humidity and temperature, join-attribute tuples are points in a two-dimensional space. The idea of a quantization is to approximate a continuous range of values by a relatively small set of discrete values. Quantization requires us to specify bounds on the ranges ([min, max]) and a resolution (step size) for each dimension. The outcome is a *restricted*, *discrete*, n-dimensional space. To complete the quantization we need to assign a symbol to each multidimensional cell. This symbol encodes

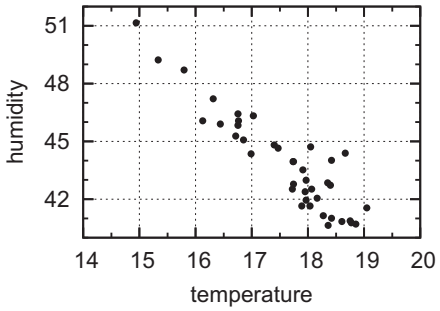


Figure 4.6: Distribution of values for 2 join attributes

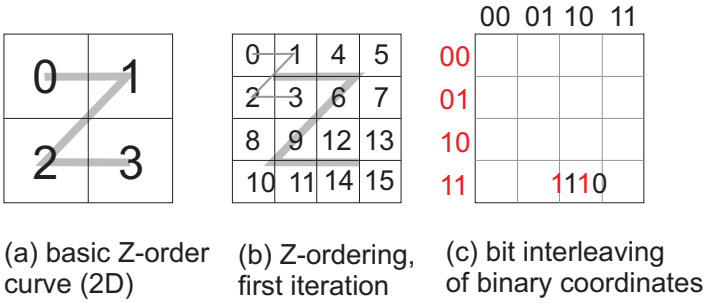


Figure 4.7: Z-ordering

a join-attribute tuple that falls into the cell. In other words, we need a numbering which maps a multidimensional point to one dimension, i.e., a space-filling curve.

For the numbering it is important that numbers corresponding to nearby join-attribute tuples are similar in order to keep the spatial correlations. Z-ordering accomplishes this, as illustrated in Figure 4.7a and b. Besides its good locality-preserving behavior, Z-ordering is easy to compute. This is important in sensor networks. We compute the Z-number of a point by bit interleaving of the coordinates of each dimension, see Figure 4.7c.

We now turn to two important details: Firstly, we need to determine an appropriate range of values ([min, max]) and a resolution of each dimension. This is done at the base station and is disseminated independent of a query. The second aspect refers to computing Z-numbers. The problem is that a sensor measurement might fall outside of the specified range. To ease presentation, we discuss the second aspect first.

Computing Z-Numbers. Each node has to encode its join-attribute tuple which is computing the Z-number. This is presented in Figure 4.8.

As a prerequisite for the bit interleaving, we need to know the length of the coordinates. This is implied by the number of cells in each dimension (Lines 2 - 5). For instance, in Figure 4.7c, we need two bits for each dimension. We compute the number of bits for each dimension separately as, in general, the dimensions are not

```
1  //compute size of each dimension
2  for all dimensions i
3      SizeOfDim[i] = {(MaxVal[i] − MinVal[i]) · 1/Resolution[i]} + 1;
4      SizeOfDim[i] = roundUpToPowOf2(SizeOfDim[i]);
5      BitPerDim[i] = log(SizeOfDim[i])
6
7  EncodeTuple(Tuple T')
8
9      //compute coordinates (P[i]) of T' in each dimension
10     for all dimensions i
11         P[i] = {(T'[i] − MinVal[i]) · 1/Resolution[i]};
12         if (P[i] < 0)
13             P[i] = 0;
14         if (P[i] ≥ SizeOfDim[i])
15             P[i] = SizeOfDim[i] − 1;
16     //apply bit interleaving to encode P (T')
17     Z = InterleaveBits(P, BitsPerDim);
18     return Z;
```

Figure 4.8: EncodeTuple

of equal size. In this case, each dimension contributes to the bit interleaving until its bits are exhausted. `EncodeTuple` starts by computing the coordinates of T' (Lines 10 - 15). By doing so, we ensure that they are within the specified range ([MinVal, MaxVal]) (Lines 12 - 15). This is necessary since the estimated range might be too narrow. In this case we map the value to the boundary of the corresponding dimension. We discuss this solution subsequently. Finally, `InterleaveBits` computes the encoding (Line 17).

Specifying Ranges and Resolution. The following parameters need to be specified: MinVal$[i]$ and MaxVal$[i]$ to bound each dimension as well as their resolution (Resolution$[i]$). These ranges are specific to the environment of the sensor network. It is therefore possible to fix them while setting up the network. While elaborate techniques exist for estimating the values, e.g., [DGM+04], for our purpose, reasonably good estimates are sufficient. If our estimated range is too large, we might need more bits to encode a point. But since our domain grows in steps of powers of two (Line 4), a moderate overestimation is not critical. For instance, there is no difference whether we specify a range containing 600 values or 900 values: They both are in the interval $[512, 1024]$ and require 10 bits. In contrast, if the range is too narrow, a value might be outside of it. `EncodeTuple` maps such a point to the boundary of the range. In the worst case this wrongly yields join-attribute tuples that

match, and we unnecessarily send their complete tuples. But this affects only a few tuples unless the range is much too narrow. It does not affect the correctness of the result.

The idea behind a quantization is to have a coarser resolution to reduce the number of different values per dimension. Again, the resolution has no impact on the correctness of the result. If it is too fine, we need more bits to encode each point. If it is too coarse, the following problem arises which also leads to an increase in the communication costs: Due to the quantization, each point represents a set of join-attribute tuples. In particular, for the precomputation, a point joins, if **any** of the join-attribute tuples joins, that is represented by the point. This is required in order not to miss a joining tuple in the final result. The problem is that the set of tuples that is represented by a point increases with a coarser granularity of the resolution. Thus, we might erroneously assume that a join-attribute tuple joins, since the corresponding point joins. As a result, SENS-Join unnecessarily sends the corresponding complete tuple in the Final-Result-Computation step. We found out experimentally that the performance of SENS-Join is insensitive to the resolution used for the precomputation as long as it is not too coarse. Thus, we simply use a fixed resolution for a particular environment. In our experiments we used steps of, e.g., 0.1°C for the temperature, or of 1m for the X- and Y-coordinates.

4.3.3 Representing Points Using a Spatial Index

To encode a *set of points* our idea is to use a spatial index. A region quadtree [Sam84] is a good choice: Firstly, our goal is to achieve a compact encoding. A region quadtree is based on a regular decomposition of an n-dimensional universe. The 2^n subspaces resulting from a partition are of equal size. This is advantageous for our compact representation since it is not necessary to encode how to divide the space. Secondly, a quadtree is closely related to Z-ordering. The Z-number of a point corresponds to the sequence of quadrants that results from a traversal of a quadtree down to the point. The following example illustrates this idea:

Example 4.4: Consider the point in Figure 4.7c. Its Z-number is 1110. Further, consider a quadtree index on such a two-dimensional domain – such an index is illustrated in Figure 4.9. For the example in Figure 4.7c, the index has two levels until the quadrant of the point is identified at full resolution (there are only 16 different points in our example). To find the point using the index, the beginning of the Z-number (11) indicates that the point is in Quadrant 3 at the upper-level index node. At the next level, the point in in Quadrant 2 (10), which is where the point actually is according to Figure 4.7c. ■

To facilitate establishing an understanding of our quadtree encoding, we removed the pseudocode from the following description. We refer the interested reader to

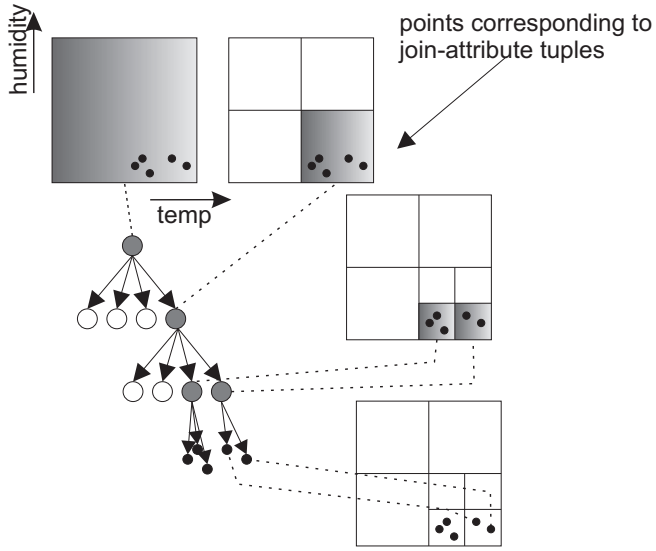


Figure 4.9: Construction of the quadtree

Appendix B.

Encoding Sets of Points With a Quadtree. To illustrate the intuition, assume a query with a single join attribute and the two (discretized) values 23.2°C and 23.4°C. To eliminate redundancy, we could represent them as 23°C plus the relative remainders 2 and 4, respectively. As an abstract illustration, Figure 4.9 shows a quadtree for five join-attribute tuples from a two-dimensional example. Each tuple corresponds to a point in a two-dimensional, quantized space. Each index node of the quadtree corresponds to a region. At each level of the tree, all dimensions are partitioned into halves. Redundancy within the set of points is eliminated as follows: The index indicates the region which is common to all of the points. If we encode each point relative to the region, then the index node represents what the points have in common. A relative remainder contains information that is unique to a point. Thus, we represent a set of points by the index nodes plus the remainders.

Note that the Z-number already includes this separation into index information and remainder, as illustrated in Example 4.4: We use the Z-number to traverse the tree by reading the Z-number from the beginning – at each level, the part of the Z-number that we have not yet used is the location of the point within the current quadrant, i.e., its relative position (remainder).

Since quadtrees are well-known, we restrict the discussion of the details to two aspects: Firstly, we need a pointerless representation of the tree. This is because we use it as a wire format. In addition, the use of pointers negatively affects the space requirements of the data structure. The second aspect refers to a general design decision with respect to quadtrees: The decomposition into subspaces is usually

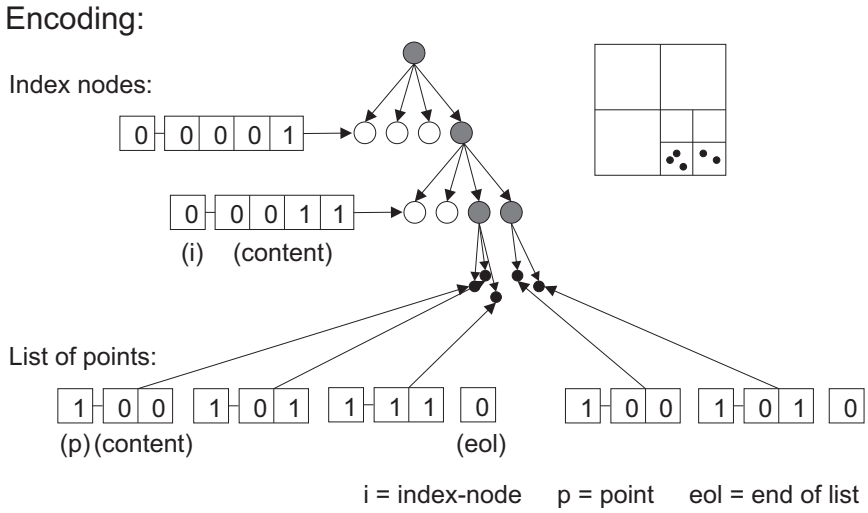


Figure 4.10: Encoding of a quadtree

continued until the number of points is below a given threshold t [GG98]. Thus, we need a criterion to decide when to stop the decomposition.

Pointerless Representation. There has been a lot of interest in pointerless quadtree representations [Sam84]. We represent the tree as a bitstring consisting of index nodes and the points which are given relative to the path. Figure 4.10 shows these elements. The pointerless representation is obtained by storing them in the order of a depth-first traversal of the quadtree. This allows us to easily implement $\texttt{Union}_{\texttt{Join\_Atts}}$ and $\texttt{Intersect}_{\texttt{Join\_Atts}}$. We elaborate on this in Section 4.3.4.

The details of our encoding are as follows: An index node starts with a '0' bit specifying that it is an index node. The remaining bits of an index node encode which of the quadrants at the subsequent level is present in the tree. If the number of points in a quadrant is below the threshold, the points are given as a list. A leading '1' indicates a point. Their encoding is relative to the path and contains only the position within the current quadrant (remainder of the Z-number). Thus, as the size of the quadrant becomes smaller with every level, so does the relative encoding of a point. As shown in Figure 4.10, we also need to mark the end of a list of points. This is done by appending a '0'.

Decomposition Threshold. In general, the threshold t for stopping the recursive decomposition of a quadtree depends on its use. In Figure 4.9, t equals 3, so a set of three points is listed explicitly. As our goal is a compact representation, t depends on the number of bits required for a further subdivision vs. the number of bits required for explicitly listing the points. Recursively subdividing a quadrant costs inserting an index node into the tree (the encoding is shown in Figure 4.10). In contrast,

since the points are specified relative to the current path, this subdivision reduces the number of bits of each point. In general, the idea is to compare both solutions and to stop the decomposition if a list of points is shorter.

Encoding of Relation Membership. To ease presentation, we have omitted so far how to encode which relation a join-attribute tuple belongs to. This is important for the base station to do the join. We prefix each point with two bits ("relation flags") indicating that the point belongs to Relation A ('10'), B ('01'), or to both relations ('11'). As a consequence of prefixing the points, when inserting them into the quadtree, the topmost index node represents the relation flags.

As a remark, the relation flags constitute the only mechanism in SENS-Join that is specific to computing joins over two input relations. In particular, SENS-Join can seamlessly process joins over an arbitrary number of input relations. The extension to joining more than two relations is straightforward: The number of relation flags has to correspond to the number of relations in the query, i.e., SENS-Join has to use more bits for the relation flags.

4.3.4 Computing Low-Level Primitives

Recall that compression algorithms are not helpful in our context, due to repeated compression/decompression [SM06]. A strength of our quadtree representation is that $\texttt{Union}_{\texttt{Join\_Atts}}$ and $\texttt{Intersect}_{\texttt{Join\_Atts}}$ (cf. Figure 4.2) can be computed directly on it. There is no need to recover the original tuples.

Section 4.2 described the semantics of the primitives in detail: Both operate on sets (sets of join-attribute tuples encoded as quadtrees). Most notably, semantically, $\texttt{Union}_{\texttt{Join\_Atts}}$ and $\texttt{Intersect}_{\texttt{Join\_Atts}}$ are simple set operations that obey the usual semantics of sets. The nodes use $\texttt{Union}_{\texttt{Join\_Atts}}$ in the Join-Attribute-Collection step to merge the sets of join-attribute tuples received from their children into a single set (a single quadtree). $\texttt{Intersect}_{\texttt{Join\_Atts}}$ is used in the Filter-Dissemination step: After the base station computed the semi-join, the join filter (set of join-attribute tuples that join) is disseminated in the network. To reduce the communication costs, the nodes intersect the join filter with the set of join-attribute tuples that appear in their subtree. Then, the nodes forward only that subset of the join filter that is required by their descendants (Selective Filter Forwarding).

We use $\texttt{Union}_{\texttt{Join\_Atts}}$ to illustrate the idea of realizing the primitives directly on quadtrees without recovering the join-attribute tuples. Again, the corresponding pseudocode is in Appendix B. $\texttt{Union}_{\texttt{Join\_Atts}}$ is similar to the merge step in Mergesort and can be done in one pass over the data: It merely requires a traversal of the two trees in parallel. Performing these operations directly on the quadtrees is simple due to the depth-first order. Since quadtrees follow a regular decomposition scheme, the shape of the tree is independent of the order of the points being added. Note that

the ability to perform set operations quickly is one of the reasons for the popularity of quadtrees [Sam84].

4.4 Experimental Study

To demonstrate the efficiency of SENS-Join we have implemented a prototype in the ns-2 network simulator [BEF+00]. It is a widely used simulator that allows for a controlled environment of our experiments and ensures repeatability. We compare its performance to the *external join*. It is state-of-the-art for our problem. In particular, we justified this choice in Section 3.2. There, we pointed out that the specialized join methods require very specific scenarios and discussed why the external join outperforms them in each of the following experiments.

Metric. Our intention is to capture the communication costs. As explained in Section 2.1, sensing and communication dominate the power consumption [AML05, YG03, MFHH03]. However, sensing is the same for every join method.

We use the number of transmissions (networking packets) as a proxy of the communication costs. This is the most common metric in the sensor network literature, e.g., [CDHH06, AML05, YLOT07]. While the number of bytes transferred is used as well, e.g., [MFHH02], we prefer the packets metric for these experiments as it accounts for the per packet overhead. For this metric, we set the maximum packet size to 127 bytes as used by SunSPOTs. We also did experiments with smaller packet sizes which were used in early systems such as TinyDB. This has yielded even better results for SENS-Join. However, we subsequently present the results for 127 bytes as this is the size that current standards such as ZigBee use.

We compare the communication costs along two lines: *overall communication costs* and *per node communication costs*. The latter is important due to the routing tree. Nodes close to the root are more loaded than leaf nodes. Thus, the per node metric is more critical: When the energy of the nodes near the root is depleted, the network ceases operation.

General Setting. For our experiments, we simulate a random distribution of the nodes. We set the communication range of each node to 50m and assume links to be bi-directional. This is a common setting in the networking community [YG03].

For the scope of this presentation we use a fixed distribution of the physical quantities, emulating real sensor data. Varying the data distribution has two effects: (a) The size of the result may change, and (b) the positions of the nodes contributing to the result may change as well. However, we found that changing the positions of nodes only has a minor influence. Further, to vary the fraction of tuples that join, we can also adapt the join conditions. This is what we do subsequently.

Parameters. There is a large number of parameters that influence the efficiency of SENS-Join. As discussed below, they can be reduced to the following parameters:

1. Fraction of nodes in the result
2. Ratio of $\frac{\text{join attributes}}{\text{attributes overall}}$
3. Number of nodes (size of the network)

The idea behind our approach is to send only the tuples that contribute to the result. Thus, the savings depend on the size of this fraction. A second parameter is the ratio of join attributes over the number of attributes in the query. While the external join sends complete tuples, we only send the join-attribute values during the Join-Attribute-Collection step. Thus, the smaller this ratio, the higher the expected savings. Finally, we are interested in the influence of the network size. Clearly, it influences the total number of transmissions. We are interested in its influence on the relative performance of the join methods.

The first two parameters in particular determine the form of the queries used in the experiments:

```
SELECT A.att_1,..., A.att_n, B.att_1,..., B.att_n
FROM Sensors A, Sensors B
WHERE join-expr(A.join-atts, B.join-atts)
AND ... AND join-expr(A.join-atts, B.join-atts)
ONCE
```

The join conditions are range conditions in the style of Q1 and Q2, used to vary the fraction of tuples in the result. The queries do not contain selection predicates. These would be handled locally and affect the number of nodes concerned. However, our third parameter already controls this number. Beyond that, we found the influence to be negligible. Finally, we query the same number of attributes from both relations. Otherwise, the tuples sent in the Final-Result-Computation step would be of different size, and the number of transmissions would depend on the fraction of the tuples that are large/small. We avoid this parameter since it does not provide any further insight: Its variation has the same effect as varying the number of attributes overall.

Default Setting. In each experiment we vary one of the parameters. If a parameter is not varied we use the following default value: The size of the network is 1500 nodes in a 1050m $\cdot$ 1050m area. The fraction of the nodes in the result is 5%. For the ratio of join attributes to attributes overall we will consider two default settings settled towards different ends of the spectrum. The first one is 33% based on one join attribute. The second one is 60% based on three join attributes.

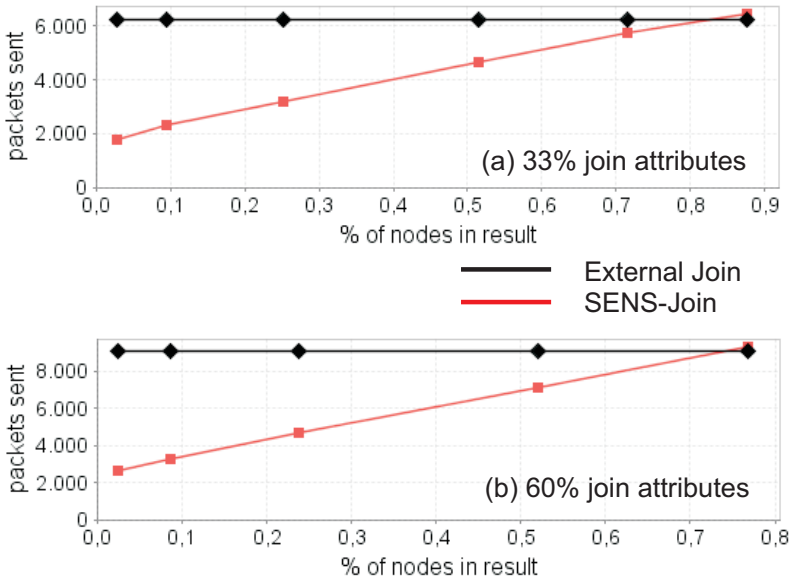


Figure 4.11: Overall savings of SENS-Join

4.4.1 Efficiency of SENS-Join

Overall Communication Costs. A first set of experiments determines the overall savings of SENS-Join as compared to the external join. The parameter is the fraction of nodes that contribute to the result. The higher this fraction, the more tuples finally need to be transmitted. Thus, we expect SENS-Join to perform better than the external join unless a high fraction of the tuples joins. There is a break-even point due to the precomputation of SENS-Join. Figure 4.11 graphs the results for 33% and 60% join attributes. For the latter, we save up to two-thirds of the overall energy consumption of the external join. For 33% join attributes, the energy savings are up to 80%. They are higher for a smaller ratio of join attributes to attributes overall, as discussed below. Next, SENS-Join is superior until more than 70% (80%) of the nodes join.

Per Node Communication Costs. In the following we investigate the relationship between the number of descendants in the routing tree and the load of the nodes. Due to the forwarding load, nodes with many descendants take up more resources and thus determine the lifetime of the network. It is critical to reduce the load on these nodes. Figure 4.12(b) shows that for 60% join attributes the most loaded nodes are unburdened by more than 80%. For 33% join attributes, Figure 4.12(a) shows a reduction of more than an order of magnitude. This difference increases as the ratio of join attributes to attributes overall decreases. The next paragraph explores this influence in detail.

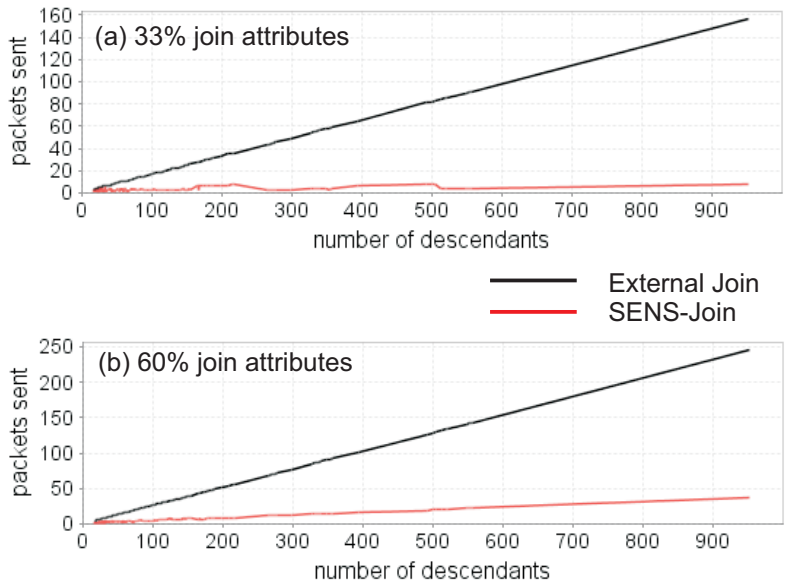


Figure 4.12: Per node savings of SENS-Join

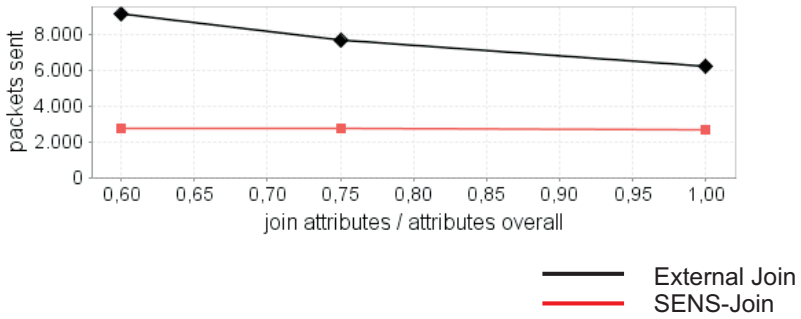


Figure 4.13: Influence of the ratio of $\frac{3 \text{ join attributes}}{x \text{ attributes overall}}$

Ratio $\frac{\text{join attributes}}{\text{attributes overall}}$. We want to verify that a smaller ratio of join attributes to attributes overall increases the savings, and we want to find out if and how these savings are bounded. This also is supposed to justify that our 33% and 60% default settings represent different ends of the spectrum.

The difficulty with the ratio is that the number of combinations is daunting. But many combinations lead to similar ratios (2:4, 4:7, 4:8, etc.) and are close to one of the defaults. Thus, they represent a large number of queries. In contrast, it is possible to increase the ratio to 100%, at least in theory. Though it is difficult to find meaningful queries with very high ratios, the analysis provides a lower bound on the savings.

We now fix the join attributes (join conditions) to have a constant rate of nodes that join (5%). In a first experiment we consider queries with three join attributes. We

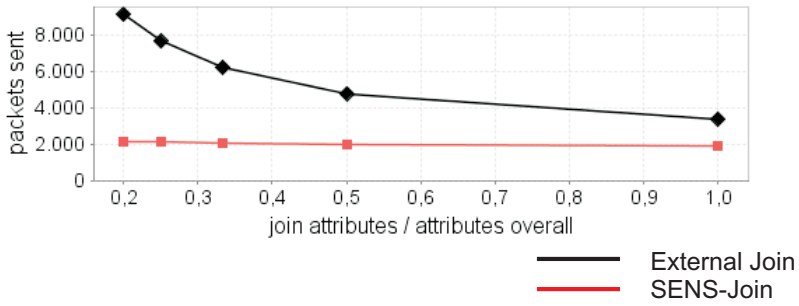


Figure 4.14: Influence of the ratio of $\frac{1 \text{ join attribute}}{x \text{ attributes overall}}$

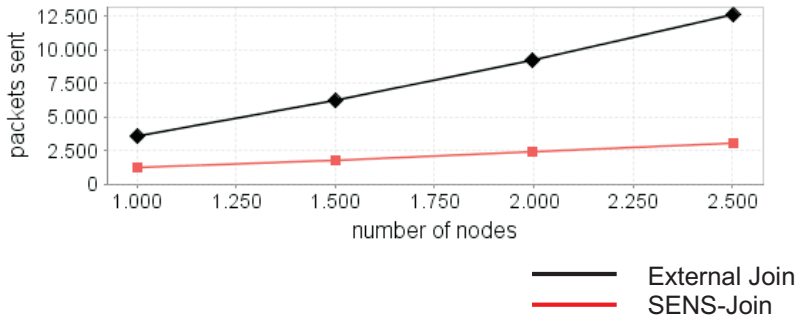


Figure 4.15: Influence of the network size

vary the number of attributes overall from five to three. Clearly, this is the minimum for three join attributes. Figure 4.13 graphs the results. As expected, the savings increase as the ratio of join attributes to attributes overall decreases. We also see that even for the worst case of 100% join attributes we save transmissions, compared to the external join. This is because of the quadtree representation. In the second experiment we take one join attribute and vary the number of attributes overall from one to five. The results in Figure 4.14 confirm our expectations.

Network Size. To determine the influence of the network size we vary the number of nodes from 1000 to 2500. At the same time we vary the area of the network to keep the node density constant. We expect the size of the network to have only a small influence on the relative results. If we assume a fixed fraction of tuples that join, the savings are proportional to the size of the network in each step. There is one exception: At the beginning of the Join-Attribute-Collection step when Treecut is applied our method is identical to the external join, and there are no savings. Intuitively, the influence of this initial phase becomes less as the size of the network increases. Figure 4.15 shows that this expectation holds. The savings are slightly superlinear with the size of the network.

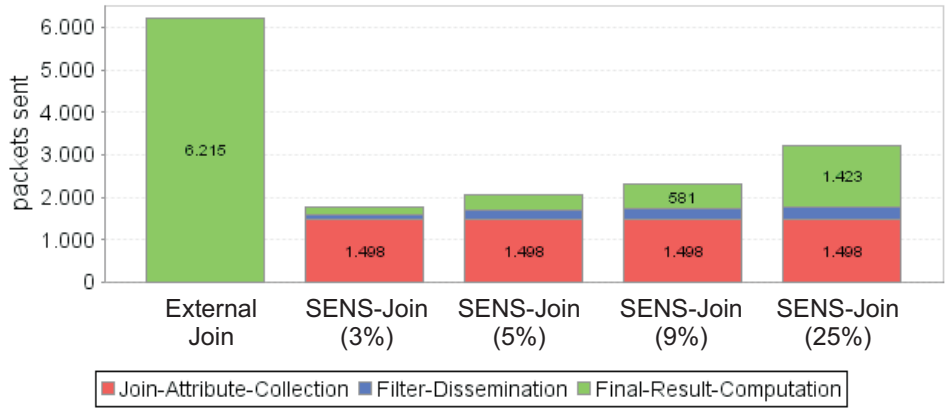


Figure 4.16: Costs in the different steps of SENS-Join

4.4.2 Costs of SENS-Join – Breakdown

We are now interested in an explanation of the savings with SENS-Join. We start by breaking down the number of transmissions to the different steps. In addition, we consider the performance of our compact representation in isolation, and we analyze its influence on the SENS-Join performance.

Costs of the Different Steps. We now assign the costs of SENS-Join to the different steps. As before, we consider the overall costs if different shares of the tuples join. The costs of the first step (Join-Attribute-Collection) solely depend on the number of join attributes and not on the fraction of tuples in the result. Figure 4.16 confirms that they are fix. If there were no result tuples (0% tuples that join), there would be no packets in the Filter-Dissemination and Final-Result-Computation steps. Thus, the costs of the first step provide a lower bound on the costs of SENS-Join for a fixed number of join attributes. Let us now look at the costs of the Filter-Dissemination: The number of nodes which need to receive the join filter depends on the fraction of tuples in the result. Thus, the smaller this fraction, the more subtrees are pruned. Finally, the costs of the Final-Result-Computation step are proportional to the fraction of nodes in the result.

Performance of Quadtree Representation. One of the reasons that compression algorithms are unsuitable for our problem is a bad compression ratio for small data volumes. We now compare our compact representation to some well-known compression algorithms of different kinds (cf. [Say00]): zlib [GA] (library form of gzip), which combines LZ77 and Huffman coding, and bzip2, [Sew], which is based on the Burrows Wheeler Transform. These algorithms do not run on current sensor nodes due to their use of memory and code size. However, by using highly optimized

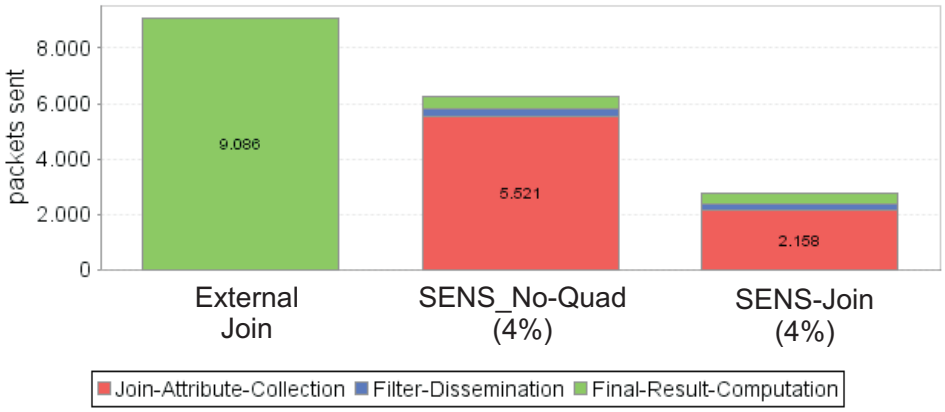


Figure 4.17: Influence of quadtree representation

algorithms we provide an upper bound on what can be achieved. We focus on lossless algorithms. Lossy compression leads to incorrect join results.

For this experiment we modified the Join-Attribute-Collection step to either send the raw join-attribute tuples (no compact representation), to use one of the compression algorithms, or to use our quadtree representation. In particular, for the compression algorithms, we decompress the join-attribute tuples received at each node. The node than adds its own tuple and re-compresses the data before forwarding. While this is computationally demanding, it gives the best performance with the respect to the compression ratio. It is thus appropriate for our upper bound on the performance of compression algorithms.

We collect three join attributes: temperature and the location coordinates. This is difficult for our approach: Two of the join attributes are uncorrelated (X- and Y-coordinates). As expected, using a standard compression is inferior our quadtree representation: For 1500 nodes, collecting the join attributes using no compression requires 6215 packets. Bzip2 requires 3230 packets and zlib requires 3097 packets. In contrast, the quadtree representation saves two-thirds of the costs (2158 packets). The difference stems from bad compression ratios of the standard algorithms near the leaves of the tree where the data volume is small. Finally, note that compression algorithms proposed for sensor networks are much less complex than bzip2 or zlib and cannot achieve their performance.

Influence of Quadtree Representation. Finally, we distinguish between the savings due to only sending join attributes and the ones due to the quadtree representation. For this discussion we use the queries from the introduction. Without the quadtree mechanism we save sending two out of five attributes for Q2. Thus, the volume of data is reduced by 40% (66% for Q1). However, the actual savings of the Join-Attribute-Collection step will be slightly smaller. This is because a reduction

in the volume of data does not reduce the number of packets if this already is the minimum of a single packet. The numbers in Figure 4.17 confirm this simple back-of-the-envelope calculation. The Join-Attribute-Collection step needs about 39% less transmissions than the external join. As discussed above, the compact representation more than halves the volume of data sent. Again, due to the lower bound of a single transmission, some nodes cannot profit from this reduction (Figure 4.17).

4.5 Tradeoffs

SENS-Join can significantly reduce the energy consumption of join processing. However, these benefits are not for free.

Response Time. SENS-Join introduces a precomputation and is thus inferior to the external join regarding response time. In a way, SENS-Join trades response time for energy consumption, as the latter arguably is the most critical resource in sensor networks. However, the response time of SENS-Join is upper bounded by at most twice the duration of the external join.

Vulnerability to Disconnection. If a link happens to go down *during the execution of a query*, our current error handling does not deliver results for the time of the outage. This problem has been discussed in Section 4.2.5 – it is an infrequent problem and could be mitigated by a more elaborate error handling.

Memory Requirements. Treecut and Selective Filter Forwarding trade memory for transmission costs. As discussed in Section 4.2, our design accounts for memory restrictions.

4.6 Summary

In this chapter, we presented SENS-Join, a general-purpose join method that can efficiently handle any number of join conditions and arbitrary distributions of the nodes involved. To obtain efficiency, SENS-Join builds upon the idea of IDEAL: It discards non-joining tuples at the sources. To provide the knowledge which of the tuples join, SENS-Join introduces a precomputation. The main contribution of SENS-Join is the design of the precomputation. By thoroughly designing the information flow and by means of a compact representation specific to the precomputation data, this precomputation becomes efficient. SENS-Join is more efficient than the state-of-the-art approach unless a high fraction of the input relations (ca. 60% - 80%) joins. We achieve a reduction of the overall energy consumption by more than 80%. The savings of the most loaded nodes is more than an order of magnitude in some situations.

SENS-Join can be used to answer one-time queries as well as continuous join queries. However, for continuous queries, using a precomputation to find out which of the tuples join is not optimal: This is because the precomputation is repeated for each execution, leaving aside temporal correlations in the data. In the following chapter, we present an approach that improves upon SENS-Join for the processing of continuous join queries. We propose to keep the knowledge, which of the tuples join, and to maintain it, instead of re-computing it for each execution.

5 Processing Continuous Join Queries: a Filtering Approach

In this chapter, our concern is processing continuous join queries. Continuous queries are important - they prevail in monitoring and surveillance. In particular, we present CJF ("Continuous Join Filtering"), a filtering-based approach. The filters are used to discard non-joining tuples. CJF exploits temporal correlation in sensor readings as well as the continuous nature of queries: The idea is to keep the filters for subsequent executions and to maintain them. The problems are determining the sizes of the filters and deciding which filters to update. Simplistic approaches result in bad performance. We show how to compute solutions that are optimal.

