QUALITY OF SERVICE AWARE OPTIMIZATION OF SENSOR NETWORK QUERIES

Citation for published version (APA):

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester’s Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.
QUALITY OF SERVICE AWARE
OPTIMIZATION OF SENSOR
NETWORK QUERIES

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2010

Ixent Galpin
School of Computer Science
Contents

Abstract 14
Declaration 15
Copyright 16
The Author 17
Acknowledgements 18
Glossary 21

1 Introduction 22
   1.1 Setting the Scene 22
   1.2 Fundamentals 25
   1.3 Motivation 28
   1.4 Aim, Objectives and Research Contributions 30
   1.5 Technical Approach 32
   1.6 A Guide for Readers 36

2 Background 38
   2.1 Query Processing 39
      2.1.1 Classical Query Processing 39
      2.1.2 Distributed Query Processing 43
      2.1.3 Stream Query Processing 46
      2.1.4 Adaptive Query Processing 48
   2.2 Sensor Networks 49
   2.3 QoS in Sensor Network Applications 54
   2.4 Conclusion 59
3 Related Work 62
3.1 Framework for Comparison 63
3.1.1 QoS Expectations 63
3.1.2 Query Optimization 64
3.2 The State of the Art in Sensor Network Query Processing 66
3.3 QoS-awareness in Other Modes of Query Processing 76
3.4 Incorporating Alternative Decision-making Policies into Query Optimizers 80
3.5 Conclusion 84

4 The SNEE Query Processing Stack 85
4.1 Query Processing Stack Overview 87
4.2 Query Language 88
4.3 QoS Expectations 92
4.4 Metadata 94
4.5 Query Processing Stack Design Issues 98
4.6 Characterization of Query Stack Steps 102
4.6.1 Single-site Phase 102
4.6.2 Routing 104
4.6.3 Where-scheduling 105
4.6.4 When-scheduling 108
4.7 Conclusion 111

5 A Fixed Goal Instantiation of SNEE 114
5.1 Routing 115
5.2 Where-Scheduling 119
5.3 When-Scheduling 126
5.4 Conclusion 130

6 A QoS-aware Instantiation of SNEE 132
6.1 Technical Approach 133
6.2 Routing 134
6.2.1 Algorithm Description 135
6.2.2 Example Output 146
6.2.3 Discussion 149
6.3 Where-Scheduling 151
The main text of this dissertation contains 62,728 words including footnotes and endnotes. The appendices contain a further 1,608 words.
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Some example logical-algebraic query operators.</td>
<td>40</td>
</tr>
<tr>
<td>2.2</td>
<td>Characteristics of some example sensor network hardware.</td>
<td>53</td>
</tr>
<tr>
<td>2.3</td>
<td>Characteristics of existing sensor network deployments.</td>
<td>57</td>
</tr>
<tr>
<td>3.1</td>
<td>Definition of QoS variables considered.</td>
<td>63</td>
</tr>
<tr>
<td>3.2</td>
<td>Range of QoS expectations purportedly supported by SNQPs.</td>
<td>68</td>
</tr>
<tr>
<td>3.3</td>
<td>Approaches to making query planning decisions.</td>
<td>69</td>
</tr>
<tr>
<td>4.1</td>
<td>Some example SNEEq operators.</td>
<td>104</td>
</tr>
<tr>
<td>6.1</td>
<td>The heuristics used to generate routing trees.</td>
<td>137</td>
</tr>
<tr>
<td>7.1</td>
<td>QoS expectations used for FG-SNEE in the experiments.</td>
<td>194</td>
</tr>
<tr>
<td>7.2</td>
<td>QoS expectations used for QoS-SNEE in the experiments.</td>
<td>196</td>
</tr>
<tr>
<td>A.1</td>
<td>The SNEEq physical algebra.</td>
<td>216</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An example logical query execution plan.</td>
<td>41</td>
</tr>
<tr>
<td>2.2</td>
<td>An example physical query execution plan.</td>
<td>42</td>
</tr>
<tr>
<td>2.3</td>
<td>The two-phase optimization approach for DQP.</td>
<td>44</td>
</tr>
<tr>
<td>2.4</td>
<td>An example distributed query execution plan.</td>
<td>45</td>
</tr>
<tr>
<td>2.5</td>
<td>Components of a sensor node.</td>
<td>50</td>
</tr>
<tr>
<td>2.6</td>
<td>Example sensor network hardware.</td>
<td>52</td>
</tr>
<tr>
<td>3.1</td>
<td>Some example TinyDB queries.</td>
<td>68</td>
</tr>
<tr>
<td>4.1</td>
<td>The SNEE query processing stack.</td>
<td>87</td>
</tr>
<tr>
<td>4.2</td>
<td>Example SNEEql schema definition. The keyword <code>sensed</code> is used to denote that these are acquisitional streams.</td>
<td>90</td>
</tr>
<tr>
<td>4.3</td>
<td>Some example queries in SNEEql.</td>
<td>91</td>
</tr>
<tr>
<td>4.4</td>
<td>Some example QoS expectations for SNEE.</td>
<td>92</td>
</tr>
<tr>
<td>4.5</td>
<td>An example sensor network.</td>
<td>95</td>
</tr>
<tr>
<td>4.6</td>
<td>An example SNEE physical schema.</td>
<td>97</td>
</tr>
<tr>
<td>4.7</td>
<td>The Physical-Algebraic Form for Q4.</td>
<td>102</td>
</tr>
<tr>
<td>4.8</td>
<td>An example routing tree for Q1, superimposed on the network topology in Figure 4.5.</td>
<td>105</td>
</tr>
<tr>
<td>4.9</td>
<td>An example Distributed-Algebraic Form for Q1.</td>
<td>107</td>
</tr>
<tr>
<td>4.10</td>
<td>An example Compact Distributed-Algebraic Form for Q1.</td>
<td>108</td>
</tr>
<tr>
<td>4.11</td>
<td>An example agenda for Q1.</td>
<td>109</td>
</tr>
<tr>
<td>4.12</td>
<td>The structure and properties of SNEE agendas.</td>
<td>110</td>
</tr>
<tr>
<td>5.1</td>
<td>The Physical-Algebraic Form for Q3.</td>
<td>116</td>
</tr>
<tr>
<td>5.2</td>
<td>The FG-SNEE ROUTING algorithm.</td>
<td>117</td>
</tr>
<tr>
<td>5.3</td>
<td>The routing tree for Q3 generated by FG-SNEE.</td>
<td>118</td>
</tr>
<tr>
<td>5.4</td>
<td>The FRAGMENT-DEFINITION algorithm.</td>
<td>120</td>
</tr>
<tr>
<td>5.5</td>
<td>The FRAGMENT-INSTANCE-ASSIGNMENT algorithm.</td>
<td>121</td>
</tr>
</tbody>
</table>
F.12 \( \lambda \) with \( \alpha = 30s \) for FG-SNEE vs. TDB-SNEE .......................... 258
F.13 \( \lambda \) with \( \alpha = 60s \) for FG-SNEE vs. TDB-SNEE .......................... 258
F.14 \( \lambda \) with \( \alpha = 150s \) for FG-SNEE vs. TDB-SNEE .......................... 259
F.15 \( \lambda \) with \( \alpha = 300s \) for FG-SNEE vs. TDB-SNEE .......................... 259
F.16 \( \lambda \) with \( \alpha = 600s \) for FG-SNEE vs. TDB-SNEE .......................... 260
F.17 \( \lambda \) with \( \alpha = 1000s \) for FG-SNEE vs. TDB-SNEE .......................... 260
F.18 \( \lambda \) with \( \alpha = 3000s \) for FG-SNEE vs. TDB-SNEE .......................... 261
F.19 \( \beta \) with \( \alpha = 3s \) for FG-SNEE vs. TDB-SNEE .......................... 261
F.20 \( \beta \) with \( \alpha = 15s \) for FG-SNEE vs. TDB-SNEE .......................... 262
F.21 \( \beta \) with \( \alpha = 30s \) for FG-SNEE vs. TDB-SNEE .......................... 262
F.22 \( \beta \) with \( \alpha = 60s \) for FG-SNEE vs. TDB-SNEE .......................... 263
F.23 \( \beta \) with \( \alpha = 150s \) for FG-SNEE vs. TDB-SNEE .......................... 263
F.24 \( \beta \) with \( \alpha = 300s \) for FG-SNEE vs. TDB-SNEE .......................... 264
F.25 \( \beta \) with \( \alpha = 600s \) for FG-SNEE vs. TDB-SNEE .......................... 264
F.26 \( \beta \) with \( \alpha = 1000s \) for FG-SNEE vs. TDB-SNEE .......................... 265
F.27 \( \beta \) with \( \alpha = 3000s \) for FG-SNEE vs. TDB-SNEE .......................... 265
F.28 \( \delta \) with \( \alpha = 3s \) for FG-SNEE vs. TDB-SNEE .......................... 266
F.29 \( \delta \) with \( \alpha = 15s \) for FG-SNEE vs. TDB-SNEE .......................... 266
F.30 \( \delta \) with \( \alpha = 30s \) for FG-SNEE vs. TDB-SNEE .......................... 267
F.31 \( \delta \) with \( \alpha = 60s \) for FG-SNEE vs. TDB-SNEE .......................... 267
F.32 \( \delta \) with \( \alpha = 150s \) for FG-SNEE vs. TDB-SNEE .......................... 268
F.33 \( \delta \) with \( \alpha = 300s \) for FG-SNEE vs. TDB-SNEE .......................... 268
F.34 \( \delta \) with \( \alpha = 600s \) for FG-SNEE vs. TDB-SNEE .......................... 269
F.35 \( \delta \) with \( \alpha = 1000s \) for FG-SNEE vs. TDB-SNEE .......................... 269
F.36 \( \delta \) with \( \alpha = 3000s \) for FG-SNEE vs. TDB-SNEE .......................... 270

G.1 \( \alpha \) with QoS 1 for QoSA-SNEE ................................................. 271
G.2 \( \alpha \) with QoS 4 for QoSA-SNEE ................................................. 272
G.3 \( \alpha \) with QoS 5 for QoSA-SNEE ................................................. 272
G.4 \( \alpha \) with QoS 6 for QoSA-SNEE ................................................. 273
G.5 \( \alpha \) with QoS 7 for QoSA-SNEE ................................................. 273
G.6 \( \alpha \) with QoS 8 for QoSA-SNEE ................................................. 274
G.7 \( \delta \) with QoS 1 for QoSA-SNEE ................................................. 274
G.8 \( \delta \) with QoS 2 for QoSA-SNEE ................................................. 275
G.9 \( \delta \) with QoS 3 for QoSA-SNEE ................................................. 275
G.10 \( \delta \) with QoS 4 for QoSA-SNEE ................................................. 276
H.10 Agenda for scenario 6 compiled against QoS 1. . . . . . . . . . . . 315
H.11 Agenda for scenario 6 compiled against QoS 6. . . . . . . . . . . . 316
H.12 Agenda for scenario 6 compiled against QoS 10. . . . . . . . . . . 317
Abstract

Sensor networks comprise resource-constrained wireless nodes with the capability of gathering information about their surroundings and have recently risen to prominence with the promise of being an effective computing platform for diverse applications, ranging from event detection to environmental monitoring. The database community proposed the use of sensor network query processors (SNQPs) as means to meet data collection requirements using a declarative query language. Declarative queries posed against a sensor network constitute an effective means to repurpose sensor networks and reduce the high software development costs associated with them.

The range of sensor network applications is very broad. Such applications have diverse, and often conflicting, QoS expectations in terms of the delivery time of results, the acquisition interval at which data is collected, the total energy consumption of the deployment, or the network lifetime. The conflicting nature of these desiderata is aggravated by the resource-constrained nature of sensor networks as a computing fabric, making it particularly challenging to reconcile the trade-offs that arise. Previously, SNQPs have been focused on evaluating queries as energy-efficiently as possible. There has been comparatively less work on attempting to meet a broad range of optimization goals and constraints that captured these QoS expectations. In this respect, previous work in SNQP has not aimed at being general purpose across the breadth of applications to which sensor networks have been applied.

This PhD dissertation presents an approach for enabling QoS-awareness in SNQPs so that query evaluation plans are generated that exhibit good performance for a broader range of sensor network applications in terms of their QoS expectations. The research contributions reported here include (a) a functional decomposition of the decision-making steps required to compile a declarative query into a query evaluation plan in a sensor network setting; (b) algorithms to implement these decision-making steps; and (c) an empirical evaluation to show the benefits of QoS-awareness compared to a representative fixed-goal SNQP.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://www.campus.manchester.ac.uk/medialibrary/policies/intellectual-property.pdf), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses.
The Author

Ixent Galpin completed a BA Joint Honours in Computer Science and Spanish at the University of Bristol in 2000. He worked as a software engineer at Reuters until 2004, when he commenced a MSc in Advanced Computer Science at the University of Manchester. His MSc project involved enhancing the temporal analysis capabilities of the Tripod spatio-temporal query processor [GFPB04] developed at the University of Manchester.

In 2005, he started a PhD in Computer Science at the University of Manchester in the area of query optimization in sensor networks, which gave rise to the SNEE query optimizer (http://snee.cs.manchester.ac.uk/). At the same time, he was employed as a Research Associate, initially for the Design, Implementation and Adaptation of Sensor Networks through Multi-dimensional Co-design (DIAS-MC) project, and currently for the Semantic Sensor Grids for Rapid Application Development for Environmental Management (SemSorGrid4Env) project. During his time at the University of Manchester, he has been co-author of the following publications:


He is looking forward to a fruitful career in research.
Acknowledgements

I would like to express my deepest gratitude to my supervisors, Alvaro A. A. Fernandes and Norman W. Paton for their constant support, patience, friendship and time. Your guidance was invaluable. I also wish to thank my advisor, Steve B. Furber.

Thank you to my friend Christian Y. A. Brenninkmeijer, with whom I worked in close collaboration in developing the SNEE (for Sensor Network Engine) query optimizer over the last four years. I also wish to express my gratitude to Alasdair J. G. Gray, who provided much support during the recent writing-up period, and Farhana Jabeen, for her involvement in the SNEE project.

The Information Management Group at the School of Computer Science of the University of Manchester is an excellent place to do research. Thank you to all my colleagues from there, who are too numerous to list individually.

The whole of the research reported in this dissertation was done as part of the Design, Implementation and Adaptation of Sensor Networks through Multi-dimensional Co-design (DIAS-MC) project – funded by the Engineering and Physical Sciences Research Council (EPSRC) – and the Semantic Sensor Grids for Rapid Application Development for Environmental Management (SemSorGrid4Env) project – funded by the European Commission’s Seventh Framework Programme. I am grateful for having been given the opportunity to take part in these projects.

I am extremely grateful to the School of Computer Science at the University of Manchester for covering my tuition fees.

Thank you, Johana, for your patience and love; I don’t think I would have made it without you.

My mother, Rosma Jimenez, took care of me during some of the more stressful periods, for which I am extremely grateful. I also wish to thank my father, John Galpin, and my friends, Tania Verdemato and Thomas Burslem, for their ongoing support.

Henk Muller, Andreas Stirnemann and Alvaro A. A. Fernandes played a role, at different points of my life, in motivating me to do a PhD in Computer Science. Quero expressar os meus mais profundos agradecimentos, Alvaro, for giving me the opportunity for this to materialize.
Para Johana.
Glossary

cDAF  Compact Distributed Algebraic Form.
CEM  Cost Estimation Model.
CQL  Continuous Query Language, a declarative language for querying push-based streams at Stanford University \cite{ABW03}.
DAF  Distributed-algebraic Form.
DBMS  Database Management System.
DDL  Data Definition Language.
DQP  Distributed Query Processor/processing.
FAF  Fragmented-algebraic Form.
GPS  Global Positioning System.
MAC  Medium Access Control.
PAF  Physical-algebraic Form.
QEP  Query Execution Plan.
QoS  Quality of Service.
RT  Routing Tree.
SNQP  Sensor Network Query Processor/processing.
SQL  Structured Query Language.
Chapter 1

Introduction

1.1 Setting the Scene

Over the last few decades, database technologies have proved hugely successful, as demonstrated by the widespread adoption of database management systems (DBMSs) as a means to store and retrieve vast quantities of data effectively, a common problem faced by most companies, governments, schools and other entities [GMUW00]. Data stored by a DBMS may be queried by users without requiring an understanding of the storage technique used or having knowledge about the physical location of the data. Database use has become pervasive throughout virtually all sectors of society, including commercial, scientific and military settings. Nowadays, it is hard to conceive how a bank, travel agent or supermarket would fare without database technology. The uptake of this general-purpose data archival and retrieval technology has been enabled by, and has in turn resulted in, the widespread availability of off-the-shelf, mature, implementations of DBMS software both commercially (e.g., Oracle) and in the open-source world (e.g., MySQL).

In contrast to databases, the emergence of sensor networks [KW05], triggered by advances in hardware technology, is a relatively recent phenomenon. As a consequence, sensor networks are a relatively less mature technology which has moved little beyond the research world. Sensor networks comprise nodes with sensors with the ability to gather data about their surroundings, interacting with one another via wireless communications, and may have a limited energy supply.

1.1. SETTING THE SCENE

Such networks potentially enable observation and analysis of complex interactions over a large, inaccessible, hazardous area in the physical world. Often, data can be viably obtained at a higher spatio-temporal resolution than would have been possible using a traditional (manual) data collection method [CM04]. As sensor data tends to be acquired periodically, such data is often streaming, i.e., it is generated and processed on-the-fly, although it may also be archived for subsequent retrieval. Examples of sensor network applications proposed in the literature include observing wildlife habitats in a non-intrusive fashion [TPS+05], flood prediction [ALE09], censusing animals [HBC+09], performing predictive maintenance of machinery, e.g., in an oil tanker or a semiconductor plant [KAB+05], and monitoring ambient temperature and energy consumption in buildings [SKG+05]. The spectrum of applications that can be conceived for sensor networks spans multiple domains, and is arguably as broad as that for databases, suggesting that, in the future, like databases, sensor networks may become pervasive.

A factor that has triggered the recent interest in sensor networks is the fall in sensor node hardware production costs, coupled with the increasing computational power of these platforms [ASSC02, HHKK04]. However, software development costs for sensor networks remain high, as programming sensor networks requires specialized knowledge, resources such as memory and energy are scarce, and debugging can be cumbersome. Implementing a simple, data collection application may be a considerable effort requiring thousands of lines in an embedded programming language. Gehrke et al. [GM04] and Madden et al. [MFHH05] observe that an efficient way to program and repurpose sensor networks is required if they are to become a viable platform, and propose the use of the database paradigm as an effective means to program sensor networks with considerably less effort. This has given rise to Sensor Network Query Processors (SNQPs), which view the sensor nodes in the network as data sources over which users may pose declarative queries specifying the data to be collected, without needing to be concerned about how this is done, or where the data is located. Examples of SNQPs in the literature are Cougar [BGS01, GM04], TinyDB [MFHH05] and SNQL [BLM+07]. SNQPs have demonstrated their effectiveness in real-life deployments, e.g., redwood tree micro-climate monitoring [TPS+05] and the Great Duck Island deployment which involved monitoring bird nests [SMP+04].

It should be emphasized that the complexity of programming sensor networks is aggravated by the fact that sensor nodes are typically very resource-constrained,
notably in terms of energy and memory available. Decisions about the use of these resources need to be made judiciously if a sensor network is to successfully fulfil the requirements of its users. For example, consider a sensor network deployed in a glacier to monitor the movement of the ice [MOH04]. Such an application should strive to be as economical with energy as possible, so that sensor nodes are not depleted of their energy stock prematurely, thus rendering the network unusable. As such, readings are taken throughout the day, and transmitted back to the gateway node once a day. However, this concern with preserving the energy stock does not apply to all applications all of the time. A group of firefighters using a sensor network to monitor forest fires will primarily be concerned with being notified as quickly as possible in the event of a fire. It would be useless to receive the notification about this at the end of the day. Given that there are trade-offs between both desiderata (i.e., that of conserving energy, and that of receiving the data in a timely fashion), the firefighters are likely to be willing to sacrifice longevity of the network in order to ensure a short notification delay.

In this dissertation, user concerns which reflect how data is collected, rather than what data is collected, are referred to as quality-of-service (QoS) expectations. The algorithms used to collect data in both cases may have several commonalities, in the sense that they need to acquire data from the source nodes, and transmit it to the gateway node, possibly performing some intermediate processing. However, the behaviour with respect to energy conservation will differ significantly. Current, first-generation, SNQPs such as TinyDB, Cougar and SNQL, could be used instead of implementing these programs manually. However, these SNQPs have, implicitly and inherently, a fixed optimization goal (viz., usually reducing energy consumption), and thus are unlikely to be suitable for both applications described. This dissertation argues that SNQPs can be made QoS-aware, and that doing so yields tangible benefits in their ability to respond to application-specific QoS, enabling them to be used for a broader range of application scenarios than existing SNQPs. Techniques for enabling QoS-awareness in the context of SNQPs are presented and their effectiveness is evaluated. The main research contributions conveyed in this dissertation centre around this idea of QoS-aware optimization as a response to both the potential existence of conflicting QoS expectations for the same query (at different points in its lifecycle) and the constrained nature of the software and hardware platforms with which to meet such expectations.
1.2. FUNDAMENTALS

The remainder of this chapter details the ideas sketched above, to introduce the research contributions made in this dissertation. Firstly, Section 1.2 provides a brief background on classical, distributed and stream query processing, introduces the reader to the highly constrained distributed computing platform that sensor networks give rise to, and describes the notion of QoS expectation in the context of query processing in sensor networks. Then, Section 1.3 compares SNQP with other, more established, forms of query processing, and describes limitations of state-of-the-art SNQPs. The aims, objectives and research contributions of the work that led to this dissertation are presented in Section 1.4. Section 1.5 describes the technical approach employed to achieve the research contributions. Finally, Section 1.6 describes the structure of the whole dissertation for the benefit of the reader.

1.2 Fundamentals

A query is a declarative statement characterizing a collection of data items that conform to a database schema. In classical DBMSs, a query is optimized and compiled into a query execution plan (QEP) which is then evaluated by a query execution engine. For a query compiler/optimizer, this involves making several decisions including how to access the data (i.e., what indices and algorithms to use) based on information about the data (e.g., its value distribution, its location in physical storage devices) and the runtime environment (e.g., the amount of memory available). This gives rise to physical independence, insofar as the logical characterization of the required data is decoupled from the physical, concrete conditions under which that data will be computed. In classical query processing, data sources are located, and queries are evaluated, at a single server. In contrast, a distributed query processor (DQP) operates over a distributed database, i.e., a database where both the data sources and the computational resources required to run a query evaluation plan are partitioned across various sites over a wide area. Queries often combine data from various sites, thus the metadata stored in the system catalog has information about where data items are placed and what capabilities each site has regarding the evaluation of query plans in order for the optimizer to generate the QEP that will make best use of resources. Classically, both centralized query processing and DQP data sources constitute stored (as opposed to streaming) data.
A query processing stack is a software architectural abstraction used to represent the process by means of which one translates a declarative query into a QEP into a series of steps. At each step, internal representations of the query undergo a sequence of transformations, each of which is the outcome of different optimization decisions. The end product is a QEP, either in interpretable or executable form. As the space of possible QEPs for a particular query is typically very large, in many cases a query processing stack will apply techniques such as dynamic programming \cite{AHU83} to prune the solution space, often giving rise to heuristic-based algorithms at each step.

A recent development in the technological landscape is the increasing availability of continuously updated data channels. Examples include market data feeds, news feeds (e.g., RSS-based ones), data obtained from global positioning satellites, or from mobile devices. These scenarios pose challenges to database technology on two basic fronts: firstly, the data required to answer a query does not lie stored in a data container (e.g., magnetic discs) but instead enters the QEP from streams of indefinite cardinality (thus, data may never cease to arrive), and secondly, because data reflects a continuously changing world view, a query may need to be re-evaluated, in principle, every time that more data arrives, i.e., in response, or as a reaction to, a change in the world as reflected in the newly-arrived data. Some background to stream query processing is given later in Section 2.1.3.

Another recent development, catalysed by advances in radio, processor and battery technology, is the emergence of sensor networks as an economically viable hardware platform with which to observe or monitor phenomena in situ, possibly over a wide area of interest and at a fine grain of observation. Such networks are formed from sensor nodes working collaboratively on a task. In the context of this dissertation, sensor nodes are devices with in-built wireless capability, limited computational power, depletable (and not easily replenishable) energy resources, and transducers which are able to gather information about their surrounding environment. The data produced by sensor networks is typically of a streaming nature. A detailed background about sensor networks is given in Section 2.2.

A sensor network may be viewed as a distributed computing platform, in which every sensor node is a computer with processing power and memory, cooperating in the execution of a distributed algorithm in order to accomplish a common goal. Each sensor node has access to limited information, and needs to communicate
1.2. FUNDAMENTALS

with other sensor nodes during the execution of a distributed application. Each sensor node has sensors that sense information about their surroundings (thereby generating data items that make the node behave like data sources in a distributed database). Furthermore, nodes may have actuators, which are able to affect the environment in which they are immersed (although this aspect is not considered in the research reported in this dissertation). For example, a flood detection warning system may automatically open a sluice gate when the river exceeds a certain water level. The ability to embed computing, sensing and actuation capabilities into the environment has led to the idea that sensor networks conceptually enable the physical world itself to be viewed as a computing platform [BGS00].

Lynch [Nan96] notes that while it is important that distributed algorithms run correctly and efficiently, their design is a difficult task. This is aggravated in the context of sensor networks due to the resource constraints of the currently cost-effective hardware platforms, the embedded nature of the software solutions, the cost of (typically wireless) communications, the limited power supply and the inherent unreliability of the hardware. Despite these constraints, most sensor network applications of greater interest must be robust enough to continue operating even when components fail, or data transmitted by radio is lost (although developing techniques to achieve this is not the focus of this dissertation). Furthermore, in the case of sensor networks, the design of the distributed algorithm, in order to satisfy performance requirements, should take into account the specific QoS expectations that are most relevant to the application in question.

In application software engineering, requirements fall under two broad categories, viz., functional requirements, which specify the task to be performed, and non-functional requirements, which specify expectations as to the performance which must be exhibited by the application, in terms of metrics of relevance to the user [Som07]. Thus, in a query processing context, the functional requirements refer to the query, and the non-functional requirements, include the QoS goal, such as a target response time (e.g., in Paton et al. [PAL+09]). In the context of this dissertation, QoS is an application-level notion, and comprises various variables that matter significantly to the user. In Section 2.3, several representative sensor network applications are described, with a focus on their QoS expectations. From the discussion in that section, it can be seen that sensor network applications have diverse QoS variables, such as (1) the frequency that sensors read data (the acquisition interval), (2) the maximum delay tolerable for
data to reach the user (the *delivery time*), (3) the *total energy consumption* of the nodes in the sensor network, and (4) the time taken for the sensor network to become non-operational due to energy depletion (the *network lifetime*).

### 1.3 Motivation

The complexity involved in designing sensor network applications means that, in contrast with the increasing affordability of the hardware, the development of sensor network applications remains an expensive process. The scarcity of generic tools that automate their development and enable software reusability means that sensor network application developers are typically faced with the task of implementing each application monolithically, and from scratch, which is time-consuming and error prone [CLL+06]. Given that for many applications of sensor networks, the functional requirements consist, almost exclusively, of data collection, research such as that conducted in the TinyDB and Cougar projects, has given rise to the idea of query processors as a means to program a sensor network application with comparatively less effort.

Construing the sensor network as a distributed computing platform and allowing users to pose queries over it potentially constitutes an effective and efficient means to obtain data about the physical environment, as users would not need to be concerned with how sensors are to acquire the data, or how nodes transform and/or transmit the data. The onus then falls on the optimizer to generate a [QEP] that satisfies the expressed needs. Note that, as with classical query processing, finding the optimal query plan may not be feasible due to the number of possible candidate plans, so an optimizer is judged on its ability to generate good query plans most of the time. However, the fact that users are provided with an effective means to avoid designing and implementing single-purpose sensor networks compensates for the fact that the generated [QEPs] may only be quasi-optimal.

A [SNQP], therefore, has characteristics in common with a [DQP] in that data sources and query evaluation takes place across several sites. However, in contrast to classical and distributed query processing, where data is stored and indexed for subsequent retrieval, sensor data is often streaming, as it relates to events that are unfolding in time, and may need to be processed in real time, possibly because they are urgent, or simply due to the fact that it is not feasible to store all the data. Continuous query languages have been developed to retrieve information
1.3. MOTIVATION

from streams (such as CQL, devised as part of the STREAM project [ABB+03, ABW03]) in which queries are repeatedly evaluated in response to newly arrived data items as the data stream keeps generating them.

An important difference between classical query processing and SNQP is that, in the former, the compiler/optimizer is mainly focussed on a fixed, unique optimization goal, viz., to minimize response time. This focus is justifiable in the classical case as the computational fabric is reliable and, in practical cases, unconstrained. However, sensor networks are a highly resource-constrained platform, given that energy supply, memory availability and processing power are all scarce. Moreover, not only are nodes inherently unreliable, but they communicate via radio, which is an unreliable technology when the corresponding electronics is as limited as is often the case with sensor networks. This leads to challenging trade-offs and characterizes a (potentially) multi-objective constrained optimization problem. For example, the need to preserve scarce resources (in particular, energy) so as to ensure the longevity of the network suggests collecting data less frequently which may conflict with the need for greater accuracy in the results. Note also, that data is more likely to be lost when it is transported wirelessly and, due to the streaming nature of the data, scanning (let alone rescanning) of the entire extent of a collection of data items is not possible. These observations characterize a violation of assumptions made in DQP about the nature and reliability of the underlying infrastructure.

The degree to which each issue is a concern will vary from application to application. Indeed, the applications described in Section 2.3 have different priorities in terms of optimization goal. This means that reconciling diverse QoS concerns is crucial, as different sensor network applications have different, possibly multiple and conflicting, QoS expectations. Current SNQPs are unable to cater for a wide range of sensor network applications, as they take very few QoS expectations into account. For example, TinyDB purports to primarily optimize for energy consumption, suggesting that it may be inefficient for event-detection applications for which the fastest possible delivery time may be required. This observation leads to the central research challenge addressed in this dissertation, viz., that for a SNQP to be effective and efficient for different sensor network applications, such as the ones described at the start of this section, it must be able to optimize according to specific QoS specifications associated with, potentially, each submitted query.
As discussed in Section 3.3, for query processing over robust networks, with relatively unconstrained and robust nodes, the query optimization goal is generally fixed, i.e., it tends to be response time (or similar). However, because sensor network hardware tends to be highly resource constrained, stark trade-offs present themselves between different QoS-related desiderata. For example, it may not be possible to achieve both a low energy consumption and a fast result delivery time, as the latter may mean powering up the radio very frequently, and as a result quickly depleting the node of energy. Given that sensor networks have a broad range of applications, each with different QoS expectations, it is expected that a SNQP with a fixed optimization goal will be unable to satisfy the needs of a broad enough range of sensor network applications. It is therefore argued that, for a SNQP to be general-purpose, i.e., suitable for the broad range of applications envisaged for sensor networks, it should be flexible in responding to different QoS expectations when optimizing a query or query workload, i.e., the QEP generated by the compiler/optimizer should be chosen in the light of specific QoS expectations that a user can explicitly state for the query or query workload.

Posing a declarative query against a sensor network has been demonstrated to be an effective and efficient means of interacting with a sensor network in data collection tasks [GM04]. However, as discussed in Section 3.2, existing SNQPs are incipient given the limitations that:

L1 Their approach to generating a QEP is simplistic, as few query planning decisions are made explicitly, often resulting in the generation of QEPs that act in a fixed or predetermined manner; and

L2 As a result of L1, existing SNQPs are unable to take diverse optimization goals and QoS expectations into account when generating QEPs.

The next section describes the research aims which address these shortcomings.

1.4 Aim, Objectives and Research Contributions

The aim of the research presented in this dissertation is to demonstrate the potential of SNQPs to take into account diverse QoS expectations, by exploring the incorporation of QoS-aware decision-making capabilities into a sensor network query optimizer.
In order to address the research aim, given limitation L1 mentioned above of existing SNQPs, it is first necessary to address the problem of identifying the required decision-making steps in a query processing stack for sensor networks, so that different kinds of query planning decisions can be made explicitly. Then, alternative approaches to how each decision is made are considered. In this dissertation, the notion of a query processing stack template refers to the specification of a sequence of decision-making steps that are required to generate a QEP. This includes specifying the interface, i.e., inputs and outputs, of each step in the sequence. This is essentially the ‘skeleton’ of a query processing stack, as the template is agnostic as to the implementations of each step, i.e., the algorithms which determine how the decisions are actually made. A query processing stack template may give rise to several different instantiations, and these are described by specifying the algorithms that implement each of the decision-making steps in the query processing stack template, in order to address L2.

The research aim of this PhD dissertation is therefore decomposed into the following objectives, from which corresponding research results were obtained, that the remainder of this dissertation describes and evaluates:

**O1** To design a functional decomposition of the query compilation process from declarative query to imperative QEP for sensor networks that allows individual query planning decisions to be made more explicitly than is done by current state-of-the-art SNQPs [GM04, MFHH05].

The corresponding research contribution is the SNEE (for Sensor NEtwork Engine) query compilation/optimization stack template for sensor network query processing presented in Chapter 4. From this, two concrete instantiations are targeted, one with a fixed optimization goal **O2**, and another which is responsive to QoS expectations **O3**.

**O2** To design and implement an instantiation of the query processing stack in **O1** with the fixed optimization goal of preserving energy (the most commonly cited concern in SNQPs). This instantiation serves to validate the sequence of steps in **O1**. Also, assuming that this instantiation outperforms existing SNQPs with similar in-built assumptions about QoS expectations, it will be used as a baseline to evaluate the benefits of having a QoS-aware SNQP compared to a fixed goal one.

The corresponding research contribution is a set of example algorithms that
CHAPTER 1. INTRODUCTION

In this chapter, we introduce the research and development of a novel query processing stack, referred to as SNEE (Sensor Network Query Execution Environment). The goal is to enable efficient and effective query processing in sensor networks while meeting Quality of Service (QoS) requirements.

**O1** Instantiation to generate QEPs with a fixed optimization goal. This instantiation is referred to as FG-SNEE and presented in Chapter 5.

**O2** To design and implement a solution that enables query planning decisions to be informed by different criteria in response to QoS expectations associated with a query.

The corresponding research contribution is an instantiation of O1 with algorithms for each optimization step that enable alternative decision-making policies to be active in the query processing stack depending on the QoS expectations specified by the user for a given query. This instantiation is referred to as QoSA-SNEE and presented in Chapter 6.

**O3** To evaluate the performance benefits of having QoS-aware decision-making capability in SNQPs for representative workloads and QoS expectations.

The corresponding research contribution is an evaluation of how favourably the QoS-aware instantiation of the SNEE query processing stack O3 fares compared to a representative fixed-goal SNQP by comparing QEPs generated at starkly contrasting points in the QoS spectrum. This implies that it is first necessary to identify a suitable candidate to represent previous state-of-the-art SNQP among systems such as TinyDB, Cougar and FG-SNEE that can be used as the baseline for comparison with QoSA-SNEE. This is presented in Chapter 7.

Note that the design of a syntax and semantics for SNEEq, the declarative query language over sensor networks, the fine-grained cost models for use by O1 above, and the corresponding algorithms that underlie the implementation of the corresponding physical-algebraic operators, are research contributions of, and described in the PhD dissertation by, Christian Brenninkmeijer [Bre09], and are not, therefore, to be in any way construed as contributions of the research reported in this dissertation. These contributions are reviewed in Chapter 4 and described in more detail in Appendices A and B.

1.5 Technical Approach

In order to address the aims and objectives described in Section 1.4, the overall problem is broken down as follows: Firstly, a functional decomposition of steps
from declarative query to imperative QEP in the sensor network context is devised, resulting in the SNEE query processing stack template created as a result of objective O1. Subsequently, FG-SNEE, a fixed-goal instantiation of the query processing stack is proposed, with the aim of validating the functional decomposition proposed, and furthermore, providing a suitable baseline for evaluating the benefits of QoS-awareness over optimizing according to a fixed goal. Then, QoS-aware decision-making is incorporated into the steps of the query processing stack, resulting in the QoSA-SNEE instantiation described in objective O3. Finally, the evaluation (objective O4) involves comparing the outcomes of O2 and O3. This section discusses the technical approach adopted to meet each objective.

O1: Functional Decomposition of Query Optimization. As discussed in Section 2.4, the DQP and SNQP problems have much in common, despite several important differences. Based on this observation, in the design of the SNEE query processing stack for sensor networks, the two-phase optimization approach [Kos00] (described in Section 2.1.2) is used as a starting point and adapted to take into account the differences with the sensor network case. This involves a query processing stack architecture broadly characterized by the initial generation of a centralized QEP in the first phase, followed by a second phase in which various distribution decisions are made that yield a QEP broken down into fragments destined to run on different sites. The SNEE query processing stack includes novel, sensor-network specific decisions, such as which network paths to use to route tuples, and the timing of tasks, both issues that are not usually a concern in DQP but need to be taken into account in this case because of the cost implications they have on a sensor network QEP. The novel decisions are incorporated into the SNEE query processing stack, the functional decomposition of the steps to optimize a SNEEql query against a sensor network, presented in Section 3.1.

The QEPs generated by SNEE are assumed to be generated at a computer outside the sensor network, using metadata such as node resource availability and node connectivity collected from the sensor network, which reflects the state of the sensor network at query compilation time. Thus, it is assumed that, in order for a query to be compiled and deployed, functionality is provided for SNEE by other software components that is responsible for the initial formation of a
network, and subsequently, the collection of metadata required by the optimizer
and shipping of the QEP to nodes in the sensor network.

Furthermore, it is noted that sensor networks are a fragile computing fabric,
as they are prone to clock drift and node failure. They therefore comprise signifi-
cantly less robust nodes than those typically used for DQP. This implies that over
a relatively short time the initial conditions assumed by the SNEE optimizer may
change, and the generated QEP may no longer function (e.g., if a sensor node
chosen by the optimizer to relay data dies), so a degree of resilience is desirable.
Dealing with resilience is beyond the scope of this dissertation, and throughout
this dissertation, an assumption is made that the volatility and brittleness of the
execution environment can be attenuated by, say, a network management layer
(e.g., a stack of adaptive protocols), the effect of which would be to ensure that
the metadata used to generate the QEPs decays in validity at a slower rate that
it would otherwise. Such functionality may perform network healing, e.g., if a
node being used to relay tuples dies, a new route would be found. It is to be
expected that such functionality will become more prevalent as sensor networks
mature. Currently, the commercially-available Moteworks software produced by
X-bow claims to provide the kind of functionality described that is required to
deploy a QEP and, to a limited extent, the functionality required to make QEPs
more resilient at runtime. In this dissertation, it is assumed that claims such as
these are true, or else that similar overall functionality could be assembled using
off-the-shelf components (e.g., from the TinyOS codebase).

In cases when changes to the initial conditions are severe, however, such net-
work management functionality is unlikely to be sufficient to ensure that the QEP
continues executing correctly and meeting the QoS expectations. An approach to
make a QEP more resilient at runtime (and continue to perform according to the
QoS expectations), also used in other modes of query processing, would be to em-
ploy adaptive query processing (AQP) techniques, that involve the query processor
responding to changing conditions as a query is evaluated, e.g., by ad-
justing the QEP. For example, AQP could enable a neighbouring node to take over
execution of a resource-intensive fragment involving a join that is executing on a
node that is low on energy. More details about AQP techniques for classical modes
of query processing are provided in Section 2.1.4. There have been preliminary

\[\text{http://www.xbow.com/support/wSoftwareDownloads.aspx}\]
\[\text{http://tinyos.cvs.sourceforge.net/}\]
investigations into AQP for sensor networks [BB03, TYD+04, MNG05, BLM+07], which are complementary to the results presented in this dissertation.

O2: Fixed-Goal instantiation of Query Processing Stack. The FG-SNEE instantiation uses heuristic-based algorithms to generate QEPs. The heuristics are essentially to minimize the amount of radio traffic between sensor network nodes, and to buffer the data as much as possible so that the radio is powered up as infrequently as possible. It is intended that these heuristics will lead to a reduction in energy consumption. By making more query planning decisions explicitly than previous SNQPs, the aim is to generate more energy efficient QEPs than the previous state of the art, enabling FG-SNEE to be a suitable representative fixed-goal SNQP in order to assess the benefits of QoS awareness during the experimental evaluation.

O3: Quality-of-Service Aware Instantiation of Query Processing Stack. The QoSA-SNEE instantiation generates QEPs that exhibit performance in light of diverse QoS expectations. The underlying desideratum for the algorithms that comprise the steps in QoSA-SNEE is that they are generally applicable for different QoS goals, i.e., that a different algorithm is not required when optimizing for a different QoS expectation. To enable this, generalized algorithms are proposed that can optimize for different goals by using QoS-specific objective functions, which depend on QoS expectations that the user has coupled with the query. These objective functions take as input an intermediate QEP and evaluate its desirability according to a specific QoS variable. Thus, additional QoS goals can be proposed by specifying an objective function to represent this variable, as long as the variable may be expressed using an expression involving time, memory or energy cost models. In some steps, the decision-making is cast as a constrained optimization problem and delegated to an off-the-shelf optimizer; another approach taken is to generate possible intermediate QEPs, rank them according to their objective function value, and discard the ones which appear to be least desirable.