The following section features an overview of the problems and the contributions of CJF. We then outline the structure of this chapter. To illustrate our ideas, we will use the simple join query from Example 1.2, which acquires data from nodes that observe similar temperature and humidity conditions:

Example 5.1:

```
SELECT A.*, B.*
FROM Sensors A, Sensors B
WHERE |A.hum - B.hum| < 2%
AND   |A.temp - B.temp| < 0.3°C
AND    A.id != B.id
SAMPLE PERIOD 30s
```

■

5.1 The CJF Approach

As with SENS-Join, our goal is to avoid shipping non-joining tuples. SENS-Join[1] uses a precomputation to compute a join filter, i.e., to compute the set S_J of tuples that join. A precomputation is good for one-time queries. However, for continuous

[1]There are further approaches that have studied the idea of not sending non-joining tuples, most notably [YLZ06, CNS07]. All of them use a semi-join based precomputation to find the set of joining tuples. A detailed discussion of related approaches is provided in Section 3.2.

queries, it is not optimal: Due to temporal correlations, changes in subsequent sensor readings are usually small. Thus, for most tuples, the property if they join remains unchanged. Repeating the computation of S_J prior to each execution is a substantial overhead. In contrast, maintaining S_J is difficult: If the Tuple t of a node is not in S_J, the node discards t. Thus, the data is unknown for maintaining S_J. The problem is to notice that, at some point in time, t enters S_J, and the node should now send its data. Otherwise, the join result is incomplete, i.e., incorrect.

To avoid such incorrect results while taking advantage of temporal correlations, CJF is based on filtering. As with IDEAL, the base station computes the final result. We propose to install filters on the nodes as they enable discarding tuples and, at the same time, can prevent incorrect results. The main contribution of CJF is to "maintain filters" in an optimal way, as overviewed after introducing filters.

In line with the literature, e.g., [OLW01], a filter is an interval: If the attribute value of a Node j is within its filter, then j does not send its data. We refer to the filter used by Node j as $filter_j$. The filter size is the width of the interval. For queries with multiple join attributes, filters are multidimensional. Section 5.3 formally specifies filters and filter sizes.

As a remark, this notion of a filter is different from a 'join filter' as used by SENS-Join. Conceptually, the join filter is a list of those join-attribute tuples that contribute to the result – a node sends its complete tuple in the final result computation step, if its join-attribute tuple is within the join filter. In contrast, CJF uses the classical notion of a filter which is an interval. A node sends its tuple if its join-attribute values are **not** contained in the interval.

Maintaining Filters. To avoid incorrect results though the input relations are incomplete due to the filtering, the base station does the following: If Node j has filtered its Tuple t_j, the base station uses the filter to check if the missing Tuple t_j still does not join (cf. Section 5.3). If the filter does not allow to rule out that t_j joins, the base station retrieves t_j. This mechanism separates computing the result from filtering – the result is correct for any setting of the filters.

Our central concern is efficiency. The key for minimizing communication is to optimize the size of the filter: If $filter_j$ is too small, its filtering capability decreases. In contrast, increasing the filter size increases the probability that the tuple of another node joins with some values inside $filter_j$. To guarantee correctness, the base station would retrieve t_j. A smaller filter could have avoided these costs. Most notably, an optimal $filter_j$ not only depends on the data of Node j but also on the data of other nodes. This makes optimizing filters for join queries different from filtering problems in the literature, such as [OLW01]. We will show that ignoring these dependencies results in bad performance. In addition, to obtain efficiency, it turns out to be essential to optimize the filters *individually for each node* and to *continuously adapt* them to changing conditions (sensor readings), as shown in Section 5.7 as well.

CJF accommodates these requirements. Optimizing filters requires knowing the sensor readings of all nodes. Thus, we base our approach on models of future readings. Further, the base station has to perform the optimization (cf. Section 5.3). Then, the problem of "maintaining filters" is as follows: (1) The base station needs to continuously compute filters that minimize communication costs. (2) For each execution, the base station has to decide which of the current filters to update. This is because updating filters means to send them to the nodes and incurs costs. However, due to temporal correlations in sensor readings, the changes of the filters are usually small, and updating might not pay off.

With respect to (1), we show how to map continuous join queries to an optimization problem under constraints. We then show how the base station continuously solves the optimization problem. This is difficult as standard optimization methods like Newton's method cannot handle constraints. In addition, the cost function of our problem is not convex. The challenge here is to avoid local minima. For the constraints, we present a solution based on the barrier method. With respect to optimizing non-convex cost functions, there is no general mathematical approach. However, we can exploit the continuous nature of the queries to relax the optimization problem. We allow CJF to *sometimes* work with local optima as long as they yield good performance. At the same time, we guarantee that this relaxation is transient: CJF will eventually find a global optimum. In particular, we propose a stochastic optimization approach for which we prove this property. In addition, we show how to avoid local optima that yield bad performance.

With respect to (2), CJF updates filters only if the expected improvement at least amortizes the costs of updating. This problem is more complex than those in the literature on updating filters (e.g.,[JKM$^+$07]), as we again need to consider dependencies among filters. In summary, our contributions regarding the processing of continuous join queries in sensor networks are as follows:

- To guarantee correctness while exploiting temporal correlations, we propose a filtering approach. CJF allows to separate computing the result from filtering such that the result is correct independent of the filters.

- We show how to map join queries to an optimization problem under constraints for optimizing filters.

- CJF continuously computes optimal filters. We present an algorithm that handles the constraints of the problem as well as the non-convexity of the cost function.

- We propose a scheme to decide which of the filters to actually update for each execution.

- We evaluate the performance of CJF based on real-world sensor data. In particular, CJF comes close to the lower bound provided by IDEAL, except for a small overhead due to maintaining the models.

Chapter Outline. In the following section, we briefly review related work on filtering. We then start the description of CJF by presenting a framework that integrates filtering with join processing (Section 5.3). In particular, we describe how to guarantee a correct join result, irrespective of the setting of the filters. In Section 5.4, we show how to compute optimal filters, i.e., we address problem (1). Updating filters, i.e., problem (2), is subject to Section 5.5. We provide some details on the implementation of our prototype in Section 5.6, before we present our experiments in Section 5.7.

5.2 Related Work on Filtering Data Streams

There is prior work on filtering in the literature on processing streaming data that is related to CJF. We review the relevant approaches in this section. We already discussed related work with respect to join processing in Section 3.2.

To briefly illustrate the context of querying streaming data, consider network monitoring as a typical application. Here, the status of network elements (e.g., routers) is monitored. That is, the elements are the data sources that continually report their status to the monitoring site, such as a network control center. Applications use queries over the data stream to specify the data that is relevant to them – this idea is similar to querying sensor networks.

The goal of the filtering is to reduce the amount of communication, as in sensor networks. The idea is to exploit temporal correlations by answering queries on cached versions of the data. This is possible as, e.g., network monitoring applications typically do not require absolute precision [OJW03]. Filters are used to discard data items at the sources, as long as the precision of the cached version is fine for the queries. The approaches share with CJF the problem of computing optimal filters.

Olston et al. [OLW01] presented one of the first solutions. In detail, the stream processor caches the most recently received data value of each source. In addition, the stream processor installs a filter on the data source. The filter is a (one-dimensional) interval which is centered around the most recent value. The data source sends its current data if it is outside of the filter.

The stream processor uses the cached data item to answer queries. In particular, the queries indicate the required quality of the answer by means of a maximum error. The stream processor can then decide whether the cached version of the data fulfills this requirement, based on the size of the current interval. If so, the cached data is used to answer the query. If not, the stream processor has to acquire the current reading from the data source.

Given this setting, Olston et al. face an optimization problem which is choosing optimal filter sizes: If a filter is too small, the current data of the corresponding source might fall outside of it and is sent. In contrast, if the filter is too large, it might not suffice to answer a query, given its quality requirements. The data must then be acquired. An optimal filter size minimizes the sum of both costs.

The solution proposed in [OLW01] is to continuously adapt the filters of each data source. The idea is to increase the filter size (with some probability) if the data is not filtered and to decrease the size if the data is acquired. As a main contribution, Olston et al. prove that this simple approach actually finds the optimum.

Our problem is different from theirs. In our context, the filters are not independent of each other and cannot be optimized in isolation. The dependencies in our case are that the filters themselves may not join. We will formalize this notion in the next section. Intuitively, if a pair of filters contains values that join, the base station has to acquire the current data from both nodes in order not to miss a result tuple. Olston et al. do not face such dependencies and thus do not account for the corresponding costs. Therefore, their solution is mathematically not optimal for our problem. Even worse, ignoring these dependencies actually results in bad performance in our case as we will show in Section 5.7.

While there exist filtering approaches that optimize filters at a central site and consider dependencies among filters, to our knowledge they target specific queries.

To give an example, Olston et al. [OJW03] showed how to optimize filters for aggregate queries. In detail, the maximum error refers to an aggregate over the values such as AVG. To allow for filtering data values at the sources, the idea is to assign a share of the maximum error to each node. Then, the shares depend upon each other as, overall, the maximum error might not be violated. In particular, Olston et al. show that it is suboptimal to give each node an equal share. Intuitively, the size of the share should correspond to the rate of change of the data at each node. Therefore, computing a share of the maximum error for each node is an optimization problem.

The proposed approach is as follows: Initially, each node gets some filter interval such that the maximum error is not violated. The idea then is that each node periodically shrinks the size of its filter. This frees a part of the error budget which can then be assigned to the nodes by the stream processor. As their main contribution, Olston et al. show how to find the node that benefits the most from an increase in the filter size. The authors set up a linear program to find this node. This solution is specific to the problem of computing aggregates and cannot be transferred to our problem.

Jain et al. [JKM$^+$07] recently also considered computing optimal filters for aggregate queries. While their solution differs from the one presented in [OJW03], it is also specific to aggregate queries. In particular, Jain et al. do not build upon a periodic shrinking. Instead, the stream processor continuously computes optimal filter sizes. This incurs the problem of deciding on the filters to update, as in our case. We present their approach in detail in Section 5.5, along with the shortcomings of applying it in our context. Most notably, Jain et al. update the filters independent of

each other. This causes problems in our case where filters are not independent.

5.3 Filtering for Join Processing

This section presents the foundations of CJF. We introduce filters in Section 5.3.1. We then present a framework which integrates filtering with join processing (Section 5.3.2). Finally, we define the "filter maintenance problem" (Section 5.3.3).

5.3.1 Filters in CJF

A filter is a (hyper-)rectangle in the join-attribute space. A tuple is filtered if it is within the rectangle.

Definition 5.1 (Filter)
A filter is a multidimensional interval $[a_i, b_i]$. *The dimensions correspond to different attributes. The semantics is: If the attribute values of a Node j are within its filter in all dimensions, then j does not send its data.* $filter_j$ *denotes the filter used by Node j.*

We distinguish between static and dynamic attributes: Static attributes have fixed values. An attribute is dynamic if its value changes over time. As explained in Section 5.4, filters for discarding non-joining tuples cover dynamic (join) attributes.

Example 5.2: The join attributes in Example 5.1 are temperature, humidity, and node_id. Node_id is static as it is fix for each node. Thus, CJF uses two-dimensional filters. The dimensions of the join-attribute space are temperature and humidity. ■

Location within Join-Attribute Space. For now, assume that we know future sensor readings. In CJF, a filter is centered around the (expected) values of the join attributes at the time of filtering. If we accurately predict the sensor readings, these are the optimal locations of the filters.

Filter Size. The size of a filter is the widths of the intervals $[a_i, b_i]$. Thus, for an n-dimensional filter, there are n different sizes per filter. To allow for a compact representation and thus a small overhead of sending filter sizes, we use a single parameter "size factor" s to specify the sizes. The size factor s is a scaling factor. The size of a filter interval $[a_i, b_i]$ in dimension i is computed by multiplying the variance σ_i of the physical quantity with s. For simplicity, we use the same σ_i for all nodes, which is a network-wide average and is fix for the duration of the query.

The expected sensor readings determine the location of the filters. Therefore, the size factor s is sufficient to specify a filter.

5.3.2 A Join-Filtering Framework

In the following, we describe our join-filtering framework, where the sensor nodes and the base station have different tasks.

Our goal is to perform close to IDEAL by discarding non-joining tuples. At the same time, we want to avoid re-computing the set S_J of joining tuples for each execution. This leads to the problem of correctness: If a Node j with $t_j \notin S_J$ does not send its data, this data is unknown. The problem then is to notice that, at some point in time, t_j enters S_J and should be sent. If we fail to realize such situations, the join result will be incomplete.

To overcome this problem, CJF is based on filtering. As in IDEAL, the base station computes the result. To discard tuples, the base station installs filters on the nodes. In particular, filters can also be used to guarantee correctness. The following mechanism accomplishes this: If a Node j has filtered its tuple t_j, the base station can use $filter_j$ to check if t_j still does not join. To do so, it exploits that t_j *has to be* inside $filter_j$. If none of the values in $filter_j$ joins with data from other nodes, then t_j cannot join. If $filter_j$ does not suffice to rule out that t_j joins, the base station retrieves t_j from Node j to compute a correct result.

Example 5.3: Consider the following join condition from Example 5.1: `|A.temp - B.temp| < 0.3°C`. If Node j has the filter [22°C, 23°C], and a Node h sends its tuple t_h with a temperature value of 24°C, then t_j and t_h cannot join. However, if h sent 23,1°C, $filter_j$ is insufficient to guarantee correctness. ■

This situation is important as it incurs costs:

Definition 5.2 (t_h collides with $filter_j$)
*A tuple t_h collides with $filter_j$, if there **exists** a value in the filter that joins with the tuple.*

Theoretically, it might also happen that two filters collide: For instance, think of filters, [22°C, 23°C] and [22.8°C, 23.5°C]. In the interest of correctness, the base station must retrieve two tuples. CJF avoids these extra costs when setting the filters (cf. Section 5.4).

The mechanism to guarantee correctness is embedded in the following overall process. It is used in each execution to compute the result. For now, suppose that each node knows its filter. The base station knows each node and the corresponding filter:

1. Each node reads the sensors that are relevant to the query.

2. Each node applies its filter: If the join-attribute values are outside of the filter interval, the node sends a tuple to the base station with all relevant attributes. Otherwise, it sends nothing. The tuple is kept in RAM until the next execution.

3. The base station receives the tuples that are not filtered and constructs the relations to be joined. For each node that has filtered its tuple, the base station adds a tuple consisting of the filter intervals $[a_i, b_i]$ as value for attribute i.

4. The base station joins the relations and retrieves missing tuples in case of collisions.

The framework achieves correctness *independent* of the setting of the filters. Filtering 'only' affects efficiency.

5.3.3 Maintaining Filters

We now introduce the central problem, namely "maintaining filters". To do so, we first illustrate how *filter sizes influence efficiency*: If the tuple t_j of a Node j joins, we should not install a filter on j. Otherwise, the filter would cause a collision, and the base station must retrieve t_j. In contrast, if t_j does not join, and we install a filter on j, we need to choose its size. If the filter is too small, t_j might fall outside of it and will be sent though it is not needed. If the filter is too large, we risk a collision. This is even more costly, since now t_j is queried *and* sent unnecessarily.

We optimize filter sizes *individually* for each node. Intuitively, to avoid collisions, filters should be smaller in regions of the join-attribute space where there are more potential join partners. Uniformly sized filters are suboptimal, as shown in Section 5.7.

As the costs corresponding to a specific filter size depend on the sensor readings of *all* nodes, it is necessary to know these readings to minimize costs. We approximate this knowledge by means of models of future readings. CJF computes filters at a central site as then, the nodes have to send model updates to only one location. In particular, the base station computes the filters. It needs to know them anyway to guarantee correctness.

Updating filters affects efficiency as well: The base station needs to send filter sizes to the nodes. This incurs costs. However, redistributing filter sizes in each execution in pursuit of a perfect filtering will cost too much if the current settings are good enough.

'Optimal filter maintenance' consists of the following problems:

1. The base station needs to determine filter sizes that minimize communication costs. This is subject to Section 5.4. Note that the problem is not choosing the locations of the filters – these are given by the expected sensor readings.

2. For each execution, the base station has to decide which filters to update (cf. Section 5.5).

Models. We need to choose a model to predict sensor readings. We used a linear model in our implementation, which we describe in Section 5.6. However, CJF is orthogonal to the choice of the model as long as it fulfills the following requirements:

- Provides an estimate (expected value) of the sensor readings of a node at the time of the next query execution.
- Provides a probability that the sensor readings of a node are in a specified interval.
- The probability function must be continuously differentiable.
- Provides the notion of variance of the measurements.
- Is simple enough such that nodes can compute it.

5.4 Computing Optimal Filters

To optimize filter sizes, the base station maps a join query to a mathematical optimization problem as shown in Section 5.4.1. We then present the algorithm used to solve it continuously (Section 5.4.2).

5.4.1 The Optimization Problem

We now specify our goal more precisely. So far, we do not consider time: We can compute filter sizes that minimize the expected communication costs *for each query execution*. Alternatively, we can optimize filters for a number of upcoming executions. We decided for the first alternative and always optimize filters for the next execution. The reason is that the model-based predictions of sensor readings are best for the near future. They degenerate with time. In addition, to consider multiple executions, we would have to decide on how many. This is difficult. Most notably, we do not know for how many executions the filters will actually be used. This depends on the optimal filters which we are about to compute.

We now say how to map a join query to a mathematical optimization problem which yields the optimal filter sizes. Our mapping works for general θ-joins as specified in our problem statement (cf. Section 3.1), subject to the following restriction: Each condition has to cover one attribute per relation. (We do not require identical attributes: For instance, 'A.temperature > B.humidity' is fine, though not meaningful.) In particular, we require both attributes to be dynamic or static, e.g., unlike '`A.node_id > B.temperature`'.

CJF starts by dividing the join conditions into those over static and over dynamic attributes. Conditions over static attributes are easy to deal with and are processed

separately: Since the base station knows the attribute values for each node, the idea is to simply evaluate the conditions and then evict them from the query. This is possible due to the conjunctions in the query: Pairs of nodes that do not fulfill one of the join conditions cannot join. As a result, we obtain for each node a set of potential join partners, which we will use in the optimization problem.

In detail, for each Node j that is involved in the query (i.e., for which $t_j \in (A \cup B)$), the base station computes a set N_j^{static}. N_j^{static} is that subset of nodes that joins with t_j based on the conditions over static attributes. Further, the base station obtains a set $N = \bigcup_j N_j^{static}$. The purpose of N is as follows: Nodes not in N are guaranteed not to join ($j \notin N \Leftrightarrow N_j^{static} = \emptyset$). We set their filter sizes to ∞. In contrast, nodes in N are considered in the optimization. We have to compute their filter size.

Example 5.4: In Example 5.1, there is one join condition over static attributes: `A.id != B.id`. For each Node j, N_j^{static} is the set of nodes in the network except j itself, as no tuple joins with itself. If the network contains at least two nodes, there are no non-joining tuples based on this condition: N contains all nodes. ■

The Optimization Problem. We will use the following variables: Let M_{ji} be a random variable modeling the measurement of Node j for Attribute i for the next query execution. m_{ji} is the expected measurement, i.e., $m_{ji} = E(M_{ji})$. $\vec{s} = (s_1, ..., s_{|N|})^T$ is a vector containing the filter sizes s_j of each Node j, as defined in Section 5.3. Let $cost_j$ denote the costs for j to send its tuple to the base station. We assume that sending to the base station incurs the same costs as in the inverse direction, to avoid a further parameter.

The Objective Function. The objective function specifies the communication costs given a specific setting of the filters $\vec{s}$. To formalize the communication costs, we first specify the costs due to a Node j, $commCost_j(s_j)$. According to our framework, a Node j sends its tuple in two cases: First, if it is outside of its filter. This incurs $outFilterCost_j(s_j)$. Second, if it is retrieved due to a collision ($collisionCost_j(s_j)$). Then,

$$commCost_j(s_j) = outFilterCost_j(s_j) + collisionCost_j(s_j)$$

$outFilterCost_j(s_j)$ is the probability that t_j is not filtered multiplied with the costs of sending:

$$outFilterCost_j(s_j) = (1 - P(t_j \text{ is filtered})) \cdot cost_j$$

According to Definition 5.1, for an n-dimensional filter of size s_j, t_j is filtered if it is within the interval $[a_i, b_i] = [m_{ji} - s_j \cdot \sigma_i, m_{ji} + s_j \cdot \sigma_i]$ in each dimension i:

$$P(t_j \text{ is filtered}) = P(\bigcap_{i=1}^{n} M_{ji} \in [m_{ji} - s_j \cdot \sigma_i, m_{ji} + s_j \cdot \sigma_i])$$

$collisionCost_j(s_j)$ is the probability of a collision, multiplied with the costs. The costs in this case are $2 \cdot cost_j$ as we query and send. However, t_j is only retrieved if it was filtered:

$$collisionCost_j(s_j) = P(t_j \text{ is filtered} \bigcap \exists h \in N_j^{static} : t_h \text{ collides with } filter_j) \cdot 2 \cdot cost_j \tag{5.1}$$

where the existence resolves to

$$P(\exists h \in N_j^{static} : t_h \text{ collides with } filter_j) = P(\bigcup_{h \in N_j^{static}} t_h \text{ collides with } filter_j)$$

The idea of computing $P(t_h \text{ collides with } filter_j)$ is to identify the subspace of the join-attribute space that collides with $filter_j$. We then only need to compute the probability of t_h being in it: $P(t_h \text{ collides with } filter_j) = P(\bigcap_{i=1}^{n} M_{hi} \in subspace_i)$.

Due to the conjunctions in the query, each join condition further narrows the subspace of tuples that collide with $filter_j$. Thus, CJF computes the subspace by iterating over the join conditions. *For each join condition* in the query, CJF takes the current subspace and restricts it. The restrictions themselves are specific to the join condition. However, the principle is always the same: The subspace is restricted to those tuples that join with one of the tuples in $filter_j$, based on the current condition.

Example 5.5: In Example 5.1, the join conditions are: $|A.att_i - B.att_i| \; \theta \; d_i^{max}$ where $\theta \in \{<, \leq\}$. Such a condition restricts dimension i of the subspace to all values that are at most d_i^{max} away from $filter_j$ ($[m_{ji} - s_j \cdot \sigma_i, m_{ji} + s_j \cdot \sigma_i]$); to have a collision, $M_{hi} \in [m_{ji} - s_j \cdot \sigma_i - d_i^{max}, m_{ji} + s_j \cdot \sigma_i + d_i^{max}]$. ■

Example 5.6: For $|A.att_i - B.att_i| > d_i^{min}$, a tuple t_h that collides with $filter_j$ has to be more than d_i^{min} away from it. This simply is: $M_{hi} \notin [m_{ji} + s_j \cdot \sigma_i - d_i^{min}, m_{ji} - s_j \cdot \sigma_i + d_i^{min}]$. ■

CJF restricts the subspace of tuples that collide with $filter_j$ for all conditions involving $\{\leq, <, >, \geq\}$ according to this principle.[2] Conditions involving equality and inequality conditions are handled as follows: Equality conditions can be expressed in terms of their absolute difference, e.g., $A.x = B.x$ is the same as $|A.x - B.x| \leq 0$. This has already been discussed. For inequality conditions, the model expects them

[2]Conditions without absolute value operators are are not symmetric. They require to distinguish between $t_j \in A$ and $t_j \in B$ for computing the probability of a collision. The details are discussed in Appendix C.

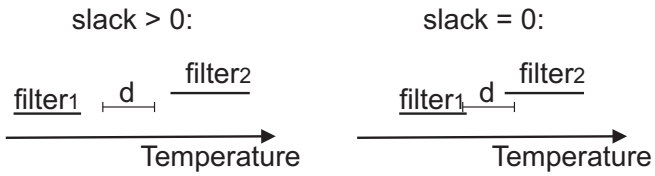


Figure 5.1: Slack between filters in dimension "temperature"

$$slack^i_{jh}(s_j, s_h) = \begin{cases} [(m_{hi} - s_h \cdot \sigma_i) - (m_{ji} + s_j \cdot \sigma_i) - d_i^{max} - \epsilon_i]^2, \\ \qquad \text{if } (m_{hi} - s_h \cdot \sigma_i) - (m_{ji} + s_j \cdot \sigma_i) - d_i^{max} - \epsilon_i > 0 \\ [(m_{ji} - s_j \cdot \sigma_i) - (m_{hi} + s_h \cdot \sigma_i) - d_i^{max} - \epsilon_i]^2, \\ \qquad \text{if } (m_{ji} - s_j \cdot \sigma_i) - (m_{hi} + s_h \cdot \sigma_i) - d_i^{max} - \epsilon_i > 0 \\ 0, \text{else} \end{cases}$$

Figure 5.2: Continuously differentiable function $slack^i_{jh}$ for $|A.att_i - B.att_i| < d_i^{max}$

to be fulfilled: The probability of two measurements being unequal is 1. Thus, according to an inequality condition, we expect tuples to join. In general, we must ignore inequality conditions when computing filter sizes.

Given $commCost_j(s_j)$, the overall communication costs is the sum of the costs due to each node, i.e., our objective function is:

$$commCost(\vec{s}) = \sum_{j \in N} commCost_j(s_j) \tag{5.2}$$

Constraints. Filter sizes must be non-negative. Thus, a first set of constraints is: $s_j \geq 0, \forall j \in N$. In addition, we need constraints to avoid colliding filters, as motivated in Section 5.3: A pair of filters may not contain a pair of tuples $t_j \in \mathit{filter}_j$ and $t_h \in \mathit{filter}_h$ that join. To formalize this constraint, we first define the function $slack_{jh}(s_j, s_h)$ such that $slack_{jh}(s_j, s_h) > 0$ if there is room to increase filter_j or filter_h without a collision. $slack_{jh}(s_j, s_h) = 0$ iff filter_j and filter_h collide. In detail, we define $slack_{jh}(s_j, s_h) = \sum_{i=1}^{p} slack^i_{jh}(s_j, s_h)$. There is one $slack^i_{jh}$ for each of the p join conditions in the query. The form of $slack^i_{jh}$ is specific to the join condition, but the principle is always the same: $slack^i_{jh}$ becomes zero if any pair of tuples $t_j \in \mathit{filter}_j$ and $t_h \in \mathit{filter}_h$ fulfills the join condition, and it is greater zero otherwise. Then, $slack_{jh}(s_j, s_h) = 0$ if the filters collide and greater zero otherwise, as intended.

Example 5.7: There are tuples that fulfill $|A.att_i - B.att_i| < d_i^{max}$ if the filters are closer than d_i^{max}, see Figure 5.1. Figure 5.2 shows how to formalize this. There is one case for $m_{ji} \geq m_{hi}$ and one for $m_{ji} < m_{hi}$, as the join condition is symmetric.

$slack^i_{jh}$ is well defined, as for any s_j, s_h, only one of the cases applies. ■

There are similar functions $slack^i_{jh}$ for other join conditions. – Two remarks conclude the discussion of $slack_{jh}$. Firstly, in Figure 5.2, instead of the distance in Dimension i we actually use the square of it. This is to ensure continuous differentiability of $slack_{jh}$, to solve the optimization problem. Secondly, $slack^i_{jh}$ turns zero if the filters become closer than $(d^{max}_i + \epsilon_i)$ instead of d^{max}_i. We will discuss the rationale behind ϵ_i at the end of Section 5.4. For now, suppose that $\epsilon_i = 0$.

Observation 5.1: Sometimes there are no filter sizes s_j, s_h that avoid colliding filters, i.e., $slack_{jh}(s_j, s_h) > 0$ cannot be fulfilled. □

If the predicted measurements of j and h join, then $filter_j$ and $filter_h$ collide, irrespective of s_j, s_h. This is because the expected measurements are always contained in a filter.

Given $slack_{jh}$, we define a constraint to avoid colliding filters:

$$(slack_{jh}(s_j, s_h) > 0) \vee (s_j = s_h = 0) \tag{5.3}$$

The disjunction guarantees that the optimization problem always has a solution, i.e., even if $slack_{jh}(s_j, s_h) > 0$ cannot be fulfilled. In this case, the only way to avoid a collision is not to filter t_j and t_h, i.e., $s_j = s_h = 0$. Summing up, our optimization problem is:

$$\begin{array}{ll} \text{minimize} & commCost(\vec{s}) \\ \text{subject to} & (slack_{jh}(s_j, s_h) > 0) \vee (s_j = s_h = 0), \forall j \in N \forall h \in N^{static}_j \\ & s_j \geq 0, \forall j \in N \end{array} \tag{5.4}$$

5.4.2 Continuously Optimizing Filters

Definition 5.3 (Optimal Filter-Size Factor $\vec{s^*}$)
$\vec{s^*}$ solves (5.4), i.e., the filter-size factors $\vec{s^*} = (s^*_1, ..., s^*_{|N|})$ minimize the expected costs under constraints.

We now turn to computing $\vec{s^*}$. The first obstacle in applying a standard technique such as Newton's method are the constraints. To handle them, we have developed a solution based on the barrier method. We give some background on this method in Section 5.4.2.1 and present our approach in Section 5.4.2.2. The second challenge in computing optimal filters is to avoid local optima. The underlying problem is that our objective function is not convex. The experiments in Section 5.7 demonstrate that this is indeed the case: There, the optimizer finds different optima for different starting points. This cannot happen for convex functions. It also shows that ending up

in a local minimum is an issue in practice. In Section 5.4.2.3, we present our solution for finding the global optimum $\vec{s^*}$.

To start with, we derive filter sizes that yield a lower bound for the cost function, $commCost(\vec{s^*})$ (recall that the location of the filters is given by the expected sensor readings):

Definition 5.4 (Isolated Minimum s_j^{iso})
s_j^{iso} is the filter-size factor that minimizes $commCost_j(s_j)$.

In contrast to s_j^*, the isolated optimum s_j^{iso} ignores the constraints (optimizes $commCost_j$ "in isolation"), i.e., the s_j^{iso} are not a solution for (5.4) in general. Note that s_j^{iso} exists as, due to the definition of $commCost_j(s_j)$, $commCost_j(0) = cost_j$ and $s_j \rightarrow \infty \Rightarrow commCost_j(s_j) \rightarrow 2 \cdot cost_j$. Finally, $commCost(\vec{s^{iso}})$ is a lower bound for $commCost(\vec{s^*})$: In the objective function (5.2), s_j affects only $commCost_j(s_j)$. Thus, the s_j^{iso} minimize $commCost(\vec{s})$.

The following proposition relates s_j^{iso} to s_j^* – *the optimal filter under constraints s_j^* is smaller than the isolated optimum s_j^{iso}*:

Proposition 5.1: s_j^* is in $[0, s_j^{iso}]$.

The proof uses the following insight, which is important in itself:

Lemma 5.1: If $filter_j$ of size s_j^{big} is feasible, i.e., causes no collisions, then every size $s_j^{small} < s_j^{big}$ is feasible as well.

Proof: s_j^{big} causes no collisions. That means, according to the definition, there exists no value inside $filter_j$ that joins. However, if we decrease the size of $filter_j$, every smaller filter is contained in the larger one (the filters have the same center). Thus, the smaller filter as well cannot collide. ■

Proof of Proposition 5.1: Assume that $s_j^* > s_j^{iso}$. Then, according to Lemma 5.1, s_j^{iso} is feasible as well. However, per definition, s_j^{iso} minimizes $commCost_j$, in contradiction to $s_j^* > s_j^{iso}$. ■

5.4.2.1 Background on Barrier Methods

To cope with the constraints, our solution is based on the barrier method. It solves large problems reliably and efficiently [Hin06]. Subsequently, we provide some basics of this method. See [BV04] for more details. It addresses problems of the following form:

$$\begin{array}{ll} \text{minimize} & f(\vec{x}) \\ \text{subject to} & c_i(\vec{x}) \geq 0, \quad i = 1, ..., m \end{array} \tag{5.5}$$

It does so by transforming the problem to an unconstrained one to which a standard optimization technique like Newton's method can be applied. The idea is to use

the constraints as a penalty in the cost function: If $\vec{x}$ approaches the bounds of the feasible region, the penalty increases significantly. Within the feasible region, it is (approximately) 0. This idea is realized as follows:

$$\text{minimize } f(\vec{x}) + (\frac{1}{t}) \sum_{i=1}^{m} -\log(c_i(\vec{x})) \tag{5.6}$$

for a constant $t \in \mathbb{R}^+$. The function $\phi(\vec{x}) = -\sum_{i=1}^{m} \log(c_i(\vec{x}))$ is called the logarithmic barrier. Irrespective of the value of t in (5.6), the logarithmic barrier grows to infinity if $\vec{x}$ approaches the bounds of the feasible region: $\exists i : c_i(\vec{x}) \to 0 \Rightarrow \phi(\vec{x}) \to \infty$.

Of course, (5.6) is only an approximation of the original problem. However, it can be shown that the solution of (5.6) converges to the solution of (5.5) as t grows to infinity [BV04].

In principle, we need to solve (5.6) for a large t such that the error of the approximation is smaller than some ϵ. However, if the parameter t is large, $f(\vec{x}) + \frac{1}{t}\phi(\vec{x})$ is difficult to minimize with Newton's method [BV04]. The solution is to solve a sequence of problems of the form (5.6), increasing the parameter t and therefore the accuracy of the approximation in each step, and starting each minimization at the solution for the previous value of t.

5.4.2.2 Coping with the Constraints

We now say how the base station computes optimal filter sizes for each node. To deal with the constraints in (5.4), the base station applies the following procedure *in each query execution*:

1. Transform the problem (5.4) to (5.6).
2. Solve the optimization problem (5.6).

Step 1: Transformation. To use the barrier method, we have to transform our optimization problem (5.4) to comply with the required form (5.5), leading to (5.6). The objective function $commCost(\vec{s})$ already does so. For the constraints $s_j \geq 0, \forall j \in N$ we can define $c_j(\vec{s}) = s_j, \forall j \in N$ to meet the form in (5.5). The challenge is to express the constraints (5.3) in the required form: Firstly, to avoid colliding filters, we need to guarantee $slack_{jh} > 0$, instead of '$\geq$' in $c_i(\vec{x}) \geq 0$. However, since we stop the iterative optimization for some $t < \infty$, we end up fulfilling the strict constraint $slack_{jh} > 0$. The more difficult problem is that (5.3) contains a disjunction. Recall that this is necessary as for some pairs of filters, $slack_{jh} > 0$ cannot be fulfilled (Observation 5.1). In that case, we must not install filters on j and h ($s_j = s_h = 0$).

1 $\vec{s}_0$:= `init`($\vec{s}_l^*$); *//$\vec{s}_l^*$: optimal assignment from last execution*
2 $\vec{s}_0$:= `neighbor`($\vec{s}_0$);
3 $t_0 := \frac{m}{f(\vec{s}_0)-p^*}$; *//m is taken from (5.6)*
4 $\mu := 20$; $\epsilon := 10^{-2}$; $k := 0$;
5
6 repeat{
7 $\vec{s}_{k+1}$:= minimize $f + \frac{1}{t_k}\phi$ starting at $\vec{s}_k$;
8 $t_{k+1} := \mu t_k$;
9 k++;
10 }**until**($\frac{m}{t_k} > \epsilon$);
11
12 *//$\vec{s}_{cur}$: current filter assignment*
13 **if** ($f(\vec{s}_{cur}) > f(\vec{s}_k)$) **return true**; *//better filters $\Rightarrow$ update $\vec{s}_{cur}$*
14 **return false**;

Figure 5.3: Optimization Method

This suggests the following approach: Prior to the optimization, the base station finds those pairs of filters having $slack_{jh}(\xi, \xi) = 0$ for a very small filter ξ. It sets their filter size to 0. This reduces the set of nodes N that have to obtain a filter to a smaller set N'. However, the nodes in N' can fulfill $slack_{jh} > 0$. Thus, CJF can now drop the disjunctive part of constraint (5.3) and obtains: $c_i(\vec{s}) = slack_{jh}(s_j, s_h)$.

Step 2: Solving the Problem. Figure 5.3 shows the algorithm for computing optimal filter sizes. For now, think of Lines 1 to 4 as initialization. In Lines 6 to 10, the barrier method is implemented as described in Section 5.4.2.1. In Line 7, the optimization problem is solved for a specific choice of t. Then t is incremented, and the procedure is repeated until a stopping condition holds (Line 10). The stopping condition guarantees that the error in the solution is $\leq \epsilon$ [BV04]. As the optimization problem in Line 7 is an unconstrained problem, we can use a standard (state-of-the-art) optimizer to solve it. In our implementation, we use the Quasi-Newton method BFGS [NW06]. Briefly, Quasi-Newton methods estimate second derivatives, in contrast to computing them, as Newton's method does. This is more robust and has lower costs per iteration. For details, see [NW06]. The code in Lines 1 to 4 and 12 to 14 is important for handling non-convexity.

5.4.2.3 Coping with Non-Convexity

The problem with Lines 6 to 10 (Figure 5.3) is that they might not find the global optimum as the objective function is not convex. There is no general approach for solving non-convex optimization problems. However, in our case, we can exploit the

1 **neighbor**($\vec{s}_0$)
2 **for** each $j \in N'$:
3 s_j^{iso} := min $commCost_j$;
4 s_j := random() $\cdot s_j^{iso}$;
5
6 **if** (feasible($\vec{s}$)) //$\forall j \in N' \forall h \in N_j^{static} : slack_{jh}(s_j, s_h) > 0$
7 **if** ($\frac{commCost(\vec{s}) - commCost(\vec{s}_0)}{commCost(\vec{s}_0)} < 0.1$)
8 **return** $\vec{s}$
9
10 **return** $\vec{s}_0$

Figure 5.4: Neighbor Method

continuous nature of the queries to relax the optimization problem. We allow CJF to (i) *sometimes* work with local optima as long as they yield good performance. (ii) At the same time, we guarantee that this relaxation is transient: CJF will eventually find a global optimum.

With respect to (i), we avoid bad local minima when choosing starting points for the iterative optimization (Line 1). See Paragraph 5.4.2.3.1. With respect to (ii), we present a stochastic optimization with the following property: We will prove that, for static environments, CJF converges to globally optimal filters. 'static' means that the sensor readings do not change. We use this requirement to capture our assumption of temporal correlation – if the sensor readings vary arbitrarily between subsequent executions, the stochastic approach does not necessarily converge.

The stochastic approach is based on the following idea: For each execution, we pick a random setting of the filter sizes $\vec{s}$ (Line 2). If and only if it is not too bad (as specified below), we substitute the starting point with this random choice. The important insight is: If, at some point in time, we choose a starting point close to the global optimum, then the optimization in Lines 6 to 10 will find the global optimum. In any case, we accept an optimum if it improves the costs of the best setting known so far (Lines 12 to 14).

Figure 5.4 shows how CJF chooses random filter sizes. In particular, this algorithm implements simple random sampling. For each node, `neighbor()` chooses a size in $[0, s_j^{iso}]$ where s_j^{iso} minimizes $commCost_j$ (Lines 2 to 4). According to Proposition 5.1, the optimal setting is in this interval. Paragraph 5.4.2.3.2 will say how to compute s_j^{iso}. We substitute our starting point with this random choice if it is feasible for (5.4) and is at most 10% worse regarding the objective function (Lines 6 to 8).

The main result of this subsection is: The probability that CJF never finds the global optimum is 0. This holds for the following reason: We do not need to sample exactly the optimum. Due to the optimization, we only need to find a point in a neighborhood where the optimization function descends to the global optimum.

Proposition 5.2: The probability of *never* generating a starting point $\vec{s}_0$ in a neighborhood $[s_j^* - \frac{\Delta_j}{2}, s_j^* + \frac{\Delta_j}{2}]$ of s_j^* is 0.

Proof: The important insight is that the probability of uniformly sampling a point in an arbitrary neighborhood of size $\vec{\Delta} > 0$ is constant ($g(\vec{\Delta}) > 0$):

$$g(\vec{\Delta}) = \int_{\Delta_1}^{0} ... \int_{\Delta_{|U|}}^{0} 1 d\alpha_1 ... d\alpha_{|U|} \cdot \frac{1}{\Delta_1 \cdot ... \cdot \Delta_{|U|}}$$
$$= \sum_{j=1}^{|U|} \Delta_j \left(\frac{1}{\prod_{j=1}^{|U|} \Delta_j}\right) = \text{const.}$$

Thus, the probability of *not* generating a point in $[s_j^* - \frac{\Delta_j}{2}, s_j^* + \frac{\Delta_j}{2}]$ is upper bounded by $1 - g(\vec{\Delta})$, where $(1 - g(\vec{\Delta})) \in [0, 1)$. Finally, $\prod_{t=t_0}^{\infty}(1 - g) = 0$ (which is the claim, starting at $t = t_0$ in time) $\Leftrightarrow \sum_{t=t_0}^{\infty} \log(1 - g) = -\infty$. This is because $\log(1 - g) \in (-\infty, 0)$. ■

5.4.2.3.1 Initialization. In Line 1 (Figure 5.3), `init()` chooses a starting point for the optimization. This must be done with care: The starting point must be feasible in our original optimization problem (5.4). This is because the constraints are used as an argument of a logarithmic function, which is undefined if the constraints of (5.4) are not met. In addition, the starting point should be as good as possible: If the optimizer only finds a local optimum, the corresponding costs are upper bounded by those of the starting point.

When choosing an initial value for s_j, we distinguish two cases: There is an optimal filter size s_j^{last} from the preceding execution. There is no s_j^{last}. The latter is the case if either Node j was not in N' in the last execution, or if we have the very first query execution.

In the first case, we initialize s_j with s_j^{last}, the optimum from the preceding execution. Given that the measurements do not change by much in between two executions, the optimal filter sizes do not change by much as well. As a detail, due to changes in the expected sensor readings, the sizes s^{last} can violate the constraints (5.3), i.e., two filters could collide. In this case, CJF decrements both filters by multiplying both s^{last} with the same factor $\in (0, 1)$. This way, we avoid penalizing one of the nodes more than the other. Note that we have to decrement: A filter collides if one of its values joins. Clearly, any larger filter collides as well.

If there is no s_j^{last}, our idea is to initialize s_j with a point close to s_j^{iso}. The motivation is that $commCost_j(s_j^{iso})$ lower bounds $commCost_j$, i.e., s_j^{iso} is optimal with respect to the communication costs. However, simply taking the s_j^{iso} will likely violate the constraints. To obtain a feasible setting, CJF decrements s_j^{iso} until the constraints are met. Specifically, if $filter_j$ of size s_j^{iso} collides with a $filter_h$ having a

```
1  S_I := {[0, s_max]};
2  ε := 10^{-1};
3  //initialize the current minimum s_min;
4  if (cC_j(0) < cC_j(s_max)){s_min := 0;}else{s_min := s_max;}
5
6  while(true){
7      pick I ∈ S_I : [cC_j(s_min) − lBound(I) > ε]∧
8                     [(cC'_j(s_min) > 0 ∧ s_min == b(I))∨
9                     (cC'_j(s_min) < 0 ∧ s_min == a(I))];
10
11     if (I == null){ //no such I ∈ S_I
12         pick I ∈ S_I : ([lBound(I) < cC_j(s_min)] ∧ [s_min > b(I)])∨
13                        ([cC_j(s_min) − lBound(I) > ε] ∧ [s_min ≤ b(I)]);
14         if (I == null) break; //all intervals fulfill the conditions
15     }
16
17     //split I
18     S_I := S_I\{I};
19     s := (a(I)+b(I))/2;
20     if ([cC_j(s) < cC_j(s_min)]∨[(cC_j(s) == cC_j(s_min))∧(s < s_min)]){ s_min := s};
21     S_I := S_I ∪ {[a(I), s], [s, b(I)]}
22 }
23 return s_min;
```

Figure 5.5: Optimizing $commCost_j(s_j)$ (= 'cC_j' in this figure)

previously optimal size s_h^{last}, then CJF decrements only s_j^{iso}. If both sizes are new, CJF decrements both with the same factor as discussed above.

Finally, for the initialization in Figure 5.3, p^* deserves explanation. It is used to initialize the parameter t of the barrier method. p^* is an estimation of $commCost(\vec{s^*})$. We estimate p^* as: $p^* = \sum_{j \in N'} commCost_j(s_j)$, where s_j is the optimum from the last execution s_j^{last}, if it exists. Otherwise, we use $s_j = s_j^{iso}$.

5.4.2.3.2 Optimizing $commCost_j(s_j)$. Finding the isolated minima s_j^{iso} is difficult due to the non-convexity of $commCost_j(s_j)$ – again, using a standard technique probably ends up in a local minimum.

To start with, we specify our goal more precisely. We are interested in s_j^{iso}, the filter size that minimizes $commCost_j$. If there are multiple minima, our goal is to find the smallest filter size that minimizes the costs. This is because it has the smallest potential of causing collisions.

Our optimization algorithm is based on the following insight: $commCost_j(s_j) = outFilterCost_j(s_j) + collisionCost_j(s_j)$, where $outFilterCost_j(s_j)$ *monotonically decreases* with increasing s_j and $collisionCost_j(s_j)$ *monotonically increases*. We can thus lower bound $commCost_j$ on an interval $I = [a, b]$:

$$\texttt{lBound}(I) = outFilterCost_j(b(I)) + collisionCost_j(a(I)) \tag{5.7}$$

In (5.7), interval I is a part of the domain of s_j, namely $\mathbb{R}_0^+$, and $a(I)$ and $b(I)$ are the endpoints of I.

Subsequently, we will establish three properties of $\texttt{lBound}(I)$ (Observation 5.2 - 5.4) which we use for finding s_j^{iso}. In particular, the observations serve as a mathematical foundation of our solution.

Suppose that we have split the domain of s_j into disjunct intervals. In addition, let s_{min} be a filter size from the domain. Intuitively, s_{min} will serve as an approximation of the result s_j^{iso}. Then, the following observation holds because $\texttt{lBound}(I)$ lower bounds the costs for sizes $s \in I$:

Observation 5.2: If $commCost_j(s_{min}) < \texttt{lBound}(I)$, then I cannot contain the minimum s_j^{iso}. □

Observation 5.2 indicates the main idea of our approach: Suppose that we have a candidate s_{min} which we suspect to minimize $commCost_j(s_j)$. Given a decomposition of the domain of s_j into intervals, we can use the lower bound to assure for each interval that it does not contain a filter size that results in lower communication costs.

We now identify situations in which the condition from Observation 5.2 cannot be fulfilled. As in case of Observation 5.2, Observation 5.3 holds because $\texttt{lBound}(I)$ lower bounds the costs for sizes $s \in I$:

Observation 5.3: $\forall s \in I : commCost_j(s) \geq \texttt{lBound}(I)$. □

Our idea is to discard intervals from the search based on Observation 5.2. However, due to Observation 5.3, we can only discard intervals $I : s_{min} \notin I$. Therefore, we refine our approach as follows: If $s_{min} \notin I$, we seek to discard I based on Observation 5.2. If $s_{min} \in I$, we require $commCost_j(s_{min}) - \texttt{lBound}(I) < \epsilon$. This condition bounds the error of our approximation s_{min} by ϵ: $commCost(s_{min}) - commCost(s_j^{iso}) < \epsilon$.

The final observation constitutes a problem if $commCost_j$ has multiple minima:

Observation 5.4: For any interval I that contains an $s' \in I$ with $commCost_j(s') = commCost_j(s_{min})$: $commCost_j(s_{min}) \geq \texttt{lBound}(I)$. □

Observation 5.4 immediately follows from Observation 5.3. As a consequence of Observation 5.4, the condition from Observation 5.2 cannot be fulfilled for any interval I that contains a minimum.

This leads to the final refinement of our approach which seeks to fulfill one of the following conditions for each interval: (A) Given a candidate s_{min} for s_j^{iso}, the algorithm uses Observation 5.2 to discard all intervals I for which $b(I) < s_{min}$. Thus, if (A) is fulfilled, we know that there is no filter which is *smaller* than s_{min} and leads to lower costs. (B1) For an interval I that contains s_{min} ($s_{min} \in I$), we require $commCost_j(s_{min}) - \texttt{lBound}(I) < \epsilon$, since Observation 5.2 cannot be fulfilled if $s_{min} \in I$. (B2) If $s_{min} > a(I)$, we also require $commCost_j(s_{min}) - \texttt{lBound}(I) < \epsilon$. This is because I might contain a further minimum and thus cannot fulfill Observation 5.2. (B1) and (B2) can be summarized as follows: (B) If $s_{min} \leq b(I)$, we require $commCost_j(s_{min}) - \texttt{lBound}(I) < \epsilon$.

Our algorithm assures conditions (A) and (B) as follows: It splits the domain of s_j into intervals. For each split, it computes $commCost_j$ at the point of splitting. Thus, we evaluate the cost function at endpoints of intervals. We maintain the current minimum, s_{min}, as an approximation of the result s_j^{iso}. In addition, we use the current s_{min} to fulfill (A) and (B). If an interval I does not fulfill the corresponding condition, we split it into I_1 and I_2. This might result in a new minimum. In any case, the lower bounds of I_1 and I_2 will be larger than $\texttt{lBound}(I)$. The algorithm keeps splitting intervals until for each interval either condition (A) (in case $b(I) < s_{min}$) or (B) ($b(I) \geq s_{min}$) holds.

Figure 5.5 shows the pseudocode of the algorithm. It implements the preceding idea along with an ordering of choosing intervals to split. The idea is to give priority to finding a good s_{min} as it allows for a better pruning. Therefore, we greedily search the minimum: We split that interval that contains the current s_{min} (as an endpoint) and into which $commCost_j$ descends, as indicated by the derivative – if it does not already satisfy Condition (B) (Lines 7 - 9). If there is no such interval, we arbitrarily choose one to split, until all fulfill Conditions (A) or (B).

Minimum Distances of Filters. Finally, we address the following problem: Suppose that two filters only marginally fulfill the constraints, i.e., $slack_{jh} = \epsilon$ for some small ϵ. Then, it might happen that in the next execution the expected values of Nodes j and h slightly change (by more than ϵ), and the filters collide. To avoid this, CJF has to update the filters. However, it is likely that the situation immediately repeats due to the trend in the sensor readings, and CJF again updates the filters.

To avoid this situation, we introduce minimum distances ϵ_i between filters, which are already included in the constraint $slack_{jh}$ in Figure 5.2. We set ϵ_i to the average rate of change of the corresponding physical quantity multiplied by the time t during which we want to avoid collisions. Note that for a static environment, $\vec{\epsilon} \rightarrow 0$. Thus, this extension maintains the property of convergence to the globally optimal filter sizes.

5.5 Efficiently Updating Filters

The base station computes optimal filter sizes for each execution. However, due to temporal correlations in the sensor readings, the changes are usually small. Thus, redistributing filter sizes for each execution will costs more than it saves. To obtain efficiency, [JKM+07] has proposed to update filters only if the expected savings outweigh the updating costs. This idea is applicable in our case as well. However, deciding which filters to update is more complex for join filters than for individual filters as in [JKM+07]. In the following, we discuss their solution before presenting ours.

Goal. To reduce the overall communication costs, our goal is to update a filter if and only if the improvement in terms of expected communication costs at least amortizes the updating costs (during the time we expect the filter to be used).

Approach from [JKM+07]. To decide if Node j should be updated, the authors compare the "charge" of continuing with s_j (costs due to a suboptimal filter) to updating ($cost_j$).

$$charge_j(s_j^*, s_j) = (commCost_j(s_j) - commCost_j(s_j^*)) \cdot (t_{curr} - t_{adjust})$$

The first factor is the expected difference in the communication costs if we use s_j instead of s_j^*. The second factor is the time for which we expect the filter size to be used. As an estimate, the authors use the time that the current filter has been valid if we update now: (current time − time of last adjustment).

$charge_j(s_j^*, s_j)$ is large if the current filter s_j is more costly than the optimal one, or if it has been a long time since the last update. Thus, in the static case, the latter factor will eventually force an update, yielding convergence to the optimum.

Problem. Our optimizer ensures that filters do not collide in the optimal setting $\vec{s^*}$. However, if we only update a subset of the filters, an old filter of, say, Node j might collide with an optimal (new) filter of Node h ($slack_{jh}(s_j, s_h^*) = 0$). Thus, in contrast to [JKM+07], we cannot update the nodes in isolation.

More precisely, the problem arises if a Node j has to shrink its filter (the current filter s_j is larger that the optimum s_j^*) and is not updated. This might prevent another Node h from enlarging its filter to s_h^*: If $s_h^* > s_h$, s_h^* can collide with s_j ($slack_{jh}(s_j, s_h^*) = 0$). If we do not update s_j, we cannot update s_h to s_h^*. Either we do not update s_h as well, or we use a size smaller than s_h^*. In both cases, not shrinking s_j leads to a suboptimal filter on h and thus incurs costs. $charge_j(s_j^*, s_j)$ does not reflect these costs.

Example 5.8: Assume that the filter of Node j is [22°C, 23°C], and j is supposed to shrink it to [22,3°C, 22,7°C]. Node h currently uses [23,4°C, 23,6°C] and is supposed to grow it to [23,1°C, 23,9°C]. For the join condition `|A.temp - B.temp| < 0.3°C`, h cannot use its new filter if j is not updated as well: [22°C, 23°C] and [23,1°C, 23,9°C] can contain joining tuples. ■

Our solution requires the following definitions: D_j is the set of nodes that "depend" on j to be shrunk, i.e., the nodes in D_j cannot use their optimum if j is not updated ($h \in D_j$ in Example 5.8):

Definition 5.5 (Dependent Nodes D_j)
$D_j := \{h | slack_{jh}(s_j, s_h^*) = 0\}$

In contrast, we say that j "blocks" h, i.e., j prevents h from growing to s_h^* if it is not updated:

Definition 5.6 (Blocking Nodes B_h)
$B_h := \{j | h \in D_j\}$

Finally, if h depends on j and we do not shrink s_j, then $s_{h,j}^{max}$ is the maximum filter size that can be assigned to h:

Definition 5.7 (Maximum Possible Filter Size $s_{h,j}^{max}$)
$s_{h,j}^{max} := \sup_{s_h}\{slack_{jh}(s_j, s_h) > 0\}$

Updating Filters in CJF. In CJF, deciding which nodes to update involves the following three tasks: (1) Resolving collisions of filters. (2) Shrinking filters. (3) Enlarging filters.

The decisions must take place in this order since the ability of enlarging filters depends on which of the filters were shrunk.

(1) Resolving collisions of filters. As filters are centered around expected sensor readings, their position in the join-attribute space changes due to trends in the readings. This can result in filters that collide based on their current filter sizes, i.e., without updating ($slack_{jh}(s_j, s_h) = 0$). In this case, the base station would retrieve the tuples of both nodes. This is twice as costly as updating them. Thus, as a first step, CJF finds pairs of nodes that collide based on their current filter sizes s_j, s_h and updates them.

(2) Shrinking filters. In the second step, CJF considers nodes for which the current filter size s_j is larger than the optimal one s_j^*. We have to decide if it is more costly to continue with s_j or to update. As in [JKM+07], using a suboptimal filter incurs $charge_j(s_j^*, s_j)$. In addition, not shrinking s_j might block a Node h, which can only use $s_{h,j}^{max}$ instead of its optimal size s_h^*:

$$blockingCost_j := \sum_{h \in D_j} \frac{1}{|B_h|}(commCost_h(s_{h,j}^{max}) - commCost_h(s_h^*)) \cdot (t_{curr} - t_{adjust})$$

The factor $\frac{1}{|B_h|}$ reflects that j is not the only node that blocks h and accounts j a share of the costs of blocking.

CJF updates s_j iff $charge_j(s_j^*, s_j) + blockingCost_j > cost_j$.

(3) Enlarging filters. Enlarging filters is simpler as it affects only the node in question. Let s_j^{max} be the maximum possible filter size ($\leq s_j^*$) given the decisions on shrinking ($= s_j^*$, if all nodes in B_j are updated). We update s_j iff $charge_j(s_j^{max}, s_j) > cost_j$.

5.6 Prototype Design

We conclude the description of CJF with an important design decision concerning our prototype. Recall that CJF is not tied to a specific model for predicting measurements. In this section, we say which model we use for our prototype.

The most important considerations when choosing a model are (1) the accuracy of the predictions and (2) the costs of maintenance. If the predictions are bad, the locations of the filters in the join-attribute space are suboptimal. Bad predictions also lead to a high variance and thus to small probabilities of the readings being in a filter. Maintaining the model is important as the quality of the predictions degrades in time. This is due to changes in the trend of the readings. Then, the nodes need to update the model to re-enable good predictions. Finally, some models involve extensive learning phases which are very costly.

For CJF, we decided on a simple linear regression model. From the literature it is known that linear models work well for a wide range of sensor data sets (see, e.g., [DGM+04, JCW04]). Our experiments substantiate this observation (cf. Section 5.7). In addition, our choice is very general: There is no need for fitting the model to our data/ application, and it avoids any extensive learning. Finally, the costs of maintaining the model is low. We will briefly give some details on the model before we discuss its maintenance.

The data is modeled as $y = \alpha x + \beta + e$ where the parameters α and β are estimated using ordinary least squares regression (OLS) on the n most recent data values received at the base station (in our implementation, $n = 6$). In our case, x is the time. e is an error term which is normally distributed for OLS, leading to simple computations of probabilities. There are standard formulas for estimating α and β as well as the variance of e. See, e.g., [JW02] for details.

[JW02] describes univariate as well as multivariate regression models. For ease of implementation, we assume independence of the attributes in our prototype: We

model each attribute in isolation using a univariate model. Although this is a simplification, we found it to work well for our purpose. In particular, the independence assumption simplifies computing the probabilities, for instance:

$$P(t_j \text{ is filtered}) = \prod_{i=1}^{n} P(M_{ji} \in [m_{ji} - s_j \cdot \sigma_i, m_{ji} + s_j \cdot \sigma_i])$$

With respect to maintaining the model, a model update is just a tuple, i.e., the current sensor readings of the join attributes. Therefore, we do not need a mechanism to decide when to update the model – this is implicit: A node sends its tuple to the base station whenever it is not filtered. Finally, as the model is based on the n most recent sensor readings for a small n, it quickly adapts to changes in the data.

Handling Selections. Our prototype handles predicates other than join predicates by a simple subscribe/unsubscribe mechanism: We use the common approach that each node checks these predicates locally. A node notifies the base station when it does not belong to any input relation due to the selection predicates. If, due to changes in the data, the node belongs to one of the relations in a later query execution, it simply re-starts sending tuples. Finally, the base station determines whether a node belongs to Relation A, B or both, simply by evaluating the selection predicates with the expected sensor readings. The nodes continuously check correctness of this membership.

Parameter $cost_j$. As a final remark on our prototype, we used a very simple cost model for $cost_j$. Not only did we use the same costs for sending from Node j to the base station and for the inverse direction. We also used the same costs for all nodes. This effectively eliminates $cost_j$ from our considerations. While we obviously could have used a more sophisticated cost model, this is not in the focus of our work. This simple constant cost model worked well for our experiments.

5.7 Evaluation

This section features an experimental evaluation. We show that maintaining filters is much more efficient than re-computing the set of joining tuples. We also show that CJF filters almost all tuples that do not join. Finally, we justify optimizing the size of the filters per node, in contrast to using uniform filter sizes. We also examine different influences on the performance of CJF, most notably of temporal correlation.

5.7.1 Experimental Setup

As with SENS-Join, we implemented CJF in the ns-2 network simulator to have a controlled environment of our experiments and repeatability. For our implementation, the runtime of optimizing filters was on the order of 2 seconds.

Data sets and Setting. Our goal is to evaluate CJF on real-world data, i.e., traces from sensor networks. In contrast to SENS-Join, CJF is sensitive to the data as it relies on temporal correlation – we wanted to avoid any bias in the results due to synthetic data. We used the publicly available LUCE data set [Senc] which has a large number of attributes. This allows for variations in the queries. To vary the characteristics of the data, i.e., the temporal correlations, we used different sections of the data set. The network consisted of 70 nodes when the data was acquired. For the experiments, we set the size of the network and the positions of the nodes to reflect the setting during the original data collection.

Alternative Approaches. We compare CJF to:

(1) External Join. The external join consolidates the entire input relations at the base station and joins them outside of the network. For details, refer to Definition 3.1.

(2) SENS-Join. SENS-Join is state-of-the-art for our problem.

(3) IDEAL. IDEAL is the lower bound from Section 3.1.3. Recall that IDEAL is infeasible in practice. However, as this is a simulation, we can provide the knowledge which of the tuples join at no costs.

(4) Adaptive Precision Setting (APS) [OLW01]. This approach continuously optimizes the filters of each node. However, in contrast to CJF, it does not directly account for the dependencies of the filters on the data of other nodes (Section 5.2 discussed APS in detail; we will repeat the important aspects below).

(5) UNIFORM. UNIFORM sets all of the filters to the same size. We present the details along with the experiment.

Queries. We adopt the approach taken in the evaluation of SENS-Join (cf. Section 4.4) for varying the queries. There, the queries correspond to the following pattern where the selected attributes and the join conditions are varied:

```
SELECT A.att_1,..., A.att_n, B.att_1,..., B.att_n
FROM Sensors A, Sensors B
WHERE join-expr(A.join-atts, B.join-atts)
AND ... AND join-expr(A.join-atts, B.join-atts)
SAMPLE PERIOD 30
```

As an illustration, consider the query from Example 5.1.

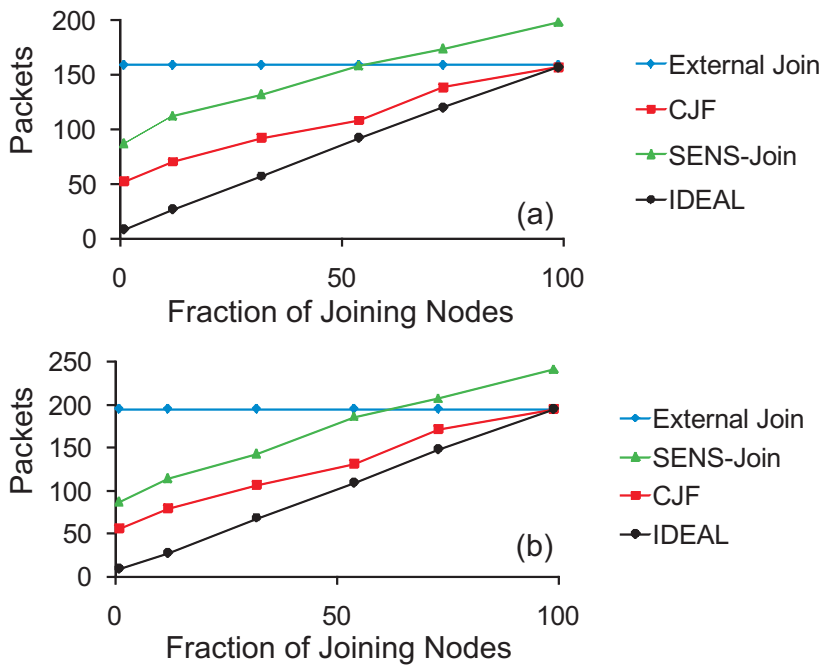


Figure 5.6: Costs of join approaches for different types of queries

Metric. We use the same metric as in Section 4.4: Number of transmissions (networking packets).

5.7.2 Comparative Experiments

Maintenance vs. Re-Computation. To demonstrate the efficiency of maintaining filters instead of re-computing them, we compare CJF to the alternative join approaches, SENS-Join, the external join, and IDEAL. For this purpose, we repeated the main experiments from Section 4.4. The parameter is the fraction of nodes that join, which is the most important influence on the communication costs. The results are based on a query with three join attributes and five attributes overall in the query (we vary the queries in turn). This ratio is important for SENS-Join. Note that CJF is not aware or tuned to these queries.

We varied the fraction of nodes in the result by adapting the join conditions, i.e., for `|A.att - B.att| <` δ, we varied the thresholds δ. The external join collects the entire relations. Thus, its costs are constant. We expect the efficiency of the other approaches to increase for less nodes in the result. In particular, CJF should be more efficient than SENS-Join as it does not involve a precomputation. The following results are based on 50 executions of the query after an initial warm up phase (CJF needs six time steps to establish the models). The results are shown in Figure 5.6(a). As expected, CJF consistently outperforms SENS-Join. CJF can reduce the overhead

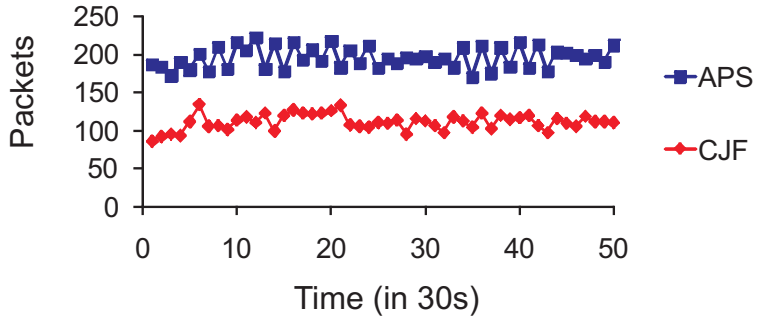


Figure 5.7: Considering dependencies among the nodes

over IDEAL by more than a factor of four. In contrast to SENS-Join, there is no break-even point with the external join as CJF does not involve any fix overhead. CJF is the best for all queries and imposes only little overhead over IDEAL. While it filters almost perfectly, this overhead is mostly due to updating the models at the base station (cf. Section 5.7.3 for details). This also explains why, for a small fraction of joining nodes, the overhead over IDEAL is larger than for a higher fraction: There are more models to update if less nodes join/ have filters.

The relative performance of SENS-Join compared to the external join is slightly better for a higher ratio of join attributes to attributes overall, as explained in Section 4.4. We therefore varied the queries in our experiments to have higher ratios of join attributes to attributes overall. For instance, Figure 5.6(b) shows the costs of a query with seven attributes overall and three join attributes. The relative performance of CJF compared to the external join is unaffected. The savings are proportional to the overall data volume in the query.

Considering Dependencies. A basis of CJF is that join filtering is different from other filtering problems. The size of a filter not only depends on the data of the node but also on the data of other nodes. We will now show that ignoring these dependencies results in bad performance. To this end, we compared CJF to APS. APS adapts the filters of each node in isolation. The idea is to increase the filter (with some probability) if the tuple is not filtered and to decrease the size if the tuple is queried [OLW01]. For this experiment, we fix the fraction of joining nodes to 32% such that a number of nodes can be filtered. Relative to the number of nodes that do not join, we obtained the same results for other queries. Figure 5.7 shows the number of transmissions in each time step. A further analysis has revealed that the reason for the bad performance of APS is a large number of collisions among the filters. Thus, it is important to consider dependencies when setting filters.

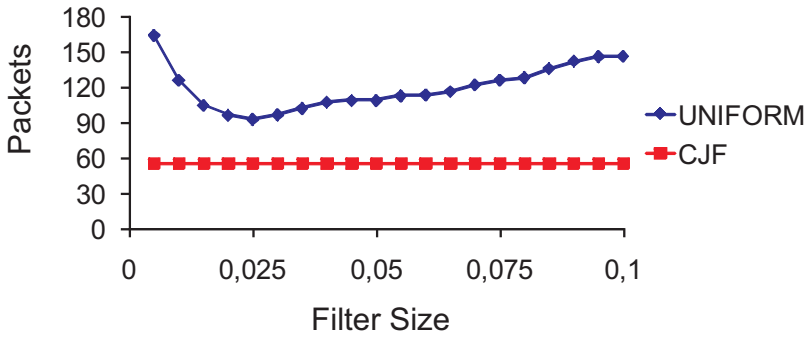


Figure 5.8: Using uniform filter settings

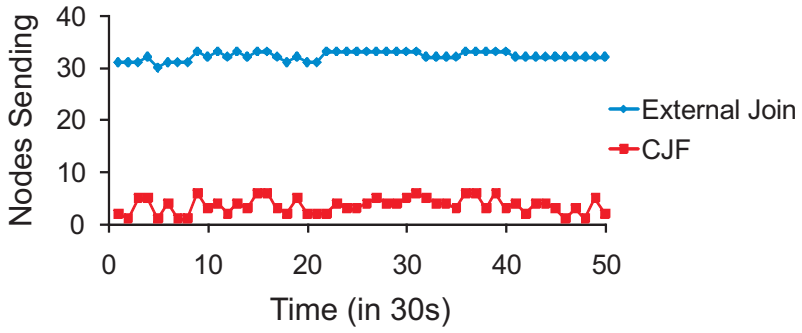


Figure 5.9: Performance of filtering of CJF

Individual Optimization. Another claim of ours has been that it is essential to optimize the filters individually and to continuously adapt them. To validate this claim, we compare CJF to UNIFORM. UNIFORM assigns the same filter to each node. However, it differentiates between tuples that join and those that do not. Only nodes that did not join in the last execution are filtered. We fix the fraction of nodes that join to 5% and use the size of the uniform filters as a parameter. This parameter does not influence CJF. Figure 5.8 graphs the results. Even if we knew which size to choose for UNIFORM, it is still significantly worse than using individually optimal sizes.

5.7.3 Analysis of CJF

Optimality of Filtering. We are interested in the filtering performance of CJF: Among the nodes that do not join, how many do not send? CJF might miss some due to filters being too small. Further, the model at the base station could be outdated. In this case, as the filters are centered around the expected sensor readings, the position of the filter is wrong, and a tuple goes unfiltered. The tuple then serves as an update of the model. We show results for a query that allows to filter 50% of the nodes

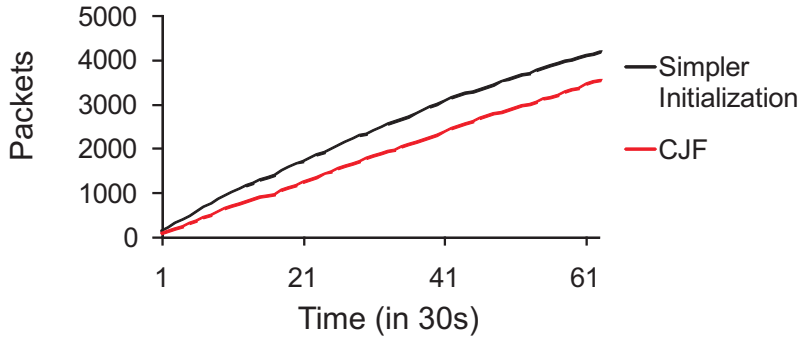


Figure 5.10: Initialization of filters and convergence

on average – again, the relative performance ($\frac{\text{nodes filtered}}{\text{non-joining nodes}}$) was independent of this setting. In Figure 5.9, we graph the number of unfiltered nodes. As the external join does no filtering, it indicates how many nodes can be filtered. On average, CJF successfully filters about 90% of the non-joining nodes. In a further analysis, we found that almost all of the unfiltered nodes are model updates. Thus, the difference of CJF and IDEAL stems from the need to maintain models at the base station. Beyond the updates, we found that the overhead of CJF is negligible: CJF updates less than one filter on average per time step. In addition, we observed less than two collisions on average. These figures apply to a broad range of queries that we have tried.

Initial Assignment and Convergence. To cope with the non-convexity of the cost function, CJF sometimes allows for local optima. The goal of the following experiment is twofold. Firstly, we want to illustrate the non-convexity of the cost function. But even more importantly, we want to show the effectiveness of our stochastic optimization approach. For these purposes, we compared CJF to an approach that uses a simpler initialization: It simply sets the initial sizes such that they filter the next reading with a probability of 99%. It then shrinks them until they fulfill the constraints. We used a query with only a few joining nodes, as a higher number of filters better highlights the difference. Figure 5.10 shows the cumulative costs of both approaches over time. Firstly, note that the optimizer indeed does not find the same setting with the different initializations. However, the difference between the two approaches becomes constant towards the end of the run: At some point both filter settings converge.

Varying Data Sets. CJF uses a model-based approach to know future sensor readings. Thus, the performance of CJF depends on the dynamics in the data. In the following, we analyze this influence. Therefore, we run CJF over four different sections of the LUCE data set. We have chosen them manually such that they have very different rates of change. For instance, during night time the data tends to be

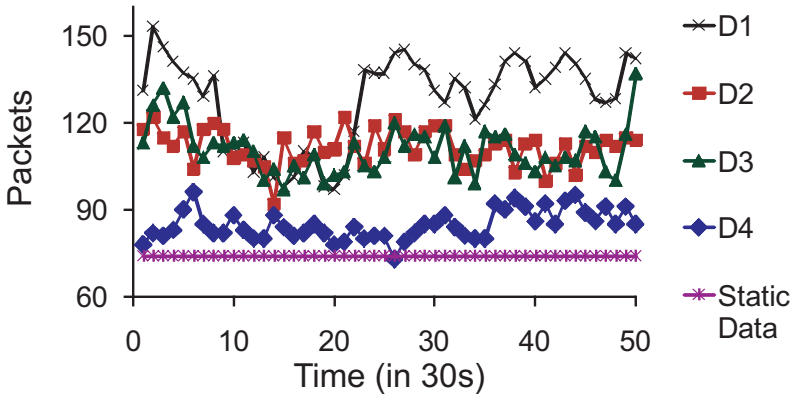


Figure 5.11: Influence of different data sets on CJF

quite stable, while, e.g., the temperatures show a sharp increase towards noon. We added a synthetic data set representing the best case for CJF: It is static. The queries were chosen to yield 50% $\pm$ 5% of joining tuples. Figure 5.11 shows the results. As expected, the performance of CJF is better for more stable data. For the static data, CJF actually matches IDEAL (not shown in the figure). The data sets in the preceding experiments performed similar to D2 and D3, i.e., they were neither particularly good nor bad.

5.8 Summary

This chapter studies continuous join queries. Our goal was to perform close to IDEAL by discarding tuples that do not join. In particular, we showed how to exploit temporal correlation in sensor readings as well as the continuous nature of queries to avoid a re-computation of the set of joining tuples for each execution. To this end, we have presented an approach that maintains join filters, CJF. What is special about join filtering is that the size of a filter depends on the data of other nodes. CJF builds upon models to consider such dependencies. It guarantees correctness in the presence of filtering. One contribution of ours is computing filters that minimize the communication costs. We have shown how to map join queries to an optimization problem under constraints and how to continuously solve it. To cope with the non-convex objective function, we have devised a stochastic algorithm that converges to the optimal solution and an approach for finding good initial filters. This assures good performance right away. Finally, CJF updates filters only if the improvement at least amortizes its costs. Our experiments indicate that the filtering is close to optimal.