O4: Evaluation Strategy. The strategy employed to evaluate the benefits of QoS-awareness in SNQPs is to compile a set of queries against a QoS-aware SNQP and a suitable candidate to represent previous state-of-the-art SNQP (which acts as a baseline for comparison) for varied QoS expectations. The focus of the
experiments is on the analysis of the properties of QEPs rather than evaluating the performance on sensor node hardware. This approach enables systematic experimentation of the effectiveness and efficiency of SNEE-generated QEPs. In this respect, the question as to whether the results would hold to the same extent in actual hardware and over more capable underlying software layers was left for further work.

The QEPs generated are evaluated against each of the QoS variables, with the intention of highlighting the degree of responsiveness of a QoS-aware SNQP over a fixed-goal SNQP. It is to be expected that, for any query and QoS expectation, a QoS-aware SNQP will perform at least as well as the fixed goal one does. The software artifacts compared experimentally to implement the above strategy are FG-SNEE (argued in Section 7.1 to be a suitable candidate to represent fixed-goal SNQPs following a comparison with previous SNQPs) and QoSA-SNEE (the only QoS-aware SNQP known to be in existence at the time of writing). By means of these experiments, the benefits of QoS-awareness in the context of sensor networks are made apparent. Based on this, it is argued that SNQPs should be QoS-aware in order to be truly considered general purpose, i.e., able to generate QEPs for a broad variety of applications.

1.6 A Guide for Readers

Chapter 2 starts with an introduction to different modes of query processing, viz., classical, distributed, and stream query processing. A description of sensor networks as a distributed computing platform follows. Then, the notion of QoS for sensor network applications is described, using a sample of sensor network applications as examples. The chapter concludes with a discussion of the issues involved in designing a query processor for sensor networks, such as the need for additional types of decisions to be made by the query optimizer in order to contend with a highly resource-constrained and inherently unreliable platform, and the broad range of QoS expectations which need to be supported.

Chapter 3 describes work related to the research contributions of this dissertation: firstly, some concepts are introduced to enable an effective discussion in the remainder of the chapter, then a survey of the state-of-the-art in SNQP is provided, which reveals shortcomings with respect to the challenges discussed in the previous chapter. The focus then turns to classical and distributed query
optimization. Research dealing with QoS is discussed, followed by work which considers approaches to extend the behaviours exhibited by query optimizers. These are both considered for implementing QoS awareness in Chapter 6.

Chapter 4 presents the SNEE query compilation/optimization stack template. Firstly, the inputs to the query stack are described: SNEEql, a declarative, expressive query language for sensor networks, associated QoS expectations, and metadata, including the space, time and energy cost-models used to inform query planning. Following a discussion on the adaptations and extensions required to DQP in light of the challenges discussed in Section 2.4, a functional decomposition from a declarative query to imperative QEP is presented. From the SNEE query stack template, two instantiations are derived, described in the next two chapters.

Chapter 5 describes FG-SNEE, an instantiation of the SNEE query processing stack for a fixed optimization goal. Example algorithms for each step in the query processing stack are provided.

Chapter 6 presents QoSA-SNEE, a QoS-aware instantiation of the SNEE query processing stack. Firstly, possible approaches to enabling QoS-awareness are discussed, based on the approaches to implementing extensibility that were reviewed in Section 3.3. For the steps in the query processing stack in Chapter 4, QoS-aware implementations are described, whose effectiveness is evaluated in the next chapter.

Chapter 7 presents an evaluation of the research contributions: Firstly, a suitable baseline to represent a state-of-the-art fixed-goal SNQP is determined based on descriptions of existing systems and an experimental evaluation. Then, the benefits of QoS-awareness are demonstrated, by comparing the performance of the baseline fixed-goal SNQP (i.e., FG-SNEE) to the only example of a QoS-aware SNQP known to exist to the author at the time of writing (i.e., QoSA-SNEE), for applications with contrasting QoS expectations.

Conclusions are presented in Chapter 8. This chapter reflects on the implications of the results with respect to the design of SNQPs and argues that they demonstrate that QoS-awareness may bring about significant savings in software development costs for sensor network applications. Finally, future work following on from the research contributions of this dissertation is proposed, such as the incorporation of QoS-aware AQP techniques into SNQP.
Chapter 2

Background

This chapter provides the technical context required to understand the research contributions reported in the remainder of this dissertation. Note that the background chapter does not describe potentially competing research; this is done in Chapter 3 where related work (or in some cases, work which seems to be related) is analysed. Within this chapter, Section 2.1 gives an overview of the different types of query processing. Section 2.2 gives an introduction to sensor networks, the distributed computing platform that the SNEE query processing stack described in Chapter 4 generates query execution plans (QEPs) for. A brief overview of the proposed applications for sensor networks is given, example sensor node hardware platforms are described, and issues, such as the need to conserve energy, and consequently, the benefits of trading off processing with radio communications, are discussed.

Then, in Section 2.3, the requirements of several sensor network applications are identified, with a focus on their QoS expectations. This exercise enables the relevant QoS variables for sensor network applications to be identified, and also confirms the view put forward in the previous chapter that QoS expectations for sensor network applications are diverse. Finally, Section 2.4 describes the challenges that arise when applying query processing techniques over sensor networks, based on the fundamental differences between sensor network query processing (SNQP) and other modes of query processing. These challenges result from the inherent scarcity of resources of the underlying platform, which motivates the need for SNQPs to consider QoS expectations to trade-off potentially conflicting user requirements (e.g., a short delivery time and high network lifetime) when generating a QEP. These challenges manifest themselves as decisions
that do not need to be made in other modes of query processing, and therefore influence the design of the query processing stack for sensor networks, capable of making QoS-aware decisions, that is proposed in Chapter 4 of this dissertation.

2.1 Query Processing

This section comprises four parts: Section 2.1.1 describes classical query processing, in which queries are evaluated on a single, central server over stored (as opposed to streamed) data. Section 2.1.2 describes distributed query processing (DQP), which operates over a distributed database, i.e., a database where both the data sources and the computational resources required to evaluate a QEP are partitioned across various sites over a wide area [Kos00]. Section 2.1.3 describes stream query processing, in which queries are posed over streams of incoming data in real-time, rather than over data which has been previously stored and possibly indexed. This paves the way for the later discussion of query processing in sensor networks, in Section 2.4, with which these modes of query processing are compared and contrasted.

2.1.1 Classical Query Processing

By classical query processing, in this dissertation, is meant the evaluation of queries on a central server over stored data (as opposed to incoming data streams). Much of the material in this sub-section is summarized from Garcia-Molina et al. [GMUW00]; relevant surveys that describe the optimization and evaluation of queries in this context are given by Graefe [Gra93], Ioannidis [Ioa97], and Chaudhuri [Cha98].

Queries may be expressed in a declarative language, such as SQL [GWO09] or OQL [CBB+00]. Generally speaking, the processing of a query involves two steps:

- The query is parsed and compiled into an optimized QEP.
- The QEP is evaluated, and results are returned to the client.

Being declarative languages, SQL and OQL do not specify how the query is to be evaluated, only what data should be returned. Therefore, the query compilation/optimization stage is critical in order to find an efficient QEP, i.e.,
### Table 2.1: Some example logical-algebraic query operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol Used</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scan</td>
<td>$\text{scan}_{R,C}$</td>
<td>Read tuples from relation $R$, discarding those that do not meet conditions in predicate expression $C$.</td>
</tr>
<tr>
<td>select</td>
<td>$\sigma_C(R)$</td>
<td>Discard tuples in relation $R$ which do not meet conditions in predicate expression $C$.</td>
</tr>
<tr>
<td>project</td>
<td>$\pi_L(R)$</td>
<td>Remove any attributes in $R$ which are not in the attribute list $L$.</td>
</tr>
<tr>
<td>aggregate</td>
<td>$\gamma_{F,L}(R)$</td>
<td>Aggregate values into a single value using the merge function $F$ (e.g., $\text{SUM}$, $\text{COUNT}$ or $\text{AVG}$). If a list $L$ of grouping attributes is specified, a result is given for each distinct value of the grouping attribute(s).</td>
</tr>
<tr>
<td>natural_join</td>
<td>$R \bowtie S$</td>
<td>Concatenates tuples from $R$ and $S$ which have equal values for attributes with the same name, removing duplicate tuples.</td>
</tr>
<tr>
<td>theta_join</td>
<td>$R \bowtie_\theta S$</td>
<td>Concatenates tuples from $R$ and $S$ for which condition $\theta$ is true. $\theta$ is a condition between attributes in relations $R$ and $S$, e.g., $R.x &gt; S.y$.</td>
</tr>
<tr>
<td>sort</td>
<td>sort$(R)$</td>
<td>Returns the tuples of $R$ in sorted order.</td>
</tr>
<tr>
<td>distinct</td>
<td>$\delta(R)$</td>
<td>Returns the tuples in $R$ with duplicates removed.</td>
</tr>
<tr>
<td>operation_call</td>
<td>$\phi(R)$</td>
<td>Applies a function $\phi$ to every tuple of $R$ drawing the input arguments from the tuple and returning the corresponding output value as an attribute in the resulting relation.</td>
</tr>
</tbody>
</table>

one that results in an acceptable response time. By *response time* is meant the time between the query being issued and the results being returned. By *acceptable* is meant within an interval (often unstated) that does not compromise the efficiency of the other processes the client is performing.

The generation of a [QEP] has a logical and a physical stage. In *logical query plan generation*, the abstract syntax tree representing the query structure is mapped to a logical [QEP] i.e., one where nodes are operators in an abstract algebra. Table 2.1 lists some such operators and provides them with brief, informal definitions. An example logical [QEP] using the operators in Table 2.1 is shown in Figure 2.1. A logical [QEP] can be seen as a term. Query optimizers can be seen as term rewriting systems [Klo92], in that the logical stages rewrite the logical [QEP] using equivalence laws over logical algebraic expressions. The transformation rules that are applied rely on various heuristics aimed at, e.g., reducing the size of intermediate results. An example of such a heuristic is to push a selection down the tree in order to discard as many tuples as possible, as early as possible, thereby decreasing the size of intermediate results that downstream
2.1. QUERY PROCESSING

select p.proteinID, blast(p.sequence)
from proteins p, proteinTerms t
where p.proteinID=t.proteinID and t.termID = 8372

Figure 2.1: A logical query execution plan resulting from the query shown above, adapted from [SWG+03].

operators will need to process. This is intended to improve query response times, as it reduces the overall workload and may reduce the number of I/O operations required. This is desirable as I/O operations have a high cost associated with them in classical query processing, as they tend to increase the response time of a QEP.

Example. An SQL statement SELECT A.y FROM (A NATURAL JOIN B) WHERE A.y>5 may be represented as the relational-algebraic expression \( \pi_{A,y}(\sigma_{A.y>5}(A \bowtie B)) \). Given that it is preferable to perform selections before joins, so that as many tuples as possible are discarded as early as possible, the expression can be rewritten into the equivalent (in the sense of being result-neutral) but more more efficient (in the sense that it is likely to have to contend with smaller intermediate results) expression \( \pi_{A,y}(\sigma_{A.y>5}(A) \bowtie B) \).

Physical query plan generation involves choosing concrete algebraic operators (i.e., algorithms) to instantiate the QEP. This stage, therefore, essentially generates executable code on-the-fly that, when run, returns the required query
results. Figure 2.2 shows an example physical QEP. For each logical-algebraic operator, there may exist more than one physical-algebraic operator (i.e., concrete algorithms on concrete data structures) that implements it. For example, for a logical join there may correspond any amongst nested-loop join, hash join, merge join, etc. Because of this one-to-many relationship, many possible physical plans can be generated from one logical QEP. The available options (e.g., which physical joins are available) are stored in a catalog. The catalog also stores statistics about the data which are parameters for a cost estimation model (CEM), used to assign a predicted cost to each candidate physical plan. The cost of each plan is estimated using cost-based plan enumeration, and the best physical plan is selected. The absolute cost estimates of a plan need not be highly accurate: in this context, relative cost estimates are more important, i.e., the aim is to select the cheapest plan even if its estimated cost is not accurate. The best physical operator to use for a logical operator depends on various factors such as available memory, the (predicted) size of the relations involved, and the ordering of the operators.

A common paradigm for implementing the interface by means of which physical-algebraic operators that participate in a QEP interact with one another is to
expose an iterator interface (i.e., one that implements only three operations, viz., \texttt{open()}, \texttt{next()}, \texttt{close()}). This enables them to be composed in arbitrary combinations, and also allows for a uniform execution model, called the \textit{iterator model}. The iterator model is pull-based, in the sense that each data tuple needs to be explicitly requested. A query is evaluated by calling \texttt{open()} on the root, then \texttt{next()}, repeatedly, while there are input tuples available, and then \texttt{close()}. All calls propagate from parent to children as one would expect. To this root-to-leaf control flow there corresponds a reverse leaf-to-root data flow which causes the full results to reach the root whence they are returned to the client. As already mentioned, this approach enables operators to be plugged into one another in arbitrary combinations. It also enables non-blocking operators to execute in pipelined-parallel fashion. An operator is blocking if it needs to see all tuples of one or all of its input relation(s) before it can generate any output. For example, \texttt{select} and \texttt{project} are non-blocking; \texttt{sort} is blocking.

\subsection{Distributed Query Processing}

While classical databases have proved very successful in achieving their aims, the increased availability and usability of network infrastructure has led to distributed infrastructures becoming the default in business, scientific and personal contexts. In particular, there is great interest in integrating remote, autonomous data resources into globally unified ones (e.g., genomic data warehouses in biology). Likewise, there is interest in harnessing remote computational resources and making use of their processing power in integrated distributed computations. These two incentives underpin current activity in Distributed Query Processing (DQP). A survey of DQP techniques is presented by Kossmann [Kos00].

A DQP operates over a distributed (potentially virtual) database, i.e., a database where both the data sources and the computational resource required to run a Query Execution Plan (QEP) are partitioned across various sites over a wide area. Queries often combine data from various sites, thus the metadata stored in the system catalog has additional information about where data items are placed and what capabilities each site has regarding the evaluation of QEPs in order for the optimizer to generate the code that will make best use of resources.

The two-phase optimization paradigm, a well-known approach to query optimization in DQP, involves generating a QEP as if it was for a single site (as described in Section \ref{iterator_model}), and then carrying out distribution decisions in a
Figure 2.3: The two-phase optimization approach for distributed query processors, as employed by the Polar* optimizer [SWG+03].

The Polar* optimizer [SWG+03], whose DQP strategies are used as an example in this section, adopts the two-phase approach, as illustrated in Figure 2.3. The single-site phase encompasses the logical and physical optimization steps described for the case of classical query processing. For the multi-site phase, the single-site plan is firstly partitioned into fragments (i.e., sub-trees of the single site plan), and then each fragment is assigned to a subset of the sites available for query evaluation so that the overall cost of the QEP is reduced.

Figure 2.4 is an example distributed QEP as could be generated in the multi-node optimizer phase for the single-site plan shown in Figure 2.2. Each dotted-line box represents a different query fragment executing on one or more sites. Each fragment is annotated with a unique identifier, and the set of sites to which it has been assigned. Note the use of EXCHANGE operators, which encapsulate flow control, data distribution, and inter-process communication, between fragments in the QEP [Gra90]. An EXCHANGE operator $e$ with a child fragment $F$ and a parent fragment $G$ has two parts, each of which may act independently: a producer $e_p$, associated with $F$, and a consumer $e_c$, associated with $G$. Flow control may be implemented by $e_c$ and/or $e_p$ carrying out buffering in order to reconcile different rates of data production and consumption between instances of $F$ and $G$. Instances of the producer $e_p$ may exhibit alternative data distribution policies by sending tuples to specific instances of $e_c$, e.g., with the aim of balancing the load as evenly as possible among instances of $G$. Communication takes place if a tuple is sent from an instance of $e_p$ to an instance of $e_c$ on a different site, in which case, the higher cost resulting from the associated communications is reflected by the CEMS when estimating the cost of the EXCHANGE operator.

The strategy employed by Polar* during the partitioning step to determine at which edges in the operator tree to insert EXCHANGE operators (and consequently demarcate fragments) involves considering how data needs to be partitioned in
order for operators to produce correct results. This is done, firstly, by identifying whether an operator is *attribute sensitive*, i.e., requires its input data to be partitioned in a specific way when several copies of it are executing in parallel [HM95]. For example, a join may be parallelized as long as each copy of the join compares tuples from each of its operands that are within the same partition. In this case, a partition may be defined as comprising tuples whose values of attributes used by the join predicate lie within a specific range. If an operator is not attribute sensitive, the way that its input data is partitioned will not affect the correctness of the results produced by the operator. However, if an operator $op$ is attribute sensitive, and the output of $op.\text{Child}$ is partitioned in a different manner to that required by $op$ (or the output of $op.\text{Child}$ is not partitioned at all), redistribution of data is required. This is achieved by inserting an \texttt{EXCHANGE} operator between $op$ and $op.\text{Child}$. 
In the scheduling step, Polar** assigns fragments to sites with data and computational resources. For the example query shown in Figure 2.1, the metadata informs the optimizer that the sources for the proteins extent are sites 2 and 3, and the sources for the proteinTerms extent is site 6. The fragments with the SCAN operators are therefore placed at these sites in order to reduce communication costs. The approach to assign sites for the fragment with the HASH JOIN involves allocating enough sites so that the hash table used by the join fits in main memory. In this example, two sites are sufficient to meet this requirement. Finally, F1 is assigned to sites 4 and 5, as the metadata informs the optimizer that these sites are the only ones capable of executing the blast operation call. In summary, decisions involving the assignment of sub-plans to sites are governed by the resource requirements of the former (notably in terms of memory), and the capabilities of the latter (i.e., the data and computational resources that sites are able to provide).

2.1.3 Stream Query Processing

Another recent development in the technological landscape is the increasing availability of continuously updated data channels. Examples include market data feeds, news feeds (e.g., RSS-based ones), data obtained from global positioning satellites, or from mobile devices. These scenarios are challenging to database technology, as data takes the form of streams of indefinite cardinality that potentially may not cease to arrive, rather than being stored and indexed as is traditionally the case for subsequent retrieval. Given that it is usually not feasible to store all of the data, it may need to be processed on-the-fly. Furthermore, given that data often relates to a continuously changing world view, a query may need to be re-evaluated, in principle, every time that data is updated, i.e., in response, or as a reaction to, a change in the world as reflected in the newly-updated data. For such reasons, data stream management has become a prominent area of database research in recent years, resulting in systems such as Aurora [BBC+04, ZSC+03], STREAM [ABB+03], TelegraphCQ [CCD+03], Gigascope [CJSS03] and Borealis [AAB+05, TAc+06]. A survey of issues raised is [GO03]. Data stream management differs from the classical case in that:

- Data is pushed to the query processor rather than being pulled on request.
then subsequently pull it on an as-needed basis.

- Tuples are ordered, whereas classically a relation is a set (or a bag). Thus, each tuple comes with ordering information (e.g., a timestamp).

- Queries are often posed and need to be answered on live data (i.e., data that if not used is lost forever) because storing all the data is impossible (even in principle), and, pragmatically, stream data often relates to events happening in the real world that one wishes to monitor and react to periodically.

Because a stream differs from a relation in that it has indefinite (potentially infinite) cardinality, significant challenges arise for query processors, because many classical operators, such as join and sort, are blocking operators (as defined in Section 2.1.1). As a consequence, for a join or a sort over a data stream to be meaningful one needs to define a subsequence on the stream and ensure it is available for the duration of the operation. This changes the semantics of blocking operators from having the scope of entire extents to having the scope of windows (often, moving windows) on those extents. As a result, a window (i.e., a bounded subsequence of the stream with definite cardinality) needs to be defined for traditional relational operators to be applicable. Also, since it is not viable to store all the tuples from a stream, often a summary is kept instead, giving rise to probabilistic queries that only give approximate answers in the sense that, as in statistical science, they represent an inference from a sample of a population to which one has no access [ABB+03].

Examples of streaming data applications include analysis of stock market ticker feeds and network traffic-monitoring applications. Streaming data lends itself to so-called continuous queries, which are repeatedly evaluated, and produce a stream of results. For example, a user may pose queries such as:

- Every five seconds, report the average of all the FTSE company share prices.

- Calculate the maximum reported price of British Airways stock over the last 5 minutes every 30 seconds (i.e., over a moving window containing 5 minutes worth of data which slides in the sense that it gets updated every 30 s).

- Every minute, report the current price of a stock if it is greater than the average price reported during the last 5 minutes. This is an example of a
landmark query, which involves comparing present data to recent data (by correlating the incoming tuples from a stream with the tuples in a window containing recent data) [GKS01].

This mode of repeatedly evaluating a query contrasts to that of queries over stored data, which tend to be evaluated only once. Naturally, it is also possible (but less commonplace) for a query to be evaluated only once over a stream, in which case it may be referred to as a snapshot query. However, such queries are not the focus of the work in this dissertation.

In stream applications, data sources are typically geographically dispersed, and hence stream data management requires the use of distributed processing techniques. In Medusa [CBB+03, ZSC+03] and Borealis [AAB+05], query evaluation takes place in a distributed fashion as in DQP. This gives rise to optimization challenges including adaptively balancing the workload across various nodes in response to changing conditions during query evaluation, such as a change in tuple production rate from a data source [CBB+03]. Also, in the presence of multiple queries executing simultaneously, the optimizer may identify common sub-plan fragments which may be shared, thereby enabling an improvement in performance.

2.1.4 Adaptive Query Processing

Traditionally, query processing has followed an optimize-then-execute approach, whereby a QEP is generated based on knowledge available at compile time, and therefore cannot take into account changes in conditions during query evaluation. The knowledge alluded to includes assumptions such as statistics about the data (which will impact cardinality estimates), predictions about the cost of operators, and information about the current availability of resources. The performance (and indeed viability) of the QEP hinges on the accuracy of the assumptions/estimates made during QEP generation. Furthermore, particularly for queries that are evaluated some time after query optimization has taken place, such as queries embedded in applications that are used on a regular basis, or continuous queries over streams, the information about the conditions may also become outdated (e.g., if workloads vary, or if a server becomes unavailable), meaning that the QEP
determined initially may become far-from-optimal. In response to this issue, approaches have been proposed to monitor, and react in response to, changing conditions during query evaluation. These are described as adaptive query processing (AQP) techniques. Surveys of AQP are given in [GPFS02, BB05, DIR07].

A query processing technique is considered to be adaptive if it incorporates a feedback loop which is able to influence the behaviour of the query processor during the evaluation of the query. One approach to implement adaptivity involves using adaptive operators, in which case an operator monitors changes in the environment and is able to change its behaviour accordingly. For example, the Eddy operator [AH00] continuously routes tuples to a different operator according to the current conditions, effectively changing the order that tuples are sent to operators. Another approach employed to enable adaptivity in a query processor is to modify the QEP at runtime in light of feedback from the environment. For example, Envririyankul et al. [EPFL10] propose an approach in which a particular join physical operator may be replaced with a different type of join algorithm (e.g., a hash join with a nested-loop join) at specific points during query evaluation, while enabling work that has been carried out prior to the adaptation to be reused. In both cases, an important issue is how often to perform adaptations. Because performing an adaptation incurs an overhead, adapting too often may outweigh the benefits of the adaptation. Similarly, consideration needs to be given as to what points during query evaluation adaptation is viable, given that state needs to be preserved, and the correctness of results must not be compromised (e.g., because data is lost) during an adaptation. As previously mentioned, AQP is not the focus of the research contributions presented in this dissertation and is left as future work.

2.2 Sensor Networks

Applications in a variety of domains have been proposed for sensor networks, including environmental monitoring, military, security and surveillance, precision agriculture, industrial applications, smart buildings, asset tracking and supply chain management, and health monitoring. The best known applications are probably environmental monitoring applications, which have been deployed by
scientists to monitor natural phenomena of interest; a survey of such applications is presented in Hart et al. [HM06]. However, numerous other applications exist, e.g., car-park monitoring, which aims to detect potential accidents or crimes [NKC09]; fence monitoring [WTV+07], aimed at detecting potential breaches of security; the use of sensor networks in precision agriculture, e.g., the deployment in a vineyard [BBB04]; industrial applications, in which sensor networks perform predictive maintenance of machinery, e.g., in an oil tanker or a semiconductor plant [KAB+05]; and building monitoring, e.g., the deployment aimed at regulating light and temperature in a building to reduce running costs and improve occupant comfort [SKG+05]. In Section 2.3, a discussion of sensor network applications is presented, with a focus on their QoS expectations.

In a sensor network, the sensor nodes which comprise the network collaborate on performing a task, which typically involves collecting data about, or detecting and responding to events within, the environment in which they are immersed. Figure 2.5 presents the conceptual model of a sensor node assumed in this dissertation. In the diagram, a component drawn in a dotted line denotes an optional component, black lines within the node denote interactions between components, and grey lines indicate interactions outside the node (e.g., a node communicating with another node via radio, or a node sensing data about the environment). The components of a sensor node shown in Figure 2.5 are:
2.2. SENSOR NETWORKS

1. One or more sensors, which gather information (e.g., a pressure or temperature reading) about the physical world at the location where the sensor node is situated.

2. A processor that has the ability to perform computations (e.g., signal processing) on sensed data, or data received via the radio, and also controls the components in the node. The processor may have inbuilt volatile and/or persistent memory for storing data and/or programs.

3. A wireless communications device, which is used to send and receive data through the environment to and from other nodes. In the survey of sensor network deployments by Römer and Mattern [RM04], although the most common form of communication is using radio, other communication modalities such as light, laser and sound are found to be in use.

4. Optionally, an external persistent store, which is typically a flash module.

5. Optionally, an actuator, which is able to, in some way, alter the environment to achieve a desired effect (e.g., a sounder is activated to restrict cattle movement in the deployment in Butler et al. [BCPR04]).

Note that a sensor network may also have nodes with specialized roles. For example, there may be one (or possibly more) gateway nodes, which provide an interface to communicate with a computer outside the sensor network (e.g., an Internet connection via a mobile telephone network), where the collected data may be sent to, or commands to the sensor network sent from. If there are no gateway nodes in the network, it is possible for sensed data to be logged to persistent memory at a subset of the nodes in the network, and then manually collected using a portable device (to give an example of a possible mode of operation). Another type of node which may be present in a sensor network is relay nodes, which may not have any sensors, and are used for bridging communications between nodes in the sensor network in a multi-hop fashion, necessary in cases where direct communication by radio is impossible otherwise (e.g., [KR04]).

Figure 2.6 shows some concrete examples of the abstract sensor node in Figure 2.5 which, at time of writing this dissertation, are available on the market. In Table 2.2, some characteristics for these hardware platforms are shown. As can be seen, the low processing power and the scarce memory mean that each node is but a very limited computer. Note that a variety of external sensors can be
connected to each of these platforms. Also note that the radio range figures given in the table are for the default, in-built radio, although it is possible to connect an external radio with a longer range, assuming that sufficient energy is available to power it.

By and large, sensor nodes are powered by batteries, and the cost of replacing these could be high. For some deployments, due to dangerous conditions or inaccessibility, replacing batteries is impossible (e.g., in the case of the sensor network used for monitoring glaciers [MPE+06]). Therefore, special consideration must be given to the conservation of power for sensor network deployments to be practically feasible. If a sensor network depletes its energy sources prematurely, the lifetime of the network may be too short to be of any practical use.

As a result, energy-efficient techniques have been developed, with a particular focus on radio communications, since this functionality tends to dominate sensor
2.2. SENSOR NETWORKS

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Mica2</th>
<th>Tinynode</th>
<th>Imote 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tec09b</td>
<td>Sho09</td>
<td>Tec09a</td>
</tr>
<tr>
<td>CPU Speed</td>
<td>8MHz</td>
<td>8MHz</td>
<td>13-416MHz</td>
</tr>
<tr>
<td>RAM</td>
<td>4K</td>
<td>8K</td>
<td>32MB</td>
</tr>
<tr>
<td>Program memory</td>
<td>128K Flash (persistent)</td>
<td>92K Flash (persistent)</td>
<td>256K SRAM (volatile)</td>
</tr>
<tr>
<td>Data Flash</td>
<td>512K</td>
<td>512K</td>
<td>32 MB</td>
</tr>
<tr>
<td>Radio range</td>
<td>150m (outdoor)</td>
<td>150m (outdoor) 50m (indoor)</td>
<td>30m</td>
</tr>
</tbody>
</table>

Table 2.2: Some characteristics of the sensor nodes in Figure 2.6

node energy consumption. For example, in the networking literature, numerous link layer medium access control (MAC) protocols have been proposed (e.g., S-MAC [YHE04] and TEEM [gGCL+09]), which strive to avoid radio packet collisions and reduce the amount of time that the radio is listening for incoming packets, so as to enable the radio to enter into a low power sleep state for as much time as possible. Other techniques that aim to save energy by managing the radio include adjusting the radio transmission power appropriately, the use of topology control protocols, and routing protocols [BJ05]. Note that significant energy savings can be achieved by carrying out several short transmissions in a multihop fashion to span a longer distance, as the energy consumption required to power the radio increases by the square of the distance [PK00].

Given the much lower cost of CPU cycles from a sensor node’s point of view, Pottie and Kaiser [PK00] observe that, rather than sending (potentially unnecessary) raw data to the destination (as, e.g., in the ALERT flood detection system [ALE09]), in-network processing, i.e., performing computations and transformations on the sensed data within the sensor network, in a distributed manner, in order to reduce the volume of radio traffic can lead to significant energy savings. The benefits of in-network processing have been postulated for diverse applications. For example, Ledeczi et al. [LNV+05] describes a sensor network deployment which performs sniper localization, and signal processing takes places at the sensor nodes to determine the location of a sniper; Subramaniam et al. [SPP+06] propose a distributed outlier detection algorithm to clean sensor data or detect interesting events; and Madden et al. [MFHH02] propose a technique in which aggregation is performed in an incremental, distributed manner within the network.

The fragile and constrained nature of this computational fabric poses other challenges in addition to energy preservation. Their distributed nature means
that time synchronization is a concern. Sensor nodes are also prone to failure and communications are unreliable. Addressing such issues, however, is beyond the scope of this dissertation, as components that provide an abstraction of a robust network are assumed to exist that support the software artifacts described later on.

This section provided a brief overview of sensor networks as a highly resource-constrained, distributed computing platform, and the main issues relating to them. In the next section, the requirements of sensor network applications, and their QoS expectations, are discussed.

### 2.3 QoS in Sensor Network Applications

In this section, some of the requirements of sensor network applications are elicited, on the basis of some paradigmatic examples, with the aim of identifying relevant QoS variables. Application requirements are divided into functional requirements and non-functional requirements. *Functional requirements* specify the specific services that a system should provide, and how a system should respond to particular inputs. *Non-functional requirements* refer to behaviours, performance or constraints that the system should exhibit. These requirements are usually specified about the overall system, rather than specific parts of the system. This broad category of requirements may include usability, reliability, portability and legislative requirements [Som07]. For example, in an airline booking system, a functional requirement may be for a travel agent to be able to order a special meal for a passenger, and for a notification to be sent to the catering department. Non-functional requirements may include the need for availability of the system, security and authentication, and for the system to be able to handle a specific volume of transactions per second.

QoS expectations fall under the category of non-functional requirements. Although, broadly speaking, the distinction between QoS and other types of non-functional requirements is not always clear, in the context of this dissertation they refer to quantifiable performance variables which directly affect the experience of the user. Note that, in the networking community, the use of the term QoS differs somewhat, as it is used to refer to low-level network properties such as bandwidth, latency and jitter. These are not the focus of this dissertation, as a user is unlikely to be directly concerned about them, although they will of
Römer and Mattern [RM04] present a survey in which they specify dimensions for sensor network applications, and identify several non-functional requirements, including some of the QoS variables considered in this dissertation. Some of these dimensions are considered in this section, and are summarized in Table 2.3. The example applications are based on:

- the Great Duck Island deployment [MCP+02], used to gain an insight into the nesting patterns of the Leach’s Storm Petrel, and the micro-climate surrounding their habitats.

- the ZebraNet deployment [ZSLM04] which tracks zebra migration in Kenya. Sensor nodes with a GPS were fitted onto zebras using collars, and their position logged periodically while they roamed freely. In this application, the connectivity of the sensor network was intermittent, and data could only be transmitted between nodes when they came into contact with each other.

- the Glacier Monitoring deployment in Norway, aimed at gaining a better understanding of the dynamics of glacier motion, and the impact of global warming on glaciers [MOH04]. Nodes were deployed by drilling holes in the glacier at different depths. The nodes transmit the data collected about their movement via radio to the gateway.

- the Sniper Localization deployment described by Lédeczi et al. [LNV+05], in which sensor nodes with acoustic sensors are used to collaboratively identify the position of a sniper, based on the soundwaves emitted by a gunshot.

- the Volcán Reventador deployment in Ecuador [WALJ+06], which collects information about volcanic and seismic events.

- the Crowden Brook deployment in the Peak District, UK [MPL+07], which studies the hydro-dynamics of surface water drainage.

- the Cane Toad monitoring deployment in Australia [HBC+09], which identifies different species of frogs in order to carry out a census. Complex signal processing takes place on the nodes with this application.
Table 2.3 presents a summary of the applications listed above, including, wherever possible, expected values for the QoS variables. Each QoS variable is now discussed in turn, with reference to the applications in Table 2.3.

The acquisition interval $\alpha$ is the amount of time between consecutive sensor data acquisitions; it therefore determines the temporal resolution at which data is collected. For some event-detection applications, a short acquisition interval may be required, otherwise an event may be missed. Indeed, in the case of the Sniper Localization, Cane Toad Monitoring and Volcán Reventador deployments, which contain acoustic sensors, sensing is effectively continuous in order to identify events of interest (i.e., a gunshot, a frog vocalization, or a volcanic explosion, respectively). In such cases, several components of the sensor node may remain in a low power state until they receive notification that an event of interest is occurring, in order to avoid depleting the battery prematurely. This is the case of the Sniper Localization deployment. In contrast, other applications only require data at a much lower temporal resolution. For example, the Crowden Brook deployment, only acquires data once every 15 minutes, and the Glacier Monitoring deployment, a reading is taken once every 4 hours. These values correspond to the temporal granularity at which the scientists responsible for the deployment consider it meaningful and useful to monitor the development of the phenomena under study, traded off against the constraints imposed by the hardware and energy resources available.

The delivery time $\delta$ is the amount of time spent between data being acquired, and the corresponding processed results reaching the gateway node. It, essentially, stipulates the urgency with which the data must reach the user. Note, however, that it is not always the case that data gets sent to the user via the gateway node. In the cases of the Volcán Reventador and Crowden Brook deployments, non-event timeseries data is logged locally at the flash memory of the source nodes, and data is collected manually; only event notifications are sent to the gateway node, as a means to conserve radio energy. In most of the applications above such an approach is not viable. For example, in the Glacier Monitoring application, the nodes are non-recoverable, so data needs to be sent by radio to the gateway node. Similarly, in the Great Duck Island application collecting the data manually would involve disturbing the birds. In the cases when data is sent to the gateway, the delay which can be tolerated to receive the data varies significantly between applications. In the sniper detection application, being informed quickly
### Table 2.3: Characteristics of existing sensor network deployments.

<table>
<thead>
<tr>
<th>Application</th>
<th>Sensing Modalities</th>
<th>Acquisition Interval $\alpha$</th>
<th>Delivery time $\delta$</th>
<th>Energy Stock $\epsilon$</th>
<th>Lifetime $\lambda$</th>
<th>Main QoS of concern</th>
<th>Description of Sensing Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>Great Duck Island [MCP+02]</td>
<td>Light, temperature, infrared, relative humidity, and barometric pressure</td>
<td>5 min to 1 hour, depending on query</td>
<td>Real-time</td>
<td>2 AA batteries (2200 mAh) per node</td>
<td>9-12 months</td>
<td>Lifetime</td>
<td>100+ nodes at fixed locations</td>
</tr>
<tr>
<td>Zebranet [ZSLM04]</td>
<td>GPS</td>
<td>8 minutes</td>
<td>Best effort</td>
<td>2 mAh lithium ion battery per node charged by solar panels</td>
<td>1 year</td>
<td>Lifetime</td>
<td>7 nodes moved by zebras over large area</td>
</tr>
<tr>
<td>Glacier Monitoring [MOH04]</td>
<td>Pressure, temperature, orientation (tilt in three dimensions), external conductivity and strain gauge</td>
<td>4 hours</td>
<td>1 day</td>
<td>Lithium Thionyl Chloride battery</td>
<td>1 year</td>
<td>Lifetime</td>
<td>9 nodes moved by ice</td>
</tr>
<tr>
<td>Sniper Localization [LNY+05]</td>
<td>Acoustic</td>
<td>Continuous</td>
<td>Two seconds or less</td>
<td>Two AA batteries per node</td>
<td>Weeks or months</td>
<td>Acquisition interval, delivery time, lifetime</td>
<td>60 nodes over 80m x 80m area</td>
</tr>
<tr>
<td>Volcán Reventador [WALJ+06]</td>
<td>Seismometer and microphone</td>
<td>10ms</td>
<td>Non-event data stored at source node; events notified to gateway within 490s</td>
<td>Two D-cell batteries per node</td>
<td>19 days (batteries replaced weekly)</td>
<td>Acquisition interval, delivery time</td>
<td>16 nodes over 3Km section of mountain</td>
</tr>
<tr>
<td>Crowden Brook [MPL+07]</td>
<td>Soil moisture, temperature, battery power; two nodes measure rainfall</td>
<td>15 min</td>
<td>Interesting events reported within 1 hour Time-series data logged at source nodes</td>
<td>19 Ah 3.6V Lithium Thionyl battery per node</td>
<td>59 days</td>
<td>Lifetime</td>
<td>4 nodes with 250m between each node</td>
</tr>
<tr>
<td>Cane-toad monitoring [HBC+09]</td>
<td>Acoustic</td>
<td>0.1 ms</td>
<td>15s</td>
<td>2 AA batteries for sensing nodes, LiIon battery for nodes which carry out complex signal processing</td>
<td>4 days</td>
<td>Acquisition interval, lifetime</td>
<td>Static nodes, location not specified</td>
</tr>
</tbody>
</table>
of the location of the sniper is paramount, as immediate action is required. The
delivery time is less of a concern for the Crowden Brook deployment, which
reports events within one hour, as gaps in the timeseries are tolerable, and it may
take a few days before someone is able to attend to node failure. In cases where
data is being collected for posterior analysis, the delivery time to the gateway is
inconsequential, e.g., Zebranet and Glacier Monitoring. It is noted that, possibly,
the Cane Toad monitoring and Great Duck Island deployments do not require
such short delivery times, and indeed, could increase their lifetimes by accepting
a lower latency.

The total energy $\epsilon$ is defined as the sum of the amount of energy required
for every node in the sensor network to execute a specific task for a specific
period. While it was not possible to obtain precise figures for $\epsilon$ in Table 2.3,
it is evident that preservation of the energy stock is a significant concern. For
example, in the Zebranet deployment, the energy stock is limited by the fact
that the sensor node collars must not be too heavy. Furthermore, using a GPS
sensor, to obtain position information is very energy intensive, so the approach
taken is to recharge the battery using solar cells. In the Cane Toad monitoring
project, the high frequency acoustic sampling and complex signal processing are
also energy intensive, and the authors report having to change the batteries every
4 days – in order to mitigate this, and extend the lifetime of the deployment, the
authors propose the use of solar panels. Another, related, variable is the average
energy consumption per node in the sensor network, denoted $\bar{\epsilon}$. Note, however,
that neither total energy nor average energy take into account the evenness of
energy depletion throughout the nodes in the network.

A significant concern, and by far most commonly studied one in the sensor
network literature, is the lifetime $\lambda$, defined as the amount of time for which suf-
ficient nodes have enough energy to execute a given task. This variable does take
into account whether the nodes in the sensor network are evenly depleted of their
energy supply. This is important for practically all sensor network applications.
For example, in the Great Duck Island deployment, replacing batteries would
involve disturbing birds, and possibly distorting subsequent sensor readings. In
the Glacier Monitoring deployment, the nodes are non-recoverable, so the lifetime
of the sensor network determines the amount of data that can be collected, and
consequently, the cost-benefit ratio of the deployment. With the Crowden Brook
deployment, changing batteries takes at least half-a-day, due to the remoteness
2.4 Conclusion

The database research community was quick to spot the fact that sensor network data management could be understood, and supported using such advances as described in Section 2.1 by viewing a sensor network as an environment for distributed stream query processing. Thus, a device may be seen as a remote data source if it has sensing capabilities, a remote computational resource if it has processing capabilities, and as a node in a network if it has communication capabilities. The sensor nodes considered in this dissertation have all three kinds of capability. It is therefore generally accepted by the database research community that it is useful to view a sensor network as a distributed database whose data sources are either the sensing devices or the stores inside the sensor nodes.
This allows users (or applications) to pose declarative queries in an ad-hoc fashion, without needing to concern themselves with the underlying location of the data sources, or how the data is to be collected, essentially providing a layer of abstraction over the sensor network.

A SNQP, therefore, has characteristics in common with the computational infrastructure over which a DQP engine runs. However, there are also fundamental differences:

- **The acquisitional nature of the query processing task.** Data is neither lying ready in stores nor is it received without being requested (as in classical streams). Tuples are generated by sources when explicitly requested by the QEP.

- **The energy-constrained nature of the sensors.** Preserving energy becomes a crucial requirement because it is a major determinant of network longevity, and requires the executing code to shun energy-hungry tasks.

- **The nature of the communication links.** In DQP, communications are assumed to be cheap and reliable. In contrast, wireless links are not robust, and often cannot span the desired distances, so the data flow topology (e.g., as embodied in a query operator tree) needs to be overlaid onto some query-specific network topology (e.g., a routing tree of radio-level links) for data to flow from sensors to clients. While in the context of this dissertation it is assumed that the issue of lack of robustness is partly mitigated existing network management components, routing decisions are likely to have significant implications on the performance of a QEP.