The presentation of CJF concludes the first part of this dissertation in which we looked into processing queries with a well-separated interest, i.e., selective queries. In particular, query processing as discussed so far has exploited this selectivity: Ef-

ficiently processing selection, projection, and join operators builds upon not sending data that is irrelevant to the result. However, this idea only results in savings if queries actually are selective. In the second part of this dissertation, we address the problem of answering non-selective queries.

6 Approximate Query Processing in Sensor Networks

We now turn our attention to the second integral part of this dissertation: We consider situations in which applications simply collect the current sensor readings without prior selections. Examples of such applications are numerous. Firstly, in scientific environments, researchers often collect the entire set of measurements periodically for later analysis. The idea of discarding portions of the data is unattractive to them in their exploratory research, so they favor transmitting more data, even at the expense of a lowered sampling period or a shorter lifetime of the network [CDHH06]. As an example of an industrial application, consider quality management, i.e., monitoring the conditions in the production. To be concrete, think of a drug manufacturer who has to comply with legal requirements for documenting the production process. In particular, law requires the manufacturer to log the entire data [Pha]. Finally, making the sensor readings available at a central location periodically is a common data-acquisition task in monitoring production lines. Staff can then take corrective action immediately.

The common characteristic of all of these applications is that the data acquisition covers the readings from all of the sensors in the network. Thus, from the point of view of query processing, the queries simply consolidate the entire readings at the base station. In particular, such queries have a *low selectivity*.

In the following, we illustrate the problem of answering non-selective queries and motivate approximate query processing as a means of addressing this problem. We then review related work on approximate query processing. This motivates our own approach, which we present in Chapter 7. In addition, it justifies our choice of comparison schemes for the corresponding experiments.

6.1 Problem Statement: Processing Non-Selective Queries

Subsequently, our concern is efficient consolidation of the state of the network, i.e., of the current snapshot. An application can request a snapshot once or periodically.

So far in this dissertation, the basic method that we have discussed for processing queries is tree-based data collection. To reduce communication, tree-based data col-

lection exploits the selectivity of queries by pushing data-reducing operators into the network. Then, only the data that qualifies for the result is sent along a routing tree to the base station.

However, the problem with tree-based data collection is that, if selectivity is low, most of the data is in the result. Thus, plain tree-based data collection is communication intensive. Making matters worse, observe that the costs do not stem from the query processing approach – they are inherent in the problem itself. It is the size of the query result that causes the costs. Further, the size of the result is not due to some computation as in case of the join – the join increases the size of the data. In this case, the increase can be avoided by delaying the processing, i.e., processing the join operator at the base station. In contrast, for non-selective queries, the result is simply the sensor data.

A widespread approach for reducing the data volume is approximate query processing. As the problem is the large query result, the idea is to actually (slightly) modify the semantics of the query/the result itself: We allow for some inaccuracy in the result. In return, we obtain some leeway for an efficient query processing.

Approximate query processing in sensor networks is motivated as follows: In a lot of applications, queries cannot dismiss any sensor readings. In contrast, accuracy tends to be less critical: An approximation of the current snapshot is usually sufficient [DGM+04, CDHH06]. For instance, for drug manufacturer, it might be fine to know that the temperature in the production is 9.03°C or even 9.0°C, instead of seeing the exact output of a sensor (9.0273...°C). At the same time, it is important to bound the error of approximations. For instance, the drug manufacturer has to document that the temperature in the production process did not exceed certain thresholds. This is not possible without error guarantees. Subsequently, our goal will be to approximate snapshots of sensor readings under *user-controlled* error guarantees.

Formal Problem Statement. Our goal is to efficiently approximate snapshots under user-controlled error guarantees. As the guarantees in our targeted applications refer to individual values, we need a maximum error metric. This is because for SSE (sum squared error), the error for individual values could be arbitrarily high [GG02].

Formally, we target queries that conform to the following structure:

```
SELECT Att_1 ± e_1, ..., Att_n ± e_n
FROM Sensors
{WHERE predicates(Att_1, ..., Att_n)}
SAMPLE PERIOD x | ONCE
```

Example 6.1: The following query acquires the current temperature readings with a maximum error of 0.1°C:

```
SELECT temp ±.1°C
```

```
FROM Sensors
ONCE
```
■

Terminology. In the following, we will sometimes refer to the maximum error e as specified in the query by "**(maximum) error bound**".

Before discussing related work on approximate query processing in sensor networks, we provide some background on wavelet transforms. We do so at this point because it is the foundation of some of the related approaches. Thus, the following section simplifies an understanding of related works. However, it is much more important than just being background for related approaches. Our own approximate query processing scheme, which we will present in Chapter 7, is based on the wavelet transform. Thus, the subsequent discussion is also required to follow the discussion in the next chapter.

6.2 Constructing Wavelet Synopses

This section provides a brief recap of wavelets. Computing a wavelet synopsis consists of two subsequent tasks, wavelet transform and thresholding.

Wavelet Transform. We illustrate the principle by means of an example. Further information can be found in the standard literature, e.g., [SDS96]. Our description is based on the Haar transform which is the basis of SNAP. We justify this choice in Section 7.2.

Consider the following dataset which might be a column of a database table: $D = [2, 6, 7, 4]$. The transform pairs neighboring values v_x and v_y and computes an *approximation coefficient* for each pair. In case of the Haar transform, the approximation coefficients are simply the averages ($\frac{v_x+v_y}{2}$): $[4, \frac{11}{2}]$. This is a "lower-resolution" representation of D. In addition, a set of *detail coefficients* is computed as pair-wise differences divided by 2 ($\frac{v_x-v_y}{2}$): $[-2, \frac{3}{2}]$. These coefficients contain the information lost by averaging: Given both the approximation and the detail coefficients, the original data can be reconstructed. Recursive application of this pairwise averaging and differencing process on the lower-resolution data (the approximation) yields the following full transform:

Resolution Level	Approximations	Detail Coefficients
0	$[\frac{19}{4}]$	$[-\frac{3}{4}]$
1	$[4, \frac{11}{2}]$	$[-2, \frac{3}{2}]$
2	$[2, 6, 7, 4]$	

The result of a Haar transform is the single overall approximation coefficient followed by the detail coefficients in the order of increasing resolution. Thus, the transform of our example data is given by $W_D = [\frac{19}{4}, -\frac{3}{4}, -2, \frac{3}{2}]$.

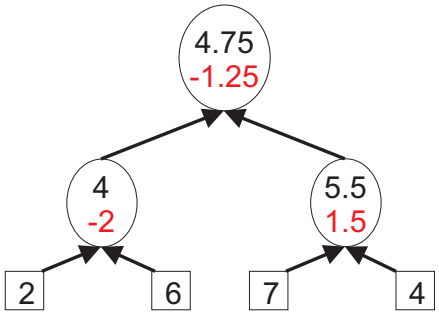


Figure 6.1: Structure graph for the Haar example

Each coefficient of the data transformed can be associated with a *coordinate* which identifies the coefficient by its resolution level, along with its position within this level. Intuitively, the coordinate simply corresponds to the position of the coefficient in W_D.

Thresholding. No information has been lost during the transform. Most notably, the original dataset D has four values, and so does W_D. The task of creating a compact representation of the dataset (synopsis) from W_D is called *thresholding*. In a nutshell, thresholding is discarding a subset of the detail coefficients. During reconstruction, the coefficients omitted are assumed to be zero. Thus, the resulting wavelet synopsis is an approximation of D.

Why is it advantageous to transform the data prior to thresholding? Wavelet transforms decorrelate the data. Most notably, if D contains similar values, the detail coefficients tend to be small. Thresholding then introduces only small errors. Thus, *in the context of constructing synopses, the goal of the transformation step is to obtain small detail coefficients.* The subsequent thresholding problem then is an optimization problem, e.g., find a minimum number of detail coefficients to retain, given a maximum error in the reconstructed data.

Structure Graph. To exploit the data reduction capabilities of wavelet transforms for an efficient data collection in sensor networks, the synopsis has to be constructed incrementally during the forwarding. That is, approximation schemes that build upon wavelet synopses have to distribute the transform. One of the major challenges in doing so is to map the logical data flow of the transform onto the routing structure of the network. Thus, in this context, the logical data flow of wavelet transforms is important [HW04].

In the following, we capture the logical data flow by means of a "structure graph". Figure 6.1 shows the structure graph for our example. A node represents a computation, i.e., nodes compute a function on the data obtained from their children. Edges represent data dependencies. For the Haar transform, the structure graph is a bal-

anced binary tree. In general, structure graphs are not necessarily trees but directed acyclic graphs (e.g., Daubechies-4, etc.). Graph structures arise for transforms that use the same approximation coefficient in more than one higher-level computation. Otherwise, we speak of a "structure tree". The term "computation node" stands for nodes of the structure graph, in contrast to sensor nodes.

6.3 Related Work on Approximate Query Processing in Sensor Networks

Data-reduction mechanisms studied by the database community include sampling, histograms, and wavelets. Among them, only wavelet-based approaches have been explored for approximate query processing in sensor networks. A reason might be that wavelets are known to achieve a higher degree of accuracy of query results compared to histograms and random sampling [CGRS00, MVW98, VW99]. Beyond these techniques from the database community, there are data reduction schemes based on modeling sensor measurements. We focus on model-based and wavelet-based synopses in what follows. We leave aside approaches that are restricted to approximating aggregates, such as [CLKB04, DKR04b, NGSA08], or [OJW03], which we described in Section 5.2.

6.3.1 Model-based Approaches

The idea behind model-based approaches is to answer queries based on a model of the current measurements, instead of querying the data from the sources. The approaches differ in the complexity of the model and their error guarantees. However, all model-based approaches estimate the measurements. If they provide error guarantees, their performance critically depends on the deviation from the actual sensor readings that an application can tolerate. For instance, estimating the temperature within $\pm 0.5°$C works well. For more accurate estimations in turn, frequent updates of the model are required, and the communication costs quickly increase. In Section 7.5, we compare our approximate query processing scheme to model-based approaches and show that it can efficiently answer queries for error guarantees that are tighter by orders of magnitude. Subsequently, we briefly introduce these approaches. This also justifies our choice of reference points for the experiments.

The simplest model is a constant. The base station caches the most recently received readings of each node. A node sends an update whenever a reading deviates by more than a certain threshold from the cached value. [OLW01] builds upon this idea – we already discussed this approach in the context of filtering in Chapter 5: The scenario considered by Olston et al. is to have a number of queries that differ in the quality requirements of the answer. The problem then is to choose a threshold that

minimizes the number of transmissions: For small thresholds, it is more likely that a new sensor reading deviates from the cached item by more than the threshold and is sent. In contrast, for large thresholds, it is likely that the maximum error requirements in the query are violated and the current reading has to be acquired. [OLW01] constantly adapt the threshold if either of the two cases occurs.

Jain et al. [JCW04] proposes to use Kalman Filters as a model. The underlying assumption for using a more complex model is that it allows for better predictions and thus results in less updates. The focus of this work is actually on the evaluation in which the authors validate this assumption. In detail, the scenario is slightly different from that in [OLW01]: Jain et al. consider a *single* query that indicates its quality requirements by means of a maximum error. The base station maintains a model of each of the current sensor readings which is a Kalman Filter. Based on this model, the base station estimates the current sensor readings and returns them as a query result. The nodes are responsible for guaranteeing correctness of the query result. To do so, the nodes maintain the same model synchronously. They compare the current estimate to the actual sensor reading and send a model update if the maximum error bound is violated (before the base station returns the query result). These approaches ([OLW01, JCW04]) strictly obey the quality requirements of the query. This corresponds to the guarantees in our problem statement.

In contrast, SAF (Similarity-Based Adaptive Framework) [TM06a] uses time series forecasting models with much weaker error guarantees. The setting is the same as in [JCW04]. However, the authors use probabilistic error guarantees. The idea is that the application knows that the current sensor readings are within a specified interval with a certain probability. Note that such a probabilistic guarantee naturally allows for less communication than a strict maximum error, as a violation of the error bound can sometimes be ignored. However, the approach suffers from the following restriction: The application can either influence the maximum error *or* the probability that the readings are within the corresponding confidence interval. That is, if the application provides a maximum error that it needs to be guaranteed, then the approach figures out with which probability it can guarantee this error (or vice versa). The application cannot set both parameters. This is in contrast to other approaches that provide probabilistic error guarantees and which we will discuss below.

The approaches mentioned exploit temporal correlations. (Additionally) capturing spatial correlations requires modeling several nodes, as [CDHH06, DGM$^+$04] do. Both use time-varying multivariate Gaussians.

As with the previous approaches, Ken [CDHH06] follows the idea that the nodes check the validity of the current estimate. To capture spatial correlation, Chu et al. propose to organize nodes in clusters and to maintain a model for each of them. Each cluster simulates the estimates of the base station and sends updates whenever error requirements are violated. This approach gives way to strict error guarantees. The main contributions are (a) to show how to optimally build clusters and (b) for each cluster to determine the minimal set of sensor readings to update the model. In

more detail, cluster formation is an optimization problem. Larger clusters can better accommodate spatial correlation. Thus, larger clusters lead to better predictions, thereby reducing the number of updates. In contrast, checking whether the current readings violate the error bounds requires to consolidate all of the current readings of a cluster at one node that maintains the model for the cluster. Thus, larger clusters increase the costs of guaranteeing the maximum error. Chu et al. propose an optimal and a heuristical approach to form clusters.

In contrast, BBQ [DGM+04] primarily executes at the base station, which maintains one probabilistic model of the entire network. The query result is estimated based on the model. If the confidence of an estimate is insufficient for a query, BBQ queries the network to fulfill the quality requirements. In more detail, the nodes do not need to synchronously maintain a model. This is in contrast to the preceding approaches. The query result as well as the confidence of the result is computed at the base station. As their main contributions, Deshpande et al. show how to do so for different kinds of queries (range queries, value queries, average queries). In addition, for querying the network if the confidence of an estimate is insufficient, the authors show how to identify the optimal set of attributes to observe. Note that this probabilistic approach does not allow for detecting sudden changes in the sensor readings or outliers, which might be interesting for an application.

While such sophisticated models enable better predictions than simple models, a downside is that they entail a very expensive learning phase. According to [TM06a], the initial training phase requires each node to transmit its readings to the base station every 30 - 60 seconds for a couple of days.

PAQ [TM06b] is similar in spirit to Ken, except that it is much less sophisticated: In principle, Tulone et al. also divide the nodes into clusters. The base station maintains a model for each cluster. This allows to capture spatial correlation. In addition, each cluster is responsible for checking the validity of the current estimates, as in Ken. However, the clustering of PAQ is much simpler. The requirements are that all nodes in a cluster must be within communication distance to each other. In addition, their sensor readings must be similar. Also, the authors use a much simpler model, i.e., a time series model. The design comes at the expense of much higher communication costs, compared to Ken.

Finally, there are two approaches that provide no error guarantees. Guestrin et al. propose to use a regression function as a model [GBT+04]. Their main contribution is to show how to distribute the regression. In particular, to reduce the complexity of the computation, the authors use so called kernel functions, that separate the network into slightly overlapping regions. Then, the regression within each region can be performed independently of the rest of the network. However, the distributed regression is quite expensive and has to be repeated every once in a while when the data distribution changes (in their experiments, the authors recomputed the regression model every 20 minutes).

Kotidis introduces snapshot queries [Kot05]. The idea is to select a set of repre-

sentative nodes. Queries are then answered exclusively based on the sensor readings of these nodes. In a way, queries are answered based on sample of the current sensor readings. As a main contribution, Kotidis presents a distributed algorithm which selects representative nodes. The choice considers the sensor data, i.e., 'representative' means representing the sensor readings in its neighborhood. While Kotidis does not use a model to answer queries, he proposes to use models for selecting representatives. The models capture the trend in the sensor readings. The intention is to ensure that the selected node will remain representative within a foreseeable time frame. Besides not providing any error guarantees, there is a further restriction of this approach: While a sample can be used to approximate aggregates, it is not appropriate to reconstruct the measurements themselves. In particular, for a missing value (non-representative value), the base station does not know by which of the received measurements it is represented.

6.3.2 Wavelet-Based Approaches

From a database perspective, there are many successful applications of wavelets as a data reduction tool. This is due to their good data reduction capabilities. The following paragraph gives an overview of successful applications of the wavelet transform in the database domain. However, the algorithms are inherently centralized. For instance, many of them use dynamic programming to establish the synopsis. In contrast, to use wavelets as a data reduction tool during data collection, we need to distribute the construction of the synopsis. The focus of this section is on reviewing distributed approaches to establishing wavelet synopses.

Overview: Centralized Wavelet-Based Approaches. Wavelets have been used in selectivity estimation [MVW98], for answering range-sum aggregate queries over data cubes in [VW99] and for general-purpose queries in [CGRS00]. [MVW00] studies dynamic maintenance of synopses. [DGR07] introduced extended wavelets for data sets with multiple measures. Recently, [SDS09] also addressed the overhead due to wavelet coordinates. While most of the early work focused on the SSE (sum squared error), [GG02, GK04] considered maximum error metrics. [GH05, GH06] showed that for metrics other than SSE, keeping the original wavelet coefficients is suboptimal and proposed approximation algorithms for optimal synopses. For streaming data, [GKMS01] proposes algorithms for building approximate synopses, and [KM05] introduces a greedy algorithm for constructing synopses with maximum error guarantees.

Distributed Wavelet-Based Approaches. The problem that has received the most attention when deploying wavelet transforms in sensor networks is mapping their regular refinement scheme/their structure tree onto the irregular network topol-

ogy. As a simple example, the classical Haar wavelet is computed on a perfectly balanced binary tree. Thus, a number of approaches studies the problem of mapping such a balanced binary tree onto a routing tree. To compute approximate query answers, a further problem has to be solved: One needs a distributed thresholding approach with error guarantees. None of the following approaches has addressed this second problem.

Acimovic et al. [ACBL05] propose a solution to distributing the computation of a Haar wavelet if the sensor network is a one-dimensional chain. In addition, the number of nodes has to be a power of two. Then, the nodes can be numbered 0 through 2^n-1. To compute the transform at the first level (level 0), the nodes with odd indices send their data to their lower-index neighbor. There, the data is transformed. Then, the nodes with odd indices are done with the transform, while the others take part in the next level. In general, at level m, the nodes with index $l \; div \; 2^m$ send their data to the nodes $l \; div \; 2^{m+1}$ where div is the usual integer division. While this scheme achieves a mapping of the transform onto the network, it is highly inefficient. This is because even if a node is not involved in the transform at level m anymore, most of the nodes are still needed for forwarding data – at every level (only neighboring nodes communicate). In addition, Acimovic et al. do not address the thresholding problem. For their experiments, they simply use a data distribution that is constant in space (i.e., within the network) except for a few discontinuities. Then, almost all neighboring nodes observe the same sensor readings, leading to zero coefficients. These can be discarded. However, zero coefficients usually do not appear in practice as it is unlikely that two nodes observe the same temperature readings. For a practical solution, we need a distributed thresholding approach.

Ciancio et al. [CO05] propose an approach that is similar to [ACBL05]. In particular, the authors assume the same network architecture, i.e., a one-dimensional chain of nodes. The main difference is that Ciancio et al. use a more sophisticated transform (5/3-wavelets). This results in the problem that a node sometimes cannot transform the received data: 5/3-wavelets require more than two coefficients as input at each level. Ciancio et al. show how to partially compute the transform on an incomplete set of input coefficients and to forward partial wavelet coefficients in this case. As in [ACBL05], the thresholding problem is not addressed. The most important shortcoming of both approaches is that they are confined to very specific topologies, i.e., one-dimensional chains. In particular, this property is exploited for the solution. There is no generalization of these approaches to arbitrary topologies.

Zhou et al. [ZLW+06] distribute the computation by means of an overlay, which is a one-dimensional chain. In principle, the transform can then be computed as described for [ACBL05]. However, the details of their approach remain unclear. In particular, it is not obvious how to construct such an overlay. Also, it is unclear where the data reduction comes from as the authors do not address the thresholding problem. We remark that Zhou et al. actually consider a slightly different problem. They do not collect a snapshot of data but each node assembles its data over a period

of time. Thus, the input to the transform is different: Each node contributes a set of sensor readings instead of a single reading per type of sensor. As a contribution, Zhou et al. present the maths of computing the Haar transform in this case. However, if the overlay idea is used for our problem, i.e., to collect a snapshot of data, the problem described for [ACBL05] applies: The forwarding significantly increases the length of the paths compared to a straightforward tree-based data collection.

Wagner et al. actually propose a transform that works on an irregular network topology [WBD+06]. In principle, the idea is to approximate a measurement by regressing a plane through the measurements of nearby nodes. The detail coefficient is then the deviation of the actual measurement from the approximation. To turn this idea into a practical approach, a hierarchical refinement scheme on the nodes is required, i.e., at each level of the transform, we have to decide on (a) the nodes that we want to approximate and (b), for each of them, nearby nodes that are used in the regression. As their main contribution, Wagner et al. show how to precompute such a hierarchical refinement scheme at the base station. However, it turns out that the resulting transform is communication-intensive. To compute a detail coefficient, the node that is approximated has to send its data to the nodes that are used in the regression. After some computation, each of these nodes returns a value from which the detail coefficient can be computed. In particular, as the transform is hierarchical, this happens at each level of the transform (at different scales) and involves many of the nodes multiple times. In their experiments, Wagner at al. show that transforming the data and subsequently collecting the coefficients is less efficient than a simple collection of the original data.

Hellerstein et al. [HHMS03, HW04] propose to integrate the wavelet transform into the routing tree. Recall that the routing tree guarantees shortest paths for each node to the base station. Thus, performing the transform along the routing tree is a good idea. We will discuss this approach in detail in Section 7.2.1. In addition, it will serve as one of the comparison schemes for evaluating our own approach in Section 7.5.

Finally, Dang et al. [DBcF07] compute wavelet synopses by assigning the nodes to clusters and transforming the data of each cluster. Thus, this approach does not perform a distributed wavelet transform but consists of multiple centralized transforms that are independent of each other. This is nice since then, all of the results from computing wavelet synopses in traditional database settings apply. In particular, it is clear how to transform the data and how to perform the thresholding under error guarantees. The problem is that only data within a cluster is decorrelated, redundancies between clusters are not removed. Thus, the performance of the data reduction is suboptimal, as we will show in Section 7.5.

6.3.3 Further Approaches

The goal of our work is to approximate current snapshots either for one-time queries or for a periodical data collection. Targeted applications are monitoring and surveillance applications that require query results in real-time. In contrast, there is some work regarding the acquisition of long-term historical records of measurements from each node. That is, the idea is for each node to assemble its data for a certain period of time and then to compress this data on the node before sending it to the base station. To give some examples, [DKR04a] proposes to exploit self-similarity, [LM03] constructs a piecewise approximation, and DIMENSIONS [GGP$^+$03] uses a wavelet transform for local compression of sensor readings at each node. Note that DIMENSIONS does not perform any distributed wavelet computation.

6.4 Discussion

In this chapter, we turned our focus to approximate query processing. In sensor networks, the motivation for approximate query processing is answering non-selective queries. The problem is that in the case of low selectivities, the data volume in the query is large, making the processing communication intensive. To obtain efficiency, the idea is to trade accuracy for communication costs in a controlled manner.

Prior work has addressed non-selective queries by approximating results based on models which serve as a synopsis of the data. The solutions suffer from the following problem: Predicting measurements is only possible with large error bounds such as ± 0.5°C for temperature readings. Higher accuracy requires frequent updates. According to [DGM$^+$04], "as epsilon gets small (less than .1 degrees), it is necessary to observe all nodes on every query...". Most notably, this problem is inherent in using predictions.

To overcome this problem, we propose a query-processing scheme based on wavelet synopses in the following chapter. The use of wavelet synopses is motivated by their effectiveness in compressing data and providing accurate answers [DGR07]. However, exploiting these capabilities in our context requires a distributed construction of synopses. Although being an area of active research, we pointed out in the preceding section that no prior work has presented a distributed transform that is more efficient than a simple tree-based data collection, let alone error guarantees in this context.

7 Efficiently Approximating Sensor Relations with Quality Guarantees

In this chapter, we present our approach, SNAP ("SNapshot APproximation"), for answering queries with a low selectivity. SNAP efficiently consolidates an approximation of sensor data based on wavelet synopses as motivated in the preceding chapter. The difficulty is that the synopsis has to be constructed incrementally during data collection to ensure efficiency. Our core contribution is to show how to distribute the construction of wavelet synopses in sensor networks. In addition, SNAP provides strict error guarantees.

The following section outlines the contributions of our approach and sketches the subsequent structure of this chapter.

7.1 The SNAP Approach

In the following, we give an overview of how to distribute the transform as well as the thresholding to achieve an efficient consolidation of sensor readings. Regarding the distributed wavelet transform, we see two design alternatives. We could use some standard transform and map it onto the network, i.e., assign the computations to sensor nodes and set up a network overlay. The overlay has to reflect data dependencies between computations. This approach has been considered before (e.g., [ZLW+06, WBD+06]). The problem is that it sacrifices shortest paths: The routes in the overlay will exceed those of a simple tree-based data collection by much (cf. Section 7.2). With SNAP, we explore an alternative design and integrate a wavelet transform into an optimal routing structure. We show that optimizing the integration is NP-hard and propose a polynomial algorithm that comes close to the optimal solution. Most notably, our solution is an online algorithm, i.e., the integration is not precomputed and can adapt to changes in the routing structure.

Regarding the thresholding process, discarding a subset of the detail coefficients yields a compact synopsis. However, doing so in a distributed environment leads to suboptimal synopses and to a huge communication overhead (see Section 7.3). Hence, we propose a distributed solution that is different and is inspired by work on image compression: Instead of discarding coefficients, we find a compact represen-

tation for them. The fundamental problem in distributing this alternative mechanism is as follows: To assign compact codes to the coefficients, we must know their frequencies in the overall synopsis. However, the synopsis is created incrementally. The frequencies cannot be known at the time of encoding. We show how to estimate the frequencies in a way that is mathematically sound. Our approach is also applicable to tuples with multiple attributes. This is an optimization problem in its own right if coefficients are discarded [DGR07]. In summary, our contributions regarding efficiently consolidating sensor relations are:

- We present SNAP, a distributed approach to compute wavelet synopses incrementally. To our knowledge, SNAP is the first distributed wavelet compaction providing error guarantees.
- To distribute the wavelet transform, we propose to integrate it into the routing structure. SNAP is first to explore this idea.
- Finding an optimal integration is NP-hard. We provide a heuristical solution and show that it performs well by means of an optimal algorithm based on dynamic programming.
- To distribute the construction of synopses, SNAP compactly encodes coefficients, instead of discarding them.
- Compacting the coefficients requires to know their frequencies in the overall synopsis. We show how to accurately estimate these frequencies.
- We evaluate the performance of SNAP based on real-world and synthetic sensor data. Our results are that SNAP reduces the communication costs by more than a factor of five compared to state-of-the-art approaches. SNAP improves the limit in the error guarantees for which data can be efficiently consolidated by more than an order of magnitude.

Chapter Outline. In the following section, we present our approach for distributing wavelet transforms. We then address the problem of distributing the construction of the synopsis, i.e., the data reduction, in Section 7.3. The following Section 7.4 concludes the description of SNAP with a further optimization. Finally, we present our experiments in Section 7.5.

7.2 Distributing Wavelet Transforms

This section presents our approach for distributing wavelet transforms. We extract design alternatives and justify our choice of integrating a transform into an unmodified routing tree (Section 7.2.1). Our integration approach builds on flexible structure trees (Section 7.2.2). We say how to optimize the integration in Section 7.2.3.

7.2.1 Design Space for Distributing Wavelet Transforms

To reduce communication compared to tree-based data collection, we have to construct the synopsis during forwarding incrementally. This requires a *distributed* wavelet transform. The problem is integrating a structure graph with the network topology: Here, 'finding an integration' means finding a wavelet transform and a mapping of its computation nodes to sensor nodes. Then the edges of the structure graph imply a communication overlay.

At a high level, we see two alternatives to address the problem: (1) One could adapt the routing structure to the transform, i.e., take a *fixed* structure graph and impose it as an overlay onto the network. However, network topologies are highly irregular. Mapping a fixed structure onto the network thus has an undesirable effect on the communication costs: It leads to nodes that merely forward data without prior wavelet compaction. The routes might not even be on the shortest paths to the base station [HW04]. (2) One could integrate the transform into the routing structure, i.e, the routing structure is fix. Each node would receive approximation coefficients from its children in the routing structure, transform them to obtain a synopsis and forward it to its parent. This approach requires specifying a structure tree that can be mapped onto the routing tree.

Related work has mostly explored the first approach (cf. Section 6.3). However, there are strong arguments for the alternative design. Firstly, communication is difficult in sensor networks, i.e., links are often fragile, and packet-loss rates are high. A lot of work has gone into maintaining routing trees that address these problems. It is desirable to build upon this mature technology. Second, recall that the routing tree is a shortest path tree. Deviating from it means doing suboptimal routes. SNAP therefore explores this second alternative.

There is one approach for building wavelet histograms [HHMS03] that operates on a routing tree, i.e., each node transforms the data received. To cope with the irregularity of the routing tree, the *input data* of the transform is adjusted ("zero padding"): Whenever a node has less than 2^n children, the input is filled up with zeros. The problem is that this results in *large* detail coefficients whenever an approximation coefficient is combined with a zero.

Example 7.1: Assume that a node has received the approximation coefficients 22.4°C and 22.6°C from its children in the routing tree. Its own sensor reading is 22.8°C. As the size of the input data to the transform has to be a power of 2, the node padds the data and obtains $D = [22.4, 22.6, 22.8, 0.0]$. This results in the following two detail coefficients: $[0.1, 11.4]$. In particular, for being able to reconstruct the 22.8°C, the node has to keep the 11.4. The problem repeats at the next iteration of the transform, when the two approximation coefficients $[22.5, 11.4]$ are averaged. Thus, each zero padding introduces a number of *large* detail coefficients, which is in the way of a compact representation. ■

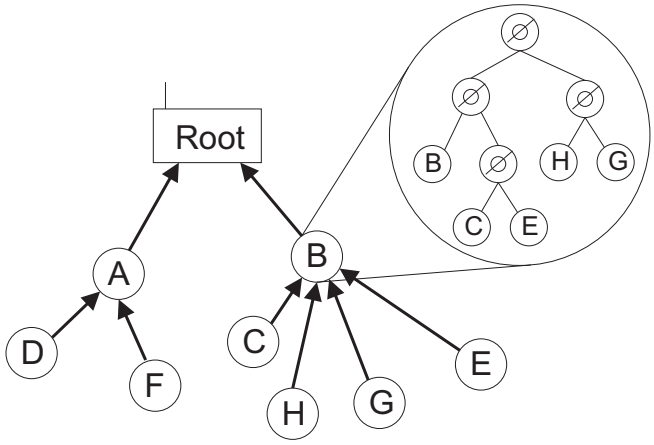


Figure 7.1: Integration approach

For our problem, zero padding turns out to be worse than a simple tree-based data collection (cf. Section 7.5). In contrast, our goal is to adjust the transform itself, not the input data.

7.2.2 Integration Approach

This section says how a structure graph can be integrated into a routing tree. The key idea for coping with the irregularity of the routing tree is to abandon the rigid structure of classical transforms.

As a first step in developing an integration approach, we need to select a wavelet transform which will be integrated (e.g., Haar, Daubechies-4, Mexican Hat, etc.). To do so, the question is which properties its structure graph must have to enable an integration. Foremost, the structure graph must be a tree. Otherwise, it would be impossible to map it onto a routing tree. It would be optimal if the structure graph complied with the routing tree. Unfortunately, no classical wavelet transform fulfills this requirement. This is because classical transforms have a regular refinement scheme. Their structure trees are balanced, and the node degree is fixed.

However, there are mathematical foundations on Haar wavelet transforms that have arbitrary (binary) structure trees [Mur07]. These transforms are more general than classical Haar transforms since the structure tree is not balanced any more. The problem remaining when integrating such a structure tree into a routing tree is that routing trees are not binary but have varying node degrees. The key observation to address this issue is that if a Node n_i has to transform m_i approximation coefficients, it is possible to arrange them in a binary tree and to apply the transform.

The preceding observation is the basis of our work, i.e., SNAP uses this flexible Haar transform. The flexibility of an *arbitrary* binary structure tree suffices to solve our integration problem. This is illustrated in Figure 7.1: Node B has set up a bi-

nary structure tree on the coefficients received from its children. This approach is pleasantly simple. At the same time, the flexibility in the structure tree leads to an optimization problem: find an optimal integration. This will be in the focus of the subsequent discussion.

7.2.3 Finding the Optimal Structure Tree

In the following, we present an online algorithm to integrate a binary structure tree into the routing tree. By performing the integration on-the-fly in contrast to precomputing it, the structure tree can react to changes in the network topology. Further, we avoid the communication overhead of an isolated precomputation.

We start by stating the optimization problem and establish its NP-hardness. As we cannot expect to find any exact polynomial algorithm, we devise a polynomial heuristic. We present an algorithm that finds an optimal integration based on dynamic programming (DP), to evaluate the heuristic.

The Optimization Problem. If a Node n_i receives m_i approximation coefficients, one per child in the routing tree, it adds its own sensor reading as a further coefficient. Then n_i transforms the data to obtain one overall approximation and m_i detail coefficients. This process reveals that the routing tree already determines the rough shape of the structure tree. Only for the $m_i + 1$ approximation coefficients at Node n_i, the structure tree is still unclear. The node needs to arrange the approximation coefficients received in a binary tree. For instance, if Node n_0 receives two approximation coefficients a_1 and a_2, there are three possible binary structure trees. n_0 can either combine a_1 and a_2 and subsequently combine the result $a_{1,2}$ with its own coefficient a_0. Alternatively, it can start combining a_0 and a_1 or a_0 and a_2. These combinations yield different detail coefficients.

Example 7.2: For the data from Example 7.1, one alternative is to first average 22.4 and 22.6. This results in an approximation of 22.5 and a detail coefficient of 0.1. Then, 22.5 and 22.8 are transformed. This results in the following set of detail coefficients: $\{0.1, 0.15\}$. In contrast, first processing $22.4, 22.8$ and then averaging the result with 22.6 yields $\{0.2, 0.0\}$. ■

Recall from Section 6.2 that the sizes of the detail coefficients determine the size of the synopsis, which is created from the data transformed. Thus, by specifying the structure tree, we determine the size of the resulting synopsis. The optimization problem then is to find the best alternative for n_0.

What is an appropriate optimization function? We want to minimize the data that n_i forwards. This implies the optimization function Ω: Ω simply is the size of the synopsis (*number of bits*) built from the transformed data. We defer constructing the synopsis to Section 7.3. Note that Ω also works for data with multiple attributes.

```
1  HeuristicalStructureTree(Set {ā_1, ..., ā_m})
2      //ā_i: approximation coefficients (from children + own measurement)
3
4      //initialization − current partial structure trees:
5      Set curTrees = ∅;
6
7      for j = 1 .. m
8          appC = ā_j;
9          NestedSet strTree = {ā_j};
10         curTrees = curTrees ∪ {(appC, strTree)};
11
12
13     until (|curTrees| == 1) do{
14
15         (cT_i, cT_j) = min_{{(cT_i,cT_j)∈curTrees}}{Ω(detCoeff(cT_i.appC, cT_j.appC))})
16
17         curTrees = curTrees \ {cT_i};
18         curTrees = curTrees \ {cT_j};
19         curTrees = curTrees ∪
20                 {(appCoeff(cT_i.appC, cT_j.appC), {cT_i.strTree, cT_j.strTree})};
21     }
22
23     return cT.strTree where cT ∈ curTrees;
```

Figure 7.2: Heuristical algorithm for computing the structure tree

Theorem 7.1: (Complexity of the problem) The problem of finding the optimal structure tree is NP-hard.

Proof: The problem of ordering joins can be reduced to our problem. Ordering joins in its most general form has recently been shown to be NP-hard [SM97]. The general problem includes considering bushy trees, i.e., there are no restrictions with respect to the ordering. It also involves cross products, i.e., every pair of relations from the query can be combined. The reduction works as follows: Every relation of the join query is mapped to an approximation coefficient. The cost function corresponds to the costs of executing the join of a particular ordering. Then, an optimal structure tree corresponds to an optimal join ordering. ■

Heuristic Approach. Due to the hardness of the problem, any algorithm that computes a structure tree on resource-constrained sensor nodes should not try to enforce optimality. We use a heuristical algorithm instead, which greedily minimizes

the optimization function: In every step, the heuristic combines those approximation coefficients yielding the smallest detail coefficient (with respect to Ω). Figure 7.2 shows the pseudocode. Initially, each approximation coefficient constitutes a separate tree (Lines 7 to 10). Then, the algorithm finds that pair of trees that yields the smallest detail coefficient (Line 15). The corresponding trees are removed from the set of current trees (Lines 17, 18). For this pair, the algorithm computes the new approximation coefficient (Line 20). Finally, the algorithm keeps track of the sequence of combining approximation coefficients based on a nested set structure where the brackets indicate the formation of the structure tree (Line 20).