- **The need to run sensor nodes according to data-dependent duty cycles.** Each element in the computational fabric must act in co-ordination with other elements on which it is dependent or that depend on it, thereby enabling energy management (e.g., by sending devices to energy-saving states until the next activity). Approaches such as the iterator model, which is both CPU- and memory-intensive, are likely to be unsuitable for the sensor network case.

Note that these points also distinguish a sensor network from infrastructures for stream processing (e.g., [ABW03] [ACc*03] [CJSS03]), which do not operate in resource constrained environments.
The above differences between SNQP and other modes of query processing imply that during the compilation of queries over sensor networks, in addition to the types of decisions made in DQP, novel types of decisions need to be made. Examples of these decisions are the need to consider the paths used for routing data, and the timing of tasks in the QEP; a comprehensive list is detailed in Section 3.1. Using the two-phase optimization architecture from DQP described in Section 2.1.2 as a basis, these decisions are incorporated as new steps in the query processing stack for sensor networks presented in Section 4.5.

Another fundamental difference between SNQP and other modes of query processing is that, for the latter, optimization is primarily concerned with optimizing for response time. However, in the case of sensor networks, applications are diverse and have varied QoS expectations (e.g., for the Glacier Monitoring application, lifetime is the main concern, and delivery time is inconsequential, whereas for the Sniper Localization application, delivery time is paramount). Furthermore, due to the limited resources available, in SNQP there are trade-offs involved that are much starker than in the DQP case. Intuitively, a lower acquisition interval should increase the total energy required to execute a QEP; similarly, a lower delivery time should decrease the lifetime of a sensor network application. This provides a compelling case for SNQP to be QoS-aware in order to be suitable for a broad range of applications, and for the QoS-aware SNQP presented in Chapter 6 to exhibit alternative decision-making query planning strategies in order to be able to reconcile multiple and conflicting QoS concerns.

As discussed in the next chapter, current state-of-the-art SNQP systems (as represented by TinyDB [MFHH05], Cougar [FSG02] and SNQL [BLM+07]) are limited in terms of responses to the challenges above, as they do not make most query planning decisions explicitly, and furthermore, even when they do so, fail to consider diverse QoS expectations.
Chapter 3

Related Work

The previous chapter presented technical background on different modes of query processing, on sensor networks, and on the QoS expectations of sensor network applications. At the end of the chapter, a brief discussion was provided about the similarities to other forms of query processing and the novel, distinctive challenges of query processing over a sensor network. It was identified that SNQPs need to consider new types of decisions when compiling a query, such as routing and the timing of tasks, and also need to take diverse QoS expectations into account. In this chapter, the focus turns to potentially competing research.

Firstly, in Section 3.1 concepts and terminology are introduced that are used in this chapter to discuss related work, and throughout the remainder of the dissertation. Then, Section 3.2 compares and contrasts existing SNQPs with respect to the QoS expectations that they consider, in order to evaluate how effectively they would meet the broad range of requirements of sensor network applications identified in Section 2.3. Also, SNQP techniques are reviewed with a focus on which query planning decisions they make explicitly, and the stage in the process of query compilation/evaluation that these are made. Section 3.3 presents a survey of QoS in classical, distributed and stream query processing, in order to identify the variables considered to be of relevance, and the approaches used to consider these variables during query planning. This is followed by an overview of techniques that have been employed to implement alternative decision-making behaviours in query optimizers (Section 3.4) insofar as this seems necessary for the aim of incorporating QoS-awareness into a SNQP. Finally, Section 3.5 summarizes the chapter and highlights gaps in the research landscape and promising approaches to implementing QoS-awareness.
### 3.1 Framework for Comparison

This section introduces concepts and terminology and, where relevant, notation used to characterize SNQPs. Section 3.1.1 considers QoS expectations, and Section 3.1.2 concepts relating to query optimization.

#### 3.1.1 QoS Expectations

Table 3.1 summarizes the QoS variables deemed to be of relevance in sensor network applications identified in Section 2.3. In this dissertation, QoS expectations of two types are considered, viz., specifying QoS variables as optimization goals, or specifying constraints over QoS variables. By *optimization goal* is meant specifying a QoS variable whose desired value should be as high or as low as possible. The optimization goals considered that correspond to the QoS variables in Table 3.1 are \{min \(\alpha\), min \(\delta\), min \(\epsilon\), min \(\bar{\epsilon}\), max \(\lambda\)\}, which specify that the acquisition interval should be minimized, that the delivery time should be minimized, that the total network energy consumption should be minimized, that the average network energy should be minimized, and that the maximum lifetime should be maximized, respectively. Zero, one or multiple optimization goals may be associated with an application. For example, the Volcán Reventador deployment described in Section 2.3 may be characterized as having a min \(\delta\) goal for interesting events relating to volcanic eruptions.

The performance of an application may also be dictated by constraints over QoS variables. A *constraint* is either a lower bound, upper bound, or equality

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\alpha)</td>
<td><em>Acquisition interval</em>, defined as the length of time between sensor acquisitions.</td>
</tr>
<tr>
<td>(\delta)</td>
<td><em>Delivery time</em>, defined as the maximum interval in time units between the time of sensor acquisition and the time for result tuples to be delivered to the user.</td>
</tr>
<tr>
<td>(\epsilon)</td>
<td><em>Total energy consumption</em>, defined as the total amount of energy stock required by all sites in the sensor network to evaluate a particular query for a specific amount of time.</td>
</tr>
<tr>
<td>(\bar{\epsilon})</td>
<td><em>Average energy consumption per node</em>, defined as the mean amount of energy stock required per site in the sensor network to evaluate a particular query for a specific amount of time.</td>
</tr>
<tr>
<td>(\lambda)</td>
<td><em>Network lifetime</em>, defined as the length of time the query can run for until the first site runs out of energy.</td>
</tr>
</tbody>
</table>

Table 3.1: Definition of QoS variables considered.
CHAPTER 3. RELATED WORK

condition, on a \textit{QoS} variable. For example, for the Glacier monitoring application, the minimum lifetime for the deployment to be effective is 1 year \((\lambda \geq 1 \text{ year})\), and for the Sniper Localization application, that the delivery time should be no more than 2 seconds \((\delta \leq 2 \text{ seconds})\). If an application is unable to meet the QoS constraints, its performance is likely to be unsatisfactory from the user’s point of view. Throughout this dissertation, a \[QoS\] expectation is denoted as a tuple \(\langle \theta, \kappa \rangle\) where \(\theta \subseteq \{\min \alpha, \min \delta, \min \epsilon, \min \bar{\epsilon}, \max \lambda\}\), \(\kappa\) is a set of expressions in the form \(q \oplus k\) where \(q \in \{\alpha, \delta, \epsilon, \bar{\epsilon}, \lambda\}\), \(\oplus \in \{=, <, >, \leq, \geq\}\), and \(k\) is a constant.

3.1.2 Query Optimization

This section enumerates the types of optimization decisions that need to be made in order to generate a sensor network query evaluation plan \(\text{QEP}\), and the different ways in which these decisions may be made. Some of these decisions are also present in \(\text{DQP}\) given that a sensor network \(\text{QEP}\) is also of a distributed nature, and others are concerns unique to sensor networks that arise from their highly constrained nature. It is necessary to consider these decisions because they may impact the performance of the \(\text{QEP}\) in terms of QoS variables. The decisions are:

\textbf{D1} \textit{Operator tree composition}, which involves deciding which operators to re-order or substitute, in order to obtain a more efficient operator tree (e.g., pushing a selection down the tree of operators to reduce the size of intermediate results);

\textbf{D2} \textit{Algorithm selection}, which involves selecting concrete operator implementations to assign to each abstract operator (e.g., deciding to use a hash-join algorithm to execute a join);

\textbf{D3} \textit{Routing}, which involves deciding which paths in the sensor network to use for transporting tuples from the sources nodes to the sink node (e.g., avoiding a node that is low on energy);

\textbf{D4} \textit{Partitioning}, which involves deciding how to fragment the query operator tree, based on operators that need to be executed at different locations, or on the need for data redistribution;

\textbf{D5} \textit{Deciding the number of instances for each operator}, which involves deciding on the number of copies to use for each operator. This may be a consequence
of the data sources being spread across different spatial locations, in which case the operator instance is referred to as a *d-instance* (for distributed operator instance), or used to enable partitioned parallelism with a view to improving performance or balancing the workload evenly when evaluating resource-intensive operators, in which case the operator instance is referred to as a *p-instance* (for parallel operator instance);

**D6** *Operator instance assignment*, which involves deciding which node to assign each operator instance to;

**D7** *Timing of sensor acquisitions*, which involves deciding when (or how often) to acquire sensor readings. This may be directly based on the acquisition interval specified by the user QoS expectations, or determined indirectly in the light of other QoS expectations (e.g., the minimum lifetime);

**D8** *Timing of computations*, which involves deciding when (or how often) to execute each fragment while respecting the precedence constraints between fragments. This is usually evident from the shape of the query operator tree, i.e., the leaf fragments need to be evaluated first;

**D9** *Timing of communications*, which involves deciding when (or how often) to engage in data transport while respecting the order of communications between nodes in the sensor network. Communications may be postponed, so that tuples from different acquisition times are buffered and sent in batches, resulting in larger data packets being sent less often, thus conserving energy, but decreasing the recency of results received by the user.

**D10** *Load shedding policy*, which involves deciding how to select tuples to discard when buffers are full.

Note that some of these decisions may be inter-related. For example, decisions **D4**, **D5** and **D6** may be taken in conjunction with each other as they all relate to the placement of operators to sensor nodes, and are therefore referred to as *where-scheduling* decisions. Similarly, decisions **D7**, **D8** and **D9** are concerned with the timing of tasks in the QEP and are referred to as *when-scheduling* decisions. It is possible for each decision to be made by different agents and at different points in the query compilation and evaluation process. For example, a decision may be made in the following different modes:
M1 *Explicitly by the user*, when specifying the query. In this case, the instruction by the user may be viewed as a compiler directive informing the optimizer how to make a decision, rather than giving the optimizer free rein to decide for itself.

M2 *Explicitly by the query optimizer*, at compile time, in which case the QEP shipped to the sensor network nodes contains the outcome of the decision made. In order to make a good decision at compile time, global knowledge of the sensor network metadata is required, and this is assumed to be provided by an underlying network management layer throughout this dissertation. Also, accurate cost models, of the nature described in Appendix B, that enable prediction of memory, energy consumption and time are assumed.

M3 *Explicitly by the query evaluation engine*, at runtime, in which case, query planning decisions are postponed until evaluation time, or decisions made at compile time may be adapted at runtime if conditions change or the initial assumptions made during query compilation turn out to be inaccurate. These may be made either in a centralized or distributed fashion.

M4 *Implicitly*, in which case no decision is made, the query processor simply acts in a predetermined manner.

Note that while M1, M2 and M3 may happen in conjunction with each other, M4 is mutually exclusive with the other options. Section 3.2 presents a survey of sensor network query processing techniques, and characterizes SNQPs in terms of the alternatives mentioned above. However, the reader is reminded that that the research contributions described in this dissertation are concerned solely with centralized decisions made at compile time; runtime decisions [AH00, GPFS02, EFP06, GSP+09, EPFL10] are a potentially fruitful topic for future work.

### 3.2 The State of the Art in Sensor Network Query Processing

Now that QoS expectations and the types of decisions relevant to SNQPs have been characterized, as well as the stages within the process at which these decisions may be made, this section compares and contrasts the approach taken by existing techniques in the research literature. Both SNQPs (i.e., systems that are complete
in the sense that they provide all the functionality required from the initial parsing of a query to the delivery of results by the evaluation of the QEP, as well as piecewise techniques (i.e., those that could form a constituent part of a SNQP) are considered with respect to how each of these decisions are made and the QoS expectations supported. The complete SNQPs considered are Cougar [BGS00, YG03], TinyDB [MFHH05] and SNQL [BLM+07]. It is noted that while a fairly detailed system description is provided for TinyDB, for the other SNQPs fewer details about their query planning techniques are available.

**QoS Expectations Supported**  For the three query processors described, queries have a syntax inspired by SQL. Queries are posed against a schema with a separate extent per sensor type in the case of Cougar and SNQL, whereas in TinyDB all sensor readings and certain metadata values (e.g., the battery level of a sensor node) are attributes of a universal relation called sensors, which is horizontally partitioned across all sensor nodes, with one row per acquisition time for each node. Examples of TinyDB queries are presented in Figure 3.1. For example, Q1 is a continuous query that requests the node identifier and voltage level of the nodes whose voltage is below a certain threshold every 10 minutes. Note the use of the additional clause to specify QoS expectations such as the acquisition interval and desired lifetime. For example, for Q2 it is specified that the system should choose an acquisition interval such that the QEP is able to run for 30 days with the energy available. In addition, Q3 requests the same minimum lifetime and a minimum acceptable acquisition interval; in case that the lifetime cannot be met with the minimum acquisition interval, the delivery time of results may be increased, and load-shedding may also take place.

For each system, the QoS expectations that are supported are shown in Table 3.2 using the QoS variables $\alpha, \delta, \epsilon$ and $\lambda$, denoting acquisition interval, delivery time, total energy and lifetime, respectively, as defined in Table 3.1. The variables $a$, $d$ and $l$ are constants that are provided, directly or indirectly, by the user within the query syntax. A vertical bar is used to indicate alternative QoS specifications that are supported. For example, Cougar only allows a fixed acquisition interval to be specified, and the optimization goal is implicitly maximising

---

1Note that in the Cougar project there are a number of publications which discuss specific techniques, including routing and communications timing, in order to conserve energy [LYD+07], and also about multiple query optimization [LYD+05]. As they are not discussed in the broader context of the Cougar query processor per se, where relevant they are treated as piecewise optimization techniques for the purposes of this survey.
**Q1**: SELECT nodeid, voltage  
WHERE voltage < k  
FROM sensors  
SAMPLE PERIOD 10 minutes

**Q2**: SELECT accel, mag  
FROM sensors  
WHERE accel > c1  
AND mag > c2  
LIFETIME 30 days

**Q3**: SELECT nodeid, accel  
FROM sensors  
LIFETIME 30 days MIN SAMPLE RATE 20 s

Figure 3.1: Example TinyDB queries from Madden et al. [MFHH05].

| **Cougar** | $\langle \text{max} \lambda, \{\alpha = a\} \rangle$ |
| **TinyDB** | $\langle \text{max} \lambda, \{\alpha = a\} \rangle \mid \langle \text{min} \alpha \{\lambda \geq l\} \rangle \mid \langle \text{min} \alpha, \{\lambda \geq l, \alpha \leq a\} \rangle$ |
| **SNQL** | $\langle \text{max} \lambda, \{\alpha \geq a, \delta \geq d, \lambda \geq l\} \rangle$ |

Table 3.2: Range of QoS expectations purportedly supported by SNQPs.

lifetime. TinyDB allows a user to specify either a fixed acquisition interval (as in **Q1**), a minimum lifetime (as in **Q2**), or a minimum lifetime and an upper bound on the acquisition interval (as in **Q3**). The optimization goal changes implicitly depending on the constraints used; it cannot be set independently of the QoS constraints. In SNQL, the user specifies an acquisition interval, a **SendInterval** which stipulates how often query results are transmitted to the gateway node (and therefore determines the delivery time), and a schedule for the evaluation of the query (from which the required lifetime is derived)\(^2\). In summary, none

\(^2\)Note that, for SNQL, \(a\) and \(d\) are lower bounds for the acquisition interval and delivery time QoS variables, respectively (i.e., they stipulate the minimum \(\alpha\), and minimum \(\delta\), respectively). Therefore, \(a\) and \(d\) correspond to the performance exhibited in terms of these variables at the start of query evaluation (when nodes have a high energy and memory availability). During query evaluation, performance in terms of \(\delta\) may adaptively degrade in an unbounded manner in order to meet the lifetime \(\lambda\).
Table 3.3: Approaches to making query planning decisions.

<table>
<thead>
<tr>
<th>Decision Type</th>
<th>Cougar</th>
<th>TinyDB</th>
<th>SNQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>(D1) Operator tree composition</td>
<td>Optimizer</td>
<td>Optimizer</td>
<td>Optimizer</td>
</tr>
<tr>
<td>(D2) Algorithm selection</td>
<td>Implicit</td>
<td>Implicit</td>
<td>Not stated</td>
</tr>
<tr>
<td>(D3) Routing</td>
<td>Optimizer</td>
<td>Implicit</td>
<td>Evaluation Engine</td>
</tr>
<tr>
<td>(D4) Partitioning</td>
<td>Optimizer</td>
<td>NA</td>
<td>Optimizer</td>
</tr>
<tr>
<td>(D5) Number of operator instances</td>
<td>Optimizer</td>
<td>Implicit</td>
<td>Optimizer</td>
</tr>
<tr>
<td>(D6) Operator instance assignment</td>
<td>Optimizer</td>
<td>Implicit</td>
<td>Optimizer</td>
</tr>
<tr>
<td>(D8) Timing of computations</td>
<td>Implicit</td>
<td>Implicit</td>
<td>Implicit</td>
</tr>
<tr>
<td>(D9) Timing of communications</td>
<td>Implicit</td>
<td>Implicit</td>
<td>User, Evaluation Engine</td>
</tr>
<tr>
<td>(D10) Load shedding policy</td>
<td>Not stated</td>
<td>User</td>
<td>Not stated</td>
</tr>
</tbody>
</table>

The SNQPs allows the user to explicitly specify an optimization goal as such, and the types of constraints allowed are restricted.

In addition to user-specified QoS expectations, certain fixed, built-in QoS-related assumptions are exhibited by the systems. Generally speaking, the systems are concerned primarily with lifetime or energy, and this is the purported optimization goal in most cases. Note, however, that Cougar and TinyDB send their data to the sink as soon as it is acquired, a design choice which means that for some applications, the delivery time is shorter than required, and a longer lifetime may have been obtained by postponing the transmission of data, and/or by sending larger packets less frequently.

**Approaches to Making Query Planning Decisions** Table 3.3 summarizes how specific types of optimization decisions are made to generate a QEP by Cougar, TinyDB and SNQL. Recall from Section 3.1 that these may either be specified by the user (denoted User), determined by the optimizer at query compile time (denoted Optimizer) and/or determined by the evaluation engine at query runtime (denoted Evaluation Engine). Alternatively, it is possible for the decision to be made in a pre-determined manner (denoted Implicit), or not to be applicable (denoted NA). The approaches to making each decision type, and any
QoS-awareness exhibited, are now discussed for the three SNQPs and for other piecewise solutions in the literature.

**Operator Tree Composition**  In TinyDB, the optimizer reorders acquisition and selection operations, so that the QEP takes less energy-intensive sensor readings first. This is done in case the tuple is discarded by the predicate because the reading taken makes the latter evaluate to false. For example, in Q2 in Figure 3.1 if an accelerometer reading consumes 0.0048 mJ, and a magnetometer reading consumes 0.2595 mJ, then the former is sensed first, and the latter only needs to be sensed if the accelerometer reading passes the predicate. This approach aims to reduce energy consumption.

**Routing**  In Cougar, the QEP is fragmented. Sensor nodes within a fragment form a cluster, and send data to the cluster leader. The cluster leaders then send data to the gateway. Depending on the operation being executed, different routing strategies may be used within a cluster. For example, for a fragment performing an AVG aggregation incrementally, a tree is used as duplicate tuples would distort the computed result. In the case of a MAX aggregation, which is not sensitive to duplicate tuples, a graph is used for increased robustness. Therefore, it would appear that routing decisions in Cougar are made explicitly, although details are not given about this.

Demers et al. [DGR+03] analyze characteristics of routing trees that are optimal for two QoS variables, viz., total energy consumption and network lifetime. It is noted that in order to maximize lifetime, the number of messages sent and received by the nodes in the network must be as even as possible throughout the routing tree. However, an algorithm for generating trees that satisfy different QoS desiderata is not proposed, nor is an experimental evaluation presented that demonstrates the effectiveness of these different properties.

In TinyDB, the QEP is not fragmented. All identical copies of the single sent to participating nodes are connected via a tree-shaped overlay network. Routing is trivial as it simply involves each node forwarding data to its parent in the tree. The query evaluation engine forms the routing tree at runtime by selecting the closest neighbour to the sink with the best link quality. This predetermined behaviour does not explicitly consider different QoS variables.

---

3This publication is associated with the Cougar project, but it is unclear as to whether the techniques described therein are incorporated into the Cougar query processor.
3.2. THE STATE OF THE ART IN SNQP

Turning to research on individual techniques, Synopsis Diffusion [NGSA04] uses a graph rather than a tree for routing tuples to the gateway. While potentially this may lead to the double counting of tuples during aggregation, it provides robustness as in the case of a tree, when a node fails, results for the entire corresponding sub-tree may be lost until the tree is reformed. A mathematical technique is used to compute approximate answers for aggregates that is insensitive to duplicates. The increased robustness means that results are likely to be based on a greater proportion of the tuples than if a tree was used. This is extended in the Tributaries and Deltas technique [MNG05], in which the sensor network comprises two regions: nodes closest to the sink employ the Synopsis Diffusion technique, while nodes further away have a tree topology as in TinyDB. An adaptive technique is proposed in which the boundary between regions is adjusted, in order to trade-off the user-specified threshold of percentage of nodes contributing while keeping the answer as accurate as possible. This work is relevant because it shows how routing decisions can be made to trade off different user desiderata. However, the decisions take place at runtime and the desiderata are user Quality-of-Data (QoD) concerns (i.e., accuracy and data completeness), not the QoS variables related to performance that are the focus of this dissertation. Furthermore, as previously stated, it is noted that the issue of robustness is not considered within the research contributions of this dissertation, as this is assumed to be the concern of an underlying network layer.

Where-scheduling Cougar and SNQL produce partitioned QEPs in which certain nodes are assigned a specific fragment. Operations such as selection and aggregation are evaluated within the sensor network wherever possible, in order to reduce radio traffic. SNQL, however, has the limitation that joins which correlate data from different spatial regions may only be executed at the gateway. In Cougar, such joins may be assigned by the optimizer to execute in the sensor network if the estimated selectivity of the join predicates is likely to make the join a cardinality-decreasing operator. Both systems therefore make where-scheduling decisions explicitly, with the aim to minimize the amount of radio traffic, and ultimately reduce energy consumption.

In the case of TinyDB, QEPs are not fragmented, and the same plan is evaluated by all the sensor nodes. This limits the complexity of the QEPs that can be executed by TinyDB, and consequently, the expressiveness of the queries that
can be posed. In order to compose a QEP with more than one fragment, it is necessary to pose separate queries, in which the upstream queries write data to a materialization point, and the downstream query reads data from the materialization point.

Moving on to piecewise techniques, Bonfils and Bonnet [BB03] consider the issue of rescheduling operators in a QEP at runtime based on metadata available at the local node. A function for computing the cost is defined recursively as the accumulated cost of operators from the acquisition operator to the operator in question. The evaluation engine periodically monitors neighbouring nodes to see if it would be cheaper to execute an operator at a neighbouring node, and if so, the operator is placed on that node. This means that other query operators may need to modify their routing, in order to ensure that query evaluation continues without interruption. The cost models used are general in the sense that they could represent any type of cost; e.g., energy consumption or delivery time, and so could be useful in a QoS-aware setting.

In Srivastava et al. [SMW05] an algorithm for operator placement is proposed, which takes into account the cost and selectivity of each operator, in order to determine an optimal operator placement. The aim is to reduce the amount of data transmitted between nodes, and therefore decrease energy consumption. It assumes that the overlay topology is a tree, and both computational power and bandwidth increase as one moves from the leaves to the root of the tree. However, in a network without such a hierarchy, the falsification of the underlying assumption reduces the usefulness of the algorithm.

**When-scheduling** In Cougar, the acquisition interval is fixed by the user and query evaluation occurs as soon as data is acquired. When-scheduling decisions are therefore made in a pre-determined manner.

For TinyDB queries that are to be repeatedly evaluated, users may specify an acquisition interval $\alpha$ or a lifetime $\lambda$. If the user specifies a desired lifetime, as in $Q3$ in Figure 3.1, the optimizer computes an acquisition interval $\alpha_{comp}$ that is predicted to meet the lifetime, using an energy cost model. During query evaluation, $\alpha_{comp}$ is reviewed periodically by the evaluation engine, and adjusted accordingly, to ensure that the lifetime requirements are met. In addition to a desired lifetime, users may optionally specify an upper bound for the acquisition interval $\alpha_{max}$ using the `MIN SAMPLE RATE` clause. If $\alpha_{max}$ has been specified, and
3.2. THE STATE OF THE ART IN SNQP

$\alpha_{\text{comp}} \leq \alpha_{\text{max}}$, then $\alpha_{\text{comp}}$ is used as the acquisition interval. Otherwise, the lifetime cannot be met with the maximum acquisition interval specified. In spite of this, $\alpha_{\text{max}}$ is used as the acquisition interval, and load shedding takes place to reduce the volume of data transmitted and conserve energy in an attempt to meet the lifetime expectations. It is not clear to what extent this approach is effective at meeting the desired lifetime; this is not discussed, nor is experimental evidence provided by Madden et al. [MFHH05]. However, this approach may not be effective because energy needs to be expended in order to acquire data which is subsequently discarded. An alternative would be for the optimizer to return an error to the user, given that this could be predicted at query compilation time, rather than silently discarding tuples and compromising the completeness of the results, without guaranteeing that the required lifetime is achieved.

For the SNQL optimizer, the user specifies an acquisition interval, and a send time, which specifies the rate at which data should be sent from the source nodes to the gateway. The delivery time requirement can be derived from these two specifications. Additionally, a schedule specifies the times at which the query should be evaluated and allows a lifetime expectation to be derived. During query evaluation, energy availability is monitored by the evaluation engine. A function is used to determine the delivery time depending on energy level of the node. If the energy level is 100%, data is sent at the delivery time requested by the user. As the energy level decreases, the delivery time is increased. It is noted that in order to reduce energy consumption, the possibility of increasing the acquisition interval could also be explored. However, the acquisition interval is only increased if there is a shortage of available memory. Note that this approach is likely to lead to an increasing delivery time as the evaluation of the query progresses, a feature which may not be desirable in applications where a constant delivery time is required.

The Wave Scheduling approach proposed by Trigoni et al. [TYD+07] combines routing and communication scheduling decisions. It produces communication schedules and a routing plan for a query workload over diverse sources and sinks aimed at either reducing total energy consumption by trading off delivery time or vice-versa. The when-scheduling approach is not QoS-aware as it acts in the same way regardless of the QoS expectations. It involves dividing the sensor network into a grid of cells, and assigning concurrent time slots for communication between adjacent cells in the grid. The communication activities are scheduled
in phases, referred to as the North, East, South and West phases. These names indicate the direction in which tuples travel from the sending cell to the receiving cell. Careful scheduling is used to avoid radio interference between simultaneous communications. Routing decisions are QoS-aware, as they exploit the fact that the path with the shortest delivery time may not be the one with the shortest number of hops, due to the waiting time required for a node to reach the correct phase to send data in a particular direction.

An experimental study compares Wave Scheduling with tree routing and scheduling approaches such as those used by TinyDB, and found that the former performs favourably for both performance variables used. However, the evaluation was carried out under the assumption that there are multiple queries active in the network and a routing tree is formed for each query that overlaps with those used by the other queries. In the case of a single query, the tree-based scheduling and routing approach is likely to perform better, as long as the relevant QoS variable is taken into account. While Wave Scheduling is likely to contend well with an unknown workload, in which data needs to be sent in various different directions within the sensor network, it does not take into account the timing and precedence constraints of computations at each node, which means that a plan that is tailored for a specific query (or workload) is likely to be more efficient. It is also noted that only two QoS variables are considered by this approach, viz., total energy and delivery time. Lifetime, for which the key desideratum would be to avoid hotspots, i.e., overloaded nodes, is not considered.

Zadorozhny et al. [ZCK04, ZCS08] address a subset of the when-scheduling problem, viz., the timing of communications, by proposing an algebraic approach to generating schedules with as many non-interfering, concurrent communications as possible. A Data Transmission Algebra is defined to represent transmission schedules, which specify the order in which communications need to take place, which communications may take place concurrently, and which communications may take place in any order but not concurrently. An initial, naïve, schedule is derived from the query routing tree, and equivalence-preserving transformations are applied as part of a randomized search to derive a more efficient transmission schedule. Cost models are proposed to rank transmission schedules. The cost models proposed in the paper evaluate a transmission schedule in terms of response time, although similar cost models for other QoS variables could be defined to make this technique QoS-aware.
Wu et al. [WLLL09] propose a technique for scheduling communications in a query workload in order to maximize the system profit. The system profit is calculated by computing the ratio of a composite quality function to the energy cost of executing the query. The quality function is a combination of functions representing the lifetime, delivery time and data completeness. An algorithm is proposed that determines the execution order for the queries in a workload that maximizes the system profit. This approach allows a subset of when-scheduling decisions (i.e., the timing of communications) to be made in view of preferences relating to QoS variables expressed by a user.

**Approximate Query Answering** An approach that cross-cuts the types of decisions described above is the use of statistical techniques which result in approximate answers and trade off accuracy of the results with energy consumption. A sensor reading is inherently inaccurate, as is the case with any instrument that senses the environment. Furthermore, sensor nodes may fail and data is regularly lost during radio transmissions, meaning that data is likely to be incomplete. This has given rise to considerable research in probabilistic queries and statistical techniques in the sensor network arena. For example, in BBQ [DGHM05, DGM*05], statistical techniques are exploited to achieve energy conservation. Users specify a declarative query with a required precision and confidence interval for the results, and a QEP is generated that aims to satisfy the constraints and yet activate as few sensors as possible as seldom as possible (and as a consequence, minimize radio communications). Other related approaches include the Ken approach [CDHH06] and PAQ [TM06], in which only tuples that do not conform a statistical model are transmitted to the gateway node. This work is focussed on Quality-of-Data concerns, which consider the usefulness of data with respect to (often subjective) criteria such as accuracy, completeness, utility etc. It is observed that this (as when considering QoS-awareness) also involves trade-offs between energy consumption and other desiderata. However, considering Quality-of-Data and approximate query answering is beyond the scope of the research contributions of this dissertation.

**Summary** All the SNQPs surveyed have a primary concern, viz., to preserve energy in order to prolong lifetime, which reflects concerns about the constrained nature of sensor networks as a platform. TinyDB exhibits a limited degree of QoS-awareness, viz., the ability to trade acquisition interval for lifetime. SNQL
considers delivery time and lifetime, although the latter always takes precedence, so the user cannot be sure of always receiving data in a timely manner. Overall, for existing complete SNQPs the QoS expectations that may be expressed are restricted to certain combinations of constraints and implicit optimization goals. From the piecewise techniques surveyed, the decisions relating to communication timing are the ones that have been explored in greatest depth with respect to QoS-awareness.

Overall, few decisions are made explicitly by the SNQPs, resulting in the QEP executing in a pre-determined manner and therefore favouring a limited subset of QoS desiderata. Although the Cougar optimizer is reported to make several decisions explicitly, these are not described in great detail, and therefore are difficult to evaluate. In short, no existing SNQP could be used for a broad range of sensor network applications, because not one of them makes all the decisions outlined in the previous section explicitly and in a QoS-aware manner. This finding is the main motivation behind the research contributions described later in this dissertation.

3.3 QoS-awareness in Other Modes of Query Processing

This section surveys QoS issues that arise in various query processing contexts. First, this section describes techniques that are internal to the optimizer and that directly impact on the decisions made when generating a QEP in the case of classical and distributed query processing, where response time tends to be the key issue. Then, this section discusses approaches that aim to meet a certain QoS for a query workload using techniques that are external to the optimizer. These approaches either involve database tuning (e.g., reallocating resources), or controlling the workload presented to the query processor. Finally, QoS issues in stream query processors are discussed. In each case, the techniques used to enable QoS-awareness are discussed, so that they may be considered in the SNQP case.

QoS Issues in Classical and Distributed Query Processing. Traditionally, in both classical centralized query processing over a single server and in DQP the main concern tends to be minimizing the response time [Kos00].
This implies that alternative QEPs need to be generated and evaluated relatively quickly, and that the QEPs that are selected most of the time, although not necessarily optimal, must be reasonably efficient. Over the last three decades, numerous systems which optimize for response time have been developed. Over and above the very successful commercial DBMS currently on sale (i.e., Oracle, IBM DB2, Microsoft SQL Server) significant research systems in this category include System R \cite{SA80}, Volcano \cite{Gra90} and OGSA-DQP \cite{LMH09}. Relatively few examples exist of classical and distributed query processors which are responsive to different optimization goals or QoS in the literature, and these are described in the following paragraphs.

In the Mariposa query optimizer \cite{SAL96}, the monetary cost of a query is considered against the response time that can be provided. Users provide a cost-response time trade-off curve, in which they specify how much they are willing to pay for different response times. The query is optimized by breaking up an operator tree into fragments, and collecting bids from the various sites where query evaluation may take place. A bid for a fragment consists of the price to be charged for evaluating the fragment and the response time that will be achieved based on factors such as the load on the server at the time. The bids for the fragments are then combined to find the costs and response times of several candidate plans. The chosen QEP is the one that yields the greatest profit to the provider, i.e., the one with the greatest difference between the total cost chargeable for the QEP and the amount that the user is willing to pay.

Papadimitriou and Yannakakis \cite{PY01} argue that Mariposa’s criteria for choosing a QEP are somewhat arbitrary, and that it is unrealistic for a user to provide a trade-off cost-response time curve. In response to this, they propose computing an approximate Pareto optimal curve consisting of the set of alternative QEPs which are undominated, i.e., those for which a QEP with both a lower cost and lower response time does not exist. Users then select the most desirable QEP according to their need.

Ye et al. \cite{YKvBO03} sketch an approach to enabling a distributed query processor to be QoS-aware. It is motivated by e-commerce applications, where it may be necessary to provide different QoS to different classes of users. Different optimization goals are considered such as response time, maximizing database throughput, monetary issues, data quality, and minimizing resource utilization. Multiple optimization goals, each with a specific weighting, may be specified by
CHAPTER 3. RELATED WORK

The QEPs are generated and costed using different cost models depending on the optimization goals provided. The overall cost of the QEP is computed using a utility function, i.e., a function which reflects the expected desirability of system state [KD07], or in this context, a QEP. The utility function used is derived from the optimization goals (and corresponding weightings) provided by the user for the query. Unfortunately, while a potentially promising idea, with techniques potentially applicable for the research contributions of this dissertation, only superficial detail is provided.

Meeting Workload QoS. A number of approaches have been proposed in the literature to provide an overall QoS for a query workload, using an optimizer which aims to minimize individual query response times. These make use of database tuning approaches such as the reallocation of resources, and/or workload adaptation. Martin et al. [MPLR02] consider a workload comprising transactions with different levels of importance, for which different levels of QoS are provided according to the user in question, in terms of the average response time for a transaction class. A buffer pool tuning algorithm is proposed that dynamically reallocates buffers used for storing database objects such as tables and indices in order to meet the current workload requirements.

Li et al. [LBR+05] propose an approach in which the QoS level (in terms of average response time) is monitored. If the QoS level is below the one that is required, data replication strategies are proposed to the database administrator, in order to enable the average response time goal to be met.

Reiss and Kanungo [RK05] consider the trade-off between response time and resource level (in terms of storage and network bandwidth, which ultimately translates to monetary cost). Each data structure in the database (e.g., tables or indices) may be stored at a different storage level (e.g., may be replicated) in order to guarantee a certain retrieval time. The aim is to find the most economic storage level at which to store each data structure in the database while meeting the response time requirements for each query in a workload. This problem is cast as an optimization problem which aims to minimize the dollar cost of data structure storage while meeting the response time requirements for each query in a workload, expressed as a quadratic optimization problem with binary variables.

Li et al. [LGB+07] consider, for a mixed workload comprising ad hoc queries as well as batch jobs over a data warehouse which needs to be completed within
3.3. QOS-AWARENESS IN OTHER MODES OF QUERY PROCESSING

A certain deadline each night, the optimal number of materialized views and indices traded-off against the amount of available disc space available to execute the workload. In this case, the QoS expectation is both completing the batch jobs within a certain time window, and providing the required average response time for ad-hoc queries.

Schroeder et al. [SHBIN06] consider an approach for online transaction processing (OLTP) workloads for which the QoS goals considered are mean response time, percentile targets for response time (i.e., where $x\%$ of response times for a class are guaranteed to be below some value $y$) and the variability in response times. Different classes of transactions are defined with different priorities. A scheduler, external to the DBMS, is used to decide on the order of transactions to ensure QoS targets are met for each class of transaction. Niu et al. [NMP+07] propose a similar technique for both OLTP and online analytical processing (OLAP) workloads. In the first instance, the technique tries to adjust resource allocation to meet QoS expectations, and then, if necessary, a scheduler adapts the workload.

Note that while the research contributions of this dissertation are focussed on the QoS-aware optimization of single queries (i.e., not a workload comprising multiple queries), the case of continuous queries resembles that of a workload over a stored data source in that it is also long running in nature.

QoS Issues in Stream Processing. In a stream processing system, where data sources push tuples into the system at high and unpredictable rates, the main concern for many applications is the delivery time. For example, in financial applications, such as a stock exchange data feed, results are only useful if they are produced in a timely manner. In the Aurora stream processor, the user may specify graphs to indicate the importance of three QoS variables: delivery time, percentage of results delivered, and a value-based QoS [CC+02]. The percentage of tuples delivered reflects the tolerable tuple loss which may result when the system is overloaded, and the value-based QoS allows the user to specify ranges of values which are more interesting than others. In Aurora, these specifications are used to inform the load-shedding policy, i.e., rules which govern when and how to drop tuples when the system is asked to go beyond its full capacity. Two types of load-shedding operators are described by Tatbul et al. [TZ+03] that may be inserted at various points in the QEP to ensure the QoS expectations are
CHAPTER 3. RELATED WORK

met as much as possible, viz., random drop, which discards a given proportion of tuples in a random manner, and semantic drop, which considers the values of the tuples, in order to preserve as many of those which contain interesting data.

Given that stream data rates are often unpredictable and bursty, the initial load-shedding policy for a QEP may turn out to be unsuitable. In this context, due to the high data rate, and the fact that bursts are inherently short-lived, adapting the QEP at runtime may not be a feasible option. In order to address this issue, Tatbul et al. [TcZ07] describe an approach in which several alternative load shedding plans are generated in advance for different combinations of input data rates, and invoked as appropriate. Such an approach is an instance of parametric query optimization, in which plans for different conditions are precompiled, and the one to be evaluated is selected at runtime [INSSD92]. Another approach to address resiliency, i.e., the ability to withstand burstiness in streaming data, is proposed by Xing et al. [XHcZ06], where operators are placed at different sites in stream query processors in order to withstand the greatest possible variation in data rates. In effect, the QEP is optimized to maximize the size of the set of feasible input rates.

Load-shedding provides the ability to trade-off the accuracy of the results against the delivery time. While research in sensor network query processing proposes load shedding techniques (e.g., in TinyDB [MFHH05]), due to scarcity of energy, the energy cost of acquiring data, and the fact that data is pulled by the query processor (since it has control over when to acquire sensor readings), a preferable approach in SNQP may be to avoid acquiring the tuple in the first place, e.g., by reducing the rate at which tuples are acquired. Also, note that the QoS expectations that are relevant for SNQPs are similar to those for stream query processors. The major difference is that because stream query processors are assumed to execute on relatively powerful servers that are connected to the electricity grid, energy-related QoS expectations are not a major issue.

3.4 Incorporating Alternative Decision-making Policies into Query Optimizers

Recall that the research contributions of this dissertation involve designing a sensor network optimizer that is QoS-aware, i.e., one that responds to different user-specified QoS expectations when making the decisions required to generate
3.4. INCORPORATING ALTERNATIVE DECISION-MAKING POLICIES

This implies that alternative decision-making behaviours will need to be exhibited by the resulting software artefact. This section discusses techniques that have been used (or potentially could be used) to incorporate different decision-making policies into query optimizers. The potential of each approach is evaluated with regard to the QoS-aware sensor network query optimizer described in later chapters.

**Query Rewriting Rules**

Query rewriting rules are a mechanism for applying equivalence preserving transformations to a QEP with the aim of improving its efficiency. For example, a query rewriting rule may state that selections should be pushed down the operator tree as far as possible, or a cross product followed a selection may be combined into an join. These rules are typically based on heuristics which aim to minimize the intermediate size of data between operators, as described in Section 2.1.1. The Starburst extensible query optimizer enables programmers to customize a database management system for specialized applications by adding new data types, physical operators, storage methods etc. The system also allows new query rewriting rules to be incorporated, specifying, for example, how to incorporate a new operator efficiently into an operator tree. For such an approach to be useful in implementing QoS awareness, the QEP would need to be in a form which can be easily transformed to equivalent QEPs, and also, appropriate heuristics would need to be defined for different QoS variables, and doing so may be far from trivial.

**Cost Models**

Selecting a QoS-dependent cost model to use during query optimization intuitively seems like a promising approach to implement QoS-awareness in a generic and principled manner. As described in the previous section, Haiwei Ye et al. propose such an technique, where different cost models are adopted depending on the optimization goal specified. The adaptive, where-scheduling technique by Bonfils and Bonnet described in Section 3.2 is also not specific to any cost model. The QoS variables for sensor network applications identified in Section 2.3 are related to both time and energy, so a QoS-aware SNQP will need to be able to estimate the relative cost of QEPs in terms of these two variables. As with classical DQP, estimating the memory consumption of QEPs is also necessary. Appendix B describes the derivation of sensor network cost models proposed by Brenninkmeijer for space, time and energy that,
as shown in subsequent chapters, can be used to derive various QoS variables for sensor network QEPs.

**Utility Functions**  A *utility function* is a function which reflects the expected desirability of a system state [KD07], or in this context, a QEP. Paton et al. [PAL+09] argue that objective functions are an effective way of making explicit the desirability of different query evaluation strategies. A methodology is proposed which consists of (1) determining the variables of relevance; (2) defining a utility function for each variable; (3) creating cost models which provide estimates for each variable; (4) designing a representation for the domain of possible solutions; and (5) selecting an optimization algorithm. It is pointed out that this approach is advantageous as it decouples the modelling of the problem from the mathematical optimization technique used. This methodology is demonstrated in the context of response time and profit associated with a particular workload (and is equally applicable for queries or workflows). Indeed, Lee et al. [LPSF09] demonstrate the application of this methodology in the context of workflows. Note that this approach, although proposed in an adaptive, runtime context, can also be used statically, at compile time, to generate an initial QEP. This methodology inspired parts of the design of the QoS-aware SNQP proposed in Chapter 6.

**Optimizing for Multiple Objectives**  Different objectives may be conflicting and give rise to trade-offs. An approach by Papadimitriou and Yannakakis [PY01] in which a Pareto curve approximation is used to propose alternative QEPs to the user, that trade off monetary cost and response time, was mentioned in the previous section. Balke and Günther [BG04] propose a technique for answering skyline queries, a class of queries that aim to identify objects in the database that meet different (possibly conflicting) objectives. A scenario that may involve a skyline query is one in which a house hunter is looking for a property that is as close to a tube station as possible, and also has the lowest cost per square metre. As properties closer to tube stations tend to be more expensive, the house hunter will most likely be forced to sacrifice, to a certain degree, one desideratum in favour of the other. This problem has strong similarities to that of trading QoS expectations for QEPs. Scoring functions are used to give objects a value in terms of a particular variable. They are combined into a single, scalar utility function. Ordered indices according to each variable are assumed to exist, and the
aim is to scan as few database objects as possible, while still retrieving the non-dominated objects in the Pareto optimal set. There are, therefore, similarities with the problem of optimizing a query according to multiple objectives.

**Parametric Optimization** In cases where the same query (or variations of the same query) need to be executed several times, e.g., when a query is embedded in an application, it may be advantageous for the query to be pre-compiled, and the same QEP reused at subsequent invocations of the query. This is likely to speed up the application, as compiling a query every time at each invocation leads to an increased response time. When compiling a plan, various parameters are considered, that inform the decisions of the query optimizer. Examples of these parameters are the current resources available in the DBMS or constants in the query (which may vary at each invocation). For example, since selectivity estimates are used to estimate intermediate output sizes between operators, and since these, in turn, influence how an optimizer orders the joins in a QEP, changing the value of a constant which is applied to a predicate may result in the QEP that was originally selected proving far-from-optimal at runtime. The same can be said if there is a change in the availability of system resources. Parametric query optimization [GW89, INSS92, INSS97] is a technique developed to contend with parameters that affect the cost of a QEP but are either unknown at the time that the query is compiled, or vary as time passes. The approach involves generating several alternative plans (or sub-plans) for different regions in the parameter space, and switching between them at runtime as appropriate.

Cole and Graefe [CG94] describe early work in this area, in which a choose-plan operator is introduced, to make decision at runtime as to which sub-plan (determined by the location of the operator in the plan) is the best one to use depending on the circumstances. This approach enables most of the optimization decisions to be made at compile time, with a few decisions being deferred to runtime. Ioannidis et al. [INSS92] argue that this approach has a high runtime decision-making overhead, and propose an alternative whereby a range of complete QEPs are generated at compile time, and the most appropriate plan is selected at runtime with minimal overhead. Bizarro et al. [BBD09] point out that it is often not necessary to generate plans for the whole parameter space, because it is often the case that only a subset of the parameter values are required. In order to address this issue, an approach is proposed in which at each query
invocation, if a suitable plan has already been generated, it is selected for evaluation, otherwise a new plan is generated on-the-fly; and is added to the set of stored QEPs so that it is available for future use.

During the execution of continuous queries, which tend to be long running, parametric query optimization may be useful to contend with changes in conditions over time. A parametric query technique in the context of stream query processing, in which alternative load-shedding plans are generated at compile time, is proposed by Tatbul et al. [TcZ07] (and was described in Section 3.3). In the case of sensor networks, parametric query optimization may prove useful as a runtime technique, in which the query processor switches between existing QEPs for a given query and QoS in order to ensure that performance meets the QoS expectations as conditions change (e.g., a range of values for the selectivity of a predicate).

3.5 Conclusion

In this chapter, various issues relating to QoS-awareness in the context of query processing were identified. First and foremost, it was noted that existing SNQPs have very limited support for QoS-aware query optimization. They would, therefore, be unable to meet the broad range of QoS expectations of the sensor network applications described in Section 2.3. In light of this observation, developing a QoS-aware SNQP is a novel and potentially useful research contribution. With regards to other modes of query processing, it was noted that they are usually exclusively concerned with a single optimization goal, typically response time. There are few query optimizers that consider different optimization goals. A survey of different techniques that have been used to incorporate alternative decision-making behaviours during query optimization was also presented; these include the provision of alternative query rewriting rules, the substitution of cost models, the use of utility functions to assess QEPs, and parametric query optimization. These techniques will be considered in the discussion of approaches for enabling QoS-awareness in the SNEE query processing stack in Chapter 6. However, before presenting the instantiations of SNEE, the next chapter presents the SNEEql language and query processing stack.
Chapter 4

The SNEE Query Processing Stack

This chapter describes the SNEE query processing stack, which is a functional decomposition of the decision-making steps required to compile a declarative query in the SNEEql language to an imperative QEP. In order to fulfill the research aims described at the start of this dissertation, these steps need to be defined because novel query planning decisions arise when generating a QEP in a SNQP context (listed in Section 3.1), that do not need to be considered in DQP. Furthermore, as was discussed in Section 3.2, existing SNQPs do not explicitly make the most of these decision-making opportunities, meaning that, generally, existing SNQPs make decisions in a fixed or predetermined manner, thereby limiting the ability to tailor certain aspects of the QEP for the given query and QoS expectations. In order for the system to generate QEPs that reflect QoS-related desiderata, such decisions need to be explicitly made, as most of them potentially have an impact on QoS variables.

Recall that the query processing stack is described as being a template because, for each step, it is possible for more than one algorithm to be used to implement its decision-making behaviour. Each step in the stack may be viewed as a template, in which the inputs and outputs of the algorithm used conform to specific types (e.g., algebraic forms that represent intermediate QEPs). Specifying the algorithms for each step in the query processing stack template gives rise to an instantiation of the stack. Defining algorithms for the steps of the SNEE query processing stack is postponed for now. Chapters 5 and 6 present two possible instantiations of the SNEE stack, one that conforms to a fixed optimization
goal (FG-SNEE), and another that is QoS-aware (QoSA-SNEE). In this chapter, the focus is on describing the inputs and outputs to the query processing stack as a whole and each of its constituent decision-making steps, and also motivating design of the resulting architecture.

Section 4.1 presents an overview of the SNEE query processing stack. The next three sections describe inputs to the overall query processing stack: Section 4.2 describes the SNEEq language, a declarative continuous query language for sensor networks. An expression in this language specifies the functional requirements of the data collection application for which a QEP is to be generated. Section 4.3 describes how users express QoS expectations, that broadly specify non-functional requirements. Note that the user-specifiable QoS expectations that each instantiation supports vary. Section 4.4 describes the metadata that informs query planning decisions throughout the query processing stack, such as network information and cost estimation models (CEMs). CEMs are an important form of metadata used at various steps in the query processing stack to evaluate the cost of (parts of) intermediate QEPs, and discard those that are undesirable (i.e., are predicted to be expensive). Where applicable, notation is introduced that is used for the remainder of the dissertation. The reader is referred to Appendix A for a description of the SNEEq physical algebra, and to Appendix B for details about the space, time and energy CEMs, including how they are derived from the algorithms that implement QEP operators.

Having described the inputs of the query processing stack, attention turns to the issues considered during the design of the functional decomposition proposed to compile a declarative query into an imperative QEP. Where relevant, the differences with the proposed approach with both DQP over robust networks and the related work described in Chapter 3 are discussed. Then, the steps of the multi-site phase of the query processing stack (i.e., where the novelty lies) are described in Section 4.6. Each step in the query processing stack is characterized by the inputs and outputs of that step. For example, the first step of the query processing stack takes as input a declarative SNEE query, and produces an abstract syntax tree. As query compilation progresses down the stack, intermediate QEP representations are generated that incorporate an increasing amount of information, as more decisions about how to evaluate the query have been made. Each type of intermediate QEP is represented using an algebraic form that is defined for each step in the stack.
4.1 Query Processing Stack Overview

The SNEE query processing stack is depicted in Figure 4.1. Its inputs are a query expression, some QoS expectations, and various metadata, and the output

Figure 4.1: The SNEE query processing stack.

Note that the description of the SNEEql language, physical operators, and CEMs (i.e., the contents of Section 4.2 and Appendices A and B) are not claimed as part of the research contributions of this dissertation. They have been included as part of this dissertation as they are required to understand the research contributions. In these sections, the text has been largely summarized, and some figures borrowed, from Brenninkmeijer’s work [BGFP08, BGFP09, Bre09].
is executable code for a QEP. The query processing stack comprises three phases, each of which consists of one or more steps. Each step is responsible for making one or more (possibly interrelated) decisions. The first two phases are similar to the two phases in classical DQP, viz., the Single-Site phase (comprising Steps 1–3, in grey boxes), and the Multi-Site phase (comprising Steps 4–6, in white, solid boxes). The Code Generation phase grounds the execution on the concrete software and hardware platforms available in the network/computing fabric and is performed in a single step, Step 7 (in a white, dashed box). It generates an executable program for each site based on the final QEP. Note, however, that the focus of this dissertation is on the second phase, which is concerned with the distribution decisions.

The first phase of the query processing stack leads to the generation of a single-site QEP and is decomposed into components that are familiar from classical, centralized query optimizers, using well-established techniques, so these are not described in detail in this dissertation. In essence: Step 1 checks the validity of the query with respect to syntax and the use of types, and builds an abstract-syntax tree (AST) to represent the query; Step 2 translates the AST into a logical algebra, the operators of which are reordered to reduce the size of intermediate results, resulting in a logical-algebraic form (LAF); and Step 3 translates the logical algebra into a physical algebra, that, e.g., makes explicit the algorithms used to implement the operators. The output of this phase is the physical-algebraic form (PAF), i.e., a tree of physical operators, characterizing a QEP for centralized execution.

In the multi-site phase, decisions pertaining to the distribution of the PAF are made. Step 4 generates a routing tree, which specifies how tuples are routed and what nodes are included in the QEP. Step 5 assigns operators in the PAF to nodes in the routing tree, resulting in a distributed algebraic form (DAF); and Step 6 times the sensing, computation and communication activities associated with the DAF, resulting in an agenda. These steps, and the algebraic form output by each step, are described in more detail later in the chapter.

4.2 Query Language

SNEEq is an expressive query language designed by Brenninkmeijer [Bre09], and is not claimed as a research contribution of this dissertation. It is inspired by
4.2. QUERY LANGUAGE

CQL [ABW03], a continuous language for classical streams. SNEEql has been designed to express data retrieval tasks over classical relations, push-based streams in which tuples are received from the sources at unpredictable rates from external sources, and acquisitional, or pull-based, streams in which the creation of tuples at the leaves of the QEP occurs when sensor readings are requested, as stipulated by the QEP. The focus of this dissertation is on compiling plans involving acquisitional streams, in which the optimizer has a high degree of control as to when tuples are created. Note that SNEEql queries, which are used to express the functional requirements of an application (i.e., in this case, a characterization of the data that is to be retrieved in the context of a schema), are coupled with QoS expectations, that define certain non-functional requirements and are discussed later, in Section 4.3. A detailed description of the SNEEql language, including a formal semantics, is given by Brenninkmeijer [BGFP08, Bre09].

As with classical databases, a schema is defined using a data definition language (DDL), described in [BGFP08] in the case of SNEEql, that defines the extents, or schema objects, that queries may be posed against. In SNEEql, extents may be acquisitional streams, push-based streams, or classical relations, as described in the previous paragraph. In a schema, the name and type of extent is specified, as are the names and types of the attributes that make up the extent. Figure 4.2 presents a schema defining four acquisitional streams, viz., tree, soil, wind and rain, which denote sets of sensors with different sensing modalities, inspired by an application scenario in which a sensor network is used to detect and monitor the risk of fires in a forest [LDI09]. The keyword sensed denotes that these four streams are acquisitional streams. Each acquisitional stream is bound to a subset of physical nodes in the sensor network, decoupled from the schema definition. The first four attributes in the tuple types defined contain information relating to the location and time of the sensor reading. The id attribute contains a unique identifier of the source node that performed the sensor readings. Co-ordinates representing the location are represented by the locx and locy attributes, and a timestamp denoting the time that the sensor reading was taken is contained in the ts attribute. The remaining attributes in the tuples correspond to sensed values.

Some example SNEEql queries are presented in Figure 4.3. These could be posed by a firefighter in a remotely located control room, with a view to monitoring the area covered by the sensor network deployment. Note that Q1–Q5
Figure 4.2: Example SNEEql schema definition. The keyword `sensed` is used to denote that these are acquisitional streams.

are continuous queries, as they are evaluated repeatedly, rather than snapshot queries, which are evaluated once. Q1 requests the current wind speed and direction at every node contributing to the `wind` stream. In Q2, the average wind speed over the last hour, updated every 30 minutes, is requested. Q3 is similar but requests the current average wind speed at every query evaluation. Q4 calculates a fire risk rating index, as proposed by Sharples et al. [SMWG09], in order to obtain an indication of the risk of a fire and its destructive potential. In this case, the danger rating $D$ is computed as $D = \frac{U}{10 - 0.25(T - H)}$, where $U$ is the wind-speed (km/h), $T$ is the temperature ($^\circ$C) and $H$ is the relative humidity (%). This index uses the fact that certain climatic conditions, i.e., higher temperature, higher wind speed, and lower humidity, lead to increased risk of fire and destructive potential of a fire. The average for each of the temperature, humidity and wind speed variables is first obtained using sub-queries, and then the formula is applied. Q5 computes, for each location where smoke has been detected, the wind speed/direction recorded nearby, as indicators of the direction the fire may spread. This query takes readings from the wind and tree streams, and joins them if smoke is detected at the given tree, and the distance between the tree sensor and the wind sensor is 40 units or less.

In SNEEql, the only structured type is `tuple`. The primitive collection types in SNEEql are `relation`, an instance of which is a bag of tuples with definite cardinality; `window`, an instance of which is a relation whose content may implicitly vary between evaluation episodes; and `stream`, an instance of which is a bag of tuples with indefinite cardinality whose content may implicitly vary throughout query evaluation. As in CQL operations construct windows out of streams and vice-versa. In all the queries, windows are used to convert from streams to relations, relational operators act on those relations, and stream operators add the
4.2. QUERY LANGUAGE

Q1: \[
\text{SELECT RSTREAM w.id, w.speed, w.direction} \\
\text{FROM wind[NOW] w;}
\]

Q2: \[
\text{SELECT RSTREAM avg(w.speed)} \\
\text{FROM wind[FROM NOW - 1 HOUR TO NOW SLIDE 30 MINUTES] w;}
\]

Q3: \[
\text{SELECT RSTREAM avg(w.speed)} \\
\text{FROM wind[NOW] w;}
\]

Q4: \[
\text{SELECT RSTREAM wavg.wsp / (10 - 0.25*} \\
(tavg.temp - tavg.rh)) \\
\text{FROM} \\
(\text{SELECT avg(w.speed) as wsp} \\
\text{FROM wind[NOW] w) wavg,} \\
(\text{SELECT avg(t.temperature) as temp,} \\
\text{avg(t.rhumidity) as rh} \\
\text{FROM tree[NOW] t) tavg;}
\]

Q5: \[
\text{SELECT RSTREAM t.id, w.speed, w.direction} \\
\text{FROM wind[NOW] w, tree[NOW] t} \\
\text{WHERE t.smoke = true} \\
\text{AND sqrt((t.locx - w.locx)^2 +} \\
(t.locy - w.locy)^2) <= 40
\]

Figure 4.3: Some example continuous queries in \text{SNEEql}.

resulting tuples into the output stream. Window definitions are of the form WindowDimension [SLIDE] [Units], where the WindowDimension is of the form NOW or FROM Start TO End, where the former contains all the tuples with the current time stamp, and the latter contains all the tuples that fall within the given range. The Start and End of a range are of the form NOW or NOW – Literal, where the Literal represents some number of Units, which is either ROWS or a time unit (HOURS, MINUTES or SECONDS). An optional SLIDE indicates the gap in Units between the Start of successive windows; this defaults to the acquisition rate specified in the QoS expectations. In the example queries, the streaming operator, RSTREAM, appends all the tuples in the window to the output stream\(^1\).

In a continuous query language, where, conceptually, data is being consumed, and thus potentially produced, on an ongoing basis, the question exists as to when

\(^1\)See Table A.1 in Appendix A for descriptions of the streaming (and other) operators.
4.3 QoS Expectations

A SNEEql query is coupled with a set of QoS expectations, which describe the desirable performance characteristics that should be exhibited as the QEP is evaluated. Section 2.3 presented examples of the diverse requirements of sensor network applications, and in Section 3.1.1 the general form of the QoS expectations considered in this dissertation is described. To recap, the QoS performance variables considered in this dissertation are acquisition interval ($\alpha$), delivery time ($\delta$), total energy ($\epsilon$), average energy ($\bar{\epsilon}$) and network lifetime ($\lambda$), and are summarized in Table 3.1. Recall that a QoS expectation is a tuple $\langle \theta, \kappa \rangle$ where $\theta$ is a set of optimization goals, and $\kappa$ is a set constraints, both in terms of QoS variables. Overall, the optimizer compiles query, $Q$, generating a query execution plan, $QP$, optimized for $\theta$ that is predicted to satisfy $\kappa$, or else returns an unsatisfiability marker if it is not feasible to do so. The intuition that underpins this dissertation is that the $QP$ for $Q$ will vary significantly depending on the $\theta$ and $\kappa$ set by the user.

Figure 4.4 presents example QoS expectations supported by QoSA-SNEE. QoS $\{1\}$ stipulates that delivery time is to be minimized, and has an upper bound for the acquisition interval of 1 minute. This QoS may be required by the firefighters in an emergency response scenario, for example, if a fire breaks out, and they need...
4.3. QOS EXPECTATIONS

...to detect the spread of the fire using Q5 in Figure 4.3. In such a situation, the lifetime of the sensor network may cease to be a concern, as once the fire is over, the sensor network may need to be redeployed. In QoS \{2\}, the main concern is maximizing network lifetime, and sensors collect data at a significantly lower temporal granularity. This will result in batteries being depleted more slowly, and data taking longer to reach the user. This may be acceptable, e.g., for Q4, in which the fire danger risk is being monitored. Note that two potentially conflicting constraints have been provided, and it may not be possible to satisfy them both, i.e., the query optimizer may use CEMs to predict that in order to achieve a lifetime of 20 months, the acquisition interval must be at least 30 minutes. In this case, the required behaviour would be for query compilation to return the verdict that the QoS expectations are unsatisfiable, so that the user relaxes the constraints. On the other hand, if the constraint on \( \alpha \) can be easily met, given that lifetime is also an optimization goal, the optimizer will attempt, as far as possible, to increase the achievable lifetime. QoS \{3\} presents a case in which a conflicting constraint and optimization goal are provided. As constraints are effectively a stricter type of QoS expectation than an optimization goal, the optimizer should select the QEP for which the delivery time is less than 20s that has the longest expected lifetime, if a feasible plan exists.

Recall that none of the SNQPs surveyed in Section 3.2 allow the user to explicitly state an optimization goal, or to specify arbitrary combinations of constraints. With Cougar, only an acquisition interval may be specified, and the purported, implicit optimization goal is broadly related to preserving energy or maximizing lifetime. TinyDB, in addition to accepting a fixed \( \alpha \) (with broadly \( \max \lambda \) as the optimization goal), allows users to specify a minimum lifetime, in which case the optimization goal becomes \( \min \alpha \). Optionally, if a minimum lifetime is specified, an upper bound for \( \alpha \) may also be specified. SNQI allows the user to specify a minimum \( \alpha \), a minimum lifetime, and (indirectly) a minimum delivery time, but the optimization goal broadly corresponds to \( \max \lambda \) or \( \min \epsilon \). Therefore, existing SNQPs only support a limited range of QoS expectations.

With regards to the QoS expectations supported by each instantiation of SNEE, the FG-SNEE instantiation, described in Chapter 5, has \( \min \bar{\epsilon} \) as the inbuilt optimization goal, subject to an equality constraint on the acquisition interval, and an optional upper bound on the delivery time. The QoSA-SNEE instantiation of SNEE (Chapter 6) allows the user to specify at most one optimization goal...
of \{\min \alpha, \min \delta, \min \epsilon, \max \lambda\}, and zero or more constraints on any of the variables. The expressiveness of the QoS expectations that may be explicitly stated by the user when compiling queries for QoSA-SNEE is therefore significantly greater than for FG-SNEE, TinyDB, Cougar or SNQL.

4.4 Metadata

As with any classical and distributed query processor, the SNEE query processing stack uses metadata about the system resources, properties of the data, and the operators available, to enable the optimizer to make decisions while generating a QEP. Some of the metadata relevant to SNQP is the same as for DQP over robust networks. However, for compiling queries against a sensor network, novel types of metadata need to be considered, as a result of the inherent resource constraints of the platform. This section discusses the types of metadata used by the SNEE query processing stack, and draws contrasts with the case of classical and distributed query processing. For illustration purposes, an example sensor network deployment is presented in Figure 4.5 over which the example queries presented in Section 4.2 execute over. Nodes are assumed to be located evenly on a grid, 20m apart from each other with a radio range of 60m. The sink node, to which query results are to be delivered, is node 0 in the top-left hand corner. The labels T, S, W and R denote sources for the tree, soil, wind and rain extents respectively, defined in the schema in Figure 4.2.

Sensor network graph. A network connectivity graph informs SNEE about which nodes are able to communicate with which other nodes, and the relative cost of such communication, in terms of energy and latency (two factors that impact the QoS variables discussed in Table 3.1). While in DQP this information would not normally be considered by a query optimizer, in SNQP the paths used by tuples in a QEP may have a significant impact on performance of the plan. A network connectivity graph, denoted \(N = \langle V, E \rangle\), is derived from the network geometry in Figure 4.5, where \(V\) is the set of vertices in the graph (i.e., the sites of the sensor network), and \(E\) is the set of edges (i.e., the pairs of nodes between which direct interaction is possible). Note that each edge \((u, v, (energy, latency)) \in N.E\) connecting site \(u\) to site \(v\) has a multi-dimensional weighting associated with it. The energy weighting reflects the relative energy
cost of transmitting data from site $u$ to site $v$, and is affected by factors such as the link quality (e.g., the average number of retransmissions required for successful packet transmission) and the distance between the sites (which affects the radio transmission power). Similarly, the latency weighting reflects the required time for the radio in site $u$ to transmit to site $v$. Sites that are further apart than the maximum radio transmission range do not have an edge in $N$.E, as they cannot communicate with each other directly.

As mentioned in Section 1.5, it is assumed that network management components are responsible for network formation resulting in the generation of a topology $N$. Topology control protocols, e.g., Geographic Adaptive Fidelity (GAF) [XHE01] and Span [CJBM01], which are responsible for creating a network by deciding which links to keep active, could be used for this purpose. In some cases, these schemes operate by forming clusters of nodes deemed to have equivalent functionality (e.g., two nodes with the same sensor type in close proximity to one another). Thus, from a set of equivalent nodes, only one node in the set may stay awake at a time, while the others sleep, in order to preserve energy. Such an approach would imply that the underlying, physical network topology is constantly changing, thus violating the assumption made by SNEE of
a relatively constant topology. In such a case, the topology graph $\mathbb{N}$ considered by SNEE should be the topology of clusters (which could be viewed as 'logical nodes'), rather than the topology constituted by the underlying physical nodes themselves. Indeed, such an approach could be used to provide SNEE with a degree of resiliency to node failure or to changes in radio propagation characteristics, in which the network management components essentially provide SNEE with an abstraction of a relatively stable and robust network. However, as previously mentioned, the exploration of approaches to make SNEE more resilient is beyond the scope of this dissertation and is left as future work.

**Node descriptions and resource levels.** A description of the nodes in the network (i.e., their sensing modalities, processor speed etc.), and their current state with respect to resource availability (i.e., energy and memory available, current workload) is required. The current state of a node $v \in V$ in the sensor network, is denoted by $v$.Attribute, where attribute $\in \{\text{MemAvail}, \text{EnergyAvail}, \text{etc.}\}$. Note that, in the network in Figure 4.5, it is assumed that all sites may contribute computing (e.g., for processing intermediate results) or communication (e.g., for relaying data) capabilities. Again, network management functionality is assumed to exist that is responsible for gathering information about the state of each node.

**Logical Schema.** A description of the extents in the schema is used for type-checking and semantic analysis when parsing a query. Note that for each extent defined in the schema, it needs to be made explicit whether it is stored relation, a push-based stream, or an acquisitional stream. An example of a schema in the context of SNEEql was presented in Figure 4.2.

**Statistical properties of the data.** Statistical properties of the data (e.g., the most common values for an attribute in the schema) is used in classical query optimization to estimate selectivities, so that the average cardinality of intermediate results between operators may be estimated, and operators ordered accordingly [GMUW00]. With acquisitional streams, as the query optimizer has control over the creation of tuples, the maximum output cardinality of an operator, denoted $\text{Size}(op)$, can be computed. For example, in the case of the $\text{SP}_{\text{ACQUIRE}}$ operator in $Q1$, there are four source sensor network nodes, so the maximum number of tuples that may be output at each evaluation episode is
4.4. METADATA

Figure 4.6: Physical schema for mapping the streams defined in the schema in Figure 4.2 to nodes in the sensor network in Figure 4.5.

In SNEE, during optimization, the maximum cardinality of operators, rather than the average cardinality, is used when making query planning decisions. This is because SNEE schedules tasks in the QEP for execution at fixed times, to enable static, compile time estimations to be made about the QEP and for sensor network nodes to be synchronized (required for radio communications, and for acquiring data at the stipulated times). Therefore, a slot for the maximum number of tuples that may be processed is allocated for each task, even if, in practice, the whole duration of the slot is not always required.

Physical Schema. The query optimizer requires a mapping from schema objects (at the logical level) to the source nodes containing sensors that can provide the data (at the physical level). The physical schema contains similar information to that stored by a distributed query processor when an extent is horizontally partitioned across various sites. However, given that sensed data is associated with a spatial location in the physical world, the mapping from the logical to physical level is of greater relevance to the user than in the DQP case. An example physical schema is shown in Figure 4.6, which maps the extents in the schema in Figure 4.2 to the nodes in the network in Figure 4.5. Note that the sensor streams generated by several individual sensors in the network often constitute a single logical extent in the schema. For example, in Figure 4.6, the Wind stream, which contains the sensed attributes speed and direction, is acquired at sites 4, 10, 22 and 24, each labelled with W in Figure 4.5.

Operators and associated cost models. The query processing stack requires a library of the available operators, and their associated properties. The SNEE physical algebra is presented in Appendix A. Moreover, CEMs are required to estimate the space, time and energy associated with these operators, to assist the query optimizer with making decisions, e.g., ranking candidate QEPs with respect to the QoS expectations. Throughout this dissertation, Memory(op), Time(op),

tree \{1, 3, 9, 12, 13, 15, 16, 19, 20, 21\}
soil \{6, 14, 17\}
wind \{4, 10, 22, 24\}
rain \{4, 10, 22, 24\}
and Energy(op) denote the memory, time and energy costs, respectively, of a given operator op. Details about the approach used in SNEE to derive the CEMs used are provided in Appendix B. The SNEEdq physical operators and associated CEMs are research contributions presented in Brenninkmeijer [Bre09].

Different types of metadata are used in different stages of the SNEE query processing stack, depending on the types of decision that needs to be made. As with other modes of query processing, inaccurate or out-of-date metadata can result in an undesirable QEP being generated by the optimizer. Note, however, that defining mechanisms for the collection and maintenance of metadata is beyond the scope of this dissertation.

4.5 Query Processing Stack Design Issues

The SNEE query processing stack takes as input a query expression, a specification of the QoS expectations on its execution, and the metadata required to generate a QEP for it, each of which are described in previous sections of this chapter. Several issues were considered in the design of the SNEE query processing stack, which generates a QEP for retrieving the data specified by the query, while approximating the performance specified in the QoS expectations, viz:

- Identifying fine-grained decisions that need to be made to generate a QEP;
- Deciding how to break up the query optimization problem, which leads to steps for QEP construction being identified. At one extreme, all the decisions may be considered together as part of a single optimization problem (meaning that the query processing stack would have a single step). At the other extreme, each fine-grained decision that needs to be made to generate a sensor network QEP may be considered separately (meaning that the query processing stack would have as many steps as there are individual decisions);
- Assuming that there is more than one step, ordering the aforementioned steps, possibly based on dependencies between them;
- Determining the inputs and outputs for each step, which algorithms implementing the step are to assume in order that they may be used interchangeably to create instantiations of the query processing stack template.
This section presents a discussion of these issues and motivates the design decisions that were taken.

Section 3.1 has already addressed the problem of identifying the types of decisions relevant to SNQP. These include the decisions required for DQP over robust networks, plus additional decisions that take into account the resource-constrained nature and fragility of sensor networks. To recap, these decisions are: operator tree composition (D1), algorithm selection (D2), routing (D3), partitioning (D4), selecting the number of instances per operator (D5), operator instance assignment (D6), adjusting the acquisition interval (D7), stipulating the timing of computations (D8), and stipulating the timing of communications (D9). The reader is referred to Section 3.1 for further details.

Dynamic programming, in which the query optimization problem is broken up into steps, and the overall QEP is constructed by combining the optimal solution at each step, is a common approach adopted by many query processors, both in commercial and research settings \cite{Ioa97}. The inherent risk with this approach is that committing to query planning decisions at earlier steps in the query processing stack may lead to a decision being made that turns out to be sub-optimal when QEP generation progresses further, or indeed, when the query is evaluated. Alternative approaches to approaching the query optimization problem involve using randomized algorithms to search the solution space of alternative QEPs to find the plan of minimum cost, e.g., \cite{Swa89, IK90, BFI91}. These, in effect, involve carrying out all the query planning construction decisions together \cite{Ioa97}. This gives less control over each individual query planning decision, and given that in SNQP there are more types of query planning decisions than classical, distributed and stream modes of query processing, such an approach was not favoured in this context. SNEE therefore adopts a dynamic programming approach, and mitigates the risk of pruning good QEPs early on by allowing each step in the query processing stack to output a collection of alternative QEPs, with the view that at least one of them will result in a quasi-optimal final QEP. Note, however, that the query processing stack architecture is not prescriptive about whether more than one candidate QEP is produced by each decision-making step – this depends on the algorithm (and associated parameters) used to instantiate each step.

Now that the overall approach (i.e., whether to make all decisions together or not) has been determined, the question of defining steps for the query processing
The query processing stack is addressed. Based on the observation that some of the decisions identified are closely interrelated and either similar in nature or a consequence of each other, one can define steps at a coarser granularity, i.e., which may comprise more than one decision. For example, in the two-phase approach in DQP the centralized QEP is fragmented into subtrees of operators and then fragments are assigned to sites. An alternative approach would be to first assign operators to sites, from which fragments would be indirectly created. The latter would enable more fine-grained control over the scheduling of operators to sites. FG-SNEE and QoSA-SNEE use each of the different approaches described. Therefore, decisions D4, D5 and D6, regarding the assignment of computations to sites in the sensor network, are interdependent, and are gathered into a single step referred to as where-scheduling. Note that, for SNEE, where-scheduling encompasses both partitioning and assigning fragments to sites in the network, whereas they may be two distinct steps in DQP e.g., as is the case of OGSA-DQP [LMH+09]. Similarly, decisions D7, D8 and D9 are related to the timing of activities in the QEP and are gathered into a when-scheduling step. The remaining decisions have a one-to-one mapping to steps in the query processing stack.

A key consideration for determining the sequence of steps is dependencies between them. For example, it is first necessary to define the logical operators in the QEP before concrete algorithms can be assigned. A well-known DQP architecture is used as a basis for this. Recall that the two-phase optimization approach, described in Section 2.1.2 involves first generating a centralized QEP (decisions D1 and D2), from which a distributed QEP is generated (decisions D4, D5, and D6). The SNEE query processing stack follows the same order of decisions. The additional steps for the case of sensor networks, viz., routing (D3) and when-scheduling (the combination of D7, D8 and D9), need to be inserted at appropriate positions in this sequence.

Given that it is necessary to first determine what sensing, computation and communication activities are taking place at each node, so that their order and duration can be computed to determine execution start times for each activity, routing and where-scheduling decisions need to be made before when-scheduling ones. Therefore, the when-scheduling step is the last decision made in the sequence.

The remaining question is to determine in what order where-scheduling and routing should take place. Given the high cost of communications in a sensor
network, the **SNEE** query processing stack first generates a routing tree, which determines the paths used to transport tuples from the source sites to the sink, prior to assigning computations to sites in the sensor network. This allows the topology of the network to be taken into account when making where-scheduling decisions, i.e., they can be informed by the shape of the routing tree, so that computations are placed onto sites in such a way that communications can flow in a unidirectional manner. The risk of committing to an inappropriate routing tree earlier may be compensated for by generating several alternative routing trees, to be considered at later steps in the stack. Conversely, making where-scheduling decisions first, and routing decisions later, would mean that the shape of the routing tree is not taken into account during where-scheduling, resulting in higher communication costs if the placement of computations implies that tuples need to travel a longer distance. Note that while it may be argued that routing and where-scheduling could be conflated into a single decision-making step, the search space would be more complex, as there would be more possible plans to consider. Considering these steps separately breaks the overall problem up into smaller ones. As a result, the routing step is inserted between the single site optimization and where-scheduling steps. This resulting sequence of decisions leads to the query processing stack presented in Figure 4.1.

None of the **SNQPs** surveyed in Section 3.2 describe a query processing stack along the lines described here. For Cougar, not enough details are presented in order to ascertain the sequence of decisions made, if any, by the query compiler. In the case of TinyDB, the query compiler limits itself to performing operator rewriting decisions, similar to those performed during logical optimization in classical query processing. For SNQL, query optimization comprises two steps: *parsing*, which includes the construction of the operator tree, and *decomposition*, which encompasses fragmenting the **QEP** and assigning fragments to groups of sensors (i.e., essentially, a coarse-grained form of where-scheduling). Decisions that involve timing acquisitions and communications are postponed until runtime. The remainder of the chapter describes the steps in the **SNEE** query processing stack in more detail.
4.6 Characterization of Query Stack Steps

Now that the motivation for the sequence of steps that comprise the query processing stack in Figure 4.1 has been presented, this section characterizes each individual step by describing the algebraic forms that constitute its inputs and outputs, with examples. The input and output types of each step are the same for all instantiations of SNEE; descriptions of the algorithms that implement the steps are postponed until the next two chapters. Firstly, the single-site phase is discussed briefly in Section 4.6.1. It is not discussed in detail as it is based on well-known techniques from classical query processing, such as the ones described in Section 2.1.1. Sections 4.6.2 and 4.6.4 then describe the steps in the multi-site phase.

4.6.1 Single-site Phase

The output of the single-site phase is an operator tree comprising a tree of physical operators. An example PAF is shown in Figure 4.7 for the example SNEEq query Q₄ from Figure 4.3. Each node in the tree represents an operator in the SNEE physical algebra (not claimed as a research contribution of this dissertation), a
subset of which is presented in Table 4.1. Note that aggregates (such as average) are computed in three phases (initialize, iterate and evaluate), each implemented as a separate physical operator, based on the incremental aggregation techniques proposed by Gray et al. [GCB+97]. In the TinyDB project [MFHH05], such an approach is shown to be effective in the context of sensor networks as it helps reduce radio traffic and allows the computation of different operands in the algebraic expression to be evaluated separately. Note also the absence of the RSTREAM and TIME_WINDOW operators which are present in the text of Q4 and have been removed by translation/rewriting (Step 2 in Figure 4.1).

The operators in Table 4.1 have argument types denoted R, S and W, for relation, stream and window respectively. Certain properties that are associated with operators are used by the optimizer when making query planning decisions, viz.,

**AttrSen.** An operator is attribute sensitive if the partitioning of tuples matters when there are multiple copies of the same operator [HM95]. If an operator is attribute sensitive, the query optimizer needs to redistribute the tuples accordingly, in order to ensure that the correct subset of the child tuples is routed to each copy of the operator. NLJOIN is an example of an attribute sensitive operator.

**LocSen.** An operator is location sensitive if there is no leeway as to which node(s) in the sensor network they may execute over. Both SP_ACQUIRE and DELIVER are location sensitive. The nodes that an SP_ACQUIRE will be assigned to are determined by the physical schema, a constituent of the metadata that was described in Section 4.4 that stipulates the sources for a logical stream in the schema. The placement of a DELIVER is determined by practical constraints, i.e., where the query results need to be sent to, and is specific to the deployment.

**Iterative.** An iterative operator op has the property that an instance of op may have an outgoing edge connected to the input of another instance of op (as opposed to an instance of op.Parent, as is usually the case).

**TimeSen.** An operator is time sensitive if it needs to be evaluated at specific times for the query results to be correct.
Appendix \=[A] presents a more comprehensive list of the S\(\text{NEE}\) algebraic operators and \(\text{PAFs}\) for the other example queries.

<table>
<thead>
<tr>
<th>Operator Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP_ACQUIRE<a href="">extentName, attrSenseList, predExpr, projExprList</a> : (S)</td>
<td>Take readings from sensors in (\text{attrSenseList}) and apply (\text{SELECT}[\text{predExpr}]) and (\text{PROJECT}[\text{projExprList}]). (\text{LocSen.})</td>
</tr>
<tr>
<td>DELIVER() ((S) : S)</td>
<td>Deliver the query results. (\text{LocSen. TimeSen.})</td>
</tr>
<tr>
<td>TIME_WINDOW<a href="S">startTime, endTime, slide</a> : (W)</td>
<td>Define a time-based window on stream (S) from (\text{startTime}) to (\text{endTime}) inclusive and re-evaluate every slide time units.</td>
</tr>
<tr>
<td>RSTREAM[] ((W) : S)</td>
<td>Emit all the tuples in window (W).</td>
</tr>
<tr>
<td>NL_JOIN[projExprList, predExpr](R</td>
<td>W, R</td>
</tr>
<tr>
<td>AGGR_INIT[aggrFunction, aggrList, projList](R</td>
<td>W) : (R</td>
</tr>
<tr>
<td>AGGR_MERGE[aggrFunction, aggrList, projList](R</td>
<td>W) : (R</td>
</tr>
<tr>
<td>AGGR_EVAL[aggrFunction, aggrList, projList](R</td>
<td>W) : (R</td>
</tr>
</tbody>
</table>

Table 4.1: Some example S\(\text{NEE}\)ql operators, adapted from \(\text{GBJ}^+09\).

### 4.6.2 Routing

\(\text{Step 4}\) in the query processing stack determines a routing tree whose edges denote the communication links that the data flows in the \(\text{PAF}\) can then rely on. This is achieved by deriving a sub-tree from the sensor network graph \(\text{N}\) that contains all the sources of the query (obtained from the physical schema in Figure 4.6), the gateway node, and any additional nodes required. For example, \(Q1\) in Figure 4.3 has as sources the sites that comprise the \(\text{wind}\) extent (i.e., sites \(\{4, 10, 22, 24\}\) as specified in Figure 4.6), and the gateway node that data is to be delivered to is site 0 (as shown in the example network in Figure 4.5). This means that at least sites \(\{4, 10, 22, 24\} \cup \{0\}\) must be included in the routing tree. Additional sites
may be used, for example, in order to enable indirect communication between these sites, e.g., as is the case with site 16.

This step has been introduced in the sensor network context due to the implications of the high cost of wireless communications, viz., that the paths used to route data between fragments in a QEP have a significant bearing on its cost. In contrast, in traditional DQP, the paths for communication are solely the concern of the network layer, so this step does not exist. Routing may be viewed as a preparatory step prior to where-scheduling, in that the routing tree imposes constraints on the data flows, and thus on where operations can be placed.

4.6.3 Where-scheduling

Step 5 assigns each operator in the PAF to sites in the routing tree. This results in the insertion of EXCHANGE operators \[\text{Gra90}\] along selected edges in the PAF giving rise to fragments. The result of this step is a distributed operator tree. A fragment in SNEE is defined as a connected sub-tree of operators in the PAF that is allocated to the same set of sites in the sensor network, and that may be scheduled to execute at the same time. Note that this notion differs subtly from the DQP case described in Section 2.1.2 because SNEE QEPs during when-scheduling (the next step), incorporate decisions relating to the timing of tasks (less of a concern in modes of query processing that involve stored data).
The `EXCHANGE` operator in SNEE also differs from that used in DQP, as these operators incorporate routing information. Like the DQP case, an instance of an upstream fragment is associated with a consumer $e_c$, and an instance of a downstream fragment a producer $e_p$. In addition, if the `EXCHANGE` encapsulates radio communications, one or more relays $e_{r1}...e_{rn}$ may be placed at intermediate sensor network nodes between a producer and consumer, reflecting the fact that routing is an additional concern in sensor network query planning.