We now say why this heuristic is appropriate. To do so, we present a DP-based algorithm that computes an optimal solution, and in Section 7.5 we will compare the heuristic to it.

Finding an Optimal Structure Tree. The idea of DP is that whenever two partial solutions are equivalent (can be substituted in the overall solution), one only needs to consider the better one with respect to the optimization function. In our case, two partial structure trees are equivalent if they cover the same set of approximation coefficients and yield the same overall approximation coefficient. The second condition is required since the overall approximation is computed based on *binary* averages. Thus, the overall approximation coefficient varies slightly with the order of the average computations, i.e., with the structure tree. Figure 7.3 shows the corresponding DP algorithm.

The algorithm incrementally constructs all partial solutions of size i. In Line 6, the partial solutions of size 1 are initialized. A partial solution consists of five fields: (1) set of input coefficients covered, (2) overall approximation coefficient, (3) the detail coefficients of the partial solution, (4) the corresponding structure tree, represented as a nested set, and (5) the costs of the partial solution (size). The actual algorithm starts in Line 8. It iterates over the size of the solutions and builds partial solutions of size i by combining pairs of partial solutions (Lines 10 - 15). Only combinations of partial solutions that do not overlap in the covered input coefficients are valid (Lines 2 - 4, Figure 7.4). In Lines 6 to 11 (Figure 7.4), the new partial solution is constructed. Lines 17 to 23 (Figure 7.3) implement the DP truncation.

7.3 Constructing Synopses

The thresholding problem is fundamentally different in a distributed setting. In Section 7.3.1, we demonstrate the shortcomings of a distributed discarding of coefficients. To avoid these problems, we use a different mechanism as starting point: Instead of discarding coefficients, we compactly encode them (Section 7.3.2). To achieve a distribution of this alternative mechanism, SNAP estimates the frequencies of the detail coefficients (Section 7.3.3).

```
1  OptimalStructureTree(Set {a_1, ..., a_m})
2      //a_i: approximation coefficients (from children + own measurement)
3
4      //initialization − partial solutions of size 1:
5      for j = 1 .. m
6        PartSolutions[1] = PartSolutions[1] ∪ ({a_j}, a_j, ∅, {a_j}, 0);
7
8      for i = 2 .. m //i == size of solution
9        for l = 1 .. (i − 1) //l == length of left subtree
10         for k = 1 .. |PartSolutions[l]|
11         PartSol lSubtree = PartSolutions[l].nextElement();
12           for o = 1 .. |PartSolutions[i - l]|
13           PartSol rSubtree = PartSolutions[i − l]. nextElement();
14
15           PartSol newSol = CombineStructureTrees(lSubtree, rSubtree);
16
17           if (∃ x ∈ PartSolutions[i]):
18               x.covAPPs == newSol.covAPPs && x.app == newSol.app)
19             //truncation: retain only the better one
20             if (x.costs > newSol.costs)
21               PartSolutions[i] = (PartSolutions[i] − {x}) ∪ {newSol}
22           else //no equivalet solution available
23             PartSolutions[i] = PartSolutions[i] ∪ {newSol}
24     return min_{x∈PartSolutions[m]}{x.costs}
```

Figure 7.3: DP for computing the optimal structure tree – a benchmark for the heuristical approach

7.3.1 Thresholding in a Distributed Setting

In the following, we discuss two problems that arise if we distribute centralized solutions. (1) Discarding coefficients leads to suboptimal synopses. (2) There is a substantial communication overhead, because discarding coefficients and providing error guarantees requires the algorithm to maintain some state. This state would have to be forwarded along the tree in addition to the data.

In centralized settings, thresholding is an optimization problem under constraints. The goal either is to minimize an error given a maximum size of the synopsis or, to minimize the size given a maximum error bound. In a distributed context, the optimization problem is different. This is because we need to forward partial synopses, which incurs communication costs. Thus, we also need to consider the size of inter-

```
1  CombineStructureTrees(PartSol lSubttree, PartSol rSubttree)
2      //disregard pairs that overlap
3      if (lSubtree.covAPPs ∩ rSubtree.covAPPs ≠ ∅)
4        return null;
5
6      Set covAPPs = lSubtree.covAPPs ∪ rSubtree.covAPPs;
7      Integer appCoeff = appCoeff(lSubtree.app, rSubtree.app);
8      Set detCoeffs = {detCoeff(lSubtree.app, rSubtree.app)}
9                      ∪ lSubtree.detCoeffs ∪ rSubtree.detCoeffs;
10     NestedSet strucTree = {lSubtree.strucTree, rSubtree.strucTree};
11     return (covAPPs, appCoeff, detCoeffs, strucTree, costs(detCoeffs));
```

Figure 7.4: Combining two partial solutions into one

mediate results, not only the size of the *final synopsis*. Formally, the problem now is minimizing the data volume of *all the synopses*.

We illustrate the intricacy of this requirement from the point of view of a single node. The problem is that the knowledge of a node does not suffice to decide which coefficients to discard: If the node discards a detail coefficient, the synopsis comes closer to the maximum error. However, there might be smaller detail coefficients to come and thus, it would have been better to save the error budget.

The second problem of discarding coefficients is metadata overhead, in two ways. First, guaranteeing a maximum error requires each node to know the error budget. Assigning fixed error budgets per node would result in very small budgets, making it difficult to discard any coefficients. [KM05] proposes a solution for the centralized case: One needs to keep track of the error that has already been introduced. They show that this can be done effectively by maintaining a range [min, max] for the error. If we transfer this approach to our scenario, this enlarges each intermediate wavelet that is forwarded by two additional numbers per attribute.

There is an even worse overhead: We need to know the coordinates of the remaining coefficients to reconstruct the data. This is done by storing the coefficients as <coordinate, coefficient>-tuples. Thus, for a synopsis of a single attribute the coordinates double the data volume. Reducing this overhead has been addressed in different ways, e.g., [DGR07, SDS09]. A distributed approach to reduce the overhead due to the coordinates is an open problem.

7.3.2 Foundations for Compact Synopses

To avoid these problems, we base SNAP on a different mechanism for obtaining synopses. It is inspired by work on image compression and consists of three steps:

1. *Quantization*

2. *Integer Wavelet Transform*

3. *Entropy Coding* (Huffman Coding)

This alternative mechanism does not solve our problem of distributing the construction of synopses – this is discussed in Section 7.3.3 – but it is key to overcome the difficulties of discarding coefficients: Our mechanism does not do any such discarding. The error bound is exploited for the quantization which affects only a single tuple. This eliminates the need to forward error budgets. Finally, we can avoid sending coordinates along with the coefficients.

Intuition. The important step to obtain a synopsis is Step (3), the entropy coding. The idea is to use short codes for frequent symbols and longer ones for less frequent symbols. In our case, the symbols to encode are the detail coefficients. The performance of the entropy coding depends on (a) the distribution of the symbols (the more skewed the distribution is, the higher the effectiveness) and (b) the number of different symbols that appear (the more symbols there are, the more bits are required).

We can regard Steps (1) and (2) as creating the optimal conditions for an entropy coding. Step (1) gives way to a small number of different symbols. Each node quantizes its sensor readings based on the error bounds of the attributes. As a result, the infinite set of possible readings is turned into a small set. Most notably, an increase in the maximum error in the query reduces the number of different symbols. The wavelet transform (Step 2) achieves a skewed distribution. Recall that the wavelet transform is used to generate small detail coefficients. Most notably, this usually results in a distribution of the detail coefficients that is bell shaped around zero [CV00]. Finally, we use an integer version of the transform: While the quantization reduces the set of possible values, the regular Haar transform involves divisions by two and thus would re-enlarge this set. An integer transform avoids this problem as it yields coefficients within the input set. We will provide the details right away.

Quantization. Step (1) combines a quantization with a scaling such that each quantized value is a (positive) integer. This is required for the integer wavelet transform. Let x_i be the sensor reading of node i of an attribute, and let e be the error bound for this attribute. Finally, let *minVal* be an arbitrary lower bound for the sensor readings. Each node quantizes its measurements to obtain the approximation coefficient to start the transform with (level 1):

$$a_{1,i} = Int(\frac{x_i - \mathit{minVal}}{2 \cdot e}). \tag{7.1}$$

Here, $Int(\cdot)$ denotes the usual integer rounding. The base station can easily reconstruct the values as $x'_i = \mathit{minVal} + a_{1,i} \cdot 2 \cdot e$. Given $a_{1,i}$, x'_i is within $[x_i - e, x_i + e]$.

Integer Wavelet Transform. The integer version of the Haar transform is the S-transform [CDSY98], where l is the level in the structure tree and p the index of the coefficient at level l:

$$a_{l,p} = \lfloor \frac{a_{l-1,2p} + a_{l-1,2p+1}}{2} \rfloor$$
$$d_{l,p} = a_{l-1,2p+1} - a_{l-1,2p}$$

We use it with a minor modification. The S-transform either does no rounding or rounds down. Thus, the approximation coefficients systematically become smaller with the levels in the structure tree. It turns out that estimating the frequencies of the detail coefficients is much easier if we avoid this systematic deviation, see Theorem 7.2. As can be seen from the corresponding proof, this systematic deviation can be avoided by equally rounding up and down:

$$a_{l,p} = \begin{cases} \lfloor \frac{a_{l',p'} + a_{l'',p''}}{2} \rfloor & \text{if } a_{l',p'} \geq a_{l'',p''} \\ \lceil \frac{a_{l',p'} + a_{l'',p''}}{2} \rceil & \text{otherwise} \end{cases} \tag{7.2}$$
$$d_{l,p} = a_{l'',p''} - a_{l',p'} \tag{7.3}$$

As the structure tree is not balanced in our case, it is not obvious how the level l is defined: We define it as the number of sensor readings that a coefficient covers. This implies that $l = l' + l''$. p still denotes the index. The inverse is given by:

$$a_{l',p'} = \begin{cases} a_{l,p} - \lfloor \frac{d_{l,p}}{2} \rfloor & \text{if } d_{l,p} \leq 0 \\ a_{l,p} - \lceil \frac{d_{l,p}}{2} \rceil & \text{otherwise} \end{cases}$$
$$a_{l'',p''} = d_{l,p} - a_{l',p'}$$

Entropy Coding. While arithmetic coding [MNW98] usually is state-of-the-art for entropy coding, it is suboptimal in our case: It requires adding a few bits to each sequence of symbols to obtain unique encodings. Since our sequences consist of only a few symbols (usually clearly less than ten, depending on the number of attributes in the query) this is a substantial overhead. We use Huffman coding [Huf52] which is well-known to yield better results for such short sequences and is standard in this case. In addition, the low storage requirements and its simplicity make it a good choice for resource-constrained sensor nodes. Irrespective of the coder used, there is a problem when it comes to distributing the approach. Assigning codes requires knowledge on the frequency of the symbols by the time of encoding. This is the main challenge when distributing the approach and is addressed in Section 7.3.3.

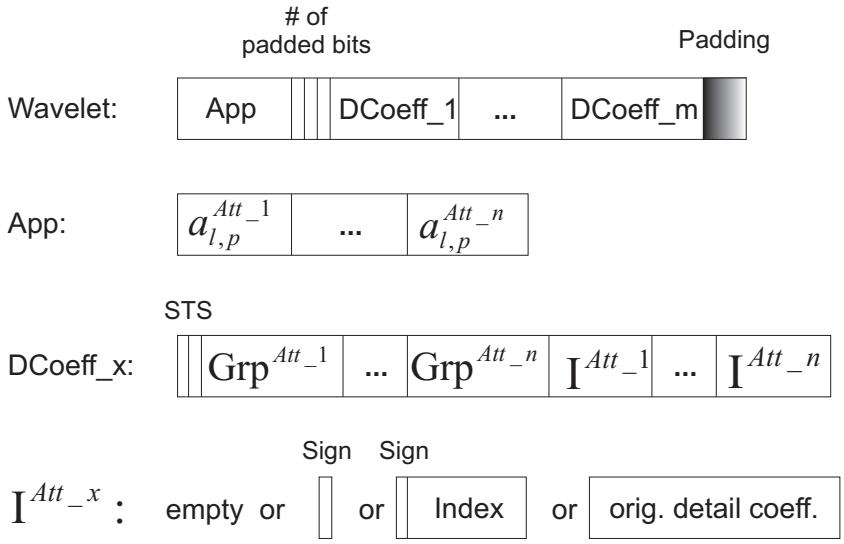


Figure 7.5: Format of Wavelets (Encoding)

Implementation Issues. We conclude the description of the framework with two details. First, we introduce the concept of "groups". It is standard in entropy coding to cope with large numbers of symbols. Second, we describe the format of wavelets sent.

"Groups" of symbols are used to reduce the number of symbols in entropy coding. The idea is to form groups of detail coefficients with similar frequencies. As a result, the number of groups is much smaller than the number of symbols, and each group is much more frequent. Each group is then assigned a Huffman code, i.e., only the group number is entropy-encoded. An index is used to discriminate between the members of each group. For instance, the detail coefficient 0 results in a single group. $\{-1, 1\}$ might be the second group. In particular, if a group contains x, it also contains $-x$, as our distribution is symmetric. $\{-3, -2, 2, 3\}$ might be the third group, etc. The final group contains the tail of the distribution. It contains all coefficients that are unlikely to appear.

The format used to send wavelets is shown in Figure 7.5. Basically, a wavelet is a bit stream which is padded at the end to align it with byte boundaries. At a high level, the wavelet starts with the approximation coefficients, one per attribute that is queried. A node then inserts three bits ("padding size") representing the size of the padding, followed by the detail coefficients. Their ordering corresponds to a depth first traversal of the structure tree. This allows to construct wavelets with ease: A node takes two wavelets as received from its children. Based on their approximation coefficients, it computes a new approximation and a detail coefficient. The latter is first in depth-first ordering. The node then simply appends the detail coefficients from the left child in the structure tree as received, followed by those from the right child.

Also, it removes the padding from its children in this append step. Given the padding size, this can be done without decoding the detail coefficients.

Figure 7.5 further shows the encoding of the detail coefficients: Each one starts with two bits called Subsequent Tree Structure (STS). As our structure tree is irregular, this is required to reconstruct the tree at the base station. STS encodes whether a node has two children, one left child, one right child, or no children. Thus, the purpose is similar to the coordinates of coefficients. Following the STS bits, the Huffman code of the group numbers are listed, one for each attribute. Finally, the indices follow. In case of the zero coefficient, no index is required. Otherwise, the indices start with a single bit which encodes the sign of the coefficient. For groups of size greater than 2, an index number follows that identifies the unsigned member. This number cannot be compressed since the group members are equally frequent. However, it makes sense to have groups whose size is a power of 2 to fully exploit this index number. The group that encodes the tail of the distribution is special: Here, the original detail coefficient is used.

7.3.3 Distribution by Estimating Frequencies

The mechanism described so far requires to know the frequencies of the symbols in the overall synopsis. This is the main challenge when distributing this approach. As the synopsis is now created incrementally, the frequencies are unknown by the time of encoding. Also, they strongly depend on the error bound e.

A common approach in entropy coding is to initially assume that each symbol is equally frequent and to adjust the frequencies continuously with each symbol. This approach is not applicable in our case. Each leaf node would start with a frequency distribution that is far from the actual one. This results in a bad compression. This distribution would only become better close to the root. Additionally, a node at a higher level of the tree receives encoded coefficients. To get to a count of the coefficients seen so far, the node would have to decode them. In the following, we propose an approach that lets each node *estimate* the frequency of a coefficient in the (unknown) final synopsis. Given this estimation approach, the previous framework can be executed in a distributed environment.

7.3.3.1 Estimating Frequencies

Our approach is based on two ideas. In the style of selectivity estimation, the base station continuously keeps track of the frequency distribution of the detail coefficients. That is, the first idea is to estimate the frequencies based on experience from previous queries. The main problem is that the frequencies strongly depend on the maximum error e. The second idea addresses this issue: The distribution is maintained for an error $e = 0$. As illustrated in the following, this corresponds to a *continuous* distribution of detail coefficients. *Given that the nodes know this distribution, it is now*

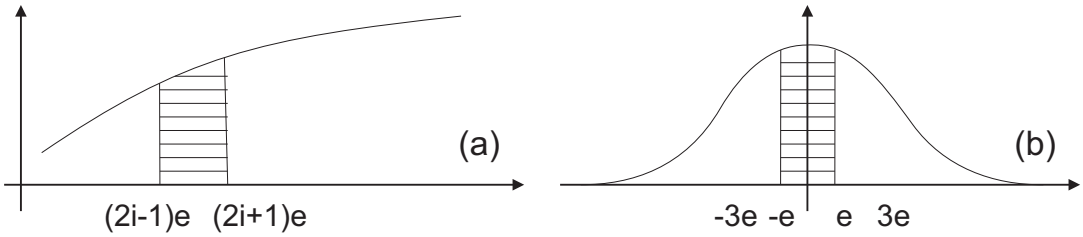


Figure 7.6: Estimating frequencies

possible for each node to estimate the frequency of a detail coefficient for $e \neq 0$ *by discretizing the continuous distribution according to* e*.*

We start by illustrating the idea of discretizing a continuous distribution. Suppose that we know the distribution $pdf(x)$ of a physical quantity, e.g., temperature. Figure 7.6(a) serves as an illustration. Formally, $pdf(x)$ is a probability distribution function. Discretizing the quantity corresponds to mapping all measurements of an interval $[(2i-1)\cdot e, (2i+1)\cdot e]$ to $2i\cdot e$ (or i after scaling). Thus, the relative frequency of i is the probability that a sensor reading is in this interval. Formally, let n be the number of sensor readings. The absolute frequency of i is:

$$freq(i) = n \cdot \int_{(2i-1)\cdot e}^{(2i+1)\cdot e} pdf(x)dx \tag{7.4}$$

To apply this idea of quantizing a continuous distribution, we need to define a "*continuous* distribution of the detail coefficients $pdf_0(x)$". $pdf_0(x)$ is the distribution of the detail coefficients of a slightly modified transform ($e = 0$): It omits the quantization as there is no error budget. In addition, as it is exact, there is no rounding in the wavelet transform:

$$\hat{a}_{1,i} = x_i - \text{minVal}$$
$$\hat{a}_{l,p} = \frac{\hat{a}_{l',p'} + \hat{a}_{l'',p''}}{2}$$
$$\hat{d}_{l,p} = \hat{a}_{l'',p''} - \hat{a}_{l',p'}$$

Given the distribution $pdf_0(x)$ of $\hat{d}_{l,p}$, we estimate the frequency of $d_{l,p}$ by integration as in Equation 7.4 (cf. Figure 7.6b).

7.3.3.2 Requirements

The previous approach is a valid estimation if the following Assumption 7.1 and Proposition 7.1 hold. We justify them in Section 7.3.3.4.

As the estimation is based on experience from previous executions, we assume that:

Assumption 7.1: *$pdf_0(x)$ is largely unchanged since its last update.*

As we base the estimation on a slightly modified transform, we have to prove the following proposition:

Proposition 7.1: The distribution of the unscaled detail coefficients $d_{l,p} \cdot 2e$ of our actual transform corresponds to the one of the continuous detail coefficients $\hat{d}_{l,p}$.

$d_{l,p} \cdot 2e$ are the quantized detail coefficients except for scaling to an integer. The scaling depends on e and therefore is not captured in the distribution $pdf_0(x)$, but in the integration – we integrate over $[(2d_{l,p} - 1) \cdot e, (2d_{l,p} + 1) \cdot e]$ (cf. Equation 7.4).

7.3.3.3 Maintaining Frequency Distributions

We have already described how the nodes estimate the frequency of $d_{l,p}$ from $pdf_0(x)$. We now discuss how to maintain $pdf_0(x)$. As a starting point for the maintenance, suppose that we have collected the continuous detail coefficients empirically. The idea is to obtain $pdf_0(x)$ by fitting a curve to their distribution.

Fitting a curve to a set of detail coefficients requires to specify a family of functions (polynomial, Gaussian, etc.) to be used. Bell curves often are well suited to describe the detail coefficients of a wavelet transform, in particular generalized Gaussian distributions, e.g., [CV00]. However, they are not a good choice in our case: They are computationally complex, and there is no antiderivative. The nodes would have to use numerical integration over a computationally complex function to arrive at an estimate. We found that the following simple bell curve describes the distribution well: $pdf_0(x) = \frac{1}{\pi} \frac{a_0}{1+(a_0 x)^2}$. The parameter a_0 describes the spread of the coefficients around 0 and has to be determined by curve fitting.

Note that, while $pdf_0(x) = \frac{1}{\pi} \frac{a_0}{1+(a_0 x)^2}$ is well-suited for our data, our technique is orthogonal to the curve function used.

Fitting $pdf_0(x)$ to the empirically determined detail coefficients is difficult due to outliers, i.e., isolated large coefficients. They yield a spread parameter a_0 which is too large. Thus, the tail of the distribution is too heavy, and the estimate of the number of frequent coefficients is too low. A simple extension would be to estimate the distribution on an α-quantile, i.e., to discard the outliers. But in this case the distribution is highly sensitive to the choice of α.

Instead of using $pdf_0(x)$, we estimate its cumulative distribution $cdf_0(x) = \frac{1}{2} + \frac{1}{\Pi} arctan(a_0 x)$. This has the following advantages over using $pdf_0(x)$:

(a) The nodes do not have to solve an integral. Given $cdf_0(x)$, the frequency of $a_{1,i}$ is simply $freq(a_{1,i}) = n[cdf_0([2a_{1,i} + 1] \cdot e) - cdf_0([2a_{1,i} - 1] \cdot e)]$.

(b) Estimating $cdf_0(x)$ can be done robust to outliers by using an α-quantile. This is now insensitive to the choice of α: By first accumulating the frequencies

and then discarding the tails, we keep the information that there were further coefficients. In a way, we only discard their "wrong" position.

(c) $cdf_0(x)$ can also be estimated from the quantized detail coefficients. This is key to react to changes in the distribution: We update $cdf_0(x)$ every time the attribute has been queried.

In summary, our approach works as follows:

1. **Initialization.** Prior to executing any query, the base station once collects the continuous detail coefficients for each attribute, determines the cumulative frequencies, and calculates a_0. In our implementation, we use a standard curve fitting algorithm (Levenberg-Marquardt, [Mar63]).

2. **Usage.** Given a query, for each of the attributes the corresponding a_0 and e are distributed in the network along with the query. Thus, every node arrives at the same estimates of the frequencies and thus at the same encoding.

3. **Update.** After execution, the base station re-estimates a_0. A moving average continuously adapts the distribution.

7.3.3.4 Justification

We now justify Assumption 7.1 and prove Proposition 7.1.

Assumption 7.1 states that the distribution of the detail coefficients $pdf_0()$ is largely unchanged since its last update. If it does not hold, the effectiveness of the coding degrades, the more the current distribution of the detail coefficients deviates from $pdf_0()$. That distribution may change due to changes in the data being transformed. More precisely, (major) changes in the *relative differences* of the data might affect $pdf_0()$. In contrast, spatial and temporal correlation of the sensor readings lead to stable differences. Changes in $pdf_0()$ can also stem from changes in the structure tree. But this is even less critical. First, recall that the underlying routing tree dictates its coarse structure. While routing trees change from time to time due to links going down, etc., such changes affect the tree only locally. Given that the differences of the approximations that a node receives are similar, the resulting structure tree is roughly the same as well.

Proposition 7.1 is that the distribution of the unscaled detail coefficients $d_{l,p} \cdot 2e$ corresponds to the one of the continuous coefficients $\hat{d}_{l,p}$. This will be shown as follows: Starting with a continuous distribution we examine the influence of (a) the quantization and (b) the rounding on the continuous coefficients. We show that:

1. The expected value of the coefficients remains unchanged.

2. The variance of the detail coefficients is small (usually much smaller than e).

(2) implies that the expected value represents the detail coefficients well. Given (1), Proposition 7.1 follows.

What is difficult in proving the statements is to model the influence of the quantization and the rounding appropriately. We will present this in detail. Given the idea, some proofs are obvious and will only be sketched. In order to calculate the expected value and variance of the detail coefficients we will introduce the random variables $A_{l,p}, D_{l,p}, R_{1,i}, T_{l,p}$. We start by defining a random variable $R_{1,i}$ modeling the quantization (cf. Equation 7.1). Without the scaling the coefficients are quantized to a multiple of $2e$. As each value in $[-e, e]$ is equally likely, $R_{1,i}$ is uniformly distributed:

Definition 7.1 ($R_{1,i}$)
$R_{1,i}$ is a random variable that is uniformly distributed in $[-e, e]$.

It follows that $R_{1,i}$ has mean $\mu = 0$ and variance $\sigma^2 = \frac{e^2}{3}$.

Definition 7.2 ($A_{1,i}$)
Let $A_{1,i} := \hat{a}_{1,i} + R_{1,i}$ be the random variable that models the quantized $\hat{a}_{1,i}$.

Lemma 7.1: $E(A_{1,i}) = \hat{a}_{1,i}$ and $V(A_{1,i}) = \frac{e^2}{3}$ where $E(\cdot)$ is the expected value and $V(\cdot)$ is the variance.

Proof: (Sketch) This can be seen by simply substituting the definition of $A_{1,i}$ and regarding Definition 7.1. ■

Definition 7.3 ($A_{l,p}$, $T_{l,p}$, $R_{l,p}$)
For $l' + l'' = l, l > 1$, let $A_{l,p} := \frac{A_{l',p'} + A_{l'',p''}}{2} + T_{l,p} \cdot e$. Here, $T_{l,p}$ is a random variable $\in \{-1, 0, 1\}$. Further, let $R_{l,p} := \hat{a}_{l,p} - A_{l,p}$ for $l > 1$.

$T_{l,p} \cdot e$ models the effect of rounding in the integer transform on the unscaled coefficients.

Lemma 7.2: $E(T_{l,p}) = 0$ and $V(T_{l,p}) = \frac{1}{2}$.

Proof: (Sketch) For the rounding, we distinguish between four cases (cf. Equation 7.2): If $a_{l',p'}$ and $a_{l'',p''}$ are both even or uneven, there is no rounding. That is, in two out of four cases, $T_{l,p} = 0$. Otherwise, we either round up ($T_{l,p} = 1$) if $a_{l',p'} > a_{l'',p''}$ or down ($T_{l,p} = -1$) if $a_{l',p'} < a_{l'',p''}$. Both cases are equally likely. The lemma then follows from the definition of $E(\cdot)$ and $V(\cdot)$. ■

Definition 7.4 ($D_{l,p}$)
Let $D_{l,p} := A_{l'',p''} - A_{l',p'}$.

Theorem 7.2: $E(A_{l,p}) = \hat{a}_{l,p}$. $E(D_{l,p}) = \hat{d}_{l,p}$.

Remark: As $D_{l,p}$ incorporates the quantization and the rounding, the second statement in the theorem is actually our first claim (1) for justifying Proposition 7.1.

Proof: The first statement can be proven by induction on l. For $l = 1$, the statement is true due to Lemma 7.1. Assuming $E(A_{l',p'}) = \hat{a}_{l',p'}$ and $E(A_{l'',p''}) = \hat{a}_{l'',p''}$ we get: $E(A_{l,p}) = E(\frac{A_{l',p'}+A_{l'',p''}}{2} + T_{l,p} \cdot e) = \frac{1}{2}E(A_{l',p'}) + \frac{1}{2}E(A_{l'',p''}) + E(T_{l,p}) \cdot e$ $= \frac{1}{2}\hat{a}_{l',p'} + \frac{1}{2}\hat{a}_{l'',p''} + 0e = \hat{a}_{l,p}$. The second statement can be obtained directly: $E(D_{l,p}) = E(A_{l'',p''} - A_{l',p'}) = E(A_{l'',p''}) - E(A_{l',p'}) = \hat{a}_{l'',p''} - \hat{a}_{l',p'} = \hat{d}_{l,p}$. ■

Lemma 7.3: $V(A_{l,p}) < e^2$.

Proof: (Induction) Note that $V(A_{l,p}) = V(R_{l,p})$ by definition. If $l = 1$, Lemma 7.3 is true due to Lemma 7.1.
If $V(A_{l',p'}) = V(R_{l',p'}) < e^2$, $V(A_{l'',p''}) = V(R_{l'',p''}) < e^2$ we get: $V(A_{l,p}) =$
$V(\frac{A_{l',p'}+A_{l'',p''}}{2} + T_{l,p} \cdot e)$
$= V(\frac{\hat{a}_{l',p'}+R_{l',p'}+\hat{a}_{l'',p''}+R_{l'',p''}}{2} + T_{l,p} \cdot e)$
$= \frac{1}{4}V(R_{l',p'}) + \frac{1}{4}V(R_{l'',p''}) + e^2 \cdot V(T_{l,p}) < \frac{1}{4}e^2 + \frac{1}{4}e^2 + e^2 \cdot \frac{1}{2} = e^2$. ■

The equality in the second to last line holds as R and T model roundings and can be assumed to be stochastically independent.

Remark: In this general case, e^2 is the smallest upper bound that can be proven for the variance. It is possible to show via induction that for a perfectly balanced structure tree that covers l readings, $V(A_{l,p}) = e^2 - \frac{1}{l}\frac{2}{3}e^2$ which quickly converges to e^2.

Theorem 7.3: $V(D_{l,p}) < 2e^2$.

Proof: $V(D_{l,p}) = V(A_{l'',p''} - A_{l',p'}) = V(\hat{a}_{l'',p''} + R_{l'',p''} - \hat{a}_{l',p'} - R_{l',p'}) =$
$V(R_{l'',p''}) + V(R_{l',p'}) < e^2 + e^2 = 2e^2$. ■

Theorem 7.3 is the second statement (2) for justifying Proposition 7.1 and thus completes our discussion.

7.4 Sending Approximations

We conclude the description of SNAP with an optimization regarding sending of approximation coefficients. It is based on the observation that the quantization not only leads to a small set of detail coefficients – this is exploited by the entropy coding. It also restricts the set of approximation coefficients to a small number. The following example motivates our optimization:

Example 7.3: Consider a system that collects temperature values with $e = 0.1$°C. Suppose that the measurements within the network range between 18.5274°C and 23.3883°C. Then the approximation coefficients must also be within this range. Most notably, there are only $\lceil \frac{23.3883-18.5274}{2 \cdot 0.1} \rceil = 25$ possible values. In this case, sending an

approximation should not require more than $\lceil log_2(25) \rceil = 5$ bits. ■

Underlying Idea. Intuitively, the approximation coefficients will mostly vary in a small range around the overall average $avg_{\texttt{net}}$. Assume for now that $avg_{\texttt{net}}$ is known. If Node n_i computes an approximation coefficient $a_{l,p}$ and then computes its difference from the quantized overall average $\bar{a}_{l,p} = Int(\frac{avg_{\texttt{net}} - minVal}{2e}) - a_{l,p}$, this difference $\bar{a}_{l,p}$ will be a small number. Thus its binary representation will be of one of the following forms: $0...01X_0...X_r$, $1...10X_0...X_r$, or $0...0$ for $a_{l,p} <, >$ or $= avg_{\texttt{net}}$ (where $X_i = \{0|1\}$). If we know the remainder ($01X_0...X_r$, $10X_0...X_r$, or $0...0$), we can restore $\bar{a}_{l,p}$ by prefixing 0's or 1's depending on the first bit. That is, there is no information in the beginning of $\bar{a}_{l,p}$. *The idea is to only send this remainder.* Technically, this requires knowing the length of the remainder, which depends on the quantization.

Approach. In general, our optimization works as follows: Assume that each node knows $avg_{\texttt{net}}$ and the range r. The latter is defined as the difference of the max and the min value measured. To send the information contained in $a_{l,p}$, n_i computes $\bar{a}_{l,p}$. It then truncates $\bar{a}_{l,p}$ to the last $\lceil log_2(\lceil \frac{r}{2e} \rceil) \rceil$ bits. The parent of n_i reconstructs $a_{l,p}$ by filling up the remainder and adding it to $avg_{\texttt{net}}$.

Note that $\lceil log_2(\lceil \frac{r}{2e} \rceil) \rceil$ bits are only sufficient if both values $avg_{\texttt{net}}$ and r are accurate. Therefore, the encoding actually used by SNAP is as follows: If $\lceil log_2(\lceil \frac{r}{2e} \rceil) \rceil$ bits are sufficient, a node starts the encoding with a '0'-bit followed by the remainder. 'sufficient' means that at least $01X_0...X_r$ or $10X_0...X_r$ can be captured to achieve uniqueness. If this condition is not fulfilled, the node starts the encoding with '10' followed by $\lceil log_2(\lceil \frac{r}{2e} \rceil) \rceil + 1$ bits. Note that this effectively doubles the range. In case of any errors, the encoding can start with '11' followed by the original $a_{l,p}$.

Finally, we say how the nodes get to know $avg_{\texttt{net}}$ and r. We do not require each node to know the max and min value but only the range which is much more stable in time. The base station simply maintains these parameters as part of its catalog. In detail, the base station keeps moving averages for $avg_{\texttt{net}}$ as well as for the min and max value per attribute. $avg_{\texttt{net}}$ or r are distributed along with the query whenever its deviation from the value currently used exceeds a threshold. Determining these thresholds is straightforward. The question is when the length of the remainder changes. If $avg_{\texttt{net}}$ is correct, the length changes if the range doubles or halves. If r is valid, we need to update $avg_{\texttt{net}}$ if it deviates by more than r. Overall, the combination of both parameters must not deviate by more than r from the values currently used. This budget should not be fully exploited since the encoding gets worse if the deviations come close to these upper bounds. In our implementation, we update r if the range halves or becomes larger by $\frac{1}{3}$. We update $avg_{\texttt{net}}$ whenever it changes by more than half of the range currently used.

7.5 Evaluation

In this section, we demonstrate the performance of SNAP based on real-world and synthetic sensor data. As in Sections 4.4 and 5.7, we use the ns-2 simulator for our study. We will show that SNAP efficiently consolidates sensor relations even for tight error bounds.

7.5.1 Experimental Setup

Query Workload. Our evaluation is based on queries that match the pattern from our problem specification (cf. Section 6.1). There are two degrees of freedom in this pattern: (1) the number of attributes in the `SELECT`-clause and (2) the error budget for each of them. Both will serve as parameters in our experiments.

Comparison Schemes. In line with the discussion in Section 6.3, we compare SNAP to the following schemes:

(1) Wavelet-based approaches. We want to highlight the benefit of our integration approach of building upon an unmodified routing tree and using an irregular transform. Two alternatives have been proposed: Using zero padding to account for the irregular routing tree [HHMS03] and subdividing the network into clusters, each of which applies a wavelet transform [DBcF07]. However, these approaches provide no error guarantees and are not designed to cope with multiple attributes. We extended them based on state-of-the-art approaches for dealing with tuples [DGR07] and for thresholding with error guarantees [KM05]. In our experiments, we do not account for any overheads due to these extensions like sending coordinates of the coefficients or forwarding the remaining error budgets. Thus, we heavily underestimate the costs of the related approaches.

(2) Model-based approaches. We compare SNAP to approaches based on multivariate Gaussians. While these sophisticated models have high training costs, they are best when it comes to normal operation. (We do not account for training costs in our experiments.) [CDHH06] has shown that Ken is superior to Caching and Kalman Filters. Among Ken and BBQ [DGM+04], we decided to compare SNAP to Ken since Ken provides the same error guarantees as SNAP. Note that the problem of model-based approaches are false predictions for tight error bounds, which is the same for BBQ and Ken.

(3) Tree-based Data Collection (TDC). We compare SNAP to simple tree-based data collection. As we will show, for the related approaches, tree-based data collection is among the best for tight error bounds.

Data Sets and Setting. As with CJF, our goal is to evaluate SNAP on real-world data sets. Again, we used the LUCE data [Senc] which is attractive for its large

number of attributes. The data is from a network consisting of 81 nodes when the data was acquired (January 2007). For the experiments based on real data, we set the size of the network and the relative positions of the nodes to reflect the setting during the original data collection.

We also use some synthetically generated data sets. This is to evaluate SNAP on large networks and to study extreme data sets. In particular, the latter usage lets us assess the influence of the data set on the performance of SNAP. For the experiments based on synthetic data, we distribute the nodes randomly.

Metrics. In contrast to the experiments in Sections 4.4 and 5.7, we use the number-of-transmissions (networking packets) and the number-of-bytes metric for this study (see Section 4.4 for a discussion). Most of the graphs in this section are based on the number-of-bytes metric: We have conducted our experiments on real data where the size of the network is 81 nodes. For such sizes, SNAP builds synopses that fit into a single networking packet. While this is nice, the problem is that varying parameters has no effect on the number of packets sent and is pointless. In contrast, the number-of-bytes metric is well suited to show the data reduction that SNAP achieves. This carries over to the number-of-transmissions metric, as we will show as well.

Default Setting. In each experiment we vary one parameter. If a parameter is not varied we use the following default value: Among the attributes of the data set, we query the node ID (no error), ambient temperature ($e = 0.1°$C), surface temperature ($e = 0.1°$C) and relative humidity ($e = 0.5\%$). These attributes were available at all nodes in the data set. Note that querying for multiple attributes without involving the IDs is rare in practice.

Varying Error Bounds. As our queries cover multiple attributes, if we want to vary the maximum errors, we have to consider all of the attributes. We simply multiply their default settings with a factor, ranging from 0.1 to 5. For instance, for the ambient temperature (default setting $e = 0.1°$C), the errors range from 0.01°C to 0.5°C.

7.5.2 Comparative Experiments

Wavelet-Based Approaches. In a first set of experiments we examine building upon an irregular transform. We compare SNAP to zero padding and cluster-based local transforms. The objective is to give evidence for the following claims: (1) Zero padding results in *many, large* detail coefficients. (2) A cluster-based local transform also suffers from inherent problems and thus cannot result in small synopses: Only the data within each cluster is decorrelated – redundancies between clusters are not removed. In addition, as the number of nodes within a cluster is rarely a power of

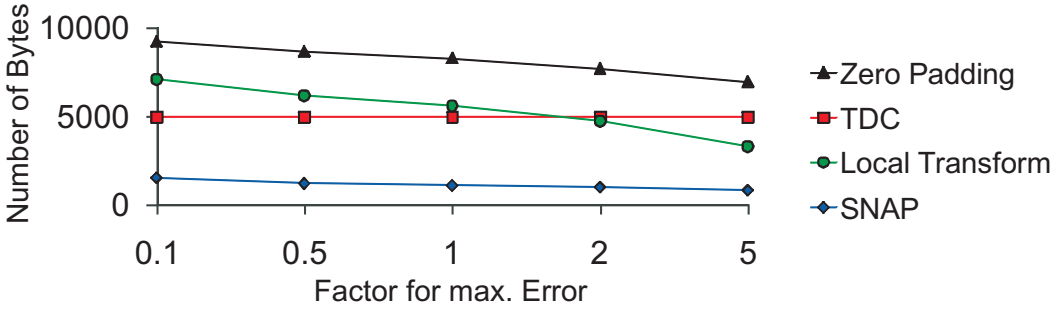


Figure 7.7: Comparison of integration approaches (real data)

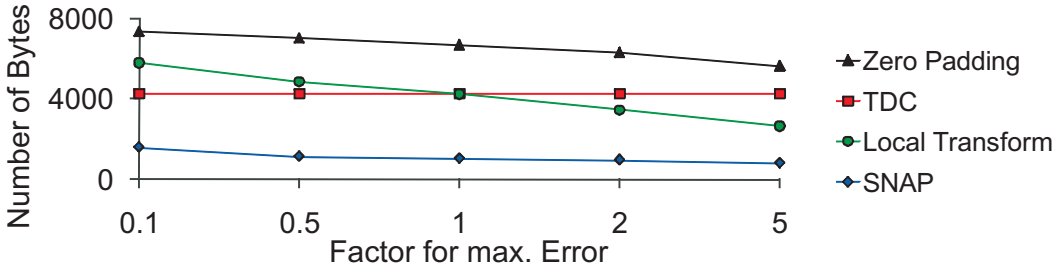


Figure 7.8: Comparison of integration approaches (synth. data)

two, we again need zero padding to perform local wavelet transforms, though less than in the tree-based approach.

We compare the approaches for different maximum errors. We expect all of them to perform better for larger error bounds. Next to the two wavelet-based approaches and SNAP, we also measure the performance of tree-based data collection (TDC), which is independent of the error. The results are shown in Figure 7.7. They confirm our expectation. The related approaches perform worse than a simple tree-based data collection. Recall that the costs of the wavelet-based approaches are underestimated substantially. In contrast, SNAP outperforms tree-based data collection by a factor of about five.

We repeated these experiments on our synthetic data, see Figure 7.8. They are in line with those on real data. All approaches including tree-based data collection are slightly better due to differences in the routing tree. For the synthetic data we placed the nodes randomly.

Model-Based Approaches. [DGM+04] has pointed out that model-based approaches degrade with tighter error bounds. We want to show that the performance of SNAP does not degrade as much. Note that model-based approaches need updates of the model from time to time, as otherwise all estimations would be out of bounds after a while. Thus, they only make sense if the network is queried continuously.

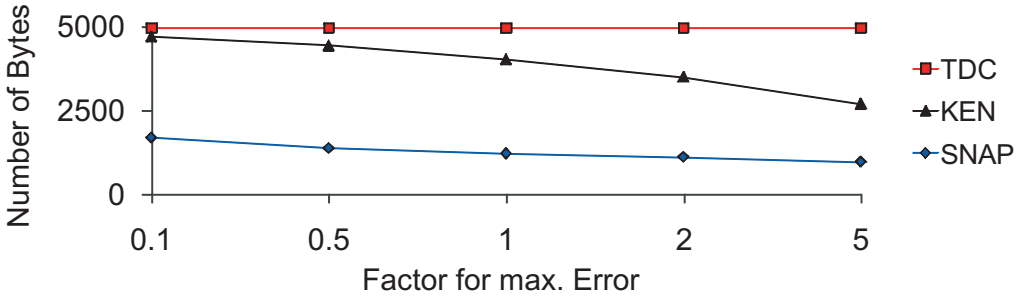


Figure 7.9: Comparison to Ken (real data)

We measure the performance of Ken by using a continuous query and averaging the costs. We query 40 consecutive snapshots from the real world data set after an extensive training phase. SNAP is executed on the same 40 snapshots, and we also average the costs. The results are shown in Figure 7.9. Note that our results for Ken are in line with those presented in [CDHH06]. Most notably, if the error bounds become less than our default setting, Ken performs similar to tree-based data collection. In contrast, even for bounds an order of magnitude tighter (factor $\frac{1}{10}$), SNAP still achieves a reduction of the communication costs by factor three.

7.5.3 Analysis of SNAP

We now study the performance of SNAP. As the performance of tree-based data collection is close to Ken for our default setting, and tree-based data collection is relatively easy to handle, we use tree-based data collection as a point of comparison.

Scalability. We are interested in the performance of SNAP for larger networks. To determine the influence of the network size we vary the number of nodes from 500 to 2000. At the same time we vary the area of the network to keep the node density constant. The experiments are conducted on synthetic data as we do not have data sets for larger networks. Intuitively, the relative savings of SNAP over tree-based data collection should be independent of the network size. This is because our synopsis achieves a constant compaction factor of the data. The results in Figure 7.10(a) confirm this expectation. To confirm that the data reduction carries over to the number of transmissions, Figure 7.10(b) graphs the results for this metric. Interestingly, even for 2000 nodes, SNAP sends only about 2500 packets. That is, SNAP reduces the data such that most of the nodes send one packet. This is the optimum for the number-of-transmissions metric, indicated by the black line.

Number of Attributes. In this set of experiments we determine the influence of the number of attributes in the query on SNAP. If the query exceeds three attributes

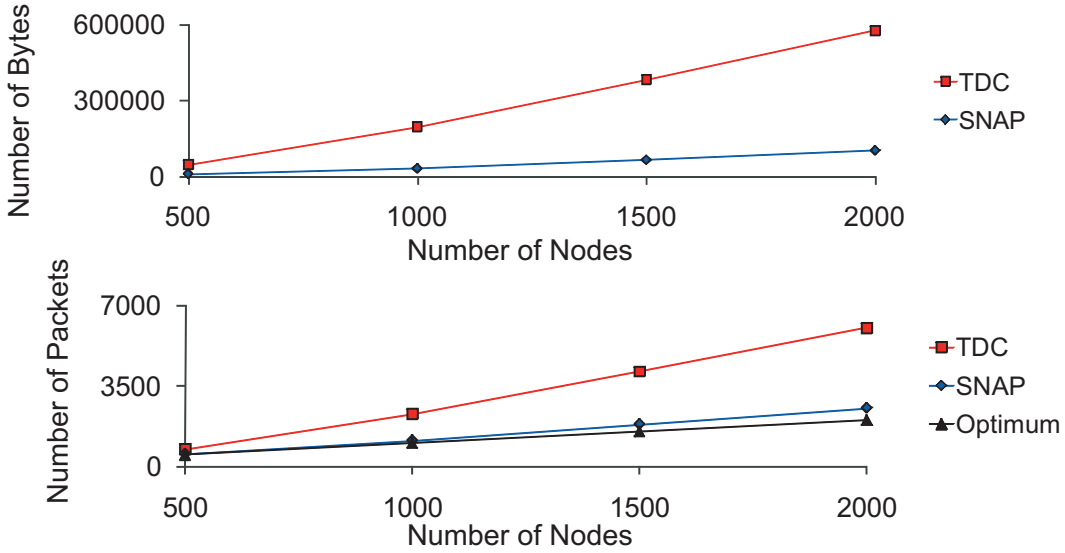


Figure 7.10: Scalability of SNAP (synthetic data)

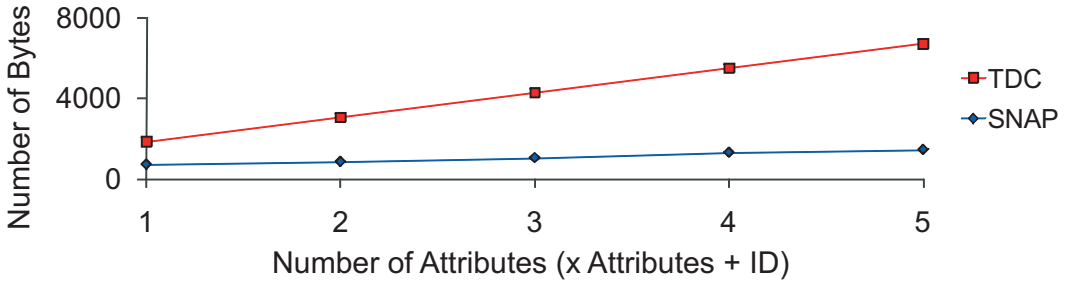


Figure 7.11: Influence of the number of attributes (synth. data)

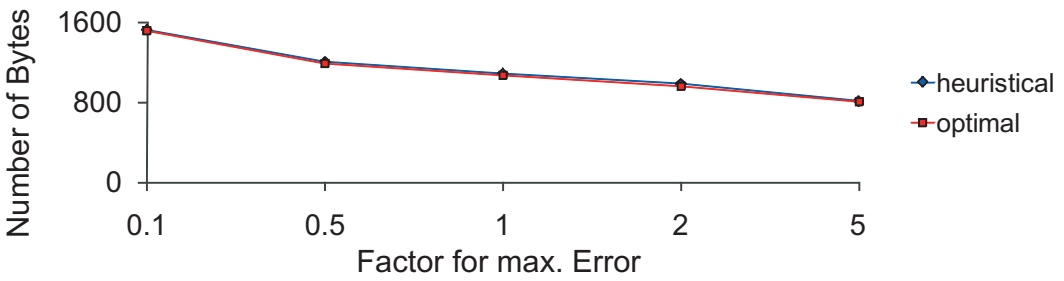


Figure 7.12: SNAP: heuristical vs. optimal integration (real data)

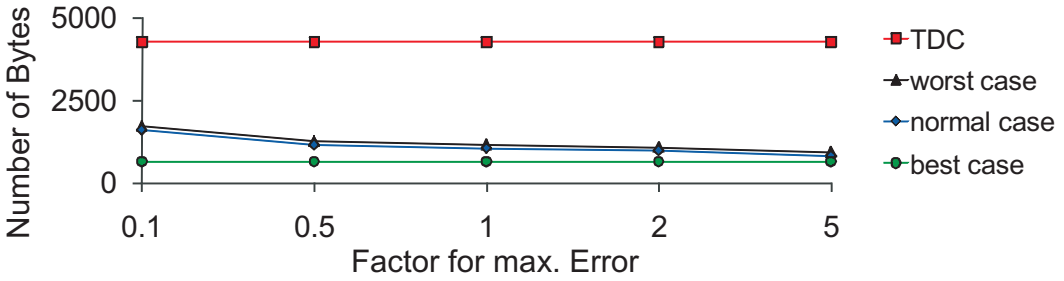


Figure 7.13: SNAP: performance on extreme data (synth. data)

(plus ID), we additionally query for solar radiation ($e = 0.5\frac{W}{m^2}$) and wind speed ($e = 0.1\frac{m}{s}$). We conduct the corresponding measurements on synthetic data because not all of the nodes in our data set have available the complete set of sensors. Intuitively, the number of attributes should not have a major influence on the relative performance of SNAP over tree-based data collection. On the one hand, more attributes better amortize the overhead in the detail coefficients for coding the tree structure (two STS bits). On the other hand, this might require tradeoffs in building the structure tree – different attributes might prefer different tree structures. Figure 7.11 confirms that both influences are minor. The data reduction becomes slightly better with the number of attributes as the ID, which is included in all queries, is difficult to compress.

Heuristical vs. Optimal Structure Tree. The following experiment highlights the appropriateness of our heuristic to devise a structure tree. We compare SNAP (which incorporates the heuristic) to a modified version of SNAP that uses the optimal algorithm. Both approaches integrate the structure tree into the routing tree. Thus, the rough structure is the same. Figure 7.12 graphs the results. In all of our experiments, the heuristic performed within 5% of the optimum.

Extreme Data Sets. Finally, we are interested in how much the performance of SNAP depends on spatial correlation in the data. Therefore, we evaluate SNAP on two extreme data sets. The first one simulates perfect correlation, i.e., each node measures the same value on corresponding attributes. This is supposed to be the best case for SNAP. The second data set simulates uncorrelated data as a worst case. We set the ranges of the attributes as observed in our real data and let each node observe a random value from the ranges for each of the attributes. The results are shown in Figure 7.13. As expected, the performance of SNAP degrades with less correlation. However, the sensitivity is limited. SNAP performs well even on the uncorrelated data.

7.6 Summary

In this chapter, we have presented SNAP, our solution to evaluating queries with a low selectivity. To achieve efficiency even for tight error bounds, we based SNAP on wavelet synopses. SNAP constructs a synopsis during data collection incrementally. As a core contribution, we have shown how to distribute the wavelet transform and the thresholding step. We have done so by integrating the transform into an unmodified routing tree. To obtain a synopsis we have explored a design that encodes coefficients compactly instead of discarding them. To distribute this mechanism, we have proposed an approach to estimate the frequencies of coefficients. SNAP is the first distributed solution to the thresholding problem with error guarantees. It achieves a data reduction by more than a factor of five and improves the accuracy for which data can be efficiently consolidated by more than an order of magnitude.

8 Conclusions and Future Directions

This dissertation extends the state-of-the-art in query processing in sensor networks. We presented two methods for processing the join operator. Along with selections and projections, the join is fundamental for focusing on relevant portions of the sensor readings. To obtain efficiency, the idea is to exploit selectivity. Moreover, we presented an approximate query processing scheme which allows to efficiently consolidate entire sensor relations at the base station. In particular, this allows for an efficient processing of non-selective queries.

In principle, approximate query processing can be applied to selective as well as non-selective queries, i.e., it is orthogonal to the selectivity of the query: It is possible to approximately collect sensor data after discarding irrelevant portions, to further reduce the data volume. However, we motivated approximate query processing with non-selective queries for the following reason: For selective queries, the data volume is already small, leaving only little room for an additional data reduction. Given that an approximation sacrifices precision of the query result, the additional savings in case of selective queries is small and might not weigh off the loss of accuracy.

In the following, we review the contributions of this dissertation in more detail before discussing some interesting opportunities of future work.

8.1 Summary

To simplify data acquisition from sensor networks, query interfaces have proven to be attractive. They hide technical details from the application. With respect to an efficient query processing, early prototypes of sensor network query processors have introduced the idea of pushing operators into the network to reduce the data that is involved in the query. That is, the idea is to exploit the selectivity in the query for an efficient processing. The prototypes explored this idea for selection and projection operators as well as for aggregations. In particular, for selections and projections, a node can decide locally if its tuple contributes to the result.

In contrast, processing join queries has been an open problem. The join relates tuples from arbitrary nodes to each other. A node cannot decide locally if its tuple joins. In particular, the tuples are distributed throughout the network, and matching tuples is costly in terms of communication.

Prior join approaches avoid an extensive matching by specializing to specific types of queries or placements of the input tuples. This comes at the expense of applicability. Our goal was the design of join methods that avoid such specializations.

To start addressing this problem, we introduced IDEAL, a concept that lower bounds the communication costs of join processing. According to IDEAL, it is optimal to discard non-joining tuples at the sources and to send the joining tuples to the base station. These insights guided the design of our join methods: They obey this framework.

We presented SENS-Join, the first general-purpose join method that does not rely on restrictive requirements: It can efficiently handle any number of join conditions and arbitrary distributions of the nodes involved. SENS-Join solves the problem of providing the knowledge, which of the tuples join, by means of a semi-join based precomputation. In particular, we showed how to design a precomputation that is efficient – this is critical since the costs due to a precomputation could easily exhaust the potential savings of discarding non-joining tuples.

A precomputation is well-suited for one-time queries. In contrast, re-computing the set of joining tuples for a continuous execution incurs unnecessary costs, given that sensor data tends to be temporally correlated. This is a particular problem as continuous queries prevail in many monitoring and surveillance applications. To this end, we have presented an approach that maintains join filters, CJF. Most notably, CJF performs close to IDEAL by avoiding the constant costs of the precomputation.

Finally, we studied the evaluation of queries with a low selectivity. Obviously, the idea of obtaining efficiency by discarding irrelevant data early in the processing does not apply to these. However, there are a number of sensor network deployments that simply collect data without prior selection. For instance, recall the scientific deployments introduced in Chapter 1. Thus, a mechanism for efficiently consolidating entire sensor relations is highly desirable.

Prior work has addressed non-selective queries by approximating results based on models. The solutions work well if the accuracy requirements are loose. For more accuracy, communication costs increase quickly.

Our solution, SNAP, is based on wavelet synopses. They are known for their good data reduction capabilities at the costs of introducing only small errors. To allow for an efficient data collection, SNAP constructs the synopsis during the collection incrementally. SNAP is the first distributed wavelet compaction in sensor networks that provides error guarantees.

At a technical level, the following aspects of this work constitute our main innovations:

- SENS-Join exemplifies that a precomputation can be done at little costs, even if it covers the entire sensor network. As a key to obtaining efficiency, we showed how to include most of the nodes only once in the processing. This is in contrast to a naive implementation of a precomputation.

- We designed a quadtree-based data representation which eliminates the redundancy in a set of join-attribute tuples. This mechanism outperforms compression algorithms in our scenario: The latter do not allow for good compression ratios if the data volume is small. However, this is the rule for a tree-based data collection except for nodes near the root of the collection tree.

- CJF continuously computes optimal filters. To do so, we showed how to map join queries to an optimization problem under constraints. We also presented an algorithm that solves this problem. It handles the constraints as well as the non-convexity of the cost function. Our solutions aggressively exploit the continuous nature of queries – on the hand by using previous filter settings as a starting point of the optimization and on the other hand by introducing a stochastic algorithm that converges to the optimal solution in time.

- We update filters only if the improvement at least amortizes the costs of updating. In particular, we showed how to do so in the face of dependencies among filters.

- SNAP integrates the data flow of a wavelet transform into the routing structure of the network. SNAP is first to explore this idea. The integration allows for a distributed transform that neither sacrifices shortest paths nor adds to the data.

- To obtain a synopsis we have explored a design that encodes coefficients compactly instead of discarding them. In particular, we showed how to accurately estimate the frequencies of detail coefficients, which are required for a distributed compaction. This design resulted in the first distributed solution to the thresholding problem with error guarantees.

Finally, we extensively evaluated all of our algorithms. SENS-Join reduces the communication of the most loaded nodes by more than an order of magnitude in some situations. The costs of CJF are close to the lower bound of join processing. SNAP achieves a data reduction by more than a factor of five while improving the accuracy for which data can be efficiently consolidated by more than an order of magnitude. Such improvements prolong the lifetime of the network significantly.

8.2 Future Work

There is a number of problems with respect to query processing in sensor networks that we did not address to limit the scope of this dissertation. We conclude this dissertation with a section that highlights the most interesting of them.

Selectivity Estimation. In our work, we explicitly distinguish between 'selective' and 'non-selective' queries. In particular, it seems to be common sense in the literature that for the latter, approximate processing schemes are appropriate. As we also pointed out, if queries are selective, conventional, i.e., exact query processing seems sufficient.

The important point about this is that in order to decide which kind of processing to use, we need to know the selectivity. As a further example, recall that the savings of SENS-Join over the external join depend on the selectivity of the operator. That is, if more than 90% of the tuples join, the external join is preferable.

For continuous queries, one might argue that selectivity estimation is less important. For a long running query, it seems appropriate to execute it once and to simply observe the selectivity. This is different for one-time queries.

It is difficult to use traditional selectivity estimation techniques (such as [BCG01, RD08]) in our context. On the one hand, sensor readings constantly change. Prior work proposes to use query feedback to constantly adapt to changes in the data. On the other hand, traditional techniques can be inaccurate for join selectivities, in particular for similarity-joins over multiple join attributes.

As an idea to address this problem, it might be possible to use a model-based approach: Since the size of sensor relations is several orders of magnitude smaller than relations in commercial applications, it should be possible to set up an approximation of the current sensor relation and simply execute the query on it to obtain the selectivity. Modeling sensor readings has been used in [DGM+04] to approximately answer queries. While this approach is much too costly to estimate selectivities, the accuracy requirements are much weaker in this case. It is an open problem whether this allows to use a model-based approach. Beyond that, this requires to figure out how to update the models, especially if there are areas in the data that are rarely queried.

Thus, selectivity estimation in sensor networks is different from traditional selectivity estimation. We need to solve this problem to integrate different processing schemes (ours as well as schemes from the literature) into a single query processor.

Nested Queries. In our problem statements, queries are not nested. To our knowledge, nested queries have not yet been considered in the literature on query processing in sensor networks. However, there are very intuitive examples of interests that an application can have and which require a nested query. For instance, assume that the application is interested in measurements that are very different from the average. Here, computing the average is an inner query.

The problem with considering nested queries is that the number of possibilities of formulating queries is daunting. For instance, the depth of the nesting can be arbitrary. However, for our example query, a precomputation-based approach similar to SENS-Join could be promising. In particular, mechanisms such as Treecut immediately apply. Beyond that, it seems possible to approximate the result of the inner

query to obtain an efficient precomputation as with SENS-Join. Finally, for continuous queries, the idea of avoiding to re-execute the evaluation of the inner query by exploiting temporal correlations in the data seems possible. At the same time, if the result of the inner query is not only used to select data, as in our example, but is part of the result, things get a lot more involved. In summary, evaluating nested queries seems to be an interesting direction for future research.

Multiple Queries. The algorithms presented in this dissertation address an isolated execution of queries. If there are multiple queries in parallel that acquire data from the network, they might overlap in the data they collect. This allows for further savings, if the shared data is collected only once.

There already exists work on multi-query optimization in sensor networks. Most notably, [TYD+05] studies sharing for region based aggregation queries. [XLTZ07] supports other types of aggregation queries, e.g., value-based aggregation queries, and data acquisition queries. It might be interesting to extend this work to join queries. In particular, recall that for an efficient join processing, we have to find out which tuples join. We proposed to do so by means of a precomputation (SENS-Join) or based on models of sensor readings (CJF). With respect to the precomputation, an obvious idea is to collect the join-attribute values of all queries with a single collection. Even if the join attributes are not identical but only partly overlap, the costs should be close to only collecting the attributes of a single query due to the compression. With respect to maintaining models, it is possible to share the models for all queries.

Note that it is unclear whether having a large number of parallel queries is realistic. All sensor networks we are aware of are dedicated to a specific task and the number of parallel queries tends to be limited. However, even for just two queries, the ideas that we just sketched should allow for substantial savings in the communication. In contrast, a real problem is for continuous queries that they need to be synchronized with respect to their execution periodicity to enable sharing, at least if the sharing concerns the data collection. This problem would not affect a sharing in the models.

Hierarchical Network Architectures. We based our work on a flat network architecture. It is an interesting research question how a different architecture affects the algorithms. For instance, [GJP+06] points out the advantages of a hierarchical network architecture which includes more powerful nodes. In a way, such a network contains our architecture as a subnet and consists of multiple of our networks at the same time. This is advantageous when it comes to scaling to very large networks (on the order of 10000 nodes) as then, the load of a simple data collection becomes prohibitive towards the base station.

What is different with respect to query processing is that queries span multiple of the networks considered in this dissertation. In particular, having a number of pow-

erful nodes in the network opens up new possibilities with respect to the processing locations of operators. This might also enable to optimize for response time besides energy costs by distributing the processing among powerful nodes.

Finally, a hierarchical architecture can be applied to reduce the number of wireless hops for each node to reach a base station. If the number of wireless hops is small, the benefit of a filtering scheme like CJF is probably much higher than in our current architecture. This is because currently, most of the communication costs are due to forwarding. Also, maintaining models is much cheaper if each node can reach the base station within a few hops.

In summary, the network architecture affects the performance of algorithms. Considering hierarchical networks thus opens up new and interesting problems. In particular, if the goal is to scale the network beyond several thousand nodes, a hierarchical architecture is probably the most realistic one.

Finding Temporal Patterns. In line with the literature, we introduced the semantics of queries as 'snapshot'-queries (Chapter 2), i.e., the result is a subset of the current sensor readings. While we support monitoring applications by means of continuous queries, these queries also obey the snapshot semantics. They are executed periodically on the most recent snapshot. In particular, this semantics excludes searches for interesting patterns in time. For instance, we might be interested in sensor readings that behave contrary to the common trend. Such interests require the ability to relate measurements from different points in time to each other.

Given that the sensor relation contains an attribute 'time stamp' that records when the measurements have been taken, it is possible to compare readings from different points in time by means of join queries. [YLOT07] studies the processing in this case. However, they imposed severe restrictions on the queries to enable an efficient processing. In particular, one of the relations is required to consist of a few tuples only. A generalization is an open problem.

In addition, it is unclear whether using such join queries to express interests that span multiple points in time is appropriate. For instance, the interest that we formulated informally cannot easily be expressed by a join query.

What makes the search for patterns in time a worthwhile research question is that an *interesting* pattern is almost by definition selective, i.e., it is rare in the data and might allow for an efficient processing.

Small Sampling Periods. Finally, there is an interesting problem which is due to bandwidth limitations. For any continuous query, there is a maximum rate at which the query can be executed before the capacity of the wireless channel becomes exhausted. For tree-based data collection schemes, the nodes close to the root are the first that run into problems. In particular, these nodes have to drop packets in case of on overload. However, these packets might already have been forwarded a couple of

hops. The corresponding energy is wasted. Madden et al. [MFHH03] addressed this problem by automatically throttling the rate at which a query is executed. Consider the following event detection scenario in which such a mechanism is appropriate: For instance, the query might search for temperature spikes at certain nodes. Despite very high sampling rates, the result is almost always empty. And if the event occurs, it might be fine to raise an alarm, i.e., only the first result is actually relevant.

In contrast, for join queries, this throttling might not be appropriate anymore. Again, for event detection applications, the result is almost always empty. However, the detection cannot be done by a single node but requires the nodes to interact. In this case, the throttling actually reduces the rate at which the query is executed. This might significantly prolong the time until we detect the event and is unacceptable in many industrial scenarios. There, relatively high monitoring rates (e.g., 100's of Hertz) are required [AML05]. It is an open problem how to support a quick detection of events that require observations at multiple nodes. A possible approach might be to decouple the observation of a time series at a single node from the frequency of interaction of nodes to join their time series. It is an interesting question to see what kind of quality guarantees such an approach can provide.

With these interesting open problems in mind, we feel that declarative querying of sensor networks remains an exciting area for doing database-related research in the future.

CHAPTER 8. CONCLUSIONS AND FUTURE DIRECTIONS

Appendix

A Optimal Locations for Join Processing

This chapter provides theoretical insights in how to efficiently process the join in sensor networks. In particular, we study at which node(s) the join should be processed.

One alternative for processing the join is centralized, i.e., at a single location. The basic variant of a centralized join is the external join (cf. Definition 3.1) which performs all the computations at the base station. In Section A.2, we analyze in which cases a centralized processing *inside the network* is more efficient that the external join. In particular, we substantiate our claim from Section 3.2: Centralized in-network approaches cannot serve as a general-purpose join method. They are inferior to the external join except for very specific settings.

In Section A.3, we then establish the following result: IDEAL, as defined in Section 3.1.3, lower bounds the communication costs of join processing.

Up front, we present some background on our analysis.

A.1 Preliminaries

We start with introducing the terminology that we will use in our study in Section A.1.1. We then specify the models that underlie our analysis. In particular, we present our network model in Section A.1.2 and our model for the communication costs in Section A.1.3.

A.1.1 Terminology

In our study, we compare the costs of different join approaches – a fundamental notion that we will use to distinguish different approaches is a "join strategy":

Definition A.1 (Strategy)
A strategy is

(a) a set of locations where tuples are joined and

(b) the routes along which tuples are sent.

Before discussing alternatives for join strategies, we introduce two basic requirements that have to hold for any efficient strategy:

Requirement A.3 (Correctness of the Result)
The join result has to be correct.

Correctness is a particular concern when it comes to distributing the computation of the join. We have to ensure that every pair of tuples that join meets at (at least) one location.

Requirement A.4 (Minimal routes)
An optimal strategy has to minimize the routes along which the tuples are sent.

This requirement helps to restrict the set of candidates for optimal strategies. A strategy that sends tuples on an unnecessarily long route has a superior strategy that sends on a shorter route.

Having described the basic requirements, we now introduce two dimensions along which we classify strategies: (1) *number of processing sites* and (2) *locations of processing*.

Dimension 1: Number of Processing Sites. Along the *number of sites* dimension, we distinguish "centralized" from "distributed" strategies:

Definition A.2 (Centralized Strategy)
A centralized strategy is a strategy that performs the join at a single node.

Our discussion in Section A.2 focuses on centralized strategies. As a generalization, we define the notion of a distributed strategy:

Definition A.3 (Distributed Strategy)
A distributed strategy may consist of multiple (≥ 1) join locations.

Most notably, according to the preceding definition, each centralized strategy is a distributed one. The rationale behind this definition is that a distributed strategy is the most general strategy that is possible: It imposes *no restrictions* on the number of processing sites. Thus, to show in Section A.3 that IDEAL lower bounds the costs of join processing, we will have to compare it to distributed join strategies.

The following remark concludes the discussion of the number of sites dimension: According to our definition, centralized strategies perform the join at one location. In practice, resource limitations might require several neighboring nodes to compute a join [CN07]. However, we want to provide bounds on the communication costs of different strategies. Using a single node will serve as a lower bound for the communication costs of centralized strategies, as we will discuss in Section A.2.

Dimension 2: Processing Locations. We distinguish between the processing locations *root* and *in-network*. 'root' restricts the location(s) of the processing sites to the base station. As with the number of sites dimension, the generalization ('in-network') imposes *no restrictions*. That is, an in-network strategy can perform the processing on any node(s), including the base station.

Given this nomenclature, a '(centralized) root strategy' is the external join. This strategy is important in our analysis as it will serve as a reference point in Section A.2. We therefore define:

Definition A.4 (Root Strategy)
The root strategy is the external join.

Measure. As a final notion, we introduce the measure that we use in our study. Our goal is to compare strategies and to find out which one is the most efficient. In order to quantify efficiency, we define the relative measure *gain*. It compares the costs of a strategy ($cost_{strat}$) to the costs of a reference strategy ($cost_{ref}$). We use a relative measure in order to abstract from communication hardware.

Definition A.5 (Gain)

$$\text{gain}_{ref} = 1 - \frac{cost_{strat}}{cost_{ref}}$$

Modeling the costs of the strategies is different for the centralized and the distributed case and will be discussed in the corresponding sections. In our analysis, we will use two strategies as reference points: In Section A.2, we seek to improve upon the external join. Thus, we will use the root strategy as a reference point. There, gain refers to gain_{root}. In Section A.3, we investigate if we can improve upon IDEAL. There, gain refers to gain_{IDEAL}. To simplify notation, we will drop the subscript if it is clear from the context.

According to the preceding definitions, the root strategy is an instance of a centralized as well as a decentralized strategy. Thus, the following holds:

Corollary A.1: *For optimal centralized and decentralized strategies,* $cost_{strat} \leq cost_{root}$. *Therefore,* $gain_{root} \in [0, 1]$. □

A.1.2 Network Model

In the following, we present and justify our network model. We refer to the two relations to be joined as A and B. The relations contain those nodes that result from applying the selection predicates (`WHERE`-clause) to a sensor relation. Figure A.1

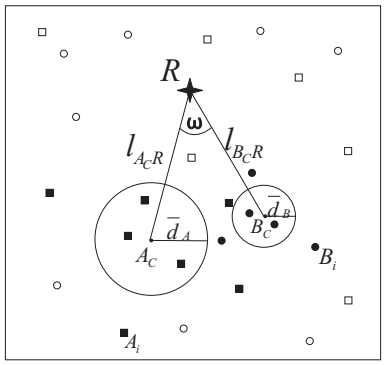


Figure A.1: Scenario

n_A, n_B	Number of tuples of Relation A, B involved in the join
$l_{\mathcal{A}_C\mathcal{R}}$, $l_{\mathcal{B}_C\mathcal{R}}$	Distance from center of mass, $\mathcal{A}_C$, $\mathcal{B}_C$ to the root $\mathcal{R}$
ω	Angle $\angle\mathcal{A}_C\mathcal{R}\mathcal{B}_C$
$\bar{d}_A$, $\bar{d}_B$	Distribution (mean distance) of nodes with respect to their center of mass
c_a, c_b	Relative costs of sending t_A, t_B for one hop with respect to t_{AB}
σ	Join selectivity

Table A.1: Overview of the parameterization

t_A, t_B, t_{AB}	Tuples of Relation A, B and a result
$\mathcal{A}_C$, $\mathcal{B}_C$	Center of mass of Relation A, B
$\mathcal{A}_i$, $\mathcal{B}_i$	Node of Relation A, B
$\mathcal{R}$	Root node

Table A.2: Further notation

represents the nodes of Relation A as filled squares. They are denoted as $\mathcal{A}_i \in \{\mathcal{A}_1, ..., \mathcal{A}_m\}$. Nodes of Relation B are depicted as filled circles. Non-filled shapes stand for nodes whose tuples are excluded by the `WHERE`-clause of the query. Note that in the case of a self-join, nodes could also belong to both relations, depending on the selection predicates. The root node $\mathcal{R}$ is depicted as a star. We refer to a tuple of Relation A or B as t_A or t_B, respectively. t_{AB} represents a tuple that results from joining t_A with t_B.

In our analysis we distinguish between different states of the network according to the following parameterization: Relation A can be described by its center of mass $\mathcal{A}_c$, the *mean* distance $\bar{d}_A = \frac{1}{n_A} \cdot \sum_i d(\mathcal{A}_c, \mathcal{A}_i)$ of each node to the center and the number of nodes n_A. Relation B can be described in the same way. The relative location of the nodes of A and B to each other and to the root node are described by the angle $\omega = \angle\mathcal{A}_c\mathcal{R}\mathcal{B}_c$ and the distances $l_{\mathcal{A}_C\mathcal{R}}$, $l_{\mathcal{B}_C\mathcal{R}}$.

Given these parameters, we can abstract from single nodes by describing the relation as $(\mathcal{A}_c, \bar{d}_A, n_A)$. On the other hand, our numerical methods require concrete sets of nodes based on the parameters $(\mathcal{A}_c, \bar{d}_A, n_A)$. We therefore assume:

Assumption A.1: *The nodes of a relation are uniformly distributed within ($\mathcal{A}_c, \bar{d}_A$).*

Finally, we will refer to the join selectivity as σ, which is defined as the ratio of the cardinality of the join result to the input, i.e., $\sigma = \frac{card(A \bowtie B)}{card(A) \cdot card(B)}$. Table A.1 summarizes these parameters. Table A.2 contains some further notation.

Appropriateness of our Network Model. Note that our set of parameters describes each relation as a whole. There is an infinite number of concrete placements of nodes, which corresponds to each parameter setting.

Proposition A.1: The results obtained for a parameter setting according to Table A.1 apply to every instance of node placements that obeys the setting.

This is true as we found that, given a specific parameter setting, the gain of sets of nodes $\{\mathcal{R}, \mathcal{A}_1, ..., \mathcal{A}_{n_A}, \mathcal{B}_1, ..., \mathcal{B}_{n_B}\}$ that obey this setting has a very small variance ($Var(gain) < 1\%$; cf. Section A.2). Thus, every instantiation of our network model results in approximately the same gain. This makes our compact model very well suited for the analysis.