As with DQP, `EXCHANGE` operators are responsible for data redistribution, e.g., an `EXCHANGE` may be required because the downstream fragment contains an attribute-sensitive operator. Note that a producer may be connected to a consumer on the same site and not involve inter-site communication, e.g., they may be merely buffering to reconcile two connected fragments that are evaluated at different rates. If they do involve radio communications, the producer and consumer invoke `transmit` and `receive` components, respectively, in which case the CEMs reflect a significantly higher time and energy cost\(^2\).

Graphically, the output of this step can be represented in two alternative ways. In the distributed-algebraic form (DAF), each node in the tree represents an operator instance of an operator type, or an instance of part of an `EXCHANGE`, at a physical site. Formally, a DAF for a PAF $P_Q$ is defined as a triple $(opInsts, flows, assignment)$ where $opInsts$ is the set of operator instances created for each operator $op \in P_Q$. $flows$ is a set of pairs $(u, v)$, $u \in opInsts$, $v \in opInsts$, which denote edges connecting operator instances; and $assignment$ is a mapping $u \rightarrow s$, $u \in opInsts$, $s \in R_Q$. $Sites$, which specifies the site in the routing tree that each operator instance is assigned to. Figure 4.9 denotes an example DAF for $Q1$, based on the routing tree shown in Figure 4.8. The dotted, rounded rectangles denote the boundaries of a site in the sensor network, with the site identifier shown underneath. $C$, $P$ and $R$ denote the consumer, producer and relay parts of an `EXCHANGE` respectively. Four instances of the location sensitive `SP_ACQUIRE` operator have been created, at the sites specified by the physical schema metadata. The paths from the routing tree in Figure 4.8 directly correspond to the DAF. For example, tuples at the `SP_ACQUIRE` instance at site 22 travel via sites 16, 10 and 5 on the way to the sink.

The DAF representation provides an operator-instance centric view of the

\(^2\)Attention is drawn to the reader of an illustration of the approach used by Brenninkmeijer [Bre09] to derive a CEM for the `transmit` component in Appendix B.
4.6. CHARACTERIZATION OF QUERY STACK STEPS

PAF operator tree superimposed over a routing tree, but uses quite a lot of space, so a compact Distributed Algebraic Form (cDAF), is used frequently throughout this dissertation. An example cDAF is depicted in Figure 4.10 which comprises a single node for each operator (irrespective of how many instances it has). The dashed, rounded rectangles denote fragments, and under each fragment, the fragment identifier and set of sites that the fragment is allocated to is indicated. EXCHANGE operators are the means by which tuples travel between fragments, but the routing-tree level details are not shown in the more concise cDAF representation.

The where-scheduling step is broadly similar to the second phase of the two-phase approach in DQP. Note, however, that, in this case, the topology of the overlay routing tree needs to be taken into account when placing operator instances on sites in the sensor network, e.g., to avoid circular tuple flows, which are likely to lead to QEPs with higher communications costs, and are unlikely to
be desirable irrespective of the QoS being optimized for. Also, unlike the case of DQP
where further operator instances are likely to be created to increase parallelization and thus
improve response time, the spatially distributed nature of data sources in a sensor network, and the constraints imposed by the tree overlay, mean that when creating operator instances, the placement of the data sources
is likely to be the greater concern.

The algorithms proposed for this step in the FG-SNEE and QoSA-SNEE instantiations only create operator d-instances, although they could potentially be extended to consider p-instances as well. Finally, the decisions made in this step in SNEE are made in the same order as in DQP for the FG-SNEE instantiation, where firstly, the QEP is partitioned, and then fragments scheduled to sites [LMH+09], but the reverse is true for the QoSA-SNEE instantiation.

4.6.4 When-scheduling

Step 6 in the query processing stack determines the timing of acquisitions, computations and communications. The result of this step is an agenda that stipulates the execution time of each task to be performed during a single evaluation episode of QEP execution. By evaluation episode, is meant a single execution of

---

3Recall that, as described in Section 3.1.2, operator d-instances are created as a result of the spatially distributed nature of the data sources, whereas operator p-instances are created to parallelize the execution of a resource intensive operator. The latter is the usual motivation for partitioned parallelism in DQP but is not considered in this dissertation.
4.6. CHARACTERIZATION OF QUERY STACK STEPS

The agenda (Note that, being in a continuous query setting, the agenda is repeatedly executed). An example agenda for Q1 is shown in Figure 4.11. The agenda can be conceptualized as a matrix, in which the rows, identified by a relative time point, denote concurrent tasks in the sites that identify the columns. In an agenda, there is a column for each site and a row for each time when some task is started. Thus, a non-empty cell \((t,s)\) with value \(a\), denotes that task \(a\) starts at time \(t\) in site \(s\). A task is either the evaluation of a fragment (which subsumes sensing), denoted by \(F_n\) in Figure 4.11 where \(n\) is the fragment number, or a communication event, denoted by \(tx\) or \(rx\), i.e., respectively, tuple transmission to, or tuple reception from, site \(n\). Blank cells denote the lack of a task to be performed at that time for that site, in which case, an OS-level power management component is delegated the task of deciding whether to enter an energy-saving state.

An agenda based on a given routing tree and DAF requires two principal parameters to be constructed, \(\alpha\), the acquisition interval, and \(\beta\), the buffering

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{start time} & \text{site 22} & \text{site 16} & \text{site 10} & \text{site 5} & \text{site 24} & \text{site 18} & \text{site 4} & \text{site 8} & \text{site 12} & \text{site 6} & \text{site 0} \\
\hline
0:00:00.000 & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow \\
0:15:00.000 & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow & F_1 \downarrow \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
4:00:00.000 & F_{17} \downarrow & F_{17} \downarrow & F_{17} \downarrow & F_{17} \downarrow & F_{17} \downarrow & F_{17} \downarrow & F_{17} \downarrow & F_{17} \downarrow & F_{17} \downarrow & F_{17} \downarrow & F_{17} \downarrow \\
4:00:00.042 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
4:00:00.087 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
4:00:00.117 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
4:00:00.134 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
4:00:00.170 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
4:00:00.198 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
4:00:00.227 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
4:00:00.245 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
4:00:00.310 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
4:00:00.375 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
4:00:00.450 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
4:00:00.600 & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} & \text{rx} & \text{tx} \\
\hline
\end{array}
\]

\(\alpha = 15\ \text{min}\)
\(\beta = 17\)
\(\delta \leq 4\text{:}15\ \text{hours}\)

Figure 4.11: An example agenda for Q1.
 factor. The acquisition interval determines the number of time units between the execution of the leaf fragments (e.g., F1 for the cDAF in Figure 4.10), which are responsible for taking sensor readings (and optionally performing transformations on the data as well). The period between sensor readings is referred to as an epoch. The buffering factor stipulates the number of epochs after which the non-leaf fragments (e.g., F0 for the cDAF in Figure 4.10) are executed. Figure 4.12 depicts how an agenda is structured. As represented by the figure, an agenda evaluation episode involves in an initial $\beta - 1$ epochs of sensing, buffering, and (if there is time before the next epoch) going into a low-power sleep state. In the last epoch of an evaluation episode a burst of communication and processing activity takes place, and tuples are transmitted up the routing tree. During this epoch, the non-leaf fragments evaluate a batch of tuples corresponding to all the epochs in the evaluation episode. After delivering the results for the evaluation episode, the QEP sleeps for any time that is left until the next evaluation episode. If $\beta = 1$, the non-leaf fragments of the cDAF are executed in the same epoch as when data is acquired (i.e., the first part of the agenda corresponding to epochs 1 to $\beta - 1$ in Figure 4.12 is not included because there is only one epoch in the evaluation episode).

In Figure 4.12 some properties of SNEE agendas in terms of the length of time between different parts of the agenda may be observed. An evaluation episode comprises $\beta$ epochs with duration $\alpha$ in total, therefore its length is $\alpha \beta$ time units. As the query is continuous, at time $\alpha \beta$, the evaluation of the agenda repeats. In the last epoch of the agenda, the time taken to evaluate the leaf and non-leaf

<table>
<thead>
<tr>
<th>Start Time</th>
<th>$s_1$</th>
<th>$s_2$</th>
<th>...</th>
<th>$s_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evaluation of leaf fragments (i.e., sensing, and possibly some processing)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sleep</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\alpha(\beta-1)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evaluation of leaf fragments (i.e., sensing, and possibly some processing)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evaluation of non-leaf fragments (i.e., data processing and forwarding)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sleep</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\alpha\beta$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>End of evaluation episode</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.12: The structure and properties of SNEE agendas.
4.7 Conclusion

This chapter has described the SNEE query processing stack. Based on the survey of existing SNQPs presented in Section 3.2, the SNEE query processing stack is the one that, to date, makes the most comprehensive set of query planning decisions in an explicit manner. By considering each decision individually for
generating a QEP, the optimizer is able to make each decision in the light of QoS considerations. This may therefore be considered to be the first step in the design of a QoS-aware SNQP.

In TinyDB and SNQL, most query-planning decisions are made in a decentralized manner at runtime, i.e., by each site participating in the QEP. As a result, decisions are made based on local information, and this has the advantage that it is not necessary to expend scarce resources collecting metadata about the whole network. However, such decisions are unable to take into account the global state of the network, meaning that the QEP may exhibit properties that are undesirable overall. In contrast, SNEE makes query planning decisions centrally at compile time, using global knowledge about the state of the sensor network. Such an approach is more likely to generate QEPs that are closer to the globally optimal one, as more useful information is available. Although there is a cost associated with collecting the required metadata for the query optimizer, the intention is that the cost of metadata collection would be offset by the benefits of being able to make better query planning decisions as a result of having knowledge about the state of the entire sensor network.

The next chapter describes FG-SNEE, an instantiation of the SNEE query processing stack that has the fixed optimization goal of minimizing average energy per node in the sensor network, and only supports a very limited set of user-specified QoS expectations. The user must specify an equality constraint on acquisition interval, and optionally, an upper bound on the delivery time. This instantiation serves to validate the query processing stack proposed, i.e., to show that the functional decomposition proposed succeeds in achieving the goal of compiling a declarative query into a QEP. It is also intended that by making more decisions in an explicit manner, it will outperform other fixed-goal SNQPs and therefore prove to be a suitable baseline for comparison with the fully QoS-aware instantiation in Chapter 6.

Now that a query processing stack template has been designed, a simplistic, naive approach to implement QoS awareness would be to provide different instantiations depending on the QoS expectations specified. In other words, different algorithms could be proposed for each QoS, and different instantiations of the query processing stack used depending on the QoS selected. However, the sheer number of possible combinations of optimization goals and constraints seems likely to make this approach impractical. The challenge, therefore, is to find a more
generalized set of algorithms to instantiate the \textit{SNEE} query processing stack, that are able to consider the full spectrum of \textit{QoS} expectations. Approaches to tackle this issue are discussed, and a \textit{QoS}-aware \textit{QoSA-SNEE} instantiation proposed, in Chapter 6.
Chapter 5

A Fixed Goal Instantiation of SNEE

This chapter presents FG-SNEE, a concrete instantiation of the SNEE query processing stack template described in Chapter 4. Like existing SNQPs, it has a fixed optimization goal relating to conserving energy/maximizing lifetime. Precisely defined, the optimization goal of FG-SNEE is $\min \bar{\epsilon}$, i.e., minimizing the average energy consumed per node in the sensor network. Unlike existing SNQPs, it makes most query planning decisions explicitly, with the intention of generating QEPs that are more energy efficient than existing SNQPs that optimize, broadly speaking, for the same goal. This instantiation of SNEE therefore serves to validate the query processing stack proposed in Chapter 4 and also, to propose a contender to be a suitable representative for a state-of-the-art SNQP for energy-related optimization goals, that can be used as a baseline to compare the performance benefits of a SNQP that is QoS-aware over a fixed-goal one. This is necessary to ensure a fair comparison takes place later on.

FG-SNEE has limited QoS-awareness because the user may specify a fixed acquisition interval and, optionally, an upper bound on delivery time. Otherwise, the exhibited behaviour is according to fixed QoS assumptions. As part of the research contributions of this dissertation, algorithms are described for the steps in the multi-site phase of the stack, i.e., those that involve the distribution of the centralized query plan. For each step in the multi-site phase, pseudocode is presented, the algorithm is described, and the output is illustrated, using a running example based on the compilation of $Q_3$ in Figure 4.3. Based on the schema in Figure 4.2 and the network in Figure 4.5. Contrasts with related work...
are also made where relevant. Algorithms for the steps in the single-site phase are not presented as they are based on well-established techniques in centralized query processing, largely as described, e.g., by Garcia-Molina et al. [GMUW00].

Throughout the pseudocode and associated descriptive text presented in this dissertation, the following font conventions are used: Algorithm-Name() denotes an algorithm that is defined using pseudocode, with a reference to the appropriate figure if necessary, FunctionName() denotes a primitive function whose semantics should be straightforward to understand (possibly with the help of a comment in the code), identifierName denotes a variable (which may have accessor fields or methods, denoted identifierName.FieldName and identifierName.MethodName). $R_Q$, $D_Q$ and $A_Q$ denote a routing tree, DAF and agenda respectively, and $R_Q$, $D_Q$ and $A_Q$ denote collections of these data structures. $N$ denotes a network topology with multi-dimensional $\langle \text{latency, energy} \rangle$ attributes. OPERATOR_NAME denotes an operator, and operator_constituent denotes part of an operator (e.g., such as the producer in an EXCHANGE).

### 5.1 Routing

Recall, from Section 4.5, that the routing step is responsible for selecting the paths within the sensor network for tuples to travel from the source nodes to the destination node. The result is a tree, derived from a network graph $N$, where the root node is the destination and the leaf nodes are the data sources. Internal nodes need not be source nodes; they may simply be used to relay tuples up towards the root, or to perform operations that do not involve acquisition. Let $P_Q$ denote the PAF operator tree that constitutes the output of the single-site phase, depicted in Figure 5.1. Let $P_Q$.Sources $\subseteq N.V$ and $P_Q$.Sink $\in N.V$ denote, respectively, the set of sites that are data sources, and the destination site, in $P_Q$. In FG-SNEE, the aim is, for each source site, to reduce the total energy cost to transport results to the destination. Karl and Willig [KW05] observe that this is an instance of the Steiner tree problem, in which, given a graph, a tree of minimal cost is derived which connects a required set of nodes (the Steiner nodes) using any additional nodes that are necessary. Thus, the optimal routing tree $R_Q$ for $Q$ is the Steiner tree for $N$ with Steiner nodes $P_Q$.Sources $\cup \{P_Q$.Sink\}. The problem of computing a Steiner tree is NP-complete, so the heuristic algorithm (given in [KW05] and claimed there to perform well in practice) is used to compute
Figure 5.1: The Physical-Algebraic Form for $Q3$.

an approximation. Figure 5.2 presents the pseudocode for the algorithm.

First, the algorithm in Figure 5.2 makes the destination site the root vertex in the Steiner tree. Then, it adds the remaining Steiner vertices to the tree being built one by one after finding the shortest path between the new vertex and some vertex already in the tree, and adding to the tree all the vertices in the computed path. The algorithm stops once all Steiner vertices appear in the tree. The result is a routing tree that is an approximate Steiner tree for $\langle \mathbf{N}, P_Q \rangle$. Figure 5.3 shows the resulting routing tree generated by the algorithm in the case of $Q3$.

This approach aims to minimize the sum of the edge energy weightings in the routing tree. The amount of data that will flow along each edge in the routing tree is not known at this stage, as the placement of operators (which, e.g., may reduce the cardinality of the data), is not determined until the where-scheduling step and therefore cannot be considered. Given that the energy cost associated with a radio transmission increases by the square of the distance, finding the minimal-cost Steiner tree using the energy weightings in $\mathbf{N}$ will favour routing trees with a higher number of short hops in the routing tree. This is intended to reduce average energy consumption, i.e., the optimization goal of this instantiation of
ROUTEING($P_Q, N$)

▷ Compute an approximate minimum energy cost Steiner
tree $(V, E)$ for $(N, P_Q, \text{Sources} \cup \{P_Q, \text{Sink}\}$), where
▷ $P_Q$ is the Physical-Algebraic Form and $N$ is the network
topology graph with multi-dimensional $(\text{energy, latency})$
▷ edge weightings.

1 $V \leftarrow \{P_Q, \text{Sink}\}$
2 $E \leftarrow \emptyset$
▷ Get network graph with uni-dimensional energy weightings.
3 $N \leftarrow \text{EnergyGraph}(N)$
▷ Define the remaining Steiner nodes
4 $V_S \leftarrow P_Q, \text{Sources}$
▷ Repeat until all nodes have been added to the tree.
5 while $V_S \neq \emptyset$
6 do ▷ Select a Steiner node (not already in the tree) at random.
7 $v_{\text{from}} \leftarrow \text{ChooseAnyOne}(V_S)$
▷ Select any node in the tree at random.
8 $v_{\text{to}} \leftarrow \text{ChooseAnyOne}(V)$
▷ Find the shortest path between these two nodes.
9 path $\leftarrow \text{ShortestPath}(N, v_{\text{from}}, v_{\text{to}})$
▷ Add the path to the tree being constructed.
10 $E \leftarrow E \cup \text{EdgesIn(path)}$
11 $V \leftarrow V \cup \text{VerticesIn}(E)$
12 $V_S \leftarrow V_S \setminus V$
13 $R_Q \leftarrow (V, E)$
14 return $R_Q$

Figure 5.2: The FG-SNEE ROUTING algorithm.


**SNEE.** Furthermore, the heuristic used to generate the routing tree does not necessarily prevent *hotspots* from occurring, i.e., nodes in the routing tree that are more congested than others, and therefore whose energy is depleted at a much faster rate, a fundamental concern when preserving network lifetime. The routing trees generated by this algorithm are therefore also unlikely to favour a short delivery time, which would involve having fewer and longer hops from the sources to the destination node.

Other approaches could be considered apart from the casting of the generation the routing tree as a Steiner tree problem. An alternative is proposed by Goel et al. [GRKL01], which describes a heuristic for the delay-constrained least-cost routing problem. However, the investigation of other approaches to generating routes from sources to the sink is left as future work.

Turning to other fixed-goal SNQPs in TinyDB, the routing tree is constructed at runtime by each node in the network selecting the parent that is closest to gateway node. Each node transmits tuples to its parent, which may be changed if network conditions change (i.e., if the parent dies, or is no longer deemed to be the closest one to the gateway node). TinyDB does not consider the location of data sources when building its routing tree, meaning that tuples may have to travel longer paths than they would otherwise. SNQL adopts a similar approach to TinyDB, except that copies of the same tuple may be sent to multiple parents, in order to reduce the impact of radio loss. It is therefore necessary to keep track
of the number of copies of each tuple in order to avoid double-counting when computing the query result. For both TinyDB and SNQL, routing decisions are made at runtime, however, it is noted that their focus is on in-network aggregation. Cougar does not describe its routing approach in sufficient detail to enable a comparison with other SNQPs.

\section*{5.2 Where-Scheduling}

The where-scheduling step, which involves assigning computation and sensing activities to nodes in the routing tree, for the FG-SNEE instantiation is broken down into two stages (see Figure 5.6). Firstly, in Fragment-Definition, \( P_Q \) is partitioned into fragments by the insertion of \texttt{EXCHANGE} operators along selected edges \((\text{child}, \text{op}) \in P_Q\) into a path \([(\text{child}, e_p), (e_c, \text{op})]\) where \(e_p\) and \(e_c\) denote, respectively, the producer and consumer parts of an \texttt{EXCHANGE} operator. This results in the intermediate fragmented algebraic form \( F_Q \). Then, Fragment-Instance-Assignment decides which sites will host each fragment, resulting in the distributed-algebraic form \( D_Q \).

In the Fragment-Definition stage, the edge selection criteria are both semantic, in the case of location- or attribute-sensitive operators (in which correctness criteria constrain placement), and pragmatic, in the case of an operator whose output size is larger than that of its child(ren) (in which case placement seeks to reduce overall network traffic). Figure 5.7(a) depicts the fragmented-algebraic form \( \text{FAF} \) for the physical-algebraic form in Figure 5.1 showing the \texttt{EXCHANGE} operators that define the four fragments. The notation \( F_n \) denotes fragment \( n \). At this stage, the set of sites where a fragment is to run has not been determined yet. Note that \texttt{EXCHANGE} has been inserted between each \texttt{AGGR_INIT} and the \texttt{AGGR_MERGE}, because the latter involves tuples from different sites, and therefore data redistribution is required. Note also that an \texttt{EXCHANGE} has been inserted below the \texttt{DELIVER}, because the latter is (as is \texttt{ACQUIRE}) location sensitive.

In the second stage, the Fragment-Instance-Assignment algorithm in Figure 5.6 computes \( D_Q \), i.e., the graph-representation of the query in distributed-algebraic form, by deciding on the assignment of fragment instances in \( F_Q \) to sites.

\footnote{Recall from Section 4.6.1 that if an operator is \emph{location sensitive} then there is no leeway as to where it may be placed, and that an operator is \emph{attribute sensitive} if it requires its input data to be partitioned in a specific way when several copies of it are executing in parallel.}
CHAPTER 5. A FIXED GOAL INSTANTIATION OF SNEE

Fragment-Definition($P_Q, \text{Size}$)

▷ Partition the Physical-Algebraic Form $P_Q$ using EXCHANGE operators,
▷ resulting in Fragmented-Algebraic Form $F_Q$. Size provides estimations
▷ of operator output sizes.

1. $F_Q \leftarrow P_Q$
2. while ▷ post-order traversing $F_Q$,
   ▷ let $op$ denote the current operator
   do for each $\text{child} \in op.\text{Children}$
   do if $\text{Size}(op) > \text{Size}(op.\text{Children})$
5.  or $op.\text{LocationSensitive} = \text{yes}$
6.  or $op.\text{AttributeSensitive} = \text{yes}$
7.  then ▷ Insert an EXCHANGE operator between
8.  ▷ $\text{child}$ and $op$.
9.  Delete($(\text{child}, \text{op}), P_Q)$
10. Insert($(\text{child}, e_p), P_Q)$
11. Insert($(e_p, e_c), P_Q)$
12. Insert($(e_c, \text{op}), P_Q)$
13. return $F_Q$

Figure 5.4: The Fragment-Definition algorithm.
5.2. WHERE-SCHEDULING

FRAGMENT-INSTANCE-ASSIGNMENT($F_Q$, $R_Q$, $\text{Size}$)

▷ Assign fragments in $F_Q$ to sites of the routing tree $R_Q$. $\text{Size}$ provides estimations of operator output sizes.

1. $D_Q \leftarrow F_Q$
2. while ▷ post-order traversing $D_Q$
   ▷ let $f$ denote the current fragment
3.   do if $op \in f$ and $op.\text{LocationSensitive} = \text{yes}$
4.      then ▷ No leeway as to where to place fragment.
5.      for each $s \in op.\text{Sites}$
6.        do $\text{Assign}(f.\text{New}, s, D_Q)$
7.    elseif $op \in f$ and $\text{Size}(f) < \text{Size}(f.\text{Children})$ and
8.        $(op.\text{AttributeSensitive} = \text{yes} \text{ or } op.\text{Iterative} = \text{yes})$
9.        then ▷ Place at deepest confluence site(s) in the routing tree.
10.       while ▷ post-order traversing $R_Q$,
11.          ▷ let $s$ denote the current site
12.          do if $s \Delta op$
13.             then $\text{Assign}(f.\text{New}, s, D_Q)$
14.     elseif $\text{Size}(f) < \text{Size}(f.\text{Children})$
15.        then ▷ Place in the same site(s) as the child fragment(s).
16.        for each $c \in f.\text{Children}$
17.          do for each $s \in c.\text{Sites}$
18.            do $\text{Assign}(f.\text{New}, s, D_Q)$
19.            else ▷ Place at the root site.
20.                $\text{Assign}(f.\text{New}, R_Q.\text{Root}, D_Q)$
21. return $D_Q$

Figure 5.5: The FRAGMENT-INSTANCE-ASSIGNMENT algorithm.

WHERE-SCHEDULING($P_Q$, $R_Q$, $\text{Size}$)

▷ Assign operators in PAF $P_Q$ to sites in routing tree $R_Q$.
▷ $\text{Size}$ provides estimations of operator output sizes.

1. $F_Q \leftarrow \text{FRAGMENT-DEFINITION}(P_Q, \text{Size})$ ▷ See Figure 5.4.
2. $D_Q \leftarrow \text{FRAGMENT-INSTANCE-ASSIGNMENT}(F_Q, R_Q, \text{Size})$ ▷ See Figure 5.5
   ▷ Merge fragments on the same node that may be scheduled at the same time.
3. $D_Q.\text{RemoveRedundantExchanges}()$
4. return $D_Q$

Figure 5.6: The FG-SNEE WHERE-SCHEDULING algorithm.
Figure 5.7: The Fragmented- and compact Distributed-Algebraic Forms for Q3 generated by FG-SNEE.
Figure 5.8: The Distributed-Algebraic Form for Q3 generated by FG-SNEE.
in the routing tree $R_Q$. The strategy that underpins decisions is as follows. If the size of the output of a fragment is smaller than that of its child(ren) then it is assigned to the deepest possible site(s) (i.e., the one with the longest path to the root) in $R_Q$, otherwise it is assigned to the shallowest site for which there is available memory, ideally the root. The aim here is to reduce radio traffic (by postponing the need to transmit the result with increased size). Semantic criteria dictate that if a fragment contains a location-sensitive operator, then instances of it are created and assigned to each corresponding site (i.e., one that acts as source or destination in $F_Q$). Semantic criteria also dictate that if a fragment contains an attribute-sensitive or iterative operator\footnote{Recall from Section 4.6.1 that an iterative operator $op$ may have instances whose parents are instances of $op$.}, then an instance of it is created and assigned to what is referred to as a confluence site for the operator.

To grasp the notion of a confluence site in this context, note that the extent of one logical flow (i.e., the output of a logical operator) may comprise tuples that, in the routing tree, travel along different routes (because, ultimately, there may be more than one sensor feeding tuples into the same logical extent). In response to this, instances of the same fragment are created in different sites, in which case EXCHANGE operators take on the responsibility for data distribution among fragment instances (concomitantly with their responsibility for mediating communication events). It follows that a fragment instance containing an attribute-sensitive operator is said to be effectively-placed only at sites in which the logical extent of its operand(s) has been reconstituted by confluence. Such sites are referred to as confluence sites. For a JOIN, a confluence site is a site through which all tuples from both its operands travel. In the case of aggregation operators, which are broken up into three physical operators (viz., AGGR_INIT, AGGR_MERGE, AGGR_EVAL), the notion of a confluence site does not apply to an AGGR_INIT. For a binary AGGR_MERGE (such as for an AVG, where AGGR_MERGE updates a (SUM, COUNT) pair), a confluence site is a site that tuples from both its operands travel through. Finally, for an AGGR_EVAL, a confluence site is a site through which tuples from all corresponding AGGR_MERGE operators travel. The most efficient confluence site to which to assign a fragment instance is considered to be the deepest, as it is the earliest to be reached in the path to the destination and hence the most likely to reduce downstream traffic.

Let $s \Delta op$ be true iff $s$ is the deepest confluence site for $op$. Figure 5.6
5.2. WHERE-SCHEDULING

describes the algorithm for assigning fragment instances. The resulting $D_Q$ for the example query $Q3$ is shown in Figures 5.7(b) and 5.8 in its compact and regular forms, respectively. It can be observed that instances of $F2$ have been created at multiple sites, as these fragments contain location-sensitive $ACQUIRE$ operators, whose placement is dictated by the physical schema definition in Figure 4.6. Also, instances of the attribute-sensitive $F1$ have been created and assigned to sites 10 and 22, the deepest confluence sites where tuples from both $F2$ and other instances of $F1$ which are further upstream are available (as it is a non-location-sensitive fragment and has been placed according to its expected output size, to reduce communication). The confluence of tuples can be seen more clearly in the DAF shown in Figure 5.8. Note also the absence of (e.g.) site 3 in Figure 5.7(b) with respect to Figures 5.3 and 5.8. This is because site 3 is only a relay node in the routing tree. A final post-processing step merges the two fragments in the FAF with the $DELIVER$ and $AGGR\_EVAL$ into a single fragment, as they have both been assigned to the same set of sites, thus eliminating a redundant $EXCHANGE$. This is an optimization performed to address the issue that $EXCHANGE$ operators may have been inserted too liberally during the first stage of this algorithm.

This algorithm aims to reduce the average node energy consumption by decreasing the amount of data sent over the radio. This is achieved by placing cardinality-reducing operators as close as possible to the leaves of the routing tree. While the decision-making exhibited by this algorithm does not explicitly consider alternative QoS expectations, reducing the amount of radio traffic is desirable for the other QoS of concern. For example, it is likely be beneficial for the optimization goal $\min \delta$, as less data needs to be sent over the radio and therefore takes less time to reach the gateway, and also for $\min \alpha$, because it will enable a higher throughput of tuples to travel from the sources to the gateway node.

Turning to related work, TinyDB does not fragment its $QEP$s, so the same $QEP$ is evaluated by all the nodes involved in a query. This approach is shown to be particularly well-suited for in-network aggregation performed in an incremental fashion. However, in the case of operators that need to see all the tuples in their inputs in order to produce a result, and where the inputs are tuples from different nodes in the network, the approach is not so well-suited. For example, TinyDB would not be able to handle a join inside the network correlating data from different regions in the network using a single query. The raw data would have to be sent outside the network to compute the join on a PC. Thus, if the join is data
reducing, it would not be possible to reap the benefits of in-network processing, as more data would be transmitted to evaluate the query, and hence more energy would be consumed. SNQL QEPs limit joins of this nature to being executed on the gateway node of the sensor network, and is therefore also unable to fully take advantage of the potential of in-network processing to yield energy savings. In the case of Cougar, insufficient detail is provided about where-scheduling policies to provide an adequate comparison.

5.3 When-Scheduling

For the FG-SNEE instantiation, the when-scheduling algorithm builds an agenda by an iterative process of adjustment. Given the memory available at, and the memory requirements of the fragment instances assigned to, each site, a candidate buffering factor $\beta$ is computed for each site. This candidate $\beta$ is used, along with the user-provided acquisition rate $\alpha$, to compute a candidate agenda. If the candidate agenda length (i.e., the time at which the execution of the last task in the candidate agenda is due to be completed as depicted in Figure 4.12) exceeds the smallest between the delivery time $\delta$ and the product of $\alpha \beta$, then the buffering factor $\beta$ is adjusted downwards and a new candidate agenda is computed. The process stops when the candidate agenda length meets the above criterion. The algorithm that computes the agenda is shown in Figures 5.9 and 5.11. The resulting agenda for Q3 is shown in Figure 5.11. It is similar to the agenda for Q1 in Figure 4.11, but note that it has been derived from a slightly different routing tree, and also, it has an additional fragment where tuples are merged during the incremental aggregation.

Figure 5.10 shows the algorithm used to generate an agenda, used by both instantiations of SNEE. The leaf fragments are added first to the agenda, $\alpha$ time units apart. In the agenda for Q3 shown in Figure 5.11, $\alpha = 15$ min, and $\beta = 17$, so each instance of the leaf fragments (only F2 in this case) is scheduled 17 times. Then the routing tree is traversed in a post-order fashion, and tasks corresponding to non-leaf fragments and exchange operators in the CDAF are scheduled. Instances of F1, the fragment that contains the AGGR MERGE operator, are

---

3Alternatively, TinyDB allows a materialization point to be defined within in the network, and subsequently queried, but Madden et al. [MFHH05] points out that the semantics of doing so are not clearly defined. SNEEql has precisely defined semantics for joins and other operators, as described in detail by Brenninkmeijer et al. [Bre09].
When-Scheduling (DQ, α, δ, Memory, Time)

▷ Schedule fragments and communications for DAF DQ with acquisition
▷ interval α and maximum delivery time δ (δ = ∞ if not specified in the
▷ QoS expectations). Time and Memory provide estimations of time
▷ and memory cost respectively.

▷ First determine the maximum buffering factor β possible
▷ according to memory available on the nodes.

1 \[ R_Q \leftarrow D_Q.rt \]
2 \[ \text{while} \quad \triangleright \text{pre-order traversing } R_Q, \]
3 \[ \triangleright \text{let } s \text{ denote the current site} \]
4 \[ \quad \text{do } \text{reqMem}_e \leftarrow \text{reqMem}_f \leftarrow 0 \]
5 \[ \quad \text{for each } f \in s.\text{AssignedFragments} \]
6 \[ \quad \text{do } x \leftarrow \text{Memory}(f.\text{EXCHANGE}) \]
7 \[ \quad \text{reqMem}_f \leftarrow \text{reqMem}_f + \text{Memory}(f) - x \]
8 \[ \quad \text{reqMem}_e \leftarrow \text{reqMem}_e + x \]
9 \[ \beta^*[s] \leftarrow \left\lfloor \frac{\text{AvailableMemory} - \text{reqMem}_f}{\text{reqMem}_e} \right\rfloor \]

▷ Now perform a binary search to find the highest β that
▷ will meet the delivery time δ.

10 \[ \beta_{\text{lower}} \leftarrow 1 \]
11 \[ \beta_{\text{upper}} \leftarrow \text{Min}(\beta^*) \]
12 \[ \text{repeat} \quad \beta_{\text{mid}} \leftarrow \left\lfloor \frac{\beta_{\text{lower}} + \beta_{\text{upper}}}{2} \right\rfloor \]
13 \[ \triangleright \text{Generate agenda using algorithm in Figure 5.10} \]
14 \[ \quad \text{A}_Q \leftarrow \text{BUILD-AGENDA}(D_Q, R_Q, \alpha, \beta_{\text{mid}}, \text{Time}) \]
15 \[ \quad \text{if } \text{A}_Q.\text{DeliveryTime} > \text{Min}(\alpha \ast \beta, \delta) \]
16 \[ \quad \text{then if } \beta_{\text{upper}} = 1 \]
17 \[ \quad \text{then exit } \triangleright \text{Error; cannot construct agenda.} \]
18 \[ \quad \text{else } \beta_{\text{upper}} \leftarrow \beta_{\text{mid}} \]
19 \[ \quad \text{else } \beta_{\text{lower}} \leftarrow \beta_{\text{mid}} \]
20 \[ \text{until } \beta_{\text{lower}} = \beta_{\text{upper}} \]
21 \[ \text{return } A_Q \]

Figure 5.9: The FG-SNEE When-scheduling algorithm.
\textbf{Build-Agenda}(D_Q, R_Q, \alpha, \beta, \text{Time})

\begin{itemize}
  \item Construct agenda for DAF \textit{D}_Q and routing tree \textit{R}_Q with acquisition interval \textit{\alpha} and buffering factor \textit{\beta}. \text{Time} provides estimations of time cost.
  \item First, schedule leaf fragments (i.e., those with \textit{SP\_ACQUIRE} operators); these execute every epoch.
  \begin{enumerate}
    \item for \(i \leftarrow 1\) to \(\beta\)
    \begin{enumerate}
      \item do for each \(s \in R_Q.\text{Sites}\)
      \begin{enumerate}
        \item do \(\text{nextSlot}[s] \leftarrow \alpha \times (i - 1)\)
        \item while \(\triangleright\) post-order traversing \textit{D}_Q
        \begin{enumerate}
          \item let \(f\) denote the current fragment
          \item do if \(f.\text{IsLeaf} = \text{yes}\)
          \begin{enumerate}
            \item then \(s.f.\text{ActAt} \leftarrow [\,]\)
            \item for each \(s \in f.\text{Sites}\)
            \begin{enumerate}
              \item do \(s.f.\text{ActAt}.\text{Append} \text{nextSlot}[s]\)
              \item \(\text{nextSlot}[s] \leftarrow + \text{Time}(s.f)\)
            \end{enumerate}
          \end{enumerate}
        \end{enumerate}
      \end{enumerate}
    \end{enumerate}
  \end{enumerate}
  \end{itemize}

\begin{itemize}
  \item Next, schedule non-leaf fragments and communications; these execute every \(\beta\) epoch(s), i.e., once per agenda evaluation episode.
  \item while \(\triangleright\) post-order traversing \textit{R}_Q,
  \begin{enumerate}
    \item do while \(\triangleright\) post-order traversing \textit{D}_Q
    \begin{enumerate}
      \item let \(s\) denote the current site
      \item do if \(f \in s.\text{AssignedFragments}\)
      \begin{enumerate}
        \item then \(f.\text{ActAt} \leftarrow \text{nextSlot}[s]\)
        \item \(\text{nextSlot}[s] \leftarrow + \text{Time}(f) \times \beta\)
      \end{enumerate}
    \end{enumerate}
  \end{enumerate}
\end{itemize}

\begin{itemize}
  \item \(\triangleright\) schedule comms between fragments
  \item \(s.\text{TX.}\text{ActAt} \leftarrow \max(\text{nextSlot}[s], \text{nextSlot}[s.\text{Parent}])\)
  \item \(s.\text{Parent.}\text{RX}(s).\text{ActAt} \leftarrow s.\text{TX.}\text{ActAt}\)
  \item \(\text{nextSlot}[s] \leftarrow + \text{Time}(s.\text{TX})\)
  \item \(\text{nextSlot}[s.\text{Parent}] \leftarrow + s.\text{Parent.}\text{RX}\)
\end{itemize}

\textbf{return} agenda

\end{itemize}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig510.png}
\caption{The \textbf{Build-Agenda} algorithm.}
\end{figure}

scheduled, firstly at site 22 (where tuples from sites 22 and 24 are combined), and then at site 10 (where the output of the \textit{AGGR\_MERGE} is combined with the tuples acquired at site 10). Instances of each fragment are connected by an \textit{EX-CHANGE} operator, which encapsulates a number of radio hops, each of which has a \textit{tx} task and corresponding \textit{rx} task. The task to execute \textit{F0} is the last task to be
5.3. WHEN-SCHEDULING

Figure 5.11: The agenda for Q3 generated by FG-SNEE with $\alpha = 15$ min.

This algorithm aims to reduce energy consumption, by increasing the buffering factor as much as possible within the delivery time constraint specified by the user. If no upper-bound for delivery time is stated, the buffering factor is limited by the available memory on a node and $\alpha \beta$. Buffering data and sending tuples in larger batches at a time conserves energy as the radio is turned on less frequently. It also allows greater scope for in-network lossless compression to take place, e.g., as proposed by Sadler et al. [SM06]. Note, however, that FG-SNEE when-scheduling assumes a fixed acquisition interval. It does not consider adjusting the acquisition interval, which is another way that energy could be conserved. A limitation of this algorithm is that it schedules communications sequentially, and therefore underutilizes the wireless medium. This is because, currently, the metadata described in Section 4.4 does not include information about the edges...
in the network connectivity graph over which communication can take place simultaneously (i.e., they are sufficient distance away from each other so that there is no radio interference, which would lead to data loss). Considering parallel communications in SNEE agendas, which would enable greater efficiency in terms of both time and energy, is therefore left as future work.

Cougar and TinyDB (in most cases) send their results right away, without performing buffering, so unlike FG-SNEE, no attempt is made to save energy by trading off the delivery time. This means that FG-SNEE when-scheduling decisions should be less energy intensive QEPs than those made by Cougar and TinyDB. The only exception for TinyDB is if a lifetime and acquisition constraint are specified that cannot be met, in which case buffering and load shedding take place in order to attempt to meet the constraints. However, load-shedding is likely to be undesirable, as it would be more energy efficient not to acquire a tuple that is to be subsequently discarded in the first place. Both instantiations of SNEE aim to avoid having to perform load-shedding with the intention of giving complete answers.

SNQL aims to conserve energy by increasing the delivery time. This is an adaptive technique, and the delivery time is increased as the energy stock decreases. This means that SNQL QEPs become more energy efficient as less energy is available, and as a consequence, it can be concluded that they could have been more energy efficient from the outset. In contrast, FG-SNEE has a constant delivery time throughout QEP evaluation, implying that FG-SNEE QEPs consume energy at a relatively constant rate. SNQL does not directly adjust the acquisition interval to save energy as TinyDB does when a lifetime constraint is specified. The only case when $\alpha$ is increased is if the memory available is low, in order to avoid the need for load shedding.

5.4 Conclusion

This chapter has described FG-SNEE, a concrete instantiation of the SNEE query processing stack template described in the previous chapter, by proposing algorithms for the multi-site steps. The collection of algorithms demonstrate that it is feasible to instantiate the query processing stack proposed in Chapter 4 using simple algorithms. While the algorithms do not enact sophisticated decision-making, they are, in this trait of theirs, following a forty-year tradition in query
5.4. CONCLUSION

optimization of preferring simple isolated steps whose composition leads to functionally useful overall outcomes. Thus, they fulfil their main purpose, viz., to validate the functional decomposition of decision-making steps from declarative query to executable code, which was the main challenge in the novel setting of wireless sensor networks as a platform for distributed query evaluation.

In particular, the algorithms are shown in Chapter 7 to be comparatively successful performance-wise in this setting. Thus, it is argued that making more query planning decisions explicitly (specifically, relating to routing, where-scheduling and when-scheduling decisions) enables FG-SNEE to generate QEPs that are more efficient in terms of energy than other fixed-goal SNQPs that are also concerned with preserving energy/maximizing lifetime.
Chapter 6

A QoS-aware Instantiation of SNEE

The previous chapter described an instantiation of the SNEE query processing stack that exhibits a fixed decision-making behaviour, with algorithms that generate QEPs based on decisions that aim to conserve energy. This chapter presents QoSA-SNEE, an instantiation of the SNEE query processing stack template with decision-making behaviours that respond to various user-specified QoS expectations. The algorithms described implement the multi-site decision-making steps of the SNEE query processing stack; the single-site steps are identical to those used in FG-SNEE, which essentially aim to reduce the size of intermediate results between operators (a goal that is likely to be beneficial for QoS expectations relating to both time and energy). Recall that, as discussed in Section 3.2, existing SNQPs such as TinyDB and Cougar have very limited QoS-awareness, as they make most decisions that affect QoS variables in a fixed manner, e.g., TinyDB QEPs generally stream back results to the gateway node as soon as they are acquired. The benefits of QoS awareness are empirically evaluated in Chapter 7, where QoSA-SNEE is compared against a suitable representative of the state-of-the-art among fixed-goal SNQPs, in order to ascertain that improvements in performance that are significant in magnitude are obtainable by making decisions in a QoS-aware manner.