A.1.3 Cost Model

Our optimization goal is to minimize the energy consumed for communication. To this end, we need to model the communication costs.

Costs of Sending a Packet Via Multiple Hops. The costs of sending a packet are computed by multiplying the one-hop costs with the number of hops. We model the one-hop costs as a parameter which is discussed at the end of this section (c_a, c_b). The number of hops is approximated by the Euclidean distance $d(\cdot, \cdot)$ between the sending and receiving node. Thus, the costs of sending a packet containing Tuple t_A from node $\mathcal{A}_1$ to $\mathcal{A}_2$ are modeled as $d(\mathcal{A}_1, \mathcal{A}_2) \cdot c_a$.

Since we are interested in the relative performance of different strategies (gain), a model that provides costs proportional to the communication costs suffices. The Euclidean distance is proportional to the number of hops given the following assumption:

Assumption A.2: *The sensor network is sufficiently dense such that the Euclidean distance between two nodes is (approximately) proportional to the number of hops.*

Nodes at Optimal Locations. In our analysis we will come up with strategies that perform computations at the mathematically optimal locations. Thus, our model assumes that there exists a node at the derived location. In practice, the node closest to this location has to be chosen. If the network is sufficiently dense, this node should

be within communication distance of the optimal location. As a result, the number of hops will be the same or will differ by at most one hop (more or less). Therefore, we expect the influence on the results to be small.

Assumption A.3: *The sensor network is sufficiently dense such that there exists a node within communication distance of each point.*

Communication Costs Per Hop. In the remainder of this section we discuss the communication costs per hop (c_a, c_b). In addition to defining them, we discuss the range of these parameters for realistic communication hardware. This is important to derive meaningful conclusions.

The costs of sending one packet over one hop can be decomposed to *fixed costs* per packet and *variable costs* depending on the size of the payload: $cost(size(t)) = cost_{fix} + cost_{var}(size(t))$. We consider the costs of sending t_A, t_B as well as of sending a result tuple t_{AB}. In order to reduce the number of parameters, we normalize the costs with respect to the costs of sending one result tuple: $c_a = \frac{cost(size(t_A))}{cost(size(t_{AB}))}$.

Variable costs depend on the size of the tuple. To interpret our results we assume:

Assumption A.4: *The maximum size of a tuple is 15 attributes of two bytes.*

Note that 15 attributes is a lot given that current sensor nodes like Mica motes are equipped with up to 8 sensors. Thus, t_{AB} can be up to 30 bytes larger than t_A if t_B is at maximum size, and no join attributes are projected out. In order to understand the lower bound on the size of t_{AB}, consider the number of join attributes. It is always possible to construct examples consisting of an arbitrary number of join attributes. However, in order to arrive at meaningful conclusions we focus on realistic scenarios:

Assumption A.5: *The number of join attributes in sensor networks is at most 8.*

Thus, t_{AB} can be up to 16 bytes smaller than t_A if all 8 join attributes from t_A and t_B are projected out.

Fixed costs for sending one packet mainly depend on the MAC and PHY layer overhead. It results from the wakeup of the transmitter, carrier sensing, RTS and CTS, preamble, etc. In order to quantify these costs we looked at a sample of prominent MAC protocols: S-MAC [YHE04], B-MAC [PHC04], and SCP-MAC [YSH06]. The minimum PHY/MAC layer overhead we observed is equivalent to the transmission of 127 bytes. This leads to the following assumption:

Assumption A.6: *The difference in the energy consumption between sending an empty frame and a frame with 16 bytes payload is less than 15%. The difference for sending 30 bytes is less than 30%.*

This percentage is further reduced by overhearing, contention, errors and collisions. Most of the measurements we are aware of refer to the 802.11 protocol (e.g.

[Fee01]). There, increasing the payload by 30 bytes results in a difference of less than 10%. Consequently, the relative costs c_a (c_b) are in the range from 0.75 (for a maximum of 30 bytes less than t_{AB}) up to 1.15 for 16 bytes more. This range should include any realistic communication hardware.

A.2 Centralized Join Processing

We now justify that centralized join approaches (e.g., [CG05, CN07]) cannot serve as a general-purpose join method. They are superior to the external join in specific scenarios only: The nodes involved need to be close to each other compared to their distance to the base station. In addition, a high selectivity is required.

[CNS07] has arrived at similar findings by means of simulation. Given Proposition A.1, our analytical approach rules out that there exist placements of nodes that are not in line with this result. In addition, the discussion in this section is important to follow Section A.3. There, we will reduce parts of the analysis to the centralized case.

A.2.1 Cost Model for Centralized Strategies

Our goal is to identify scenarios where using a centralized processing at a single site J results in energy savings compared to the root strategy, i.e., where there is a gain ($1 - \frac{cost_J}{cost_{root}} > 0$). In the following, we provide a model of the costs of centralized strategies based on our model of communication costs (cf. Section A.1).

The cost of computing the join at (any) point J is the sum of the costs of sending the tuples of Relations A and B to J and sending the result to the root node subsequently:

$$cost_J = \sum_{i=1}^{n_A} c_a \cdot d(\mathcal{A}_i, J) + \sum_{i=1}^{n_B} c_b \cdot d(\mathcal{B}_i, J) + n_A \cdot n_B \cdot \sigma \cdot d(J, \mathcal{R}) \tag{A.1}$$

$d(P_1, P_2)$ denotes the Euclidean distance. Recall that a root strategy is a centralized strategy, i.e.,

$$cost_{root} = \sum_{i=1}^{n_A} c_a \cdot d(\mathcal{A}_i, \mathcal{R}) + \sum_{i=1}^{n_B} c_b \cdot d(\mathcal{B}_i, \mathcal{R})$$

What remains to be specified for the model is the join location J. The optimal join location is not a parameter but depends on the placement of the nodes.

Proposition A.2: It is impossible to derive a closed formula for the gain, irrespective of the parameterization of the network.

Proof: The join location that minimizes $cost_J$ is optimal. This corresponds to the Fermat problem [CNS07]: For a given set of points $\{P_1, ..., P_n\}$ and their corresponding weights $\{w_1, ..., w_n\}$, find a point J that minimizes $\sum_i w_i \cdot d(P_i, J)$. It has been shown that there is no closed expression for computing the Fermat point [Baj88]. ■

The Fermat problem can only be solved numerically.

A.2.2 Method

Proposition A.2 results in two problems: (1) We need a method to compute the gain numerically. (2) We must be able to analyze the gain-function in order to identify global and local optima.

(1) Computation of the Gain Function. Our approach for numerically computing the Fermat point F of a set of points ($\{\mathcal{R}, \mathcal{A}_1, ..., \mathcal{A}_m, \mathcal{B}_1, ..., \mathcal{B}_n\}$) is based on Weiszfeld's algorithm [Kuh74]. In order to provide strict confidence bounds, we compute the function gain($l_{\mathcal{A}_\mathcal{C}\mathcal{R}}, \bar{d}_A, n_A, c_a, l_{\mathcal{B}_\mathcal{C}\mathcal{R}}, \bar{d}_B, n_B, c_b, \omega, \sigma$) based on the Monte Carlo method as follows:

For a setting ($l_{\mathcal{A}_\mathcal{C}\mathcal{R}}, \bar{d}_A, n_A, c_a, l_{\mathcal{B}_\mathcal{C}\mathcal{R}}, \bar{d}_B, n_B, c_b, \omega, \sigma$) do:

1. Generate a random set of points $\{\mathcal{R}, \mathcal{A}_1, ..., \mathcal{A}_m, \mathcal{B}_1, ..., \mathcal{B}_n\}$ that follows the parameter setting.
2. Compute the Fermat point F
3. Compute the expected gain based on $cost_J$ with $J = F$

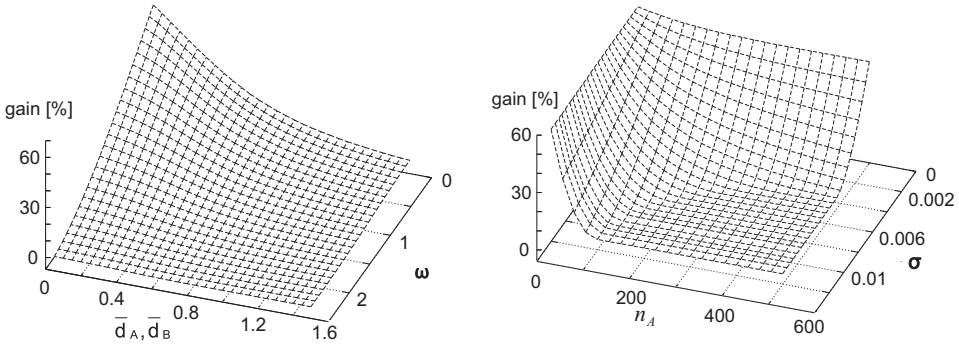
Aggregate the expected gain with the results from former trials and repeat until the confidence for the expected gain over all trials is within 0.01% with 98% probability.

According to our analysis, the variance of the gain is small ($Var(gain) < 1\%$) for different sets of points that obey the same parameter setting. Thus, the expected gain is a reasonable measure to compare join strategies.

(2) Analyzing the Gain Function. We want to identify the optima of the gain function. Analytically finding optima requires differentiating the function. However, this is impossible for the gain due to Proposition A.2. It is also problematic to find optima based on numerical methods: Such methods inspect a discrete number of values and make assumptions about the values in between. Thus, making reasonable assumptions is essential for ensuring not to miss local optima. For our analysis, we approach the problem twofold: In Section A.2.3 we observe that the parameters are monotonic within the range defined in Section A.1.3. In Section A.2.4 we prove that our numerical approach finds the single global optimum for the gain. This proof is

n_A, n_B	Number of tuples of A, B	200, 300
$l_{\mathcal{A}_C\mathcal{R}}$, $l_{\mathcal{B}_C\mathcal{R}}$	Distance $\mathcal{A}_C$, $\mathcal{B}_C$ to $\mathcal{R}$	1.0, 1.0
ω	Angle $\angle\mathcal{A}_C\mathcal{R}\mathcal{B}_C$	0.5 (30°)
$\bar{d}_A$, $\bar{d}_B$	Mean distance to $\mathcal{A}_C$, $\mathcal{B}_C$	0.5, 0.5
c_a, c_b	Relative costs of sending t_A, t_B	1.0, 1.0
σ	Selectivity	0.002

Table A.3: Standard setting for the analysis



(a) (b)

Figure A.2: (a) Influence of $\angle\mathcal{A}_c\mathcal{R}\mathcal{B}_c$ & distribution; (b) Influence of the result's size

independent of our monotonicity assumptions. In addition, the proof further substantiates the monotonicity assumptions as they are in line with the global optimum.

A.2.3 Monotonicity Assumptions

This section deals with deriving the assumptions required for ruling out local optima of the gain-function. In particular, we derive monotonicity assumptions based on reasoning about the underlying problem. Note that the gain function is not differentiable and therefore it is impossible to prove its monotonicity. Thus, we first discuss the rationale behind our assumptions, and then complement our discussion by computing the values of the gain function at discrete points. Since the gain depends on 10 parameters, it is impossible to provide an exhaustive numerical scan over the full parameter space. We use a systematic approach similar to partial differentiation, i.e. we consider the parameters in isolation. In particular, we compute the gain numerically by systematically varying one of the parameters and set the other ones to a standard setting (Table A.3). This setting is chosen such that it yields a medium gain (30%), i.e., it enables us to observe increases as well as decreases in the gain.

We now establish the monotonicity for the parameters related to the locations of the nodes, followed by the other parameters.

Influence of the Tuple Locations. We start discussing location-related parameters ($l_{\mathcal{A}_C\mathcal{R}}, \bar{d}_A, l_{\mathcal{B}_C\mathcal{R}}, \bar{d}_B, \omega$) by providing an explanation based on characteristics of the Fermat point. Figure A.1 serves as an illustration. The Fermat point is located "in between" the relations. If the angle or the mean distance of the nodes to their center becomes larger, the root node comes closer to also being "in between" the relations. In this case the root strategy is similar to the centralized strategy that performs the join a the optimal location, and the relative gain of the centralized in-network strategy becomes small.

Assumption A.7: *The gain increases monotonically as the angle ω between the centers of the relations or the mean distance of the nodes to their centers ($\bar{d}_A$, $\bar{d}_B$) decreases. It decreases monotonically as the difference between $l_{\mathcal{A}_C\mathcal{R}}$ and $l_{\mathcal{B}_C\mathcal{R}}$ increases.*

Figure A.2(a) illustrates the monotonicity for the mean distances of the nodes to their centers of mass ($\bar{d}_A$, $\bar{d}_B$) and the angle ω. The discussion for $l_{\mathcal{A}_C\mathcal{R}}$ and $l_{\mathcal{B}_C\mathcal{R}}$ is analogous. Besides the monotonicity, Figure A.2(a) shows that the in-network strategy yields the maximum energy savings if all nodes except the root node are located close to each other ($\bar{d}_A = \bar{d}_B = 0$ and $\omega = 0$). This minimizes the routes along which the input tuples are sent.

Influence of the Result Size. Again, we start the discussion of the parameters related to the size of the result ($n_A, n_B, \sigma, c_a, c_b$) by providing an explanation based on the Fermat point. It is known that as soon as the weight of one of the points is more than half of the total weight, it is the Fermat point [TL03]. Furthermore, if the weight of one of the points is close to half of the total weight, the Fermat point will be near that point. In our context, the weight of a node is the number of tuples that it sends multiplied with the transmission costs. More specifically, the weight of the result is $n_A \cdot n_B \cdot \sigma$, and the weights of the input tuples are $n_A \cdot c_a$ and $n_B \cdot c_b$. If $n_A \cdot n_B \cdot \sigma \geq m \cdot c_a + n \cdot c_b$, the Fermat point will coincide with the root node. In this case, $cost_J$ and $cost_{root}$ are identical, and the gain will become 0. In addition, the larger the size of the result, the closer will the Fermat point be to the root node.

Assumption A.8: *Computing the result inside the network is only beneficial if the cardinality of the result is smaller than the input. In particular, this requires a high selectivity. Thus, the gain is monotonic in the parameters that determine the size of the result: $c_a, c_b, n_A, n_B, \sigma$.*

We complement the explanation by computing the gain function. Figure A.2(b) shows the gain depending on n_A (the number of nodes of Relation A) and the selectivity σ. The remaining parameters correspond to the standard setting. The figure confirms our assumption: As soon as the size of the result outweighs the input tuples, the gain becomes small, and performing the join inside the network does not pay off.

Furthermore, the gain is monotonic in the parameters that determine the size of the result.

A.2.4 Gain of Centralized Strategies

In the following, we present the single global optimum of the gain function:

Proposition A.3: The gain of centralized in-network strategies has its maximum if the nodes are co-located ($l_{\mathcal{A}_C\mathcal{R}} = l_{\mathcal{A}_C\mathcal{R}}$, $\bar{d}_B = \bar{d}_B = 0$, $\omega = 0$) and the size of the result is minimal (0).

Proof. We have to show that the scenario in Proposition A.3 is the global optimum. This can be seen by considering the range of the gain-function $[0, 1]$ (cf. Corollary A.1). In the situation that we identified as a candidate for an optimum, $cost_J = 0$, while $cost_{root}$ takes on some fixed amount. In this case, the gain becomes 1 and thus is indeed a global optimum. Finally, this is the only global optimum. We conclude this from the formula (gain = $1 - \frac{cost_J}{cost_{root}}$) since $cost_J > 0$ (cf. Equation A.1) for any other setting. ■

Due to its monotonicity (cf. Assumptions A.7 and A.8) the gain-function has no local optima.

In summary, centralized approaches can be more efficient than computing the join at the base station in rather specific scenarios only. The nodes involved need to be close to each other compared to their distance to the base station. In addition, a high selectivity is required.

A.3 IDEAL Join Processing

In this section, we establish IDEAL, a lower bound on the costs of join processing in sensor networks. IDEAL requires each node to know if its tuple joins and is thus not a practical approach. However, identifying parts of the problem for which we can derive optimal solutions has helped us in the design of our join algorithms in Chapters 4 and 5. In addition, we have used IDEAL as a comparison scheme for CJF, as our goal there has been to come close to the optimum for continuous join queries.

IDEAL involves the following steps: Firstly, each node discards its tuple if it does not join. Then, the remaining tuples are sent to the base station where the join is computed.

The justification of the first step is straightforward: Tuples that do not join have no influence on the result. Thus, for minimizing the communication costs, it is optimal not to send them at all.

The actual problem that we address in this section is where to join the remaining input tuples. Most notably, discarding non-joining tuples is orthogonal to distributing the result computation: While IDEAL computes the result at the base station, it would

also be possible to choose processing sites inside the network. Thus, the question that we will answer is: Can we improve upon IDEAL by choosing other processing sites? If not, IDEAL actually lower bounds the costs of join processing.

The subsequent discussion is restricted to joining tuples. This is because, in theory, any strategy that sends non-joining tuples can be improved by not sending them – the *only* degree of freedom that we can exploit for finding a better strategy compared to IDEAL is the join location. Subsequently, we analyze if there is a gain if we allow for distributed strategies. This corresponds to dropping the restriction of processing the result at the base station. The major difficulty in finding a good distributed strategy is that optimal locations and optimal routes mutually depend on each other. We start the discussion with a concept that identifies subsets of tuples that can be regarded in isolation:

Definition A.6 (Group)
A group of tuples G is a subset of the union of Relations A and B such that if tuple $t \in G$ then every tuple t' that joins with t is in G as well.

Example A.1: For an equi-join a group is a subset of Relations A and B that yields a cross product. In this case, if we restrict the join to a single group its selectivity $\sigma = 1$. ■

The following corollary is a consequence of Definition A.6:

Corollary A.2: *The processing locations of different groups are independent of each other.* □

The reason is that tuples from different groups do not have to be brought together at one location[1]. This property lets us restrict our analysis to a single group in order to find the optimal distribution. We start with a simple example of a distributed processing that we will use in subsequent discussions:

Example A.2: Figure A.3 shows two nodes $\mathcal{A}_1, \mathcal{A}_2$ belonging to Relation A and one node $\mathcal{B}$ of Relation B. Assume that their tuples form a group and all tuples have the same size. The figure shows a strategy where t_B is first joined with the tuple from $\mathcal{A}_2$. The result is sent to the root node. t_B is also sent to a second location where it is joined with the tuple from $\mathcal{A}_1$. ■

Our analysis how to distribute the result computation is structured as follows:

1. For a group of tuples: identify the optimal number of processing sites
2. For a group of tuples: analyze where these sites are located

[1]Recall Requirement A.3: To compute a correct result, we have to ensure that every pair of tuples that join meets at (at least) one location.

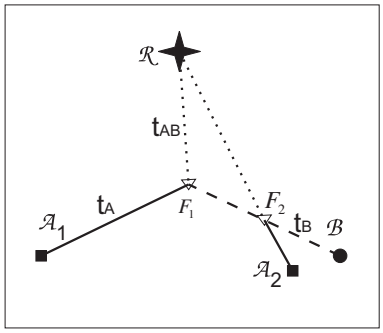


Figure A.3: Example of two join locations

A.3.1 Optimal Number of Processing Sites

The difficulty in identifying the optimal number of sites is that our problem cannot be reduced to a solved mathematical one. However, we can upper bound the gain by identifying the scenario with the highest gain when distributing the computation of the join result:

Definition A.7 (n_A:1 Scenario)
An n_A:1 scenario is a group consisting of n_A tuples of Relation A that pair up with one tuple of Relation B.

Proposition A.4: The n_A:1 scenario is the optimal case for multiple join locations, compared to all other groups ($n_A : n_B$), for a fixed set of nodes from Relation A.

Proof. If $n_A, n_B \geq 2$ (otherwise we have an n_A:1 scenario), the optimal centralized join location is the root, as the cardinality of the result is larger than the cardinality of the input (property of the Fermat point [TL03]). Consider increasing n_B by 1. If the computation is centralized, this means that we have to send t_B to the root. In the distributed case, we have to send t_B to each of the processing locations. Afterwards, n_A result tuples have to be sent from the processing sites to the base station. Thus, the cost increase for the centralized setting is less in relative terms. Therefore, the n_A:1 scenario yields the largest gain for a distributed computation, compared to a centralized one. ■

In the n_A:1 scenario, the tuples can be joined at many locations and in different orders, resulting in a combinatorial problem. We devise a method for upper bounding the gain that consists of:

- an algorithm for computing optimal strategies for two or three tuples in Relation A ($n_A \in \{2, 3\}$), and
- an estimator for lower bounding the costs for $n_A > 3$.

Computing Optimal Strategies. We can compute optimal strategies based on the following proposition:

Proposition A.5: Any join location J in a distributed strategy is a Fermat point. It minimizes the routes of the nodes from which a tuple is sent to J and to which a tuple is sent from J.

Proof. The proof is the same as the proof of the corresponding property of Steiner trees [Mel61]. The idea is that the routes could be further optimized if the join location was not the Fermat point. ■

For the case $n_A = 2$, Proposition A.5 restricts the number of possible strategies to three:

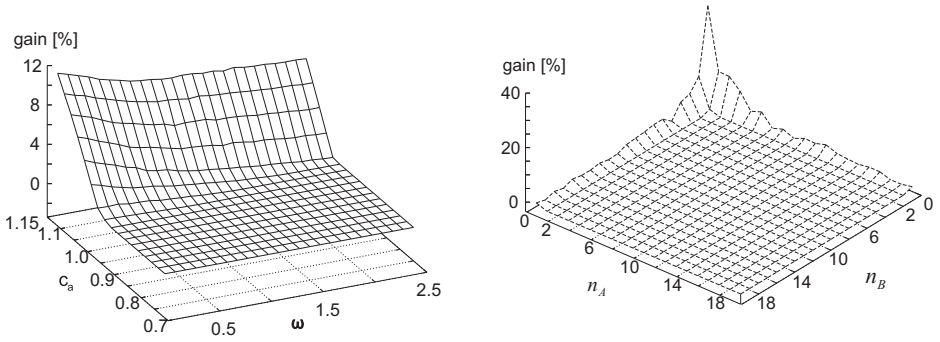
1. Join $\mathcal{A}_2$ with $\mathcal{B}$ at Fermat Point $\mathcal{F}_2$ and send t_B on to $\mathcal{F}_1$ and join it with the tuple from $\mathcal{A}_1$ (cf. Figure A.3).
2. Join $\mathcal{A}_1$ with $\mathcal{B}$, then join $\mathcal{A}_2$ with $\mathcal{B}$.
3. Use one Fermat point for all nodes $\mathcal{A}_1$, $\mathcal{A}_2$, $\mathcal{B}$ and $\mathcal{R}$.

By computing the join locations in all three cases and comparing the overall costs we find the optimal strategy. For $m = 3$ we can follow the same procedure except that there are 13 possible constellations.

Lower Bound Estimation. For $n_A > 3$ we lower bound the costs (upper bound the gain) of processing the join by reducing the problem to $n_A = 3$. We accomplish this reduction by choosing two distinguished nodes from Relation A and assume that the rest was located at one third point. We choose the two distinguished nodes by taking the node closest to the root as the first point and the node N that maximizes $d(N, \mathcal{R}) + d(N, \mathcal{B})$ as the second. The intuition is to take the distribution of the nodes into account in order to arrive at a meaningful estimation of the communication costs. The remaining nodes of Relation A are assumed to be located at a single point on the line $\mathcal{R}\mathcal{A}_\mathcal{C}$. In this way, we keep the mean distance from the root node unchanged. Assuming the remaining nodes to be co-located leads to an underestimation of the costs as it reduces the routing lengths to pair the tuples. Figure A.1 serves as an illustration of the third point.

Gain of Several Join Locations. In the following we compute a lower bound on the costs of strategies that are allowed to use multiple join locations and compare it to IDEAL. Again, the following discussion is based on monotonicity assumptions of the gain. This is analogous to Section A.2.

Proposition A.6: Using multiple join locations per group of pairing tuples results in energy savings of at most 12% as compared to choosing a single site (IDEAL).



(a) (b)

Figure A.4: (a) Gain: optimal number vs. single site; (b) Gain: optimal location vs. root node

This is an upper bound on the savings for the optimal scenario (n_A:1) and holds if the tuples of Relation A are larger than the result tuples.

We found the costs c_a of sending a tuple of Relation A to be the most influential parameter. Figure A.4(a) shows its influence along with the angle ω between Relation A and B. Intuitively, if the result tuples are larger than the ones of Relation A ($c_a < 1$), sending the input tuples directly to the root node is a good choice. Thus, the optimal centralized as well as the optimal decentralized strategy are alike. Only if the result tuples are much smaller than the ones of Relation A, multiple join locations can reduce the energy consumption. Our analysis confirms this intuition. If $c_a < 1$ using a single join location is optimal. In addition, Figure A.4(a) already shows the maximum gain: In analogy to Section A.2, the gain computed confirms the monotonicity assumptions. Therefore, we have identified the globally optimal gain.

Our analysis of the scenario with the maximum gain reveals that the upper bound for the energy savings of multiple locations per group of joining tuples is small. Thus, the number of join locations should be one per group.

A.3.2 Location of Optimal Site per Group

In order to identify the single optimal join location per group, we compare computing the result for a group at its Fermat point to computing it at the root (IDEAL).

Proposition A.7: After discarding non-joining tuples, the optimal location per group is the root node if $n_A, n_B \geq 2$; the location is the same for every group.

The analysis for one group is the same as the one in Section A.2. The only difference is that the selectivity factor is high (selectivity is low), at least $\frac{n_A}{n_A+1}$ for the n_A:1 scenario, and the number of tuples per group might be small. Thus, the results

directly apply. The most important influence is the size of the result, which is larger than the input if $n_A, n_B \geq 2$. Figure A.4(b) visualizes this influence, depicting the percental gain of the optimal location. As soon as the group consists of more than one tuple per relation ($n_A, n_B \geq 2$) the root node is the optimal location. Only if $n_A = n_B = 1$ a distributed join strategy can save up to 50% energy if two input tuples lead to one result tuple. Note that $n_A = n_B = 1$ means that none of the two joining tuples has a further join partner.

As a final remark, the preceding analysis that justifies Proposition A.7 corresponds to the intuition that we provided in Section 3.1.3: After discarding non-joining tuples, the join increases the cardinality, i.e., there are more tuples in the result than in the input, except for pathological cases.

Summary. Our most important insight is that the result computation is optimally performed at the root node, if tuples that do not join are discarded in advance. Joining tuples at multiple locations does not increase the energy-efficiency. Thus, IDEAL lower bounds the costs of join processing.

A.4 Conclusions

In this chapter, we theoretically substantiated two claims related to join processing. Firstly, processing the join at a central site inside the network is inferior to the external join except for very specific scenarios. In particular, the input relations have to stem from two small regions. In addition, the regions need to be close to each other, compared to their distance to the base station, and the selectivity needs to be very high. Secondly, we showed that IDEAL lower bounds the costs of join processing. This implies the following approach for efficiently processing joins in sensor networks: We can turn IDEAL into a practical approach by providing the knowledge if its tuple joins to each node. Doing so in an efficient manner is the underlying idea of SENS-Join (cf. Chapter 4) and CJF (cf. Chapter 5).

B Primitives for Constructing the Quadtree Datastructure

This appendix features a formal description of our quadtree encoding based on pseudocode. It complements the presentation in Section 4.3.

In Section B.1, we discuss the method $\texttt{Insert}_{\texttt{Join\_Atts}}$. SENS-Join uses it to insert a join-attribute tuple into a quadtree during the Join-Attribute-Collection step (cf. Figure 4.2). We then show how to union two quadtrees in Section B.2. This is to underline our argument that $\texttt{Union}_{\texttt{Join\_Atts}}$ and $\texttt{Intersect}_{\texttt{Join\_Atts}}$ (cf. Figure 4.2) can be computed directly on the quadtrees. There is no need to recover the original tuples as with compression algorithms.

Scope of the Following Discussion. In the following, we discuss quadtrees that store Z-numbers of join-attribute tuples, as discussed in Section 4.3.2. For an actual implementation, the Z-numbers have to be prefixed with the relation flags (cf. Section 4.3.3). We eliminated their treatment from the figures as it clutters the pseudocode. However, recall from Section 4.3.3 that the relation flags simply constitute a further index level in the quadtree – their treatment will be clear at the end of this discussion. The purpose of this appendix is to clarify the quadtree encoding.

B.1 Constructing Quadtrees

According to Section 4.2, the quadtree is constructed incrementally while SENS-Join collects join-attribute tuples. In this section, we illustrate the structure of our quadtree representation in detail by discussing the insertion of a join-attribute tuple. Figure B.1 provides pseudocode of the method $\texttt{Insert}_{\texttt{Join\_Atts}}$ which is called during the Join-Attribute-Collection (cf. Figure 4.2). At a high level, inserting a join-attribute tuple into a quadtree requires two steps: Firstly, the join-attribute tuple is quantized, i.e., SENS-Join computes its Z-number in the quadtree domain (Line 11). This step has been discussed in Section 4.3.2 in detail. Then, the point is actually inserted into the quadtree (Line 14).

We will use Figure B.2 to illustrate our explanation of the insertion. It summarizes the idea of the quadtree (Figure 4.9) and its encoding (Figure 4.10). Recall from Section 4.3.3 that the Z-number of a join-attribute tuple corresponds to a sequence of quadrants that result from traversing the index top-down until we reach the full

```
1
2  T' = π_JoinAttr(T);
3
4  Insert_Join_Atts(Quadtree Q, Tuple T')
5
6      //compute the quadtree domain (size of each dimension)
7      for all dimensions i
8          SizeOfDim[i] = {(MaxVal[i] − MinVal[i]) · 1/Resolution[i]} + 1;
9          SizeOfDim[i] = roundUpToPowOf2(SizeOfDim[i]);
10
11     P_index = EncodeTuple(P, SizeOfDim); //cf. Figure 4.8
12
13     //insert P (T') into quadtree
14     Q = InsertPoint(P_index, Q, SizeOfDim);
15     return Q;
```

Figure B.1: $\text{Insert}_{\text{Join\_Atts}}$

resolution (as specified by the quantization). For instance, for the two-dimensional case that is illustrated in Figure B.2, the Z-number is composed as a sequence of numbers $\in \{0, 1, 2, 3\}$ (cf. Figure 4.7). Each number indexes the quadrant at an increasing level of detail. Thus, by reading the Z-number from the beginning, we can use the numbers of the quadrants to traverse the tree. Example 4.4 illustrates this traversal.

Further, recall some details of the encoding from Section 4.3.3 (cf. Figure B.2): A quadtree is a bitstring that captures the tree in depth-first order. We use a '0' for indicating an index node and a leading '1' for specifying that the following is a point. We also need to mark the end of a list of points. This is done by appending a '0' to the list.

`InsertPoint` actually accomplishes this encoding of a quadtree. Before presenting the method, we briefly discuss some information that it builds upon. As a quadtree is a bitstring, all of the methods for modifying quadtrees need to navigate within bitstrings. In particular, we need to extract index nodes, points, lists of points, or entire subtrees from the bitstring. Navigating in the bitstring requires to know the size (number of bits) of index nodes and points. Most notably, the sizes of index nodes and points are not fix, but vary with their level in the tree: Points are stored relative to the region, i.e., relative to the index nodes – we discussed in Section 4.3 that the encoding of the relative position becomes shorter with an increasing level in the tree. In fact, this is what our compact encoding exploits.

For index nodes, it might be less obvious that their size can shrink with the level in the tree. The reason is that the dimensions of the join-attribute space are not necessarily of equal size. Thus, some dimensions might reach their full resolution

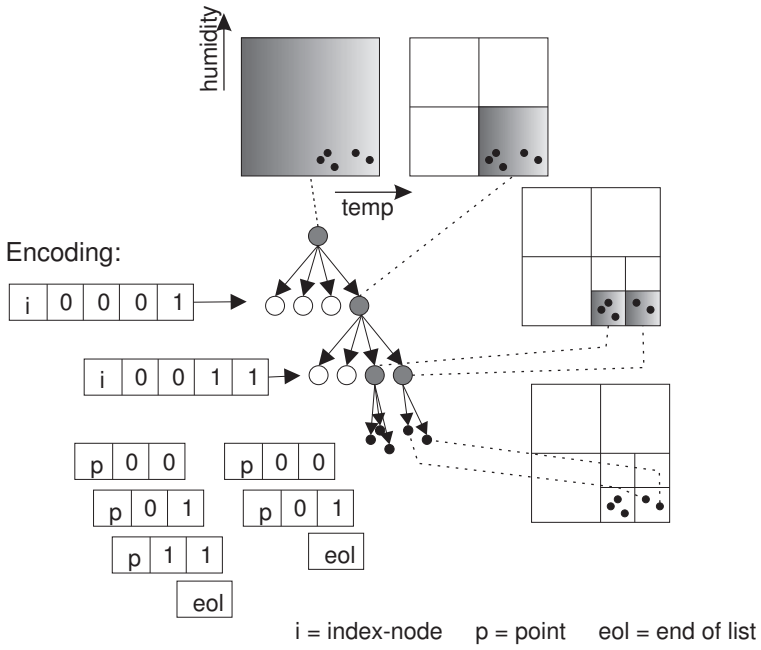


Figure B.2: Representing points with a quadtree

```
1 Split(Int[] SizeOfDim)
2
3 for all dimensions i
4       SizeOfDim[i] = SizeOfDim[i]/2;
5 for all dimensions i
6       if (SizeOfDim[i] == 1)
7             remove dimension i from SizeOfDim;
8 return SizeOfDim;
```

Figure B.3: Method *Split*

before others do. At each level in the tree, we can only split dimensions that did not already reach their full resolution – in a way, the dimensionality of the quadtree can decrease as we traverse the tree top-down and thus the index nodes can become smaller.

We used the array 'SizeOfDim' (cf. Figures 4.8 and B.1) to keep track of the number of different points in each dimension. It has one entry per dimension of the join-attribute space. In particular, we will subsequently use the method `Split` (cf. Figure B.3) to compute the size of the dimensions at the next lower level of the tree. `Split` simply divides the size of each dimension by two (Lines 3, 4). In addition, it finds those dimensions that reached their full resolution and eliminates them from the array. Most notably, as a consequence, the number of dimensions always corresponds

to the size of the array 'SizeOfDim'.

The pseudocode for inserting a point into a quadtree is presented in Figure B.4. `InsertPoint` first checks if the quadtree is empty. If so, the tree that results from inserting P is just a list of a single point, which is encoded as described (Lines 3, 4). In Line 5, the algorithms finds out how many dimensions the join-attribute space has. The number of dimensions dictates the size (number of bits) of index nodes, as illustrated in Figure B.2.

If the quadtree is not empty, `InsertPoint` checks whether the tree begins with an index node (Line 8) or is a list of points (line 24). The insertion is different in these cases. If it is an index node, the quadrant of the Point P at the current level is determined by reading the beginning of P (Lines 10, 11). Afterwards, the corresponding entry in the index-node is marked since the quadtree will contain points in that subtree after inserting P (Line 13). Intuitively, the insertion is now done recursively: We need to insert the remainder of the point P into the subtree that we just identified (which is quadtree as well). Therefore, `InsertPoint` extracts the corresponding subtree Q' in the bitstring (Lines 14 - 17) as well as the remainder P_{rest} of P (Line 19) and calls itself (Line 20). Finally, the subtree Q' is substituted by the quadtree that results from inserting P_{rest} (Line 22).

In the case that the current quadtree is a list of points (Line 23), we first check whether P is already contained (Lines 25 - 29). Only if P is not contained it is inserted (duplicate elimination). The insertion is handled subsequently: Lines 30 to 35 take the current list of points and convert it into a quadtree with an index-node. In Line 31, an empty index node Q' is established. Then, in Lines 32 to 34, `InsertPoint` iterates over the points in the list and inserts each of them into Q'. As Q' is a quadtree, this can be done by a recursive call (Line 34). P is inserted as well (Line 35). Afterwards, `InsertPoint` checks whether the resulting quadtree is shorter than a corresponding list of points. This implements the 'decomposition threshold' which has been discussed in Section 4.3.3. `InsertPoint` returns the shorter alternative[1].

In Lines 14 to 17, `InsertPoint` extracts a subtree that corresponds to one of the quadrants. To do so, it uses the subroutine `GetPositionOfSubtree` for finding the beginning of the subtree. It is a simple walk through a depth-first tree until the beginning of the subtree is reached. The corresponding method is shown in Figure B.5. Among the input parameters, 'PointLength' indicates the number of bits that are required to encode a point. This will be used by `GetPositionOfSubtree` to step over a subtree if it is a list of points.

At a high level, `GetPositionOfSubtree` maintains a pointer ("offset") into the quadtree bitstring which it received as input. The idea is to forward this pointer until the quadrant 'Quadrant', which is searched for, is reached. The method starts

[1]It is possible to decide on the shorter alternative without actually constructing both. For clarity of presentation we stick to the underlying idea at this point.

by determining the size of the dimensions as well as the length of points at the next lower level of the subtree (Lines 4 to 8). As with `InsertPoint`, this is needed for recursive calls.