QoS variables that are relevant to sensor network applications were enumerated in Section 4.3. To recap, a QoS expectation may involve specifying an optimization goal or constraints over QoS variables. The QoSA-SNEE instantiation supports a subset of the types of QoS expectations that are considered:
a user may specify a single optimization goal and zero or more constraints, all assumed to have an equal weighting. The ability to specify multiple optimization goals, or different weightings for each constituent, is left as future work.

The remainder of this chapter is structured as follows: Section 6.1 proposes some desiderata for the algorithms used to implement the steps in the QoSA-SNEE instantiation, and discusses the approach used to fulfil these desiderata. Sections 6.2–6.4 describe the algorithms for each step, motivating the approach used, and drawing contrasts with related work. Certain parts of the algorithms are described in more detail in the appendices, and are referenced throughout the chapter. The final section of this chapter summarizes the research contributions made herein.

### 6.1 Technical Approach

For a given step in QoSA-SNEE, while it would be possible to devise a different algorithm to implement decision-making in the light of a particular range of QoS expectations or optimization goals (e.g., FG-SNEE would constitute one such set of algorithms), this would be highly impractical, as with such an approach, different algorithms would need to be designed for different ranges in the spectrum of QoS expectations. A more effective approach would be to devise a single, generalized algorithm that is able to differ in its mode of decision-making depending on the QoS specified. Potentially, such an algorithm would be able to contend with different QoS variables by means of extensibility points, without having to undergo any changes. In other words, the core algorithms for the QoS-aware decision-making steps would be independent of the QoS expectations, with QoS specific parameters so that they exhibit alternative decision-making policies accordingly.

Several techniques exist that may be useful to implement alternative decision-making behaviours in a generalized manner. Possible suggestions discussed in Section 3.4 are the use of query rewriting rules, parametric optimization, multi-objective optimization, alternative cost models and utility functions. Query rewriting rules are useful in classical query optimization for reducing the size of intermediate results [GMUW00]. However, it is likely to be complex to represent QoS-enhancing transformations as rewriting rules because doing so would require a rewriting algebra that operated over not only an operator tree but also
CHAPTER 6. A QOS-AWARE INSTANTIATION OF SNEE

agendas. Moreover, heuristics would need to be identified for each QoS variable and doing so seems far from trivial. Parametric query optimization, in which alternative QEPs are generated at compile time for different points in the parameter space, although useful for cases when the parameters change at runtime (e.g., if conditions change), is less likely to be useful in QoS-aware query processing as the QoS expectations are expressed by the user alongside the query, and therefore determined beforehand. Multiple objective optimization, which may involve generating a Pareto curve of non-dominated results, is a promising approach, but is left for future work.

The most promising initial approach seemed to be the use of QoS-specific utility functions derived from time and energy cost models and other QEP parameters. Therefore, the QoS-aware steps of QoSA-SNEE employ utility functions to represent each QoS variable, which act as performance indicators for candidate QEPs with respect to QoS expectations. The utility functions are phrased in terms of CEM expressions that predict the time or energy cost of (part of) a candidate QEP based on the implications of decisions that may be made by the query optimizer. The utility functions either serve to rank and discard possible alternative solutions that have been generated (e.g., as done in QoSA-SNEE routing), or to model a constrained optimization problem which, when solved, will result in an appropriate solution (e.g., as done in QoSA-SNEE where- and when-scheduling). The intention is that alternative decision-making policies can be captured by casting the required decisions as a constrained optimization problem. Such a problem may then be delegated by QoSA-SNEE to an external, off-the-shelf optimizer. The intention is, therefore, that different QoS expectations are taken into account by generalized algorithms for the decision-making steps in QoSA-SNEE.

6.2 Routing

Recall from Section 4.3 that the routing step is responsible for deciding the nodes in the sensor network that will be used by tuples travelling from the sources to the gateway. For a given QEP it determines the nodes in the sensor network graph that will form its underlying routing tree. Given $P_Q$, the PAF that constitutes the output of step 3 of the SNEE stack, the routing step produces a collection
of candidate trees $\mathcal{R}_Q$. Every candidate tree includes the sources of the extents referenced in $P_Q$ as well as the destination node of the query, and may include additional relay nodes.

Routing decisions are likely to impact on QoS variables for several reasons. Firstly, the topology of the routing tree determines the number of communication hops from the source nodes to the gateway. Communication hops that span a shorter distance require the radio to transmit at a lower power than for longer distances [B.05]. Energy consumption is highly sensitive to the distance to be traversed and, indeed, it is proportional to the square of that distance [P.K.0]. However, having an increased number of hops will increase the time taken to reach the sink, albeit in the order of milliseconds. Another issue that routing decisions may give rise to is of hotspots, i.e., nodes with a substantially higher workload than the average within a QEP. Hotspot nodes are likely to be depleted of energy faster than others, and will therefore die earliest. Hotspots may also act as bottlenecks, and delay the delivery of results.

QoSA-SNEE routing uses a generate–evaluate–discard approach. Firstly, a collection of routing trees are generated using a generalized version of the FG-SNEE Steiner tree algorithm (described in Section 5.1). This function is invoked with parameters to specify different combinations of heuristics, with the intention of generating a diverse collection of routing trees. Secondly, the routing trees are scored using a QoS-dependent utility function, ranked in order of expected desirability, and the ones with worse-performing scores are discarded. This algorithm is intended to be generic and therefore able to support different QoS variables. Its QoS-dependent decision-making behaviour depends on the utility function used for routing tree evaluation. In Section 6.2.1 a detailed description of the algorithm is presented. Then, Section 6.2.2 shows routing trees produced for different optimization goals, and Section 6.2.3 presents a discussion on related work.

### 6.2.1 Algorithm Description

The overall algorithm for this step is shown in Figure 6.1. Firstly, at least $k_1$ routing trees are generated by computing approximations of a Steiner tree over the weighted connectivity graph $\mathcal{N}$ using different combinations of heuristics $H$ from the collection $\mathcal{H}$. An example of such a heuristic is whether to consider the energy or latency weightings in $\mathcal{N}$, and these are used with the aim of generating diverse routing trees. Each generated routing tree is scored, using a QoS-specific
CHAPTER 6. A QOS-AWARE INSTANTIATION OF SNEE

**Routing** $(P_Q, N, \theta)$

- Generate routing trees for PAF $P_Q$ and network $N$ for QoS optimization
- goal $\theta$, $k_1$, $k_2$ and $\mathbb{H}$ are SNEE global parameters that would be set by, e.g.,
- someone with a DBA role. $k_1$ specifies the minimum number of routing
- trees to generate, and $k_2$ specifies the number of routing trees to keep. $\mathbb{H}$ is a
- collection of different heuristics to use for generating diverse routing trees.

- Populate $R_Q$, a collection of candidate routing trees
- stored in order of descending score.

1. $R_Q \leftarrow \emptyset$
- Define the Steiner nodes.

2. $V_S = P_Q . Sources \cup \{ P_Q . Sink \}$
- Generate at least $k_1$ routing trees.

3. **while** $|R_Q| < k_1$
- Select different combinations of heuristics.

4. **do for each** $H \in \mathbb{H}$

5. **do** $\langle \Phi, \chi, \Psi, \Omega \rangle \leftarrow H$ **see** Table 6.1
- Generate routing tree using selected heuristics.

6. $R_Q \leftarrow$ **Meta-Steiner-Tree**$(\langle \Phi, \chi, \Psi, \Omega \rangle,$
- $V_S, P_Q . Sink, N)$ **see** Figure 6.2
- Calculate score for routing tree.

7. score $\leftarrow$ **Score**$(R_Q, \theta, N)$ **see** Figure 6.8.

8. $R_Q . Add(R_Q, score)$
- Delete any duplicate routing trees in $R_Q$.

9. $R_Q . RemoveDuplicates()$
- Keep the best $k_2$ routing trees, discard the rest.

10. **return** $R_Q \leftarrow R_Q[1: k_2]$

Figure 6.1: The QoSA-SNEE Routing algorithm.

utility function, and only the $k_2$ highest scoring trees form the input of where-scheduling, the next step in the SNEE query processing stack.

**Generating Candidate Routing Trees**

The **Meta-Steiner-Tree** algorithm (Figure 6.2) is used for generating alternative routing trees. It is a generalized algorithm for computing an approximation of a Steiner tree, based on the FG-SNEE routing algorithm described in Section 5.1. It receives four types of heuristics as parameters, denoted $\langle \Phi, \chi, \Psi, \Omega \rangle$, which dictate how different aspects of the tree construction process take place.
6.2. ROUTING

Meta-Steiner-Tree(⟨Φ, χ, Ψ, Ω⟩, VS, vroot, N)

▷ Compute the approximate Steiner tree (V, E) with Steiner nodes
▷ VS ∪ {vroot} for network graph N with multi-dimensional ⟨energy, latency⟩
▷ edge weightings, using combination of heuristics ⟨Φ, χ, Ψ, Ω⟩ (defined in
▷ Table 6.1) for tree construction.

▷ Select the first vertex to add to the tree according to the policy stipulated
▷ by heuristic Φ.
1  V ← FIRST-VΦ(VS, vroot) ▷ See Figure 6.3.
2  E ← ∅
3  VS ← VS \ V
4  while VS ≠ ∅ do
5      ▷ Select the type of weighting in N to use (e.g., energy or latency)
5      ▷ according to heuristic Ψ.
6          N ← SELECT-METRICΨ(N) ▷ See Figure 6.5.

▷ Update weighting associated with edges in N according to
▷ policy stipulated by heuristic Ω.
7  N′ ← UPDATE-WEIGHTINGSΩ(N, {V, E}) ▷ See Figure 6.4.

▷ Select the next branch to add to the tree according to
▷ the policy stipulated by heuristic χ.
8  ⟨vfrom, vto⟩ ← NEXT-BRANCHχ(N′, VS, V) ▷ See Figure 6.6.
9  path ← ShortestPath(N′, vfrom, vto)
10     ▷ Add vertices and edges to the tree accordingly.
11  E ← E ∪ EdgesIn(path)
12  V ← V ∪ VerticesIn(E)
13  VS ← VS \ V
14  return (V, E)

Figure 6.2: The Meta-Steiner-Tree algorithm.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Φ</td>
<td>select first vertex to add to tree</td>
<td>{sink, random}</td>
</tr>
<tr>
<td>χ</td>
<td>select next vertex to add to tree</td>
<td>{closest_sink, closest_any, random}</td>
</tr>
<tr>
<td>Ψ</td>
<td>network graph for current iteration</td>
<td>{energy, latency, random, mixed}</td>
</tr>
<tr>
<td>Ω</td>
<td>penalize nodes already in tree</td>
<td>{true, false}</td>
</tr>
</tbody>
</table>

Table 6.1: The heuristics used to generate routing trees.
The algorithm has the following general form: firstly, an initial node is added to the Steiner tree. Then, the remaining Steiner nodes are added in turn. Each added node is connected to an existing node in the Steiner tree via a path which may include both Steiner and non-Steiner nodes. The heuristics that \textsc{Meta-Steiner-Tree} takes as parameters determine the policies for taking each of the aforementioned decisions, e.g., which node to add first to the tree, or the next node to add to the tree.

\textbf{First-Vsink}(V_S, v_{root})
\hspace{1cm} \triangleright \text{First node added is the root of the Steiner tree.}
1 \hspace{0.4cm} \text{return } \{v_{root}\}

\textbf{First-Vrandom}(V_S, v_{root})
\hspace{1cm} \triangleright \text{First node added is a random Steiner node.}
1 \hspace{0.4cm} \text{return } \{\text{ChooseAnyOne}(V_S \cup \{v_{root}\})\}

Figure 6.3: Steiner tree algorithm heuristics for selecting the first node.

Table 6.1 summarizes the \(\langle \Phi, \chi, \Psi, \Omega \rangle\) heuristics that are parameters to the algorithm. The \(\Phi\) and \(\chi\) heuristics directly affect the order that nodes and edges are added to the graph. The \(\Phi\)-heuristic determines how the first node to be added to the tree is selected, which may either be set to the gateway node, or a node at random (Figure 6.3). At each iteration in Figure 6.2, the next node to be added to the tree needs to be selected. This is determined by the \(\chi\) heuristic used (Figure 6.6), which may either be the closest node to the sink, the closest node to any node already in the tree, or any a node at random not yet in the tree.

The \(\Psi\) and \(\Omega\) heuristics indirectly affect the order that nodes and edges are added to the graph by influencing the cost estimation of paths by the \textsc{ShortestPath} function. The \(\Psi\)-heuristic determines the metric to be used for the computation of the cost of paths in the graph, as depicted in Figure 6.5. This may be latency or energy, a randomly selected metric at each iteration of the algorithm (i.e., as each branch is added), or a mixed graph with a randomly selected metric for each edge in the tree. Using latency is intended to result in the generation of routing trees that favour time-related QoS (i.e., acquisition interval or delivery time), whereas using energy should result in trees that favour lifetime or total energy.
Update-Weightings\textsubscript{static}(\(N, T\))

▷ Do not penalize the use of edges in the network graph \(N\) already in the routing tree \(T\).

\(1\) return \(N\)

Update-Weightings\textsubscript{dynamic}(\(N, T\))

▷ Penalize the use of edges in the network graph \(N\) already in the tree \(T\), in order to avoid hotspots.

\(1\) \(E \leftarrow \text{EdgesIn}(N)\)
\(2\) \(V \leftarrow \text{VerticesIn}(N)\)
\(3\) for \(u \in \text{VerticesIn}(T)\) do
\(4\) \(\triangleright\) Obtain the number of sources in the sub-tree routed at node \(u\)
\(5\) \(n \leftarrow u.\text{OutCard}\)
\(6\) for \(\langle u, v, w \rangle \in E\) do
\(7\) \(\triangleright\) For each incoming/outgoing edge of \(u\):
\(8\) \(\triangleright\) Increase the cost using node \(v\) substantially to discourage use of edges for tuples from more sources.
\(9\) return \(\langle V, E \rangle\)

Figure 6.4: Steiner tree algorithm heuristics for penalizing edges already in the Steiner tree.
CHAPTER 6. A QOS-AWARE INSTANTIATION OF SNEE

SELECT-METRIC\textsubscript{energy}(N)
\begin{itemize}
  \item Given network graph \(N\) with multi-dimensional \(\langle\text{energy, latency}\rangle\) edge weightings, returns graph with energy edge weightings only.
\end{itemize}
1\ return EnergyGraph(N)

SELECT-METRIC\textsubscript{latency}(N)
\begin{itemize}
  \item Given network graph \(N\) with multi-dimensional \(\langle\text{energy, latency}\rangle\) edge weightings, returns graph with latency edge weightings only.
\end{itemize}
1\ return LatencyGraph(N)

SELECT-METRIC\textsubscript{random}(N)
\begin{itemize}
  \item Return either the energy- or latency-weighted graph.
\end{itemize}
1\ \(N_{\text{energy}} \leftarrow\) EnergyGraph(N)
2\ \(N_{\text{latency}} \leftarrow\) LatencyGraph(N)
3\ return ChooseAnyOne(\(\{N_{\text{energy}}, N_{\text{latency}}\}\))

SELECT-METRIC\textsubscript{mixed}(N)
\begin{itemize}
  \item Return a graph with uni-dimensional weightings mixing both the energy and latency values, with the aim of generating diverse trees.
\end{itemize}
1\ \(V \leftarrow\) VerticesIn(N)
2\ \(E \leftarrow\) \(\emptyset\)
3\ for \(\langle u, v, \text{energy, latency}\rangle\) \(\in\) EdgesIn(N)
4\ do \(w \leftarrow\) ChooseAnyOne(\(\{\text{energy, latency}\}\))
5\ \(E \leftarrow E \cup \{(u, v, w)\}\)
6\ return \(\langle V, E\rangle\)

Figure 6.5: Steiner tree algorithm heuristics that determine the radio link cost metric type that is used.
6.2. ROUTING

\textbf{Next-Branch\textit{random}}(N, V_S, V, v_{\text{root}})
\begin{algorithmic}[1]
\State $v_{\text{from}} \leftarrow \text{ChooseAnyOne}(V_S)$
\State $v_{\text{to}} \leftarrow \text{ChooseAnyOne}(V)$
\State \textbf{return} $\langle v_{\text{from}}, v_{\text{to}} \rangle$
\end{algorithmic}

\textbf{Next-Branch\textit{closest\_sink}}(N, V_S, V, v_{\text{root}})
\begin{algorithmic}[1]
\Comment{Next node added is the Steiner node closest to the root, connected to the root.}
\Comment{Obtain the node in $V_S$ closest to $v_{\text{root}}$ in graph $N$.}
\State $v_{\text{from}} \leftarrow \text{FindClosest}(N, v_{\text{root}}, V_S)$
\State $v_{\text{to}} \leftarrow v_{\text{root}}$
\State \textbf{return} $\langle v_{\text{from}}, v_{\text{to}} \rangle$
\end{algorithmic}

\textbf{Next-Branch\textit{closest\_any}}(N, V_S, V, v_{\text{root}})
\begin{algorithmic}[1]
\Comment{Next node added is the Steiner node closest to any node in the tree, connected to that node.}
\Comment{Obtain the pair of nodes $\langle v_{\text{from}} \in V, v_{\text{to}} \in V_S \rangle$ closest to each other in graph $N$.}
\State $\langle v_{\text{from}}, v_{\text{to}} \rangle \leftarrow \text{FindClosestAny}(N, V, V_S)$
\State \textbf{return} $\langle v_{\text{from}}, v_{\text{to}} \rangle$
\end{algorithmic}

Figure 6.6: Steiner tree algorithm heuristics for selecting the next branch to add to the tree.


\[ H = (\Phi, \chi, \Psi, \Omega) = \{ \langle \text{sink, random, latency, false} \rangle, \langle \text{sink, random, random, false} \rangle, \langle \text{sink, random, mixed, false} \rangle \} \]

(a) \( \text{opt\_goal} = \max \lambda \)

\[ H = (\Phi, \chi, \Psi, \Omega) = \{ \langle \text{sink, random, energy, true} \rangle, \langle \text{sink, random, random, true} \rangle, \langle \text{sink, random, mixed, true} \rangle, \langle \text{sink, closest\_sink, latency, true} \rangle \} \]

(b) \( \text{opt\_goal} \in \{ \min \alpha, \min \delta, \min \epsilon \} \)

Figure 6.7: The combinations of heuristics used for different optimization goals during routing tree generation.

QoS variables. The possibility of using a mixed graph, or selecting the graph randomly at each iteration, is intended to increase the diversity of trees obtained, attempting to, thereby, avoid the generation of locally optimal but globally suboptimal candidates. The \( \Omega \)-heuristic (see Figure 6.4) determines whether graph edges that are already in the tree should be penalized, by adding a weighting to such edges which increases exponentially according to the number of upstream sources. This is intended to discourage hotspots in the routing tree, and is therefore expected to promote the generation of routing trees that result in QEPs with longer lifetimes.

As shown in Figure 6.1, Steiner trees are generated using different combinations of these heuristics, with a view to creating a diverse collection of candidate routing trees. The different combinations of heuristics used in \( H \), which depend on the optimization goal given in the QoS expectations, are presented in Figure 6.7. Only a small set of combinations of heuristics are used in each case, with the aim of ensuring that most of the trees generated are unique. The \( \Phi \)-heuristic, which determines the first node to be added to the Steiner tree, was not found to have a significant impact on the results, and is always set to be the gateway node. Diversity in the trees generated is obtained by setting the \( \chi \)-heuristic, which selects the next node to be added to the tree, to be random in most cases, and also by varying the kind of weightings used in the \( \Psi \)-heuristic. For the \( \max \lambda \) goal, the \( \Omega \)-heuristic is set to \text{true}, thus penalizing the existence of nodes that are more congested. Conversely, for other optimization goals, the existence of hotspots is less of a concern, so the \( \Omega \)-heuristic is set to \text{false}. Note that only the optimization goal, and not the constraints in the QoS expectations, inform
the selection of heuristics. This is because, at this stage in query optimization, where query plan operators have not yet been placed, nor activities scheduled, insufficient decisions relating to the QEP have been taken to enable the CEMs to estimate values for the QoS variables in order to determine whether a constraint is easily achievable or completely infeasible.

**Scoring Routing Trees using QoS-dependent Utility Functions**

\[
\text{Score}(R_Q, \theta, N)
\]

\[\triangleright\] Compute the score for routing tree \( R_Q \) associated with network graph \( N \), using the version of the scoring function depending on optimization goal \( \theta \).

1. \textbf{if} \( \theta = \text{max} \lambda \)
2. \textbf{then return} \( 1/ \text{Score}_\theta(R_Q, \text{Root}, N) \)
3. \textbf{else return} \( \text{Score}_\theta(R_Q, \text{Root}, N) \)

\[
\text{Score}_{\text{Min} \alpha, \delta}(v, N)
\]

\[\triangleright\] Scoring function for \( \text{min} \alpha \) and \( \text{min} \delta \) optimization goals.

1. \( \alpha \leftarrow v.\text{OutCard} \)
2. \textbf{for each} \( u \in v.\text{Children} \)
3. \textbf{do} \( \alpha \leftarrow \alpha + \text{Score}_{\text{Min} \alpha, \delta}(u, N) \)
4. \textbf{return} \( \alpha \)

\[
\text{Score}_{\text{Min} \epsilon}(v, N)
\]

\[\triangleright\] Scoring function for \( \text{min} \epsilon \) optimization goal.

1. \( \epsilon \leftarrow \text{Get-Site-Energy-Consumption}(v, N) \) \[\triangleright\] See Figure 6.9
2. \textbf{for each} \( u \in v.\text{Children} \)
3. \textbf{do} \( \epsilon \leftarrow \epsilon + \text{Score}_{\text{Min} \epsilon}(u, N) \)
4. \textbf{return} \( \epsilon \)

\[
\text{Score}_{\text{Max} \lambda}(v, N)
\]

\[\triangleright\] Scoring function for \( \text{max} \lambda \) optimization goal.

1. \( \epsilon' \leftarrow \text{Get-Site-Energy-Consumption}(v, N) \) \[\triangleright\] See Figure 6.9
2. \( \lambda' \leftarrow v.\text{EnergyAvail}/\epsilon' \)
3. \textbf{for each} \( u \in v.\text{Children} \)
4. \textbf{do} \( \lambda' \leftarrow \text{Min}(\lambda', \text{Score}_{\text{Max} \lambda}(u, N)) \)
5. \textbf{return} \( \lambda' \)

Figure 6.8: QoSA-SNEE Routing and where-scheduling QoS scoring functions.
CHAPTER 6. A QOS-AWARE INSTANTIATION OF SNEE

Get-Site-Energy-Consumption\( (v, N) \)

\( \triangleright \) Calculate relative energy consumption of site \( v \in \text{VerticesIn}(N) \).

\( \triangleright \) Fixed, hardware-specific, energy overhead for each site.
1 \( \epsilon' \leftarrow v.\text{EnergyOverhead} \)

\( \triangleright \) Energy consumed by radio receiving data
2 \textbf{for each} \( u \in v.\text{Children} \)
3 \textbf{do} \( \triangleright \) Get the energy weighting for the edge in \( N \) from \( u \) to \( v \)
4 \( e \leftarrow \text{GetEnergyWeighting}(N, u, v) \)
5 \( \epsilon' \leftarrow \epsilon' + (e * u.\text{OutCard}) \)

\( \triangleright \) Energy consumed by radio sending data
6 \textbf{if not} \( v.\text{IsRoot} \)
7 \textbf{do} \( e \leftarrow \text{GetEnergyWeighting}(N, v, v.\text{Parent}) \)
8 \( \epsilon' \leftarrow \epsilon' + (e * v.\text{OutCard}) \)
9 \textbf{return} \( \epsilon' \)

Figure 6.9: Computing an approximation of relative site energy consumption, used by the routing and where-scheduling QoS scoring functions.

Once \( k_1 \) candidate routing trees have been generated, scoring takes place in order that the pruning of those candidates least likely to fulfil QoS-expectations can occur. The scoring functions employed, which vary depending on the optimization goal specified by the user, are defined in Figures 6.8 and 6.9. A scoring function is essentially a utility function that computes a score that reflects the expected desirability of the candidate routing tree with respect to the optimization goal specified. This is done by traversing the nodes in the routing tree. During the traversal, the \textit{OutCard} attribute, which gives an indication of the amount of data that a site in the routing tree will transmit to its parent, together with network metadata (i.e., the time and energy weightings associated with edges in the network graph \( N \)), are used to compute the score. A lower score indicates that a routing tree is more likely to positively impact the \textit{QEP} with respect to the QoS optimization goal.

These scoring functions are used both for evaluating candidate routing trees during the current step, and evaluating \textit{DAFs} in the subsequent where-scheduling step. However, in each case, the \textit{OutCard} associated with each routing tree site is
computed differently, as it depends on the information that is available based on
the decisions made until that point in the query processing stack. During where-
scheduling, it is possible to more precisely estimate the amount of data that will
flow between sites, as operator instances have been assigned to them. In routing,
less information is available, as fewer decisions have been made. A routing tree
defines the sites that participate in the QEP, the resources at each of these sites,
and the sites that are sources for the logical streams in the schema. It is silent
about the placement of operator instances. The best indication of the amount of
data that will travel along each edge in the routing tree is the number of stream
sources whose tuples will travel along it. Therefore, for QoSA-SNEE routing, the
OutCard used in functions in Figures 6.8 and 6.9 reflects the number of source
tuples in the sub-tree rooted at a given node, i.e., it is the maximum number of
tuples that may be acquired by the descendants of a node in the tree. Essentially,
it is equivalent to the amount of data that would flow between routing tree sites
in a SELECT * query without any predicates.

In general, for a routing tree to be amenable to the optimization goal of
minimizing the acquisition interval, it is postulated that it should aim to maxi-
mize throughput, by avoiding hotspot nodes where congestion is likely to occur.
However, in SNEE, given the constraint that stipulates that the data forwarding
phase must be shorter than the acquisition interval (described in Section 4.6.4),
and furthermore, the current limitation that SNEE schedules all communications
sequentially, for now the aim is to have as few edges (and consequently, as few
hops) as possible in the routing tree, so that it takes less time for tuples to travel
from the sources to the gateway. Considering the throughput in a less constrained
setting with parallel communications was left as future work. The ScoreMin α
function therefore returns the expected number of messages sent along the rout-
ing tree. The number of messages sent along a given edge is assumed to be equal
to the OutCard attribute of the source node of that edge. Like minimizing ac-
cquisition interval, minimizing delivery time also favours fewer, longer hops in the
routing tree. Relay nodes are avoided, as they delay the arrival of data at the
sink.

The total energy scoring function involves summing the energy used by routing
tree nodes to receive and transmit data, by considering the edge energy weighting
together with the site OutCard based on the amount of data it is predicted to
send towards the root. For maximizing lifetime, the routing tree should avoid
bottlenecks and consume energy throughout the network as evenly as possible. The lifetime function therefore computes the radio energy consumption of each node, and given the energy supply available at each node, returns the lifetime of the first node to die. It therefore strongly penalizes hotspots in the routing tree, and favours trees with more edges (i.e., shorter, lower-power radio hops).

\( \text{SCORE}_{\text{Min}} \epsilon \) and \( \text{SCORE}_{\text{Max}} \lambda \) both use the Get-Site-Energy-Consumption helper function (Figure 6.9) to calculate the relative energy consumption for a given site. It sums EnergyOverhead, an energy overhead associated with having an extra site in the routing tree, with the energy used for receiving and transmitting by a particular node. This value is an order of magnitude greater than sending or receiving an additional tuple via radio, to heavily penalize the inclusion of an additional node in the routing tree. The effect of this is to significantly impact the total network energy consumption \( \epsilon \), but to have no effect on the lifetime \( \lambda \), as the energy consumption for every site is increased by the same value (assuming that the network is homogenous). Note that the focus is on radio energy, as it is assumed that it will be dominant over other components in the node, and furthermore, radio is the hardware component which is most likely to be affected by routing tree selection.

Once an initial \( k_1 \) routing trees have been generated and scored, these are ranked, and pruning takes place, keeping only the best \( k_2 \) routing trees.

### 6.2.2 Example Output

Example routing trees for the queries in Figure 4.3, derived from the network in Figure 4.5, are depicted in Figures 6.10 and 6.11. The arrows between the nodes indicate the flow of tuples. Nodes which are grayed out do not participate in the QEP. To the right of each routing tree, a table ranking the lifetime of the nodes is given, in ascending order. Nodes which appear in the same cell of the table have the same lifetime. Note that the sink node does not appear in the lifetime ranking table as, throughout this dissertation, it is assumed to be tethered.

Figure 6.10(a) shows the routing tree for \( Q1 \), \( Q2 \) and \( Q3 \) compiled for the max \( \lambda \) optimization goal. The routing tree is the same for all three queries because they all have the same nodes as sources. From the lifetime ranking table, it can be seen that node 11 consumes the most energy. This is because it is the relay node that sends tuples for a longer distance than any other node in the QEP, and therefore needs to use a higher transmission power setting. Note that node
6.2. ROUTING

(a) \( \text{query} \in \{Q_1, Q_2, Q_3\} \), \( \text{optGoal} = \max \lambda \).

(b) \( \text{query} \in \{Q_1, Q_2, Q_3\} \), \( \text{optGoal} \in \{\min \alpha, \min \delta, \min \epsilon\} \).

Figure 6.10: Routing trees obtained for queries 1–3.
CHAPTER 6. A QOS-AWARE INSTANTIATION OF SNEE

(a) query ∈ \{Q4, Q5\}, optGoal = \max \lambda

(b) query ∈ \{Q4, Q5\}, optGoal ∈ \{\min \alpha, \min \delta\}

Figure 6.11: Routing trees obtained for queries 4 and 5.
11 could have sent its tuples via node 6 in order to use a lower transmission power; however, this would result in node 6 becoming a hotspot as it would be receiving tuples from two nodes, and sending twice the number of tuples of other nodes in the QEP. This tree is therefore characterized by (1) first and foremost, minimizing the maximum amount of data received and transmitted by any node in the routing tree, and (2) giving preference to communications made up of shorter hops, compared to longer ones. Relay nodes are used as much as possible to achieve (1) and (2). The same characteristics apply to the max $\lambda$ routing tree for $Q^3$ and $Q^4$ (Figure 6.11(a)). Note, however, that in this case, as most nodes in the network are source nodes, there is less scope for improving network lifetime by having shorter hops in the routing tree by using relay nodes; longer hops are present in order to reduce the maximum degree of any node in the routing tree.

The routing tree in Figure 6.10(b) is obtained by compiling $Q^1$–$Q^3$ with the optimization goal $\in \{ \min \alpha, \min \delta, \min \epsilon \}$. This routing tree is characterized by having fewer, longer hops than the previous one. There is also no concern about avoiding hotspots in the tree topology. This enables data to reach the source node in less time, and therefore yields a shorter delivery time. It also allows a shorter acquisition interval, as the routing tree allows a higher throughput of data from the sources to the sink. For the $\min \epsilon$ optimization goal, this routing tree is preferred because it has the lowest number of sites participating in the QEP. In the case of $Q^4$ or $Q^5$ when compiling for $\min \epsilon$, a different routing tree to the one in Figure 6.11(b) is obtained, albeit with similar characteristics.

### 6.2.3 Discussion

This section has proposed a QoS-aware instantiation of the routing step in the SNEE stack. Using a QoS-dependent scoring function, routing trees are assessed with regards to their suitability to a specific optimization goal. The proposed implementation could be extended with scoring functions for other QoS variables (e.g., average energy consumption) without changing the core algorithm. Similarly, using the same framework, it is conceivable that composite scoring functions reflecting more than one QoS variable could be implemented. This could be used if there is more than one optimization goal, or to take constraints in the QoS expectations into account. Furthermore, weightings in the QoS could be supported by combining functions and applying the respective weighting. These possible extensions show the extensibility of QoSA-SNEE routing; they are however left as
future work.

There are no examples known to the author of complete \textit{SNQPs} that make any routing query planning decisions in a \textit{QoS}-aware manner. \textit{FG-SNEE} routing only considers the link energy variable in the network graph, therefore deriving a routing tree that aims to reduce average node energy consumption. In TinyDB, routing decisions are made at the sensor network sites during query evaluation. Each site transmits tuples to its parent site, which is selected on the basis that it is the node with the best link quality closer to the gateway of the network. \textit{QoSA-SNEE} routing does not necessarily select a parent in the routing tree with the best link quality, and may instead opt to pick one with lesser link quality in order to distribute the load across network nodes more evenly. Neither TinyDB nor \textit{FG-SNEE} make any attempt to find routes that spread the load evenly throughout the network (e.g., to increase lifetime), or to find routes that involve longer but fewer hops (e.g., to reduce delivery time).

\textit{Wave Scheduling} \cite{TYD+07}, while not a complete \textit{SNQP} like \textit{SNEE} or TinyDB, proposes a technique in which routing decisions are made with a degree of \textit{QoS}-awareness. In \textit{Wave Scheduling}, routing decisions can either aim to decrease total energy consumption, or decrease delivery time. Unlike \textit{SNEE}, in which timing decisions are made after routing decisions, in this case routing decisions are made based on a previously determined time schedule. The agenda used also differs from the type used in \textit{SNEE} in that while timeslots for communication activities are stipulated, timeslots for computation or sensing activities are not. The routing decision-making algorithm adopted in \textit{Wave Scheduling} is an extension of Dijkstra’s shortest path algorithm \cite{CLRS01}, in which the metric used varies depending on whether it is optimizing to minimize total energy consumption or delivery time. To minimize total energy consumption, paths with the fewest hops are favoured. This may result in a higher delivery time, as tuples may have to wait en-route for the required phase in the agenda\footnote{Recall from Section \ref{sec:wave-scheduling} that the Wave Scheduling approach involves periodic phases, referred to as the North, South, East and West phases, indicating the direction that tuples travel.} To minimize delivery time, paths are selected by considering the delay that a tuple will incur at a node before the next phase. The path chosen is the one that will incur the shortest delivery time. This may however result in a path which involves a greater number of hops, and consequently, a higher energy consumption. In this way, routing decisions are able to trade-off delivery time and energy consumption \textit{QoS} variables. However,
other QoS variables such as lifetime or acquisition interval are not considered by Wave Scheduling.

6.3 Where-Scheduling

Where-scheduling (i.e., step 5) of the SNEE query processing stack in Figure 4.1 involves placing operators from the PAF onto physical sites of the sensor network in the routing tree. It is essentially responsible for deciding where different computations in the distributed algorithm that makes up the QEP are executed. QoSA-SNEE where-scheduling takes as input a collection of routing trees $R_Q$ and a single PAF $P_Q$, and generates one DAF for each routing tree in $R_Q$. The final result is a collection of DAFs denoted $D_Q$, with cardinality $|R_Q|$.

The decisions in the where-scheduling step directly impact the amount of data that needs to be transmitted along edges in the routing tree, an important concern both for time- and energy-related QoS. For example, placing a data-decreasing operator as deep as possible in the routing tree is likely to reduce the amount of radio traffic during the execution of the QEP, reducing both energy consumption and delivery time. Where-scheduling decisions may also cause nodes to be overloaded, e.g., if too many operators are allocated to a single node, suggesting that they may, potentially, play a role in avoiding hotspots.

In QoSA-SNEE where-scheduling, although the QoS-dependent utility functions have a similar form to those used in QoSA-SNEE routing, these are used differently. In routing, several possible solutions are generated, and then evaluated using the utility function. In where-scheduling, the utility function is used as the objective function for a constrained optimization problem which comprises constraints such as memory available at each site. This problem steers a search using an external optimizer for an allocation of operator instances that yields the most favourable value of the objective function. Section 6.3.1 describes the algorithm in more detail, Section 6.3.2 provides example output, and Section 6.3.2 presents a discussion on issues such as related work.

6.3.1 Algorithm Description

The QoSA-SNEE where-scheduling algorithm, presented in Figure 6.12 generates a DAF for each routing tree in the input collection $R_Q$. Recall from Section 4.6.3 that a DAF $D_Q$ is a triple $(opInsts, flows, assignment)$. For each
WHERE-SCHEDULING($P_Q, R_Q$)

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Define $D_Q = []$</td>
</tr>
<tr>
<td>2</td>
<td>For each $R_Q \in R_Q$ do</td>
</tr>
<tr>
<td>3</td>
<td>- Generate the DAF with partial assignment</td>
</tr>
<tr>
<td>4</td>
<td>- $D_Q \leftarrow$ GENERATE-PARTIAL-DAF($P_Q, R_Q$)</td>
</tr>
<tr>
<td>5</td>
<td>- Obtain an assignment $a$ for the nodes in the DAF</td>
</tr>
<tr>
<td>6</td>
<td>- $\langle a, objFnVal \rangle \leftarrow$ METASEARCH($R_Q, D_Q, \theta$)</td>
</tr>
<tr>
<td>7</td>
<td>- Combine instances of the same operator in the DAF that have been assigned to the same site</td>
</tr>
<tr>
<td>8</td>
<td>- $D_Q, RemoveRedundantOplnsts()$</td>
</tr>
<tr>
<td></td>
<td>return $D_Q$</td>
</tr>
</tbody>
</table>

Figure 6.12: The QoSA-SNEE WHERE-SCHEDULING algorithm.

Routing tree, firstly an operator instance tree (referred to as a partial DAF) is derived from the PAF, which involves determining how many instances of each operator are created, and how the operator instances are connected to each other. In this initial step, a set of $D_Q, opInsts$ and $D_Q, flows$ is determined. Then, the problem of assigning operator instances to routing tree nodes is cast as a constrained optimization problem, with a QoS-dependent objective function and constraints, and delegated to an external optimizer which aims to find the assignment of operator instances to sites that yields the best value for the objective function. This results in the population of the $D_Q, assignment$ mapping. Finally, some post-processing takes place with the aim of removing redundant operator instances, in which instances of the same operator that are co-located on a site are merged into a single operator instance.

Construction of the Partial DAF

The algorithm presented in Figure 6.13 constructs the partial DAF in a bottom-up fashion, by performing a post-order traversal of the PAF. At each step, depending on the properties of the operator being visited, the number of new operator instances spawned and the way that they are connected to existing
6.3. WHERE-SCHEDULING

\textbf{Generate-Partial-DAF}(P_Q, R_Q)

\begin{itemize}
  \item $P_Q$ is the Physical-Algebraic Form.
  \item $R_Q$ is a routing tree.
\end{itemize}

\begin{itemize}
  \item Generates the partial DAF, an operator instance tree derived by superimposing the PAF on the RT.
\end{itemize}

\begin{verbatim}
1 $D_Q \leftarrow \text{new DAF}(P_Q, R_Q)$
2 \textbf{while} post-order traversing $P_Q$
3 \hspace{1em} let $op$ denote the current operator
4 \hspace{1em} \textbf{do if} $op$.LocationSensitive
5 \hspace{2em} \textbf{then} ADD-LOC-SEN-OP-INSTS($op$, $D_Q$) \textbf{\triangleright} See Figure 6.14
6 \hspace{1em} \textbf{elseif} $op$.Iterative
7 \hspace{2em} \textbf{then} ADD-ITER-OP-INSTS($op$, $D_Q$, $R_Q$) \textbf{\triangleright} See Figure 6.15
8 \hspace{1em} \textbf{elseif} $op$.AttributeSensitive
9 \hspace{2em} \textbf{then} ADD-ATTR-SEN-OP-INSTS($op$, $D_Q$) \textbf{\triangleright} See Figure 6.16
10 \hspace{1em} \textbf{else} ADD-OTHER-OP-INSTS($op$, $D_Q$) \textbf{\triangleright} See Figure 6.17
11 \textbf{return} $D_Q$
\end{verbatim}

Figure 6.13: The \textbf{Generate-Partial-DAF} algorithm.