`InsertPoint` calls `GetPositionOfSubtree` only if the quadtree at hand starts with an index node. Therefore, `GetPositionOfSubtree` first forwards the pointer by the size of an index node (Line 11). `GetPositionOfSubtree` then iterates over entire subtrees until the subtree corresponding to 'Quadrant' is reached (Lines 14 to 26). In Line 14, the algorithm checks if this is the case. If not, `GetPositionOfSubtree` has to forward the pointer over the next subtree. To do so, the algorithm reads the index node to see if there is a subtree corresponding to the next quadrant. If the next quadrant is empty, there is no need to forward the pointer – there is no subtree. If the quadrant is not empty, `GetPositionOfSubtree` checks whether the subtree starts with an index node (Line 17) or is a list of points (Line 22). Forwarding the pointer if the subtree starts with an index node is done recursively (Lines 19, 20). If the subtree is a list of points, `GetPositionOfSubtree` exploits that it knows the length of the points. It forwards the pointer, point by point, until it reaches the end of the list (Lines 23 - 25). This completes the forwarding as the following bit is the beginning of the next subtree.

B.2 Merging Quadtrees

In this section, we discuss merging two quadtrees. As the details of handling the bitstring are the same as for `InsertPoint`, we present the pseudocode at a more abstract level.

Intuitively, $\texttt{Union}_{\texttt{Join\_Atts}}$ is similar to the merge step in a Mergesort procedure and can be done in one pass over the data. In particular, quadtrees are bitstrings that store the tree in depth-first order. We can therefore traverse two quadtrees in parallel for merging them. Note that quadtrees are based on a regular decomposition of the space. The shape of the tree is independent of the order of the points being added.

$\texttt{Union}_{\texttt{Join\_Atts}}$ first handles the simplest case: Merging two quadtrees if (at least) one of them is empty. Then, the other one can be output as a result (Lines 4 to 7). In Lines 9 to 14, $\texttt{Union}_{\texttt{Join\_Atts}}$ considers the case that (at least) one of the trees is a set of points. Then, the trees can be merged by iterating over the set and inserting it into the other tree, point by point. The details have been illustrated for `InsertPoint` in Lines 32 to 34. Note that due to the use of `InsertPoint`, $\texttt{Union}_{\texttt{Join\_Atts}}$ eliminates duplicates, as intended. Finally, Lines 16 to 31 handle the case that both trees start with an index node. Again, this can be done recursively as, intuitively, we only need to extract the subtrees and union them. In detail, $\texttt{Union}_{\texttt{Join\_Atts}}$ iterates over the quadrants of the index node. If a quadrant is empty in both trees, the subtree that results from merging them is empty as well (Lines 18, 19). Otherwise, the entry in the index node of the result must be set to 1 (Line 21) as the subtree

exists in the result. If one of the subtrees is empty, the result from merging is the other subtree – it is simply written into the result (Lines 22 to 27). If both subtrees exist, $\texttt{Union}_{\texttt{Join\_Atts}}$ extracts the corresponding subtrees and calls itself recursively. Again, we already discussed the details of extracting a subtree in Figure B.4, Lines 15 to 17.

B.3 Summary

In this appendix, we presented the details of our compact representation which uses a spatial index to eliminate redundancy when storing a set of join-attribute tuples. In particular, we provided pseudocode for inserting join-attribute tuples into a quadtree. The code illustrates the structure of our encoding at a fine level. Finally, we substantiated our claim that set operations are easy on this structure. In particular, they can be performed in one pass over the data, without recovering the original tuples. This is a major strength of our quadtree representation.

```
1  InsertPoint(BitString P, Quadtree Q, Int[] SizeOfDim)
2
3  if (Q == ∅)
4      return '1' + P + '0';
5  NoOfDims = SizeOfDim.size();
6  //compute size of dimensions for recursive calls (split into halves)
7  SizeOfDim' = Split(SizeOfDim);
8  if (Q[0] == 0)
9      //this is an index−node; read P's quadrant at this level
10     P' = P[0, NoOfDims − 1];
11     Quadrant = P'.toInteger();
12     //set entry for P's quadrant in the index−node
13     Q[Quadrant + 1] = 1;
14     //get subtree (Q') of that quadrant
15     beg = GetPositionOfSubtree(Quadrant, Q);
16     end = GetPositionOfSubtree(Quadrant + 1, Q) − 1;
17     Q' = Q[beg, end];
18     //insert the remainder of P into subtree
19     P_rest = P[NoOfDims, P.size() − 1];
20     Q' = InsertPoint(P_rest, Q', SizeOfDim');
21     //substitute the modified subtree (compose result)
22     Q = Q[0, beg − 1] + Q' + Q[end + 1, Q.size() − 1];
23     return Q;
24 else //this is a leaf node (points)
25     //check if P is already contained (duplicate elimination)
26     for (i = 0; i < (Q.size()−1)/(P.size()+1); i++)
27         P' = Q[i · (P.size() + 1) + 1, (i + 1) · (P.size() + 1) − 1];
28         if (P' == P)
29             return Q;
30     //form an empty index node and insert all current points (P') and P
31     Q'[0, 2^NoOfDims] = 0;
32     for (i = 0; i < (Q.size()−1)/(P.size()+1); i++)
33         P' = Q[i · (P.size() + 1) + 1, (i + 1) · (P.size() + 1) − 1];
34         Q' = InsertPoint(P', Q', SizeOfDim');
35     InsertPoint(P, Q', SizeOfDim');
36     //is this shorter than simply inserting P into the list?
37     if (Q'.size() ≤ 1 + P.size() + Q.size())
38         return Q';
39     //else: insert P as a further point into the list
40     return '1' + P + Q;
```

Figure B.4: Method *InsertPoint*

```
1  GetPositionOfSubtree
2    (Int Quadrant, Quadtree Q, Int[] SizeOfDim, Int PointLength)
3
4  NoOfDims = SizeOfDim.size();
5  //compute size of dimensions for recursive calls (split into halves)
6  SizeOfDim′ = Split(SizeOfDim);
7  NoOfDims = SizeOfDim′.size();
8  PointLength′ = PointLength − NoOfDims;
9  //Precondition of GetPositionOfSubtree: Q starts with an index−node
10 //set initial offset (= result) behind index node
11 offset = 1 + 2^NoOfDims;
12 //forward offset to subtree of 'Quadrant'
13 currentQuad = 0;
14 while (currentQuad < Quadrant)
15     if (Q[currentQuad + 1] == 1)
16         //walk over subtree
17         if (Q[offset] == 0)
18             //subtree starts with an index−node
19             Q′ = Q[offset, Q.size() − 1];
20             offset += GetPositionOfSubtree(2^NoOfDims,
21                                 Q′, SizeOfDim′, PointLength′);
22         else //walk over leaf nodes
23             while (Q[offset] == 1)
24                 offset += PointLength + 1;
25             offset ++; //end of list−flag
26     currentQuad++;
27 return offset;
```

Figure B.5: Method *GetPositionOfSubtree*

```
1  Union(Quadtree Q1, Quadtree Q2)
2
3  //case: one of the trees is empty
4  if (Q1 == ∅)
5      return Q2;
6  if (Q2 == ∅)
7      return Q1;
8  //case: one of the trees consists only of points
9  if (Q1[0] == 1)
10     insert all points into Q2;
11     return Q2;
12 if (Q2[0] == 1)
13     insert all points into Q1;
14     return Q1;
15 //general case: both trees start with an index−node
16 create empty index−node Q;
17 for all quadrants q
18     if (Q1[1 + q] == 0) && (Q2[1 + q] == 0)
19         continue;
20     //one of the subtrees exists:
21     Q[1 + q] = 1;
22     if (Q1[1 + q] == 0)
23         Q = Q + subtree of quadrant q of Q2;
24         continue;
25     if (Q2[1 + q] == 0)
26         Q = Q + subtree of quadrant q of Q1;
27         continue;
28     //else ((Q1[1 + q] == 1) && (Q2[1 + q] == 1))
29     Q'1 = subtree of quadrant q of Q1;
30     Q'2 = subtree of quadrant q of Q2;
31     Q = Q + Union(Q'1, Q'2);
32 return Q;
```

Figure B.6: Method *Union*

APPENDIX B. PRIMITIVES FOR CONSTRUCTING THE QUADTREE DATASTRUCTURE

C CJF: Generalization of the Optimization Problem for Asymmetric Join Conditions

In the following, we provide the details of mapping join queries to an optimization problem, if the conditions are not symmetric. In particular, the following description generalizes the one in Section 5.4.1 to handling asymmetric as well as symmetric join conditions. As an example of a query with an asymmetric join condition, recall Query Q1 from Chapter 4. There, the join condition indicates a search for pairs of nodes with a temperature difference of more than ten degrees:

Example C.1:

```
SELECT MIN(distance(A.x, A.y, B.x, B.y))
FROM Sensors A, Sensors B
WHERE A.temp - B.temp > 10.0
SAMPLE PERIOD 30s
```
■

We already stated in Chapter 5 that if the join conditions are not symmetric, we need to know to which relation(s) a node belongs to for the mapping. To capture this knowledge, we define the sets N_A and N_B:

Definition C.1 (Set of Nodes N_A, N_B)
Let N_A be the set of nodes having a tuple $t \in A$. Analogously, let N_B be the set of nodes having a tuple $t \in B$.

Note that the sets N_A and N_B are not necessarily disjunct. For instance, for the query in Example C.1, $N_A = N_B$.

The process of mapping a query to an optimization problem is very similar to the one described in Section 5.4.1. In the following, we review the mapping and highlight the modifications that are required for the more general case.

As before, the first step is to compute N_j^{static} for each node in the query ($N_A \cup N_B$), i.e., the set of nodes that join with Node j based on the static join conditions.

The Objective Function. Recall from Section 5.4.1 that for the objective function, the join conditions affect only the collision costs. The structure of the overall cost function is unaffected – it is the sum of the costs due to each node:

$$commCost(\vec{s}) = \sum_{j \in N} commCost_j(s_j)$$

where the costs due to a Node j, $commCost_j(s_j)$, comprises the costs for sending, if the join-attribute values are outside of $filter_j$ and the costs of retrieving t_j in case of a collision:

$$commCost_j(s_j) = outFilterCost_j(s_j) + collisionCost_j(s_j)$$

$outFilterCost_j(s_j)$ does not depend on the join conditions and remains unmodified. It is the probability that t_j is not filtered multiplied with the costs of sending:

$$outFilterCost_j(s_j) = (1 - P(t_j \text{ is filtered})) \cdot cost_j$$

According to Definition 5.1, for an n-dimensional filter of size s_j, t_j is filtered if it is within the interval $[a_i, b_i] = [m_{ji} - s_j \cdot \sigma_i, m_{ji} + s_j \cdot \sigma_i]$ in each dimension i:

$$P(t_j \text{ is filtered}) = P(\bigcap_{i=1}^{n} M_{ji} \in [m_{ji} - s_j \cdot \sigma_i, m_{ji} + s_j \cdot \sigma_i])$$

As in Section 5.4.1, $collisionCost_j(s_j)$ is the probability of a collision, multiplied with the costs. The costs in this case are $2 \cdot cost_j$ as we query and send. t_j is only retrieved if it was filtered:

$$collisionCost_j(s_j) = P(t_j \text{ is filtered} \bigcap \exists h \in N_j^{static} : t_h \text{ collides with } filter_j) \cdot 2 \cdot cost_j$$

where the existence resolves to

$$P(\exists h \in N_j^{static} : t_h \text{ collides with } filter_j) = P(\bigcup_{h \in N_j^{static}} t_h \text{ collides with } filter_j)$$

Computing $P(t_h \text{ collides with } filter_j)$ is slightly different in the general case: The idea of computing $P(t_h \text{ collides with } filter_j)$ is still to identify the subspace of the join-attribute space that collides with $filter_j$ and to compute the probability of t_h being in it. However, as indicated in Section 5.4.1, we now distinguish two cases: $j \in N_A$ and $j \in N_B$. For each case, we compute the subspace separately. We compute the $subspace^A$ of tuples that join with any value in $filter_j$, if j is in N_A (and thus, $t_h \in B$). $subspace^B$ is that subspace of the join-attribute space that collides with $filter_j$ if $j \in N_B$ ($t_h \in A$). If $j \notin N_A$ or $j \notin N_B$, the corresponding subspace is empty – this case cannot cause collisions. A collision of $filter_j$ with t_h occurs if t_h is either in $subspace_i^A$ or in $subspace_i^B$. As a consequence, $P(t_h \text{ collides with } filter_j)$ is computed as the probability of t_h being in either of those subspaces:

$$P(t_h \text{ collides with } filter_j) = P((\bigcap_{i=1}^{n} M_{hi} \in subspace_i^A) \bigcup (\bigcap_{i=1}^{n} M_{hi} \in subspace_i^B))$$

The subspaces are computed as described in Section 5.4.1: Due to the conjunctions in the query, each join condition further narrows the subspace of tuples that collide with $filter_j$. Thus, CJF computes the subspace by iterating over the join conditions. *For each join condition* in the query, CJF takes the current subspace and restricts it.

Example C.2: In Example C.1, the join condition is $A.temp - B.temp > 10.0$. This condition restricts dimension i ('temperature') of $subspace^A$ (i.e., $j \in A, h \in B$) to all values that are at least 10 smaller than any value in $filter_j$ ($[m_{ji} - s_j \cdot \sigma_i, m_{ji} + s_j \cdot \sigma_i]$); to have a collision, $M_{hi} \in (-\infty, m_{ji} + s_j \cdot \sigma_i - 10.0]$.

For $subspace^B$ (i.e., $j \in B, h \in A$), t_h collides with $filter_j$ if its temperature is at least 10 degree higher than any value in $filter_j$; to have a collision, $M_{hi} \in [m_{ji} - s_j \cdot \sigma_i + 10.0, \infty)$. ■

Constraints. Again, filter sizes must be non-negative and we need constraints to avoid colliding filters. In Section 5.4.1, we formalized the latter constraint by means of the $slack_{jh}(s_j, s_h)$ functions. $slack_{jh}(s_j, s_h) = 0$ iff $filter_j$ and $filter_h$ collide.

To cope with asymmetric join conditions, the solution for the constraints is conceptually the same as the solution for the objective function: We distinguish two cases: $j \in N_A$ and $j \in N_B$. For each case, we set up a separate slack function. The resulting optimization problem is then:

$$\begin{array}{ll} \text{minimize} & commCost(\vec{s}) \\ \text{subject to} & (slack_{jh}^A(s_j, s_h) > 0) \vee (s_j = s_h = 0), \forall j \in N_A \forall h \in (N_j^{static} \cap N_B) \\ & (slack_{jh}^B(s_j, s_h) > 0) \vee (s_j = s_h = 0), \forall j \in N_B \forall h \in (N_j^{static} \cap N_A) \\ & s_j \geq 0, \forall j \in N \end{array} \tag{C.1}$$

The structure of the slack functions is the same as in Section 5.4.1: For $X \in \{A, B\}$, $slack_{jh}^X(s_j, s_h) = \sum_{i=1}^{p} slack_{jh}^{Xi}(s_j, s_h)$. There is one $slack_{jh}^{Xi}$ for each of the p join conditions in the query. The form of $slack_{jh}^{Xi}$ is specific to the join condition, but the principle is always the same: $slack_{jh}^{Xi}$ becomes zero if any pair of tuples $t_j \in filter_j$ and $t_h \in filter_h$ fulfills the join condition, and it is greater zero otherwise.

Example C.3: For the join condition of Example C.1, $A.temp - B.temp > 10.0$, Figure C.1 shows the slack function. $slack_{jh}^{Ai}$ covers the case that $j \in N_A$ and $h \in N_B$ as indicated in Equation (C.1). Intuitively, the function takes the largest value in $filter_j$ $(m_{ji} + s_j \cdot \sigma_i)$ and computes its distance to the smallest value in $filter_h$

$$slack_{jh}^{Ai}(s_j, s_h) = \begin{cases} [(m_{ji} + s_j \cdot \sigma_i) - (m_{hi} - s_h \cdot \sigma_i) - d_i^{min} + \epsilon_i]^2, \\ \qquad \text{if } (m_{ji} + s_j \cdot \sigma_i) - (m_{hi} - s_h \cdot \sigma_i) - d_i^{min} + \epsilon_i > 0 \\ 0, \text{else} \end{cases}$$

$$slack_{jh}^{Bi}(s_j, s_h) = \begin{cases} [(m_{hi} + s_h \cdot \sigma_i) - (m_{ji} - s_j \cdot \sigma_i) - d_i^{min} + \epsilon_i]^2, \\ \qquad \text{if } (m_{hi} + s_h \cdot \sigma_i) - (m_{ji} - s_j \cdot \sigma_i) - d_i^{min} + \epsilon_i > 0 \\ 0, \text{else} \end{cases}$$

Figure C.1: Continuously differentiable functions $slack_{jh}^{Ai}$ and $slack_{jh}^{Bi}$ for the join condition $A.att_i - B.att_i > d_i^{min}$

$(m_{hi} - s_h \cdot \sigma_i)$. If the distance is more than d_i^{min}, i.e., 10.0°C in our example, then the filters collide. For $slack_{jh}^{Bi}$ ($j \in N_A$ and $h \in N_B$), the filters collide if the smallest value in $filter_j$ $(m_{ji} - s_j \cdot \sigma_i)$ is more than d_i^{min} away from the largest value in $filter_h$ $(m_{hi} + s_h \cdot \sigma_i)$. As in Section 5.4.1, we included the ϵ_i in the function which slightly sharpen the join condition, as motivated at the end of Section 5.4. ■

Finally, for symmetric join conditions, the slack functions are as described in Section 5.4.1. In particular, in this case, $slack_{jh}^{Ai} = slack_{jh}^{Bi}$.

Conclusion. We illustrated the idea of "distinguishing between $t_j \in A$ and $t_j \in B$" in the optimization problem for asymmetric join conditions, which we mentioned in Section 5.4.1. In particular, the mapping presented here is the most general and can handle any join query subject to our problem statement. It subsumes the one presented in Section 5.4.1, i.e., it can handle symmetric as well as asymmetric join conditions.

Bibliography

[ACBL05] Jugoslava Acimovic, Razvan Cristescu, and Baltasar Beferull-Lozano. Efficient Distributed Multiresolution Processing for Data Gathering in Sensor Networks. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '05)*, pages 837–840, March 2005.

[AML05] Daniel J. Abadi, Samuel Madden, and Wolfgang Lindner. REED: Robust, Efficient Filtering and Event Detection in Sensor Networks. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*, pages 769–780, August 2005.

[Baj88] Chanderjit Bajaj. The Algebraic Degree of Geometric Optimization Problems. *Discrete and Computational Geometry*, 3(2):177–191, 1988.

[BB04] Boris Jan Bonfils and Philippe Bonnet. Adaptive and Decentralized Operator Placement for In-Network Query Processing. *Telecommunication Systems*, 26:389–409, June 2004.

[BCG01] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. STHoles: A Multidimensional Workload-Aware Histogram. *SIGMOD Record*, 30(2):211–222, 2001.

[BEF+00] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in Network Simulation. *Computer*, 33(5):59–67, 2000.

[BGW+81] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie, Jr. Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems*, 6(4):602–625, 1981.

[Blo70] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[CDHH06] David Chu, Amol Deshpande, Joseph M. Hellerstein, and Wei Hong. Approximate Data Collection in Sensor Networks Using Probabilistic Models. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE '06)*, page 48, April 2006.

[CDSY98] A. R. Calderbank, Ingrid Daubechies, Wim Sweldens, and Boon-Lock Yeo. Wavelet Transforms that Map Integers to Integers. *Applied and Computational Harmonic Analysis*, 5:332–369, 1998.

[CEH+01] Alberto Cerpa, Jeremy Elson, Michael Hamilton, Jerry Zhao, Deborah Estrin, and Lewis Girod. Habitat Monitoring: Application Driver for Wireless Communications Technology. In *Workshop on Data Communication in Latin America and the Caribbean (SIGCOMM LA '01)*, pages 20–41, 2001.

[CG05] Vishal Chowdhary and Himanshu Gupta. Communication-Efficient Implementation of Join in Sensor Networks. In *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA '05)*, pages 447–460, April 2005.

[CGRS00] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate Query Processing Using Wavelets. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB 2000)*, pages 111–122, September 2000.

[CLKB04] Jeffrey Considine, Feifei Li, George Kollios, and John Byers. Approximate Aggregation Techniques for Sensor Databases. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*, pages 449–460, April 2004.

[CN07] Alexandru Coman and Mario A. Nascimento. A Distributed Algorithm for Joins in Sensor Networks. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management (SSDBM '07)*, page 27, July 2007.

[CNS07] Alexandru Coman, Mario A. Nascimento, and Jörg Sander. On Join Location in Sensor Networks. In *Proceedings of the 2007 International Conference on Mobile Data Management (MDM '07)*, pages 190–197, May 2007.

[CO05] Alexandre Ciancio and Antonio Ortega. A Distributed Wavelet Compression Algorithm for Wireless Multihop Sensor Networks Using Lifting. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '05)*, pages 825–828, March 2005.

[Cou] http://cougar.cs.cornell.edu.

[CV00] S. Grace Chang and Martin Vetterli. Adaptive Wavelet Thresholding for Image Denoising and Compression. *IEEE Transactions on Image Processing*, 9(9):1532–1546, 2000.

[Daw98] Todd E. Dawson. Fog in the California Redwood Forest: Ecosystem Inputs and Use by Plants. *Oecologia*, 117(4):476–485, 1998.

[DBcF07] Xuan Thanh Dang, Nirupama Bulusu, and Wu chi Feng. RIDA: A Robust Information-Driven Data Compression Architecture for Irregular Wireless Sensor Networks. In *Proceedings of the 4th European Conference on Wireless Sensor Networks (EWSN '07)*, pages 133–149, January 2007.

[DGM+04] Amol Deshpande, Carlos Guestrin, Samuel R. Madden, Joseph M. Hellerstein, and Wei Hong. Model-Driven Data Acquisition in Sensor Networks. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB '04)*, pages 588–599, September 2004.

[DGR03] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate Join Processing Over Data Streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*, pages 40–51, 2003.

[DGR07] Antonios Deligiannakis, Minos Garofalakis, and Nick Roussopoulos. Extended Wavelets for Multiple Measures. *ACM Transactions on Database Systems (TODS)*, 32(2):10, 2007.

[DKR04a] Antonios Deligiannakis, Yannis Kotidis, and Nick Roussopoulos. Compressing Historical Information in Sensor Networks. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*, pages 527–538, June 2004.

[DKR04b] Antonios Deligiannakis, Yannis Kotidis, and Nick Roussopoulos. Hierarchical In-Network Data Aggregation with Quality Guarantees. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT '04)*, pages 658–675, March 2004.

[EN07] Ramez A. Elmasri and Shankrant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, fifth edition, 2007.

[ESW78] Robert Epstein, Michael Stonebraker, and Eugene Wong. Distributed Query Processing in a Relational Data Base System. In *Proceedings of*

the 1978 ACM SIGMOD International Conference on Management of Data (SIGMOD '78), pages 169–180, May 1978.

[Fee01] Laura Marie Feeney. An Energy Consumption Model for Performance Analysis of Routing Protocols for Mobile Ad Hoc Networks. *Mobile Networks and Applications*, 6(3):239–249, 2001.

[GA] Jean Gailly and Mark Adler. zlib. http://www.zlib.net.

[GBT+04] Carlos Guestrin, Peter Bodik, Romain Thibaux, Mark Paskin, and Samuel Madden. Distributed Regression: an Efficient Framework for Modeling Sensor Network Data. In *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks (IPSN '04)*, pages 1–10, April 2004.

[GG98] Volker Gaede and Oliver Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[GG02] Minos Garofalakis and Phillip B. Gibbons. Wavelet Synopses With Error Guarantees. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*, pages 476–487, June 2002.

[GGP+03] Deepak Ganesan, Ben Greenstein, Denis Perelyubskiy, Deborah Estrin, and John Heidemann. An Evaluation of Multi-Resolution Storage for Sensor Networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*, pages 89–102, November 2003.

[GH05] Sudipto Guha and Boulos Harb. Wavelet Synopsis for Data Streams: Minimizing Non-Euclidean Error. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '05)*, pages 88–97, August 2005.

[GH06] Sudipto Guha and Boulos Harb. Approximation Algorithms for Wavelet Transform Coding of Data Streams. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm (SODA '06)*, pages 698–707, January 2006.

[GJP+06] Omprakash Gnawali, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. The Tenet Architecture for Tiered Sensor Networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*, pages 153–166, October 2006.

[GK04] Minos Garofalakis and Amit Kumar. Deterministic Wavelet Thresholding for Maximum-Error Metrics. In *Proceedings of the Twenty-Third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '04)*, pages 166–176, June 2004.

[GKMS01] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin Strauss. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*, pages 79–88, September 2001.

[HAE08] Moustafa A. Hammad, Walid G. Aref, and Ahmed K. Elmagarmid. Query Processing of Multi-Way Stream Window Joins. *The VLDB Journal*, 17(3):469–488, 2008.

[HHMS03] Joseph M. Hellerstein, Wei Hong, Samuel Madden, and Kyle Stanek. Beyond Average: Toward Sophisticated Sensing with Queries. In *Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks (IPSN '03)*, pages 63–79, April 2003.

[Hin06] Haitham Hindi. A Tutorial on Convex Optimization II: Duality and Interior Point Methods. In *Proceedings of the American Control Conference (ACC '06)*, June 2006.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. *SIGPLAN Notices*, 35(11):93–104, 2000.

[Huf52] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[HW04] Joseph M. Hellerstein and Wei Wang. Optimization of In-Network Data Reduction. In *Proceedings of the 1st International Workshop on Data Management for Sensor Networks (DMSN '04)*, pages 40–47, August 2004.

[Int] http://db.csail.mit.edu/labdata/labdata.html.

[JCW04] Ankur Jain, Edward Y. Chang, and Yuan-Fang Wang. Adaptive Stream Resource Management Using Kalman Filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*, pages 11–22, June 2004.

[JKM+07] Navendu Jain, Dmitry Kit, Prince Mahajan, Praveen Yalagandula, Mike Dahlin, and Yin Zhang. STAR: Self-Tuning Aggregation for Scalable Monitoring. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, pages 962–973, September 2007.

[JW02] Richard A. Johnson and Dean W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, fifth edition, 2002.

[KK00] Brad Karp and H. T. Kung. GPSR: Greedy Perimeter Stateless Routing for Wireless Networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom '00)*, pages 243–254, 2000.

[KM05] Panagiotis Karras and Nikos Mamoulis. One-Pass Wavelet Synopses for Maximum-Error Metrics. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*, pages 421–432, September 2005.

[KNV03] Jaewoo Kang, Jeffrey F. Naughton, and Stratis D. Viglas. Evaluating Window Joins over Unbounded Streams. In *Proceedings of the 19th International Conference on Data Engineering (ICDE '03)*, pages 341–352, March 2003.

[Kot05] Yannis Kotidis. Snapshot Queries: Towards Data-Centric Sensor Networks. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, pages 131–142, April 2005.

[Kuh74] Harold W. Kuhn. Steiner's Problem Revisited. In *G. B. Dantzig and B. C. Eaves, editors, Studies in Optimization. Studies in Mathematics, 10, The Mathematical Association of America*, pages 52–70, Washington, DC, 1974.

[LM03] Iosif Lazaridis and Sharad Mehrotra. Capturing Sensor-Generated Time Series with Quality Guarantees. In *Proceedings of the 19th International Conference on Data Engineering (ICDE '03)*, page 429, March 2003.

[LST91] Hongjun Lu, Ming-Chien Shan, and Kian-Lee Tan. Optimization of Multi-Way Join Queries for Parallel Execution. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB '91)*, pages 549–560, 1991.

[Mad03] Samuel Ross Madden. *The Design and Evaluation of a Query Processing Architecture for Sensor Networks*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 2003.

[Mar63] Donald W. Marquardt. An Algorithm for Least-Squares Estimation of Nonlinear Parameters. *SIAM Journal on Applied Mathematics*, 11(2):431–441, June 1963.

[MCP+02] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '02)*, pages 88–97, 2002.

[Mel61] Z. A. Melzak. On the Problem of Steiner. *Canadian Mathematical Bulletin*, 4(2):143–148, 1961.

[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 the 5th Symposium on Operating System Design and Implementation (OSDI 2002)*, December 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 (SIGMOD '03)*, pages 491–502, June 2003.

[Mic] Sun Microsystems. Sun SPOT World. http://www.sunspotworld.com/.

[MNW98] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic Coding Revisited. *ACM Transactions on Information Systems (TOIS)*, 16(3):256–294, 1998.

[Mur07] Fionn Murtagh. The Haar Wavelet Transform of a Dendrogram. *Journal of Classification*, 24(1):3–32, 2007.

[MVW98] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-Based Histograms for Selectivity Estimation. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD '98)*, pages 448–459, June 1998.

[MVW00] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Dynamic Maintenance of Wavelet-Based Histograms. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB '00)*, pages 101–110, September 2000.

[NGSA08] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis Diffusion for Robust Aggregation in Sensor Networks. *ACM Transactions on Sensor Networks (TOSN)*, 4(2):1–40, 2008.

[NW06] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer Verlag, second edition, 2006.

[OJW03] Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive Filters for Continuous Queries Over Distributed Data Streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03)*, pages 563–574, June 2003.

[OLW01] Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive Precision Setting for Cached Approximate Values. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD '01)*, pages 355–366, May 2001.

[OV99] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, second edition, 1999.

[PG06] Aditi Pandit and Himanshu Gupta. Communication-Efficient Implementation of Range-Joins in Sensor Networks. In *Proceedings of the 11th International Conference on Database Systems for Advanced Applications (DASFAA '06)*, pages 859–869, April 2006.

[Pha] SensiNet Enables FDA-Compliant Temperature Monitoring and Data Collection. http://www.sensicast.com/uploadedFiles/CS-Pharma.10.06_casestudy.pdf.

[PHC04] Joseph Polastre, Jason Hill, and David E. Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, pages 95–107, November 2004.

[RD08] Florin Rusu and Alin Dobra. Sketches for Size of Join Estimation. *ACM Transactions on Database Systems*, 33(3):1–46, 2008.

[RK91] N. Roussopoulos and H. Kang. A Pipeline N-Way Join Algorithm Based on the 2-Way Semijoin Program. *IEEE Transactions on Knowledge and Data Engineering*, 3(4):486–495, 1991.

[RLM87] James P. Richardson, Hongjun Lu, and Krishna Mikkilineni. Design and Evaluation of Parallel Pipelined Join Algorithms. *SIGMOD Record*, 16(3):399–409, 1987.

[SA80] Patricia G. Selinger and Michel E. Adiba. Access Path Selection in Distributed Database Management Systems. In *Proceedings of the International Conference on Data Bases (ICOD '80)*, pages 204–215, July 1980.

[Sam84] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[Say00] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, second edition, 2000.

[SBB08] Mirco Stern, Erik Buchmann, and Klemens Böhm. Where in the Sensor Network Should the Join Be Computed, After All? In *Proceedings of the International Workshop on Ubiquitous Knowledge Discovery (UKD)*, September 2008.

[SBB09a] Mirco Stern, Erik Buchmann, and Klemens Böhm. A Wavelet Transform for Efficient Consolidation of Sensor Relations with Quality Guarantees. In *Proceedings of the 35th International Conference on Very Large Data Bases (VLDB '09)*, August 2009.

[SBB09b] Mirco Stern, Erik Buchmann, and Klemens Böhm. Towards Efficient Processing of General-Purpose Joins in Sensor Networks. In *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE '09)*, pages 126–137, March 2009.

[SBB10] Mirco Stern, Erik Buchmann, and Klemens Böhm. Processing Continuous Join Queries in Sensor Networks: a Filtering Approach. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*, June 2010.

[SDS96] Eric J. Stollnitz, Tony D. Derose, and David H. Salesin. *Wavelets for Computer Graphics: Theory and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[SDS09] Dimitris Sacharidis, Antonios Deligiannakis, and Timos K. Sellis. Hierarchically Compressed Wavelet Synopses. *VLDB Journal*, 18:203–231, 2009.

[Sena] http://www.sensicast.com/.

[Senb] Sensorscope. http://sensorscope.epfl.ch/.

[Senc] http://sensorscope.epfl.ch/index.php/Environmental_Data.

[Sew] Julian Seward. bzip2. http://www.bzip.org.

[SM97] Wolfgang Scheufele and Guido Moerkotte. On the Complexity of Generating Optimal Plans With Cross Products (Extended Abstract). In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '97)*, pages 238–248, May 1997.

[SM06] Christopher M. Sadler and Margaret Martonosi. Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*, pages 265–278, October 2006.

[Tec] Crossbow Technology. Mica2 Wireless Measurement System. http://www.xbow.com/ products/ Product_pdf_files/ Wireless_pdf/ MICA2 _Datasheet.pdf.

[Tin] http://telegraph.cs.berkeley.edu/tinydb.

[TL03] Gregory P. Tollisen and Tamas Lengyel. On Minimizing Distance by the Road Less Travelled. *Elemente der Mathematik*, 58(3):89–107, 2003.

[TM06a] Daniela Tulone and Samuel Madden. An Energy-Efficient Querying Framework in Sensor Networks for Detecting Node Similarities. In *Proceedings of the 9th ACM International Symposium on Modeling Analysis and Simulation of Wireless and Mobile Systems (MSWiM '06)*, pages 191–300, October 2006.

[TM06b] Daniela Tulone and Samuel Madden. PAQ: Time Series Forecasting for Approximate Query Answering in Sensor Networks. In *Proceedings of the 3rd European Workshop on Wireless Sensor Networks (EWSN '06)*, pages 21–37, February 2006.

[TYD+05] Niki Trigoni, Yong Yao, Alan J. Demers, Johannes Gehrke, and Rajmohan Rajaraman. Multi-query Optimization for Sensor Networks. In *Proceedings of the 1st IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS '05)*, pages 307–321, July 2005.

[VW99] Jeffrey Scott Vitter and Min Wang. Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD '99)*, pages 193–204, June 1999.

[WBD+06] Raymond S. Wagner, Richard G. Baraniuk, Shu Du, David B. Johnson, and Albert Cohen. An Architecture for Distributed Wavelet Analysis and Processing in Sensor Networks. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks (IPSN '06)*, pages 243–250, April 2006.

[WC01] Alec Woo and David E. Culler. A Transmission Control Scheme for Media Access in Sensor Networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking (MobiCom '01)*, pages 221–235, 2001.

[Wer] Jay Werb. Making Sense of the Sensor Network Value Chain. http://www.sensicast.com/ uploadedFiles/ Resource_Center/ Making _Sense _of _the _Sensor _Network _Value _Chain.pdf.

[WTC03] Alec Woo, Terence Tong, and David Culler. Taming the Underlying Challenges of Reliable Multihop Routing in Sensor Networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems (SenSys '03)*, pages 14–27, November 2003.

[XLTZ07] Shili Xiang, Hock Beng Lim, Kian-Lee Tan, and Yongluan Zhou. Two-Tier Multiple Query Optimization for Sensor Networks. In *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS '07)*, page 39, June 2007.

[YG03] Yong Yao and Johannes Gehrke. Query Processing for Sensor Networks. In *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR '03)*, January 2003.

[YHE04] Wei Ye, John Heidemann, and Deborah Estrin. Medium Access Control With Coordinated Adaptive Sleeping for Wireless Sensor Networks. *IEEE/ACM Transactions on Networking*, 12(3):493–506, 2004.

[YLOT07] Xiaoyan Yang, Hock Beng Lim, Tamer M. Özsu, and Kian Lee Tan. In-Network Execution of Monitoring Queries in Sensor Networks. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*, pages 521–532, June 2007.

[YLZ06] 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, May 2006.

[YMB07] Man Lung Yiu, Nikos Mamoulis, and Spiridon Bakiras. Retrieval of Spatial Join Pattern Instances from Sensor Networks. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management (SSDBM '07)*, page 25, July 2007.

[YSH06] Wei Ye, Fabio Silva, and John Heidemann. Ultra-Low Duty Cycle MAC With Scheduled Channel Polling. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*, pages 321–334, Boulder, Colorado, USA, October 2006.

[ZGT09] Xianjin Zhu, Himanshu Gupta, and Bin Tang. Join of Multiple Data Streams in Sensor Networks. *IEEE Transactions on Knowledge and Data Engineering*, 99(1), January 2009.

[ZLW+06] Siwang Zhou, Yaping Lin, Jiliang Wang, Jianming Zhang, and Jing cheng Ouyang. Compressing Spatial and Temporal Correlated Data in Wireless Sensor Networks Based on Ring Topology. In *Proceedings of the 7th International Conference on Advances in Web-Age Information Management (WAIM '06)*, pages 337–348, June 2006.