\textbf{ADD-LOC-SEN-OP-INSTS}($op$, $D_Q$)

\begin{itemize}
  \item Add operator instances as specified by \texttt{RequiredSites};
  \item link all child operator instances to every instance of $op$.
\end{itemize}

\begin{verbatim}
1 \textbf{for each} site $\in op$.RequiredSites
2 \hspace{1em} \textbf{do} $opInst \leftarrow \text{new OpInstance}(op, site.Id)$
3 \hspace{1em} $op$.OutputInstances.Add($opInst$)
4 \hspace{1em} $D_Q$.AddNode($opInst$)
5 \hspace{1em} $D_Q$.SetAssignment($opInst$) $\leftarrow$ site
6 \hspace{1em} CONVERGE-SUBSTREAMS($D_Q$, $op$.Child.Instances, $opInst$) \textbf{\triangleright} See Figure 6.18
\end{verbatim}

Figure 6.14: Adding location sensitive operator instances to the Partial DAF.
\textbf{ADD-ITER-OP-INSTS}(\textit{op}, \textit{D}_Q, \textit{R}_Q)

\begin{verbatim}
  \triangleright Add operator instances for iterative operator, and
  \triangleright connect any child operator accordingly.
  1 \textit{childOp} \leftarrow \textit{op}.Child
  2 \textit{\Omega} \leftarrow \textit{childOp}.OutputInstances
  3 \textbf{while} \triangleright \text{post-order traversing} \textit{R}_Q
    \triangleright \text{let} \textit{site} denote the current routing tree node
    4 \textit{\omega} \leftarrow \text{GET-CONFLUENCE-OP-INSTANCES}(\textit{site}, \textit{\Omega}) \triangleright \text{See Figure 6.18}
    \triangleright \text{Exit loop if no more iterative instances}
    5 \textbf{if} \textit{\Omega} = \textit{\omega}
    6 \textbf{then} \textbf{break}
    \triangleright \text{Add operator instance for site if there is}
    \triangleright \text{confluence of tuples from the current or}
    \triangleright \text{child operator instances.}
    7 \textbf{if} |\textit{\omega}| > 1
      \textbf{do} \textit{opInst} \leftarrow \text{NEW OpInstance}(\textit{op}, \textit{site}.Id)
      \textit{D}_Q.\text{AddNode}(\textit{opInst})
      \textit{CONVERGE-SUBSTREAMS}(\textit{D}_Q, \textit{\omega}, \textit{opInst}) \triangleright \text{See Figure 6.18}
      \textit{\Omega} \leftarrow \textit{\Omega} \cup \{\textit{opInstance}\} \setminus \textit{\omega}
  10 \textit{op}.OutputInstances \leftarrow \textit{\Omega}
\end{verbatim}

\textbf{Figure 6.15}: Adding iterative operator instances to the Partial DAF.

\textbf{ADD-ATTR-SENS-OP-INSTS}(\textit{op}, \textit{D}_Q)

\begin{verbatim}
  \triangleright Add a single operator instance for attribute sensitive
  \triangleright operators, and link all child operator instances to this
  \triangleright operator instance.
  1 \textit{opInst} \leftarrow \text{NEW OpInstance}(\textit{op}, 1)
  2 \textit{op}.OutputInstances.Add(\textit{opInstance})
  3 \textit{D}_Q.\text{AddNode}(\textit{opInst})
  4 \textbf{for each} \textit{childOp} \in \textit{op}.Children
    5 \textbf{do} \text{CONVERGE-SUBSTREAMS}(\textit{childOp}.Instances, \textit{opInst}) \triangleright \text{See Figure 6.18}
\end{verbatim}

\textbf{Figure 6.16}: Adding attribute sensitive operator instances to the Partial DAF.
ADD-OTHER-OP-INSTS\((op, D_Q)\)

▷ Add operator instances for operators which are not
▷ attribute-sensitive, location-sensitive or iterative.

1 \textbf{for each} childOp \in op.Children
2 \textbf{do for each} childOpInst \in childOp.OutputInstances
3 \hspace{1em} \textbf{do} opInstance \leftarrow \text{new OpInstance}(op,
4 \hspace{1em} \quad \text{childOpInst.Id})
5 \hspace{1em} op.AddOutputInstance(opInst)
6 \hspace{1em} D_Q.AddNode(opInst)
7 \hspace{1em} D_Q.AddEdge(childOpInstance, opInst)

Figure 6.17: Adding other types of operator instances to the Partial DAF.

CONVERGE-SUBSTREAMS\((D_Q, childOpInstances, opInst)\)

▷ Link each instance in childOpInstances to opInst

1 \textbf{for each} childOpInst \in childOpInstances
2 \hspace{1em} \textbf{do} D_Q.AddEdge(childOpInst, opInst)

GET-CONFLUENCE-OP-INSTANCES\((R_Q, rootSite, allOpInstances)\)

▷ Helper method for ADD-ITERATIVE-OP-INSTANCES. Return the
▷ operator instances whose tuple source is a site in the subtree of \(R_Q\)
▷ rooted at \(rootSite\)

1 \text{opInstances} \leftarrow \{\}
2 \textbf{while} \hspace{1em} \triangleright \text{post-order traversing } R_Q \text{ rooted at } rootSite
3 \hspace{1.5em} \triangleright \text{let } site \text{ denote the current routing tree node}
4 \hspace{1.5em} \textbf{do for each} opInst \in allOpInstances
5 \hspace{2em} \textbf{if} opInst.Site = site
6 \hspace{2.5em} \textbf{then} opInstances.Add(opInst)
7 \hspace{2em} \textbf{return} \text{opInstances}

Figure 6.18: Helper algorithms used when generating a partial DAF.
CHAPTER 6. A QOS-AWARE INSTANTIATION OF SNEE

(a) PAF for Q3.

(b) Routing tree for Q3.

(c) Step-by-step generation of the corresponding partial DAF.

Figure 6.19: PAF, routing tree and partial DAF for Q3 with opt_goal=min δ.
operator instances in the tree may differ. Figures 6.14–6.17 present the algorithms that implement these variations, described in the following paragraphs in detail. Helper methods used by these algorithms are shown in Figure 6.18. At this stage, the $D_{Q, assignment}$ mapping is only populated for location-sensitive operators, i.e., those whose placement is fixed. The generation of the partial DAF is described by considering the compilation of $Q_3$, an example which illustrates how all the operator types are dealt with. Figures 6.19(a) and 6.19(b) show the example PAF and routing tree, denoted $P_Q$ and $R_Q$ respectively, that are the input of the where-scheduling, and Figure 6.19(c) shows the resulting partial DAF. Note the steps in Figure 6.19(c) labelled 1–5, indicating how the partial DAF is constructed. In the partial DAF the name of each operator instance is suffixed by an underscore followed by its deepest confluence site, i.e., the node furthest away from the root of the routing tree where this operator instance may be placed, an approach used to uniquely identify each operator instance. This is not necessarily the site that it will be assigned to; a non-location-sensitive operator instance may be placed at any site on the path from the deepest confluence site to the root site, subject to certain constraints, e.g., there must be sufficient memory available on the site.

The first operator encountered during the post-order traversal of $P_Q$ is the SP_ACQUIRE operator, which is location sensitive (i.e., it is a requirement for it to execute at specific sensor network nodes for the QEP to produce correct results). Figure 6.14 presents the algorithm used for creating operator instances in this case. The number of operator instances generated is fixed, as an operator instance is required at each source site where data is available according to the physical schema metadata. As depicted in Step 1 of Figure 6.19(c), four SP_ACQUIRE operator instances comprise the leaves of the DAF, and the assignment mapping is populated at this stage (denoted by the rounded rectangle drawn with a thick dotted line around these operators, and the number indicating the assigned site).

The next operator in $P_Q$ is AGGR_INIT. This operator is not location sensitive, attribute sensitive or iterative. For such operators, the algorithm in Figure 6.17 creates as many instances as there exist for the operator’s child, i.e., $|op.Instances| = |op.Child.Instances|$. For example, Step 2 of Figure 6.19(c) shows that four instances of the AGGR_INIT operator have been created, one for each SP_ACQUIRE instance. This is done because, e.g., in the case of Figure 6.19(c)
CHAPTER 6. A QOS-AWARE INSTANTIATION OF SNEE

an AGGR_INIT may (if deemed advantageous by the optimizer when assigning operator instances to sites) be co-located with each SP_ACQUIRE in the next stage of where-scheduling in which assignment is populated. This, in effect, creates the maximum possible number of operator instances for an operator (although redundant ones are removed after the assignment mapping has been populated). For each child operator instance, childOpInst$_{i=1...n} \in$ op.Child.Instances, an outgoing edge is added to $D_Q$ such that opInst$_{j=1...n} \in$ op.Instances, where $i = j$. For example, in Figure 6.19(c) SP_ACQUIRE_22 is connected to AGGR_INIT_22, SP_ACQUIRE_10 is connected to AGGR_INIT_10, etc., thus creating parallel tuple flows from the child instances to the parent instances.

AGGR_MERGE, used for incremental aggregation, is an iterative operator and is therefore handled by the algorithm in Figure 6.15. The number of operator instances to create, and how to connect them, is dictated by the topology of the routing tree and the locations of child operator instances. For example, in Step 3 of Figure 6.19 two instances of AGGR_MERGE have been created; one which combines tuples from sites 10 and 22, whose deepest confluence site is 10, and another which combines tuples from sites 4 and 24, whose deepest confluence site is 12. While performing a post-order traversal of the routing tree $R_Q$, an operator instance is created for each site in the routing tree where there is confluence of tuples from more than one operator instance in $\{op.Child.Instances\cup op.Instances\}$, although it is not necessarily assigned to that site in the subsequent steps. It is possible for no instances of an iterative operator to be created, in which case all instances of the child of the iterative operator are connected directly to instances of the parent operator (i.e., in this case, it would be the AGGR_EVAL).

AGGR_EVAL is an attribute-sensitive operator. The algorithm employed for such operators is presented in Figure 6.16. In this case, a single operator-instance is created and all instances of the child operator(s) have an outgoing connected edge to it, effectively causing the child instance outputs to converge to a single operator instance. In Step 4 of Figure 6.19 it can be seen that a single instance of AGGR_EVAL has been created, and that both AGGR_MERGE instances are

---

2Recall that an iterative operator $op$ has the property that an instance of $op$ may have an outgoing edge connected to the input of another instance of $op$ (as opposed to an instance of $op.Parent$, as is usually the case).

3Although more the one operator instance could be created in this case, this would lead to the creation of a p-operator instance, i.e., an operator instance that is being parallelized because it is resource intensive. This type of parallelism is not considered in this dissertation, and is complementary to the work presented herein.
connected to it. For the location-sensitive **DELIVER** operator in \( P_Q \), a single operator-instance is created, and all instances of the child of the **DELIVER** in the PAF (i.e., in this case, only the single instance of the **AGGR_EVAL**) have an outgoing edge connected to it. As shown in Step 5 of Figure 6.19, a single instance of **DELIVER** is assigned to site 0, the gateway node to which results are sent.

The result of this initial stage is a tree of operator instances comprising distributed logical data flows. The data flows are distributed, as inherently sensor streams are generated from spatially dispersed locations in a sensor network. The shape of the partial DAF tree captures properties of the QEP, such as operator instances that may (but are not necessarily required to) execute in different sites (e.g., as shown in Figure 6.19, there may be up to four **AGGR_INIT** operator instances in as there are four source sites). Another relevant property is the specific operator instances where all the tuples need to converge (e.g., the **AGGR_EVAL** operator above, or a **JOIN** correlating data from different regions of the network that cannot be partitioned). In the next stage, non-location sensitive operator instances are assigned to sites in the routing tree, effectively superimposing the tree of operator instances onto the physical network fabric. This approach differs from that of FG-SNEE where-scheduling, in which the decision about where to place computations and the number of instances created per operator are conflated. With QoSA-SNEE where-scheduling, the optimizer has more fine-grained control about the number of operator instances to create. Also, the intent behind this initial stage in QoSA-SNEE where-scheduling is to characterize the search space of alternative DAFs that can be explored during the randomized search in the next stage.

**Assigning Operator Instances to Sites in the Routing Tree**

Once the partial DAF has been created, the domain of the assignment mapping is known. The next step, described by the pseudocode in Figure 6.20, involves assigning a routing tree site to each operator instance, in order to populate the range of the assignment mapping. This is achieved by generating a constrained optimization problem, and invoking an external optimizer that performs a randomized search of the solution space, and returns the solution with the best QoS-dependent objective function value found.

The optimizer used is NOMADm [Abr09], which performs a randomized search
CHAPTER 6. A QOS-AWARE INSTANTIATION OF SNEE

\begin{align*}
\text{Metasearch}(R_Q, D_Q, \theta) & \\
\triangleright & \text{Invokes external optimizer to obtain assignment for DAF } D_Q \text{ and} \\
\triangleright & \text{routing tree } R_Q \text{ which is quasi-optimal for optimization goal } \theta. \\
\triangleright & k_3\text{–}k_6 \text{ are SNEE global parameters that would be set by a DBA.} \\
\triangleright & \text{Set the initial number of neighbors } (k_3) \text{ for} \\
\triangleright & \text{optimizer neighborhood function to generate.} \\
\end{align*}

\begin{verbatim}
1 nn ← k_3
2 count ← 0
3 objFnVal_{\min} ← \infty
\triangleright k_4 \text{ is the number of consecutive } objFnVal \text{ values within} \\
\triangleright a k_5\% \text{ threshold after which to stop searching.} \\
4 while count < k_4 do
5 \langle a, objFnVal \rangle ← \text{InvokeExternalOptimizer}(nn, R_Q, D_Q, Score_\theta, \\
\quad \text{InitialPoint, IsFeasible, Neighbourhood}) \\
6 \quad if objFnVal = prevObjFnVal ± k_5\% then count ← count + 1 \\
7 \quad else count ← 1 \\
8 \quad if objFnVal < objFnVal_{\min} then a_{\min} ← a \\
9 \quad objFnVal_{\min} ← objFnVal \\
10 \quad prevObjFnVal ← objFnVal \\
\triangleright k_6 \text{ is the value by which to increase} \\
\triangleright nn \text{ at each iteration.} \\
11 nn ← nn + k_6 \\
12 return \langle a_{\min}, objFnVal_{\min} \rangle
\end{verbatim}

Figure 6.20: Invoking the optimizer to obtain an assignment.
using mesh adaptive direct search algorithms [AD06] to solve constrained optimization problems. It starts from a user-provided set of initial points. At each iteration, a neighbourhood function generates a set of points from the current point. Each point is tested for feasibility, discarded if it is not feasible, and then evaluated according to the QoS-dependent objective function. The point that gives the best value for the objective function is used as the current point for the next iteration. The optimizer stops searching when a better point cannot be found in the neighbourhood of the current point.

The NOMADm optimizer is used for the QoSA-SNEE where-scheduling step because it supports problems with categorical domains, i.e., it supports variables whose domain is limited to a finite set (as opposed to the set of integers or real numbers) [Kim98]. This is the case with the problem of populating in the assignment mapping, in which operator instances are mapped to elements in the set of routing tree sites. NOMADm is a black-box optimizer because it does not require an understanding of the function being optimized. Thus, the onus is on the user of the optimizer to provide effective functions for generating a set of initial points, generating a neighbourhood for a particular point, checking the feasibility of points in the neighbourhood, and for evaluating the desirability of points. The effectiveness of the optimizer is crucially reliant on these user-provided functions that enable it to traverse the search space to find a solution that is close to the global optimum (without being trapped in local optima).

As can be seen in Figure 6.20, for a given partial $D_Q$, the external optimizer may be invoked several times, using a different value for the $nn$ parameter, which dictates the maximum number of points that the neighbourhood function should generate at each iteration of the search. If $nn$ is too low, not enough of the search space may be covered to find the optimal point (or a point that is close enough). Conversely, having a $nn$ value that is too high may result in more of the search space being explored than is necessary, a potentially time-consuming issue, especially for large routing trees. In order to gauge the best value of $nn$ to use, QoSA-SNEE where-scheduling sets $nn$ at the initial value of $k_3$, and invokes the external optimizer. $nn$ is then incremented by $k_6$, and the external optimizer invoked again with the new value of $nn$. This is repeated until $k_4$ consecutive invocations of the external optimizer lead to values of the objective function within $k_5\%$ of each other, indicating that the solution is converging at what is perceived to be the global minimum.
IsFeasible($R_Q$, $D_Q$, assignment, Memory)

▷ Used by optimizer to check that assignment is valid for routing tree $R_Q$
▷ and DAF $D_Q$. Memory is used to estimate memory consumption.

▷ Every location-sensitive operator instance must be placed at the
▷ specified site.
1 for each $opInst \in D_Q$
2 do if $opInst$.LocationSensitive
3 then if assignment($opInst$) $\neq$ $opInst$.Site
4 then return false

▷ Memory cost of operator instances must not exceed site
▷ memory available.
5 for each $site \in R_Q$
6 do memAvail ← $site$.MemAvail
7 memCons ← 0
8 for each $opInst \in$ assignment$^{-1}$(site)
9 do memCons ← memCons +
10 Memory($opInst$)
11 if memAvail < memCons
12 then return false

▷ No circular flows of tuples allowed in the routing tree.
13 for each $opInst \in D_Q$
14 do site ← assignment($opInst$)
15 parentOpInst ← $opInst$.Parent
16 parentSite ← assignment($parentOpInst$)
17 if $R_Q$.IsDescendantOf($parentSite$, site)
18 then return false
19
20 return true

Figure 6.21: Checking whether a given assignment is feasible.
6.3. WHERE-SCHEDULING

**INITIALPOINT**(\(R_Q, D_Q\))

- Used by optimizer to generate an initial assignment for routing tree \(R_Q\) and
- DAF \(D_Q\) which is guaranteed to be feasible. The approach is to place
- all operator instances except location-sensitive ones at the sink site.

```
1 for each \(opInst \in D_Q\)
2    do if \(opInst.\text{LocationSensitive}\)
3        then assignment(opInst) ← opInst.Site
4    else assignment(opInst) ← \(R_Q.\text{sink}\)
5 return assignment
```

Figure 6.22: Generating a feasible initial point.

The external optimizer requires **QoS-SNEE** to provide a feasibility function
to check that certain constraints are met during the search, shown in Figure 6.21.
In essence, this function checks that the following constraints are met:

**C1** The *location sensitivity constraint*, i.e., that location-sensitive operator in-
stances are placed at the designated sites;

**C2** The *memory availability constraint*, i.e., that the memory cost of the operator
instances placed at a given site \(s\) does not exceed the available memory at
\(s\); and

**C3** The *confluence constraint*, i.e., that there are no circular flows of tuples in the
routing tree. In other words, tuples between all operator instances should
flow from the leaves to the sink node; it is not permissable for an instance
of \(op.\text{Parent}\), where \(op\) is assigned to site \(s\), to be situated on a node that is
a descendant in the routing tree of \(s\). This ensures that tuples never flow
from the root towards the leaves.

Figure 6.22 presents the **INITIALPOINT** function, used to generate an initial
point that is used at the start of the search. It is a requirement of **NOMADm**
for the initial point to be feasible. In order to guarantee that the initial point
is feasible, location sensitive operator instances are placed as required, and all
remaining operator instances are placed at the root node of the routing tree. In
this way, it is ensured that the location sensitivity and confluence constraints are
not violated. For the purpose of checking memory constraints, the memory at
the gateway is assumed to be infinite. This is a plausible assumption because
the gateway is a node directly connected to a computer that is order of magnitudes more powerful, and potentially, operator instances at the sink could be executed outside the sensor network, although this feature is not implemented in the current version of SNEE and is left as future work.

At each iteration in the search undertaken by the external optimizer, the Neighbourhood function (Figure 6.23) generates up to \( nn \) neighbouring points in the search space which correspond to alternative assignments. From the current point in the search space, a neighbouring point is generated by choosing a single operator instance that is not location sensitive, and assigning it to a *valid confluence site* at random. The valid confluence sites for an operator instance \( opInst \) are the ones on the path from the deepest confluence site of \( opInst \) to the gateway. The \texttt{MoveOpInst} function moves a given operator instance to a valid confluence site, and any other ascendant or descendant operator instances are also shifted, to ensure that confluence constraints are maintained. Section C.1 presents an illustrated example of the \texttt{MoveOpInst} function in operation.

When a neighbour is generated (and it has not previously been considered), \texttt{IsFeasible} is invoked to check that the point is valid. If this is the case, the QoS-dependent objective function is evaluated for the newly-generated neighbour. All the points in the neighborhood are evaluated in this manner, and if there is one with a better objective function value, it is used as the current point for the next iteration.

The QoS-dependent objective functions, used by \texttt{NOMADm} to evaluate each neighbour that is generated, are the same as the routing scoring functions described in Figure 6.8. The \texttt{OutCard} attribute can, however, be calculated more accurately. At the routing step, the placement of operators has not been determined, so the amount of data that will be transmitted between sites in the routing tree cannot be estimated with much accuracy. A candidate \texttt{DAF} has more information in this regard, as the assignment function has been populated. Therefore, the \texttt{OutCard} attribute is able to reflect a more accurate estimate of the number of tuples being transmitted at each site.

The intent behind the approach adopted in \texttt{QoSA-SNEE} where-scheduling is to induce the optimizer to exhibit appropriate operator instance assignment policies depending on the QoS objective function applied. As with the \texttt{QoSA-SNEE} routing step, the core algorithm employed is independent of the QoS expectations, and it is envisaged that an additional QoS variable could be supported by
Neighbourhood($R_Q, D_Q, assignment$)

\[\text{Used by optimizer to generate neighbours for given assignment pertaining to}
\]
\[\text{DAF $D_Q$ and routing tree $R_Q$.}\]

\[\text{for } i \leftarrow 1..\text{nn} \quad \text{nn is the number of neighbours to generate.}\]

\[\text{do } a_i \leftarrow \text{assignment}\]

\[\text{Select a random operator instance to move}\]

\[\text{opInst} \leftarrow \text{ChooseAnyOne}(D_Q.\text{OperatorInstances})\]

\[\text{if } \text{opInst}.\text{LocationSensitive}\]

\[\text{then continue} \quad \text{Proceed to next iteration of loop.}\]

\[s_{\text{old}} \leftarrow a_i(\text{opInst})\]

\[\text{Identify valid sites where opInst may be placed.}\]

\[s_{\Delta} \leftarrow D_Q.\text{DeepestConfluenceSite}(\text{opInst})\]

\[\text{sink} \leftarrow R_Q.\text{Sink}\]

\[\text{possSites} \leftarrow R_Q.\text{GetPath}(s_{\Delta}, \text{sink})\]

\[\text{Choose a site at random to move opInst to.}\]

\[s_{\text{new}} \leftarrow \text{ChooseAnyOne}(\text{possSites})\]

\[\text{if } s_{\text{old}}=s_{\text{new}}\]

\[\text{then continue} \quad \text{Proceed to next iteration of loop.}\]

\[a_i \leftarrow \text{MOVEOpInst}(R_Q, D_Q, a_i, \text{opInst}, s_{\text{old}}, s_{\text{new}})\]

\[\text{return } [a_1..a_{\text{nn}}]\]

\text{MOVEOpInst}(R_Q, D_Q, a, \text{opInst}, s_{\text{old}}, s_{\text{new}})

\[\text{Moves opInst from } s_{\text{old}} \text{ to } s_{\text{new}.} \text{ Any descendants or ascendants}\]

\[\text{of opInst are shifted to avoid any circular flows of}\]

\[\text{tuples in the routing tree.}\]

\[a(\text{opInst}) \leftarrow s_{\text{new}}\]

\[\text{path} \leftarrow R_Q.\text{GetPath}(s_{\text{old}}, s_{\text{new}})\]

\[\text{if } R_Q.\text{IsDescendantOf}(s_{\text{new}}, s_{\text{old}})\]

\[\text{then for each } \text{opInst'} \in D_Q.\text{Descendants}(\text{opInst})\]

\[\text{do if } a(\text{opInst'}) \in \text{path}\]

\[\text{then } a(\text{opInst'}) \leftarrow s_{\text{new}}\]

\[\text{else for each } \text{opInst'} \in D_Q.\text{Ascendants}(\text{opInst})\]

\[\text{do if } a(\text{opInst'}) \in \text{path}\]

\[\text{then } a(\text{opInst'}) \leftarrow s_{\text{new}}\]

\[\text{return } a_i\]

Figure 6.23: Generating neighbours of the current assignment.
CHAPTER 6. A QOS-AWARE INSTANTIATION OF SNEE

specifying an additional objective function and/or constraints. Also, given that the optimizer used performs a randomized search of the solution space, various options in the search space are considered by moving random operator instances around the routing tree, to generate and assess diverse alternatives. This may also prove particularly useful in heterogeneous network settings, where sites have different resources. In contrast, FG-SNEE where-scheduling is a heuristic-based algorithm, in which fragments are assigned in a post-order fashion either to the deepest confluence routing tree site(s), or the sink, depending on whether they are data reducing or not. This gives rise to deterministic behaviour, that results in a single solution, in which the resource differences at individual sites are not taken into account.

The post-processing phase of QoSA-SNEE where-scheduling is responsible for, firstly, removing redundant operator instances, and subsequently, inserting the EXCHANGE constituents (consumer, producer and relay) between operator instances in the DAF where radio communication needs to take place, resulting in the demarcation of fragments in the QEP. More details together with an illustrated example are given in Section C.2.

6.3.2 Example output

The DAF generated and deemed optimal by QoSA-SNEE where-scheduling for Q3 with \(\text{optGoal} = \min \delta\) is shown in Figure 6.24 and the one with \(\text{optGoal} = \max \lambda\) is shown in Figure 6.25. The DAF in Figure 6.24 is considered to be favourable for delivery time because, firstly, the underlying routing tree is made up of fewer edges than, e.g., the one shown in Figure 6.25 which is based on a routing tree that is intended to spread the workload evenly. Secondly, instances of AGGR_INIT (which increases the size of a tuple, as it adds an extra attribute) and AGGR_MERGE (which reduces several tuples into a single tuple) are co-located, consequently reducing the amount of data transmitted over the radio. Both these factors lead to an (albeit slight) decrease in the delivery time. There is no scope to reduce data over the radio for the \(\max \lambda\) DAF in Figure 6.25 due to the confluence of tuples determined by the shape of the routing tree, which results in no AGGR_MERGE instances being created. However, this would also be done if the shape of the routing tree made this possible, as reducing the amount of data to be transmitted over the radio is also beneficial for the \(\max \lambda\) optimization goal.
Figure 6.24: Where-scheduling optimal candidate for \( Q3 \) optimized for \( \min \ \delta \).

It is noted that the shape of the routing tree, which is the fundamental difference between the two DAFs shown, is not a decision that is made during the where-scheduling step. The decisions relating to placing operators in the PAF made during where-scheduling, aim to reduce data being transmitted over the radio whether the QoS variable is time- or energy-related, as in both cases doing so leads to a more favourable outcome. For example, reducing the amount of data transmitted over the radio means that the tasks in the agenda can be executed in a shorter time, as processing time is traded for radio time, meaning that a higher throughput is possible (making a shorter \( \alpha \) possible) and a shorter \( \delta \) can be achieved. It also saves energy, leading to improvements in \( \epsilon \) and \( \lambda \). It is noted that further testing, involving the use of diverse queries, network topologies and QoS expectations, revealed that for a constant routing tree and PAF, the same DAT is considered by QoSA-SNEE to be the optimal one regardless of the optimization goal considered. This indicates that the different QoS-dependent objective functions (which have the same form as the ones used in the routing step, which does lead to different routing trees depending on the QoS variable in question) all induce the external optimizer to exhibit the same behaviour.
Figure 6.25: Where-scheduling optimal candidate for \( Q3 \) optimized for \( \max \lambda \).
6.3. WHERE-SCHEDULING

In other words, with QoSA-SNEE where-scheduling, all QoS optimization goals favour the same decision-making policies.

6.3.3 Discussion

This section has presented the QoSA-SNEE where-scheduling step. A QoS-dependent utility function, similar to the QoSA-SNEE routing one, is used with the intent of modifying the step’s decision-making policies depending on the user QoS expectations. The routing step differs however, in that it generates several alternative solutions and then uses the utility function to select a subset of them. In contrast, this step uses the utility function to steer a randomized search (delegated to an external optimizer) to find the solution most suited to a particular QoS expectation. As with QoSA-SNEE routing, the resulting algorithm is generalized and independent of the QoS expectations, and the utility functions essentially provide an extensibility point for alternative decision-making strategies.

As previously mentioned, QoSA-SNEE where-scheduling was found not to be sensitive to different QoS variables, given that for a given PAF and RT, a constant DAF is returned irrespective of the QoS expectations. This is because assigning computations so that the amount of data to be transmitted via radio is reduced is desirable for all the QoS variables considered. The randomized search approach employed by QoSA-SNEE where-scheduling differs significantly from the approach used in FG-SNEE where-scheduling, which applies a deterministic heuristic algorithm that aims to reduce the amount of data transmitted between sites in the QEP. This is because QoSA-SNEE where-scheduling allows a more diverse solution space to be considered. Also, FG-SNEE defines fragments first, and then assigns them to sites, as is done in traditional DQP. In the sensor network setting, defining an operator instance tree based on the spatially distributed nature of schema data sources enables reasoning to take place about confluence between operator instances, thus giving more fine-grained control to the optimizer to explore alternative solutions. Thus, particularly in heterogeneous networks comprising nodes with different resources, QoSA-SNEE where-scheduling generates DAFs with a greater degree of in-network processing than those produced by FG-SNEE, in which fragments are more likely to be delegated to the gateway node, leading to higher radio communication costs.

With regards to related work, it is noted that none of the research discussed in Section 3.2 directly considers QoS expectations when making where-scheduling


6.4 When-Scheduling

The last decision-making step in the multi-site phase of the SNEE query processing stack is when-scheduling, which is responsible for timing the DAF operator instances assigned to each routing tree site. This involves selecting suitable values for the acquisition interval and buffering factors used to construct the agenda, based on the optimization goal and constraints specified in the QoS expectations. This results in the generation of an agenda for each DAF in $D_Q$, giving rise to a collection of agendas $A_Q$. From this collection, a single agenda is selected as the final QEP for which executable code is generated.

Timing decisions have a significant impact on QoS variables on a resource-constrained platform like a sensor network where energy is severely limited. Certain applications require a high frequency of acquisition, or results to be delivered without delay, both examples of requirements that, intuitively, would lead QoSA-SNEE to generate energy intensive QEPs. In other cases, network lifetime is paramount, and QEPs should be generated in which acquisition and radio activity take place less frequently, to reduce the rate of energy depletion.

The approach taken in QoSA-SNEE when-scheduling is to determine suitable values (in respect of the QoS expectations) for the acquisition interval $\alpha$ and the buffering factor $\beta$. Subsequently, these values are passed as parameters to the agenda construction algorithm (which is also used in FG-SNEE and was presented in Figure 5.10). In order to derive an $\alpha$ and $\beta$, the when-scheduling problem is modelled as a constrained optimization problem, with a QoS-dependent objective function and constraints in terms of these two variables. An external optimizer then solves the problem and returns appropriate values for $\alpha$ and $\beta$. The algorithm is detailed in Section 6.4.1, the agendas produced by different optimization goals are shown in Section 6.4.2, and a discussion ensues in Section 6.4.2.

6.4.1 Algorithm Description

Figure 6.26 presents the QoSA-SNEE when-scheduling algorithm, which aims to time the QEP tasks in the most appropriate manner to meet the QoS expectations. In the algorithm, for each each $D_Q$ in the input DAF collection, values for the acquisition interval $\alpha$ and the buffering factor $\beta$ (the two parameters for
When-Scheduling($D_Q$, QoS, Time, Energy, Memory)

▷ Generate agenda collection $A_Q$ for each DAF in $D_Q$ that satisfies the QoS expectations QoS, Time, Energy and Memory are cost estimation models.

▷ Populate $A_Q$, a collection of candidate agendas stored in order of descending score.

1 $A_Q = [ ]$
2 for each $D_Q \in D_Q$
3 do ▷ Generate constrained optimization problem $P$
4 $P \leftarrow \text{Generate-Problem}(D_Q, D_Q.R_Q, QoS, Time, Energy, Memory)$
5 ▷ Obtain $\alpha$ and $\beta$, QoS-dependent objective function value and status
6 $\langle \alpha, \beta, f, status \rangle \leftarrow \text{Get-Alpha-Beta}(P)$
7 if $status = \text{error}$
8 then continue ▷ Proceed to next iteration of loop.
9 ▷ Construct agenda (See Figure 5.10).
10 $A_Q \leftarrow \text{Build-Agenda}(D_Q, D_Q.R_Q, \alpha, \beta, Time)$
11 $A_Q.\text{Add}(A_Q, f^{-1})$
12 return $A_Q.\text{GetBest}()$

Figure 6.26: The QoSA-SNEE When-Scheduling algorithm.
generating an agenda) are obtained by formulating a constrained optimization problem which is delegated to an external optimizer. This step aims to obtain values for $\alpha$ and $\beta$ that will lead to an agenda that, first and foremost, satisfies the constraints specified by the user QoS, and also, as much as possible, reflects the QoS optimization goal. In cases in which the constrained optimization problem is solvable, an agenda $A_Q$ is created using $\alpha$ and $\beta$, and added to the output agenda collection $A_Q$. Finally, the agenda with the best value for the QoS-dependent optimization function is selected as the final QEP which is most appropriate for the user-specified QoS expectations.

The central challenge in QoSA-SNEE when-scheduling is that of modelling the when-scheduling problem effectively. This is done by deriving, from the DAF, mathematical expressions for each of the QoS variables in terms of $\alpha$, $\beta$ and the time and energy CEMs. Thus, expressions are generated that comprise the constrained optimization problem for the optimization goal in the QoS (which corresponds to the objective function of the problem), any constraints specified in the QoS, and other constraints that reflect system properties, i.e., they reflect the behaviour of SNEE agendas. It is found that the set of expressions derived constitute a constrained optimization problem with numerical variables that can be mapped to a geometric program \cite{BKVH07}, a type of problem that can be handled by the cvx optimizer \cite{GB09}. More information about geometric programs and cvx is given in Appendix D. This appendix also contains details or worked examples showing how expressions are derived for the constrained optimization problem and adapted for cvx, and are referenced throughout this section as appropriate.

Before describing how the when-scheduling problem is cast as a constrained optimization problem, a clarification is made about the information provided by the CEMs in this step. When modelling the when-scheduling problem, the expressions generated for the various QoS variables and system properties are themselves constituted by sub-expressions representing the time, energy and memory consumption of constituents of the DAF. For each operator (and fragment) in the DAF, there is a fixed cost $F$, associated with the invocation of the operator itself, and a variable cost $V$, which is proportional to $\beta$. For example, for an EXCHANGE, the fixed cost may be due to switching on the radio, and the variable cost reflects the per-epoch cost of sending the tuples. It is noted that these costs are a function of the buffering factor $\beta$, a variable whose value has not been
decided at the time of modelling the when-scheduling problem, so the CEMs are unable to generate a constant value to represent the cost. QoSA-SNEE where-scheduling therefore uses a version of the CEMs that emit expressions in which the fixed and variable elements of the cost models are decomposed, in the form $F + V\beta$. These are denoted throughout as $\text{Memory}_\beta()$, $\text{Time}_\beta()$ and $\text{Energy}_\beta()$.

### Modelling system properties in the when-scheduling problem

The values of $\alpha$ and $\beta$ obtained are required to result in an agenda that is valid, i.e., one that conforms to the properties discussed in Section 4.6.4. In order to ensure that this is the case, a number of system properties are defined that act as constraints for the optimization problem that is generated. These system properties are expressed as conditions that are derived from the DAF, i.e., the part of the QEP that has been decided up to this point in the query processing stack. The constraints arising from these system properties are:

- **C1** The value of $\beta$ must not be such that the QEP exceeds the memory available at each site.

- **C2** The acquisition interval must be longer than $\pi$, the time taken to carry out the computations and communications in the last epoch of an agenda evaluation\(^4\).

- **C3** The acquisition interval must be a non-zero positive integer.

- **C4** The buffering factor must be a non-zero positive integer.

If any of these constraints are not met for a given assignment of $\alpha$ and $\beta$, the assignment is an infeasible solution. It is possible, for some DAFs and QoS expectations, that no feasible solutions can be found. If no feasible solution is found for any DAF, it would require the user to relax the QoS expectations or else improve the resources in hand with which to meet them.

In order for system constraints C1–C4 to be taken into account by the external optimizer, mathematical expressions that model these constraints need to be derived, and added to the constrained optimization problem being generated. 

\(^4\)This is required to create a valid SNEE agenda with the properties described in Section 4.6.4, e.g., recall that the inequality $\alpha(\beta - 1) + \pi < \alpha\beta$ needs to be satisfied.
operator instances assigned to it in the DAF do not exceed the memory available at that site, i.e.,
\[ m_i \leq M_i, \ i \in D_Q \text{-} \text{sites} \] (6.1)

where
\[ m_i = \sum_{\text{fragInst} \in \text{i.fragInstances}} \text{Memory}_\beta(\text{fragInst}) + \sum_{\text{exchOpInst} \in \text{i.exchOpInstances}} \text{Memory}_\beta(\text{exchOpInst}) \] (6.2)

and \( M_i \) represents the memory available at site \( i \). A detailed example showing how the expressions are derived is presented in Section D.2.1 of the appendices.

Constraint \( C_2 \) is expressed by adding a single inequality for the DAF to represent the constraint that \( \pi \leq \alpha \), explained in detail in Section 4.6.4. An expression for \( \pi \) is approximated by adding the time cost of each task that is executed in the final epoch of an agenda execution episode. The value for \( \pi \) has two components, the time for the tasks corresponding to the leaf fragments in the DAF, and the non-leaf ones, i.e.,
\[ \pi = \pi^L + \pi^N \] (6.3)

The assumption is made that leaf fragments of the DAF execute concurrently, whereas non-leaf fragments in the DAF execute sequentially. This approach may result in a slight over-approximation of \( \pi \), but this avoids having to construct an agenda to compute \( \pi \) (as done in FG-SNEE when-scheduling). Details about how \( \pi^L \) and \( \pi^N \) are derived are presented in Section D.2.2 together with a worked example.

The \( C_3 \) and \( C_4 \) constraints are used to ensure that the optimization variables \( \alpha \) and \( \beta \) are greater than or equal to one. This can be done simply by adding the constraints \( \alpha \geq 1 \) and \( \beta \geq 1 \) to the constrained optimization problem.

**Modelling QoS variables in the when-scheduling problem**

The system properties described in the previous section (i.e., \( C_1 \text{--} C_4 \)) are directly derived from the DAF and routing tree, and are independent of the QoS expectations specified by the user. Now, the focus turns to how to incorporate
expressions for optimization goals and constraints from the QoS expectations into
a constrained optimization problem for delegation to an external optimizer. This
is achieved by deriving expressions for each QoS variable ($\alpha, \delta, \epsilon$ and $\lambda$, defined
in Table 3.1). For acquisition interval $\alpha$, the mapping is straightforward, as it
can be made directly to $\alpha$. Delivery time $\delta$ is approximated as $\alpha \beta$, which is the
length of a single agenda evaluation.\footnote{Recall from Section 4.6.4 that it is a property of the way that agendas are constructed that
tuples are delivered after $\beta$ sensor acquisitions, each of which is $\alpha$ time units apart.}

In order to be able to derive expressions for total network energy and lifetime
QoS variables, it is first necessary to define the expression for computing $\hat{e}_i$, the energy consumption at a site $i$ during an agenda evaluation episode, which
corresponds to

$$
\hat{e}_i = \sum_{\text{fragInst} \in i.\text{fragInstances}} \text{Energy}_\beta(\text{fragInst}) + \sum_{\text{exchOpInst} \in i.\text{exchOpInstances}} \text{Energy}_\beta(\text{exchOpInst})
$$

(6.4)

This ignores the energy consumed while sensor nodes are sleeping, which is con-
sidered to be negligible compared to the energy consumed by nodes during radio
communications, sensing or processing. The energy consumption for a site per
unit of time (as opposed to the whole duration of an agenda evaluation episode)
is given by

$$
e_i = \frac{\hat{e}_i}{\alpha \beta}
$$

(6.5)

Total network energy $\epsilon_t$ is computed by summing the energy consumption values
associated with each site in the DAF for a period of time $t$, i.e.,

$$
\epsilon_t = t \sum_{i \in \text{DAF.sites}} e_i
$$

(6.6)

Section D.2.3 presents an example of how the total energy variable is derived.

The network lifetime $\lambda$ is calculated by finding the length of time that it takes
for the first site to be depleted of energy. The lifetime for site $i$, denoted $\lambda_i$, is
calculated by dividing the energy stock $E_i$ by the energy consumption per time
### Optimization goal:

Minimize $\alpha \beta$

Delivery time

### QoS Constraints:

- $\alpha \geq 10000$
  - Acquisition interval
- $\alpha \leq 20000$
  - Acquisition interval
- $0.67\alpha^{-1} + 2.22\alpha^{-1}\beta^{-1} \leq 0.05$
  - Total energy from (D.16)

### System properties:

- $57\beta + 175 \leq \alpha$
  - Processing time $\pi$ from (6.13)
- $48 + 32\beta \leq 4096$
  - Memory check for site 22 from (D.6)
- $85 + 32\beta \leq 4096$
  - Memory check for site 0

### Consistency Conditions:

- $\alpha \geq 1$
  - Acquisition interval $\geq 1$
- $\beta \geq 1$
  - Buffering factor $\geq 1$
- $\alpha \in \mathbb{Z}$
  - Variable is an integer
- $\beta \in \mathbb{Z}$
  - Variable is an integer

### Figure 6.27: An example when-scheduling problem.

unit $e_i$:

$$\lambda_i = \frac{E_i}{e_i} \quad (6.7)$$

Subsequently, the time taken for the first site to die is given by the expression generated by:

$$\lambda = \text{Min}_{i \in Q, \text{sites}}(\lambda_i) \quad (6.8)$$

In which $\text{Min}$ is a function that returns the lowest value in a collection, i.e., in this case the lifetime of the site that is first to die in the QEP. Section D.2.3 explains certain workarounds that are necessary to adapt lifetime expressions to the external optimizer, and gives a concrete example of how this is done.

### Example When-scheduling Problems

Following the descriptions on how expressions are derived to enforce system properties and the QoS expectations, this section presents examples of how these are combined to form a constrained optimization problem. All the examples in this section are based on the same DAF (the one in Figure 6.24 for which expressions have been derived in Section D.2) so the system properties are constant. The expressions for the optimization goal and QoS constraints reflect the QoS expectations in each case.

Figure 6.27 presents the when-scheduling problem generated for the QoS
6.4. WHEN-SCHEDULING

<table>
<thead>
<tr>
<th>Item</th>
<th>Expression</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization</td>
<td>Maximize ( \text{Min} \left( \frac{3132000}{0.1} \alpha + \frac{3132000}{0.53} \alpha \beta, \ldots, \right) )</td>
<td>Lifetime adapted from ( \text{D.20} )</td>
</tr>
<tr>
<td>goal:</td>
<td>( \frac{3132000}{0.18} \alpha + \frac{3132000}{0.64} \alpha \beta )</td>
<td></td>
</tr>
<tr>
<td>QoS Constraints:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>System properties:</td>
<td>( 57 \beta + 175 \leq \alpha )</td>
<td>Processing time ( \pi ) from ( \text{D.12} )</td>
</tr>
<tr>
<td></td>
<td>( 48 + 32 \beta \leq 4096 )</td>
<td>Memory check for site 22 from ( \text{D.3} )</td>
</tr>
<tr>
<td></td>
<td>\ldots</td>
<td></td>
</tr>
<tr>
<td></td>
<td>( 85 + 32 \beta \leq 4096 )</td>
<td>Memory check for site 0</td>
</tr>
<tr>
<td>Consistency Conditions:</td>
<td>( \alpha \geq 1 )</td>
<td>Acquisition interval ( \geq 1 )</td>
</tr>
<tr>
<td></td>
<td>( \beta \geq 1 )</td>
<td>Buffering factor ( \geq 1 )</td>
</tr>
<tr>
<td></td>
<td>( \alpha \in \mathbb{Z} )</td>
<td>Variable is an integer</td>
</tr>
<tr>
<td></td>
<td>( \beta \in \mathbb{Z} )</td>
<td>Variable is an integer</td>
</tr>
</tbody>
</table>

Figure 6.28: An example unbounded when-scheduling problem.

The expressions under Optimization goal and QoS constraints are derived directly from the QoS expectations and the DAF. System properties are derived from the DAF and are implementation-dependent, in the sense that these expressions are a consequence of the algorithm used for generating an agenda, and/or of the sensor network fabric that the QEP executes over (but not the QoS expectations). Consistency Conditions are implementation-independent, and are the same regardless of the DAFs or QoS expectations. The third column of each when-scheduling problem provides explanatory comments, and if the derivation of an expression is given in Appendix D.2.3, the corresponding equation is referenced.

Figure 6.28 shows the derived constrained optimization problem for the QoS expectation \( \langle \min \delta, \{10s \leq \alpha \leq 20s, \epsilon \leq 50\text{mJ}\} \rangle \). The expressions under Optimization goal and QoS constraints are derived directly from the QoS expectations and the DAF. System properties are derived from the DAF, and are implementation-dependent, in the sense that these expressions are a consequence of the algorithm used for generating an agenda, and/or of the sensor network fabric that the QEP executes over (but not the QoS expectations). Consistency Conditions are implementation-independent, and are the same regardless of the DAFs or QoS expectations. The third column of each when-scheduling problem provides explanatory comments, and if the derivation of an expression is given in Appendix D.2.3, the corresponding equation is referenced.

Figure 6.28 shows the derived constrained optimization problem for the QoS expectation \( \langle \max \lambda, \emptyset \rangle \). Intuitively, in order to maximize lifetime, one would expect that the QEP should acquire data as infrequently as possible, and maximize buffering so that the radio is switched on less often. For \( \beta \), the upper-bound depends on the memory available on the nodes, and there are constraints in the constrained optimization problem to reflect this. For the acquisition interval, while there is a lower bound defined, no upper bound is specified, meaning that \( \alpha \) could be arbitrarily high. In this case, the external optimizer would identify that this problem is unbounded and is therefore unable to provide a solution. It is therefore necessary, in the QoS expectations supported by SNEE, to provide an upper-bound for \( \alpha \), either directly (e.g., by adding a constraint of the
CHAPTER 6. A QOS-AWARE INSTANTIATION OF SNEE

<table>
<thead>
<tr>
<th>Item</th>
<th>Expression</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization goal:</td>
<td>Minimize $\alpha \beta$</td>
<td></td>
</tr>
<tr>
<td>QoS Constraints:</td>
<td>$\alpha = 20000$</td>
<td>Acquisition interval</td>
</tr>
<tr>
<td></td>
<td>$\alpha \beta \leq 30000$</td>
<td>Delivery time</td>
</tr>
<tr>
<td></td>
<td>$\frac{31320000}{0.1} \alpha + \frac{31320000}{0.53} \alpha \beta \geq 5$ years</td>
<td>Site 12 lifetime (D.20)</td>
</tr>
<tr>
<td></td>
<td>$\frac{31320000}{0.18} \alpha + \frac{31320000}{0.64} \alpha \beta \geq 5$ years</td>
<td>Site 22 lifetime (D.20)</td>
</tr>
<tr>
<td>System properties:</td>
<td>$57 \beta + 175 \leq \alpha$</td>
<td>Processing time $\pi$ from (D.13)</td>
</tr>
<tr>
<td></td>
<td>$48 + 32 \beta \leq 4096$</td>
<td>Memory check for site 22 from (D.6)</td>
</tr>
<tr>
<td></td>
<td>$85 + 32 \beta \leq 4096$</td>
<td>Memory check for site 0</td>
</tr>
<tr>
<td>Consistency Conditions:</td>
<td>$\alpha \geq 1$</td>
<td>Acquisition interval $\geq 1$</td>
</tr>
<tr>
<td></td>
<td>$\beta \geq 1$</td>
<td>Buffering factor $\geq 1$</td>
</tr>
<tr>
<td></td>
<td>$\alpha \in \mathbb{Z}$</td>
<td>Variable is an integer</td>
</tr>
<tr>
<td></td>
<td>$\beta \in \mathbb{Z}$</td>
<td>Variable is an integer</td>
</tr>
</tbody>
</table>

Figure 6.29: An example infeasible when-scheduling problem.

form $\alpha \leq k$), or indirectly (e.g., in terms of the delivery time, i.e., by adding a constraint of the form $\alpha \beta \leq k$).

The when-scheduling problem in Figure 6.29 presents an example of a constrained optimization problem with QoS expectations that are too demanding, i.e., $\langle \min \delta, \{\alpha = 20s, \delta \leq 30s, \lambda \geq 5 \text{ years}\} \rangle$. An acquisition interval of 20 seconds means that the sensor nodes cannot spend much time in sleep mode, and delivery time less than 30 seconds leaves no scope for buffering. This means that energy is too rapidly depleted, and a lifetime of 5 years is not possible with the amount of energy available. The external optimizer is therefore unable to find values of $\alpha$ and $\beta$ that satisfy the QoS expectations, and returns an infeasibility marker. Note that, when a QoS lifetime lower bound is provided, it is necessary that all the sites in the QEP live for at least the lifetime requested. This requirement is expressed by adding a constraint per site to the constrained optimization problem.

6.4.2 Example Output

The agendas for query $Q3$ when it is compiled for different optimization goals and an upper-bound constraint on $\alpha$ of 1 minute can be observed in Figure 6.30.
6.4. WHEN-SCHEDULING

(a) min $\delta$ time agenda; $\alpha = 1$ min, $\beta = 1$, $\delta = 253$ ms.

(b) min $\epsilon$ agenda; $\alpha = 1$ min, $\beta = 46$, $\delta = 45$ min 4.983 s.

(c) max $\lambda$ agenda; $\alpha = 1$ min, $\beta = 46$, $\delta = 45$ min 8.241 s.

Figure 6.30: Alternative agendas for $Q3$. 
The min $\alpha$ agenda is not shown; it is however the same as the min $\delta$ agenda, with the exception that $\alpha = \delta = 253$ ms, i.e., the processing time $\pi$, the lowest possible value. The agenda for max $\lambda$ has more columns than the others, because the routing tree selected for this optimization goal (Figure 6.10(a)) has many more sites in it than the routing tree selected for the other optimization goals, as it aims to consume energy as evenly as possible throughout the network, and it is preferable to have more sites participating in the QEP in order to avoid hotspots. In contrast, the agendas for min $\delta$ and min $\epsilon$ have fewer columns, as their routing tree (shown in Figure 6.10(b) in both cases) involves fewer sites. Recall that having fewer sites is favoured for minimizing delivery time because it leads to fewer radio hops in the routing tree, and also for reducing total network energy, in which it is beneficial to have as few sites as possible.

The agendas for the min $\alpha$ and min $\delta$ optimization goals also have a comparatively small number of rows, as $\beta = 1$, i.e., minimal buffering takes place, to enable the processing time $\pi$ and/or the delivery time $\delta$ to be as low as possible. For min $\epsilon$ and max $\lambda$, the agendas have many more rows as the highest possible value of $\beta$ is selected (insofar as permitted by the available memory). Not all the rows are shown due to space limitations; in cases where agenda rows have been omitted this is indicated using ellipses.

With regards to the $\alpha$ value, the optimization goals min $\epsilon$ and max $\lambda$ aim to make this variable as high as possible (i.e., 1 minute), in order to conserve energy. When $\beta = 1$, as is the case with the min $\delta$ optimization goal, the $\alpha$ value does not affect the delivery time achievable, because delivery time $\delta = \alpha(\beta - 1) + \pi$. If $\beta$ is set to 1, a reasonable assumption because increasing the buffering factor will significantly increase the delivery time, varying $\alpha$ will not have any effect on the $\delta$. Note that the agenda in Figure 6.30(c) has one fragment less than the other agendas because no instances of the iterative AGGR_MERGE operator have been created during where-scheduling.

Based on these observations, general characteristics can be associated with agendas generated by QoSA-SNEE, depending on the QoS goal that they optimize for. An agenda is said to be ‘short’ if it has few rows, or ‘tall’ if it has many rows. Following the same analogy, a ‘narrow’ agenda has fewer columns than a ‘wide’ one, in which many sites comprise the routing tree. Thus, the tendency when only a minority of sites are sources for a particular query (as is the case with $Q3$) appears to be as follows: For the min $\alpha$ and min $\delta$ optimization goals, agendas
that are ‘short and narrow’ are favoured. The min $\epsilon$ results in agendas that are ‘tall and narrow’. For the max $\lambda$ optimization goal, QoSA-SNEE generates agendas that are ‘tall and wide’. Note that if the majority of sites are sources for a query, during the routing step the optimizer has less leeway with respect to the choices it makes about which sites to include in the routing tree. The decisions made during the when-scheduling step only affect whether an agenda is ‘tall’ or ‘short’.

6.4.3 Discussion

As can be seen from the example when-scheduling problems in Section 6.4.1, a variety of user requirements involving an optimization goal and constraints on QoS variables may be expressed by means of a constrained optimization problem. Indeed, there is a close resemblance between the forms of the constrained optimization problems shown and of SNEE QoS expectations (described in Section 4.3), in that they both comprise an objective function and a collection of associated constraints. This approach therefore, in principle, supports a broad range of QoS expectations to be expressed for a query, and is empowering to the user.

A key advantage of this approach is that, in a constrained optimization problem, an arbitrary number of variables can be fixed beforehand, or left as unknowns (as long as none of the variables are left unbounded, as in the example in Figure 6.28). Thus, the variables for which a solution needs to be found are not predetermined beforehand, i.e., either only $\alpha$ may be fixed, only $\beta$ may be fixed, or the optimizer may have free rein to set both. This makes this technique general enough to handle diverse QoS expectations, so long as the optimization goal and constraints can be translated into expressions in terms of $\alpha$ and $\beta$ that can be handled by the external optimizer. Thus, this approach is not limited to the four types of QoS variables considered in this QoSA-SNEE; potentially additional types of QoS variables could be supported.

This is in stark contrast to FG-SNEE when-scheduling, which only accepts a fixed $\alpha$ and is therefore unable to vary the acquisition interval. FG-SNEE is only capable of adjusting $\beta$ by performing what constitutes a binary search on the value of $\beta$. Its sole heuristic is essentially to maximize $\beta$, taking into account the available memory on each site. An upper bound on delivery time may, optionally, be specified, in which case $\beta$ is reduced to create an agenda to meet the delivery
time. The algorithm is unable to support diverse QoS expectations as it lacks the versatility inherent in the QoSA-SNEE when-scheduling algorithm. Another advantage of the approach in QoSA-SNEE when-scheduling is that it can model the properties of the agenda, unlike FG-SNEE which constructs an agenda for every value of $\beta$ considered in the search it undertakes, which is potentially time-consuming.

With regards to the degree of QoS-awareness exhibited by TinyDB [MFHH05], given that by default it transmits tuples up the routing tree in the same epoch as they are acquired, it is unable to trade off delivery time against lifetime or total energy like SNEE does. This predetermined behaviour effectively means that $\beta = 1$ for TinyDB queries, meaning that its performance in terms of energy-related QoS is likely to be worse than FG-SNEE, as suggested by the experiments in Section 7.1. In TinyDB, another type of QoS expectation that may be specified is the desired lifetime for a query, which in turn results in the acquisition interval being adjusted at query evaluation time (subject to an upper bound that may be optionally specified by the user) to attempt to meet that lifetime. This decision is made in light of the energy stock remaining on a node and an energy consumption cost model used to predict the energy consumption of the query. Compared to QoSA-SNEE, which has considerable flexibility to take into account diverse QoS optimization goals and constraints, TinyDB has a very limited degree of QoS-awareness in its when-scheduling decisions.

With SNQL [BLM+07], buffering between epochs is possible, but the user needs to specify a send time parameter explicitly, rather than giving a declarative delivery time constraint. The estimated energy consumption is monitored as the query is evaluated, which leads to the initial delivery time increasing in an unbounded manner. Therefore, while a lower bound for $\delta$ may be specified in SNQL, it is not possible to specify an upper-bound, and arguably the latter would be more useful and more challenging (given that preserving energy and decreasing delivery time are conflicting goals, whereas preserving energy and increasing delivery time are not). The acquisition interval is adjusted by the optimizer if there is a shortage of memory, but (unlike QoSA-SNEE) not in order to reduce energy consumption. SNQL also, therefore, like TinyDB, exhibits limited QoS awareness when making when-scheduling decisions.
6.5 Conclusion

In this chapter, QoSA-SNEE, a concrete instantiation of the SNEE query processing stack template has been presented. Algorithms that exhibit alternative decision-making policies, depending on the QoS expectations, have been described for the multi-site steps in the SNEE query processing stack. At each step, a mechanism is employed for customizing extensibility points to make alternative query planning decisions that favour diverse QoS variable(s), viz., scoring functions for routing and where-scheduling, and a generated constrained optimization problem for when-scheduling.

As mentioned throughout the chapter, there is little related work in the SNQP area in which decisions are made about different QoS variables, viz., routing decisions in Wave Scheduling [TYD+07] trade off delivery time and total energy consumption, when-scheduling decisions in TinyDB [MFHH05] trade off lifetime and acquisition interval, and in SNQL [BLM+07], when-scheduling decisions in SNQL increase delivery time to prolong lifetime. However, there is no related work that provides a general framework for supporting different types of QoS expectations (i.e., optimization goals and constraints) for diverse QoS variables.

In the next chapter, the benefits of QoS-awareness in SNQPs is evaluated, by comparing the QoSA-SNEE instantiations with a suitable representative of the state-of-the-art among fixed-goal SNQPs by means of systematic experimentation.
Chapter 7

Evaluation

This chapter provides experimental evidence in support of the underlying thesis presented in this dissertation, i.e., that being QoS-aware enables a SNQP to generate QEPs suitable for a broader range of applications than optimizing for a fixed or implicit optimization goal. Recall that, by QoS-awareness, is meant that the decision-making policies that result in the generation of the QEP reflect potentially diverse QoS expectations that may be associated with a query. Therefore, it is postulated that, a QoS-aware SNQP generates QEPs that exhibit performance characteristics that are consistent with the stated QoS expectations. In contrast, a SNQP that is not QoS-aware (or only has a limited degree of QoS awareness) is less able to tailor QEPs for specific QoS expectations, as query planning decisions are made based on wholly or partially assumed QoS expectations.

In order to demonstrate the usefulness of QoS-awareness in SNQPs, it is necessary to ascertain that a SNQP that purports to be QoS-aware (i.e., QoSA-SNEE) indeed responds to diverse QoS expectations, by generating QEPs that reflect a variety of optimization goals and (possibly conflicting) constraints. Once it has been established that QoSA-SNEE is QoS-aware, in order to evaluate the benefits of QoS awareness, a contrast needs to be drawn between generating QEPs in a QoS-aware manner, as opposed to optimizing according to a fixed-goal or implicit QoS expectation. In every case, it is to be expected that, all other things being equal, a QoS-aware SNQP with optimization goal $\text{opt} \ \theta$ will achieve a performance no worse, in terms of the QoS variable $\theta$, than a suitable candidate to represent a state-of-the-art SNQP that specializes in the optimization goal $\theta$.

By a suitable candidate to represent state-of-the-art SNQPs for a particular
7.1. IDENTIFYING A BASELINE: FIXED-GOAL SNQP EVALUATION

optimization goal opt \( \theta \), is meant one that outperforms other SNQPs that are specialized for the same (or similar) optimization goal. Existing fixed-goal SNQPs, viz., TinyDB, Cougar or FG-SNEE, all broadly aim to conserve energy or maximize network lifetime, so in that respect, they roughly have the same concerns in terms of implicit QoS expectations. Therefore, in order to evaluate QoSA-SNEE, the fixed-goal SNQP with the best performance in terms of energy-related QoS variables will be used as a baseline for comparison.

This chapter is structured as follows: Section 7.1 addresses the question of selecting a fixed-goal optimizer to act as a baseline for comparison with QoSA-SNEE, by considering performance and other practical considerations. Section 7.2 then evaluates QoSA-SNEE, with the aim of demonstrating its flexibility over the fixed-goal SNQP used as a baseline. A discussion follows in Section 7.3 summarizing the results, and contrasts experimental evidence provided for other SNQPs in the literature, with the aim of highlighting the novelty of the results obtained. Finally, Section 7.4 concludes the chapter.

7.1 Identifying a Baseline: Fixed-Goal SNQP Evaluation

This section aims to identify a suitable candidate to represent a state-of-the-art fixed-goal SNQP to be used as a baseline for comparing the performance of a QoS-aware SNQP over one that generates QEPs according to a fixed optimization goal, i.e., one that cannot be explicitly stated by the user. By state-of-the-art, for a particular QoS variable, is meant the one that produces the most efficient QEPs in terms of that QoS. Performance between fixed-goal SNQPs is comparable providing that they all support the same (or similar) QoS expectations. While none of the SNQPs surveyed earlier in Section 3.2 precisely defined their optimization goal in terms of a specific QoS variable, as FG-SNEE does, they are all concerned with conserving energy in order to prolong the lifetime of a QEP. When only an equality constraint over \( \alpha \) is provided, both TinyDB and Cougar have, broadly speaking, a similar, implicit optimization goal, and can be reasonably compared with one another in order to identify the one that has the best performance, i.e., in this case the one that produces the most energy efficient QEPs\(^1\). In the case

\( ^1 \)This is not the case when a minimum \( \lambda \) constraint is specified in TinyDB, as the implicit optimization goal becomes \( \min \alpha \), and therefore, in this case TinyDB’s performance is not
CHAPTER 7. EVALUATION

of SNQL, the assumed QoS expectation is fundamentally different to the other SNQPs, as $\alpha$ and $\delta$ may be arbitrarily high in order to meet $\lambda$, i.e., $\alpha$ and $\delta$ are effectively unbounded. Given that SNQL’s performance is not directly comparable with that of TinyDB or Cougar, it is only feasible to directly compare TinyDB and Cougar.

Assessing the relative performance of TinyDB and Cougar empirically would be impractical, as this would involve running QEPs on two very different infrastructures, so performance figures were sought in publications associated with these software artifacts. However, the focus of most experimental evaluations on complete SNQPs in the literature has been on the benefits afforded by in-network processing (with a particular emphasis on aggregates). For example, Yao and Gehrke [YG03] demonstrate how the in-network processing of aggregations in Cougar leads to an improvement in both average energy $\bar{\varepsilon}$ and delivery time $\delta$, as it trades-off computations with communications within the sensor network. Similar benefits are shown by Madden et al. [MFHH02], using TinyDB, by measuring radio packets, a variable that is correlated with both energy and time. These experiments focus on demonstrating that in-network processing, as opposed to sending all the raw data to a machine outside the network for processing, can be beneficial for both time- and energy-related QoS variables. However, the experimental evidence provided by publications associated with TinyDB or Cougar do not provide figures that would enable a comparison of the performance between Cougar and TinyDB, or with a different SNQP such as FG-SNEE, to take place (in terms of any QoS variable, including $\bar{\varepsilon}$ or lifetime $\lambda$).

As a last resort, it would be possible gain an idea of the performance of a SNQP and deduce the properties of its QEPs based on a description of its query optimization decision-making policies. Unfortunately, in the case of Cougar, little detail is given about how queries are compiled, and how the decisions are made, so this approach is not possible. However, for TinyDB, there is a much more complete account about how the query optimizer operates. Based on this description, it is observed that TinyDB source nodes acquire tuples and transmit them up the routing tree towards the gateway in the same epoch without buffering, i.e., the acquisition interval and transmission interval are the same. Therefore, from a when-scheduling point of view, TinyDB QEPs can be expected to behave in a similar manner to a FG-SNEE agenda with a buffering factor $\beta$ of one. Note, that directly comparable with either Cougar or SNQL.
there is an exception when TinyDB does not exhibit this behaviour, viz., when a lower bound for $\lambda$, and upper-bound for acquisition interval $\alpha$ are specified, and TinyDB predicts that it will be unable to meet both requirements, in which case tuples are buffered and load-shedding occurs if the queues become full. However, in the general case, when only an equality constraint on $\alpha$ is specified for TinyDB, its reported behaviour corresponds to FG-SNEE with $\beta = 1$.

Therefore, in order to determine a suitable candidate to represent the behaviour of state-of-the-art fixed goal SNQPs, it is assumed that the QEP generated by FG-SNEE with $\beta$ coerced to 1 (referred to as TDB-SNEE) is sufficiently indicative as to how TinyDB would behave for the same $\alpha$. In other words, TDB-SNEE is considered to sufficiently emulate the behaviour exhibited by TinyDB. Note that this assumption ignores the fact that routing decisions and where-scheduling decisions are not made the same way in TinyDB and FG-SNEE. However, given that routing decisions in FG-SNEE consider the location of data sources (i.e., so as to reduce the energy cost of the edges in the routing tree that are required to answer the query), and where-scheduling decisions in FG-SNEE support a greater degree of in-network processing (e.g., unlike TinyDB, data-reducing joins that correlate data from different nodes may be placed in-network with FG-SNEE), this assumption is likely to bias comparative outcomes to favour TDB-SNEE over FG-SNEE.

In the first set of experiments of the chapter, TDB-SNEE is compared against FG-SNEE, and energy-related QoS variables such as $\bar{\epsilon}$ and $\lambda$ are measured in both cases, for a varying $\alpha$. The query optimizer that, overall, exhibits the best performance for $\bar{\epsilon}$ and $\lambda$ is considered to be a suitable representative of state-of-the-art fixed-goal SNQPs and acts as a baseline for evaluating QoSA-SNEE in Section 7.2.

### 7.1.1 Experiment Setup

In the experiment, 15 random scenarios comprising a query, network topology and physical schema are generated and compiled using FG-SNEE and TDB-SNEE for varying values of $\alpha$. No other QoS constraints are specified because different types of QoS constraints are supported by TinyDB and FG-SNEE, and would therefore not be directly comparable. The resulting QEPs are analysed, and their performance in terms of various QoS variables is obtained using an analytical model. The mean for each QoS variable is computed across the 15 scenarios,
and these are then plotted to enable comparison of the properties of the types of QEPs generated by each QoS expectation.\(^2\)

The intention behind using different, randomly generated scenarios is to ensure that the results obtained are robust, i.e., do not hold only for specific cases. The scenarios that were generated (comprising a query, physical schema and network topology) are presented in Appendix E. The query is produced using a random SNEEq query generator that produces queries with arbitrary levels of nesting, joins and aggregations. Based on the extents used by the query, a physical schema is randomly generated that maps logical extents to physical nodes in a fixed network of size 30. For each extent, a random number of source nodes are assigned to be sources.

Networks of varying densities are generated with the nodes scattered in a random manner. The density of a network may affect the choices that the SNEE optimizer makes, particularly during the routing step where, if the nodes are relatively close together, it may have to choose between a routing tree with more nodes and a higher number of shorter hops, or fewer nodes with a lower number of longer hops. A network has an \(r\text{Value}\) associated with it, defined as the reciprocal of the distance between nodes in terms of the maximum radio range supported by the nodes. For example, for the Mica2 platform [Tec09b], which is the hardware assumed in these experiments, the default radio range \(R=60\text{m}\), so if \(r\text{Value} = 1\), new nodes are added to the network at most 60m from an existing node, and if \(r\text{Value} = 30\), nodes are added to the network at most 2m from an existing node. The network generation algorithm selects an \(r\text{Value}\) in the range \([1..30]\), and after adding a node at the origin, repeatedly selects a node at random in the network, and adds a new node \(\frac{1}{r\text{Value}}\) units away from it at a random angle. This results in a connected network (i.e., there is no node in the network that cannot communicate directly with at least one other node) in which nodes are dispersed in a non-uniform fashion (i.e., the nodes will be situated in a manner not unlike a real world deployment, where physical features are likely to make a spatially uniform distribution of nodes throughout a sensing field impractical or infeasible). For each generated network, a graph \(N\) with energy weightings used as metadata for SNEE, as described in Section 4.4, is derived analytically using the Free Space model [Rap02].

The suitability of the QEPs generated with respect to the QoS expectations is

\(^2\)Individual, per scenario, results for all experiments are presented in Appendices F and G.
7.1. IDENTIFYING A BASELINE: FIXED-GOAL SNQP EVALUATION

assessed using a model for each QoS variable derived from the agenda and the time and energy CEMs described in Appendix B. For the acquisition interval $\alpha$ and delivery time $\delta$, deriving such models simply involves reading the relevant values off the agenda. For total network energy $\epsilon$, firstly the energy consumption at each site $\epsilon_i$ in the routing tree (except the gateway) is computed by iterating over the agenda and adding the associated energy cost, obtained from the CEMs for a given fragment. For periods above a certain threshold, where there is no activity assigned by the agenda for a given node, the node is assumed to be in a low-power sleep mode, and an energy consumption value is calculated by multiplying the length of the period with a constant value that represents the CPU energy consumption in this state, using the approach proposed by Breninkmniejer et al. [BGFP09]. Total network energy is then obtained by summing the energy consumptions at each node in the routing tree. Note that nodes in the network that do not participate in the routing tree are excluded. Average node energy $\bar{\epsilon}$ is obtained simply by calculating the average $\epsilon_i$. In order to enable comparisons between different QEPs, the $\epsilon$ and $\bar{\epsilon}$ values are scaled to give values for a fixed period of time.$^3$ Network lifetime $\lambda$ is computed first by obtaining the lifetimes for each node $\lambda_i$. This is done by dividing each $\epsilon_i$ by the length of an agenda (to give the energy consumption per time unit), and then dividing the energy stock available by the result, giving $\lambda_i$. The value for $\lambda$ is the lowest $\lambda_i$.

7.1.2 Results

This section presents the results obtained by comparing the performance of FG-SNEE and TDB-SNEE. Results for average energy (the optimization goal of FG-SNEE) and lifetime (the purported optimization goal for TinyDB) are shown in Figures 7.1 and 7.2, aggregated for 15 randomly-generated scenarios. Appendix F presents a breakdown of each graph in this section showing the individual results obtained for each randomly-generated scenario. The graphs show that in all cases, the energy consumption of TDB-SNEE is the same as or higher than that of FG-SNEE. Similarly, the lifetimes obtained for TDB-SNEE are shorter than, or the same as, the lifetimes obtained for FG-SNEE. For total network energy (not shown here), the trend follows closely that of $\bar{\epsilon}$. In other words, FG-SNEE consistently outperforms or performs the same as TDB-SNEE for energy-related

$^3$The value of six months was chosen for this as it is approximately in the middle of the range of the lifetime figures in Table 2.3.
Figure 7.1: Average $\bar{\epsilon}$ obtained across scenarios for FG-SNEE and TDB-SNEE for various $\alpha$.

QoS variables.

The average buffering factor $\beta$ chosen by the optimizer in each case is shown in Figure 7.3. For TDB-SNEE, $\beta$ is inherently 1. For FG-SNEE, the optimizer always tries to maximize $\beta$. It can be observed that the amount of buffering increases as $\alpha$ increases, and then levels off. This is because at lower values of $\alpha$, a higher throughput needs to be sustained, meaning that there is less scope for buffering. The amount of buffering that is feasible increases until it is constrained by the amount of memory available on the nodes. Buffering is particularly beneficial for $\epsilon$, $\hat{\epsilon}$ and $\lambda$ for lower and middle values of $\alpha$ in the range shown, e.g., for $\alpha = 15s$, FG-SNEE shows an improvement in $\lambda$ over TDB-SNEE by an average of 80.7%. Benefits are much less significant for higher values of $\alpha$, e.g., only an average 1.11% lifetime is gained for $\alpha = 3000s$ – a result of the overhead of having the nodes in a lower-power state becoming the dominant energy cost.

For delivery time, Figure 7.4 shows that TDB-SNEE delivers results in a constant time, whereas for FG-SNEE, $\delta$ increases substantially as $\alpha$ increases. This demonstrates that TDB-SNEE is significantly faster than FG-SNEE, as the latter trades delivery time in favour of energy efficiency. This trade-off is significant for lower and middle values for $\alpha$. For higher values of $\alpha$, it may be argued
Figure 7.2: Average $\lambda$ obtained across scenarios for FG-SNEE and TDB-SNEE for various $\alpha$.

Figure 7.3: Average $\beta$ obtained across scenarios for FG-SNEE and TDB-SNEE for various $\alpha$. 
that the higher $\delta$ is not compensated by the energy saving. However, considering solely energy-related variables (the purported optimization goals of both SNQPs), the performance of FG-SNEE is better than for TDB-SNEE in all cases tested. Therefore, based on the previous assumption that TDB-SNEE behaves in a similar manner to FG-SNEE, and furthermore, considering that these results do not consider the difference in routing and where-scheduling decision-making policies between TinyDB and FG-SNEE, it can be concluded that FG-SNEE performs better than TinyDB when energy-related QoS variables are considered. During the remainder of this chapter, FG-SNEE is therefore used as the baseline SNQP for the purposes of evaluating the benefits of QoS awareness.

## 7.2 Evaluation of QoS-awareness in SNQPs

This section presents experiments that aim to demonstrate the benefits of QoS-awareness by analysing the properties of QEPs generated by QoSA-SNEE. This involves confirming the claim that QoSA-SNEE is QoS-aware (i.e., reflects QoS expectations), and also comparing its performance to FG-SNEE, assumed to be
the baseline fixed-goal SNQP when optimizing for energy-related QoS variables. By reflecting the QoS expectations is meant, specifically, two things: firstly, for a generated QEP it is predicted that the constraints given in the QoS expectations will be met; secondly, that the performance of the QEP in terms of the QoS variable in the optimization goal should equal or exceed the performance of QEPs optimized against the same constraints for other optimization goals. Note that in order to make a direct comparison between the performance of QEPs using different optimization goals, the constraints should be kept constant as these may affect the optimal value achievable for a QoS variable, in particular if there is a correlation between the optimization variable and a variable used in the QoS constraints.

FG-SNEE cannot be described as not having any QoS awareness. This is because, although it has a fixed optimization goal of minimizing average node energy, it accepts an equality constraint over $\alpha$, and an upper-bound constraint for delivery time $\delta$. In effect, FG-SNEE makes query planning decisions taking into account only a narrow range of the QoS spectrum. TinyDB can also be viewed as having some QoS-awareness, as despite having a fixed goal, it is possible for a user to stipulate as QoS constraints a lower bound for $\lambda$, and optionally an associated upper bound for $\alpha$. However, neither FG-SNEE nor TinyDB are considered to be QoS-aware in the context of this dissertation, as unlike QoSA-SNEE, they do not accept diverse QoS expectations, i.e., QoS expectations that comprise optimization goals and constraints over different, potentially conflicting, QoS variables relating to time and energy. Furthermore, the expression of QoS expectations in TinyDB and in FG-SNEE is extremely limited in its form in comparison with the flexible, expressive form of QoS expectations that QoSA-SNEE responds to. Therefore, the benefits of incorporating QoS-awareness into an SNQP are shown by comparing QEPs between QoSA-SNEE and FG-SNEE, i.e., QEPs induced by flexible, expressive QoS-awareness compared to QEPs induced by fixed, less expressive QoS awareness.

The expectation is that each software artifact will not be able to generate QEPs that perform as well (in terms of the relevant QoS variables) for QoS expectations that it does not support. For example, for a comparable set of QoS constraints, QEPs generated by QoSA-SNEE that are optimized to maximize lifetime should exhibit a longer lifetime than QEPs generated by FG-SNEE (or those generated by QoSA-SNEE for different optimization goals). Conversely, given
that the FG-SNEE optimization goal of $\min \bar{\epsilon}$ is not supported by QoS-SA-SNEE, FG-SNEE should be able to produce better QEPs in terms of average energy consumption $\bar{\epsilon}$ for cases when comparable QoS constraints are used. By comparable QoS constraints is meant that the constraints are the same, or that the optimization goal coerces them into being equivalent. For example, the QoS expectations $\langle \max \lambda, \{\alpha \leq 15s\} \rangle$ and $\langle \min \epsilon, \{\alpha = 15s\} \rangle$ have comparable constraints because, intuitively, both optimization goals involve conserving energy and consequently, doing less work. This means that data should be acquired as infrequently as possible, therefore maximizing the value of $\alpha$. Therefore, the resulting QEPs in both cases are likely to have $\alpha$ set to 15s.

### 7.2.1 Experiment Setup

In this experiment, the same 15 random scenarios are used as for the experiment in Section 7.1 against FG-SNEE and QoS-SA-SNEE for various QoS expectations. Table 7.1 presents the QoS expectations used for FG-SNEE, labelled using letters, which have a single optimization goal and limited types of constraints. The set of QoS expectations used for QoS-SA-SNEE are depicted in Table 7.2 and are labelled using arabic numerals, and involve diverse optimization goals and constraints over QoS variables. The second column in Table 7.2 briefly characterizes each QoS expectation, by describing the type of application in which a QoS expectation of that nature (i.e., with the same optimization goal and constraints, but with possibly different constant values) would be useful. In some cases, these are illustrated with situations that may arise in the example sensor network applications that were described in Section 2.3. Then, the third column of the same table comments on whether the QoS expectation is supported by other SNQPs and the performance that would be likely to be exhibited by them. This is based on the system descriptions, including the QoS expectations supported by each

<table>
<thead>
<tr>
<th>FG-SNEE</th>
<th>QoS {A}: $\langle \min \bar{\epsilon}, {\alpha = 5s} \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>QoS {B}: $\langle \min \bar{\epsilon}, {\alpha = 15s} \rangle$</td>
</tr>
<tr>
<td></td>
<td>QoS {C}: $\langle \min \bar{\epsilon}, {\alpha = 25s, \delta \leq 100s} \rangle$</td>
</tr>
</tbody>
</table>

Table 7.1: QoS expectations used for FG-SNEE in the experiments.
Finally, the fourth column states the closest QoS expectation supported by FG-SNEE (of the ones presented in Table 7.1).

The QoS expectations used for QoSA-SNEE show, in some cases, potentially conflicting expectations in order to observe trade-offs that may arise. For example, in QoS {4}, having a shorter acquisition interval is, intuitively, likely to result in greater energy consumption, and hence, worse performance in terms of total energy. In QoS {6}, again, the optimization goal of maximizing lifetime is likely to be at odds with having a shorter delivery time and acquisition interval. Correlation between QoS variables is also explored, e.g., in QoS {2} and QoS {3}, which explore whether varying the acquisition interval affects the delivery time. Also, the result of optimizing for different goals is compared: QoS {4,5,B} all have similar QoS expectations, given that the optimization goals are broadly concerned with conserving energy, so the constraint over $\alpha$ is likely to result in $\alpha = 15$ in all cases. However, the optimization goal in each case may result in different characteristics being exhibited by the QEPs.

The QoS expectations used are intended to be appropriate for significantly different kinds of applications. For example, QoS {1} requires the shortest possible acquisition interval (e.g., useful when a sensor network is polling for an event); QoS {4} and QoS {5} are more focused on energy-related QoS variables and a delay in receiving data is tolerable; and QoS {6} is concerned both with network longevity, and receiving results in a timely manner, two potentially conflicting goals. It is also argued that other SNQPs are likely to perform poorly in terms of many of the QoS expectations shown. For example, TinyDB does not support QoS {1,4,5,6,9,10}; its behavior by default should be similar to that exhibited by QoS {2,3} even though its purported optimization goal is lifetime, and not delivery time. If a LIFETIME clause is used with TinyDB, its decision-making behavior would be similar to QoSA-SNEE with QoS {7,8}.

QoSA-SNEE has several parameters that affect the search space of QEPs explored by the optimizer. As in classical modes of query processing, there are conflicting desiderata, viz., to compile a query in the shortest possible time, and to find a QEP that is as close as possible to the optimal one. As making modest changes to the parameters was not found to significantly affect the overall results presented in this section, sensible values for these parameters were determined by empirical observation based on the desiderata previously mentioned, as follows:
## Table 7.2: QoS expectations used for QoSA-SNEE in the experiments.

<table>
<thead>
<tr>
<th>QoS Expectation</th>
<th>Description</th>
<th>TinyDB Support</th>
<th>Cougar Support</th>
<th>SNQPL Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>{( \min \alpha ), ( { \delta \leq 5\text{s} } )}</td>
<td>A polling rate (i.e., to detect an event that may otherwise be missed) and fast result delivery (i.e., to notify the user immediately), similar to what may be required with sniper localization application described by Ledeczi et al.</td>
<td>1 (( \alpha \geq 3\text{months} ))</td>
<td>0 (( \alpha = 1\text{year} ))</td>
<td>0 (( \alpha = 25\text{s} ))</td>
</tr>
<tr>
<td>{( \min \delta ), ( { \alpha = 5\text{s} } )}</td>
<td>Fixed acquisition interval and result delivery in &quot;real time&quot;, e.g., as with the Great Duck Island deployment [MCP02+]. TinyDB/Cougar exhibit similar behaviour by default, when purportedly optimizing for lifetime. Not supported by SNQPL.</td>
<td>1 (( \delta \geq 3\text{months} ))</td>
<td>0 (( \delta = 1\text{year} ))</td>
<td>0 (( \delta = 25\text{s} ))</td>
</tr>
<tr>
<td>{( \min \delta ), ( { \alpha = 15\text{s} } )}</td>
<td>Result delivery may be indefinitely postponed; preserving energy is an important concern, e.g., similar to what may be required with the ZebraNet deployment [ZSLM04].</td>
<td>1 (( \delta \geq 3\text{months} ))</td>
<td>0 (( \delta = 1\text{year} ))</td>
<td>0 (( \delta = 25\text{s} ))</td>
</tr>
<tr>
<td>{( \max \lambda ), ( { \alpha \leq 15\text{s} } )}</td>
<td>Result delivery may be delayed, but required within a specified time; preserving energy is an important concern, e.g., as with the Glacier Monitoring deployment [MOH04].</td>
<td>1 (( \lambda \geq 3\text{months} ))</td>
<td>0 (( \lambda = 25\text{s} ))</td>
<td>0 (( \lambda = 1\text{year} ))</td>
</tr>
<tr>
<td>{( \min \alpha ), ( { \lambda \geq 3\text{months} } )}</td>
<td>Greater concern about obtaining data for a specific period than a precise acquisition interval. This may be a concern, e.g., in the Crowden Brook deployment [MPL07+]. Not supported by TinyDB, but supported by Cougar.</td>
<td>1 (( \alpha \geq 3\text{months} ))</td>
<td>0 (( \lambda = 25\text{s} ))</td>
<td>0 (( \alpha = 25\text{s} ))</td>
</tr>
<tr>
<td>{( \min \alpha ), ( { \alpha \leq 25\text{s} ), ( \lambda \geq 1\text{year} }) )}</td>
<td>As with QoS {7}, except that a maximum acquisition interval has been stipulated. TinyDB will shed tuples in order to be able to transmit results less often if the ( \alpha ) constraint does not allow the minimum ( \lambda ) to be met.</td>
<td>1 (( \alpha \geq 3\text{months} ))</td>
<td>0 (( \lambda = 25\text{s} ))</td>
<td>0 (( \alpha = 25\text{s} ))</td>
</tr>
<tr>
<td>{( \min \delta ), ( { \alpha = 25\text{s} ), ( \lambda \geq 3\text{months} } )}</td>
<td>The query should run for a minimum period of time and return results as quickly as possible. Similar QoS may be used to query events of interest for an application such as the Volcán Reventador deployment [WALJ06+]. QoS expectations not supported by TinyDB, Cougar or SNQPL.</td>
<td>1 (( \delta \geq 3\text{months} ))</td>
<td>0 (( \lambda = 25\text{s} ))</td>
<td>0 (( \delta = 25\text{s} ))</td>
</tr>
<tr>
<td>{( \min \epsilon ), ( { \alpha = 15\text{s} ), ( \lambda \geq 3\text{months} } )}</td>
<td>A lifetime for a minimum period is required, where the main concern is to reduce total energy consumption. A QoS expectation of this nature may be posed, e.g., by a scenario similar to the Crowden Brook deployment, in which regular visits are carried out for maintenance of the sensors, and users wish to reduce the number of heavy batteries that need to be carried in order to ensure that detector readings are recorded in the &quot;live&quot; sensor network.</td>
<td>1 (( \epsilon \geq 3\text{months} ))</td>
<td>0 (( \lambda = 25\text{s} ))</td>
<td>0 (( \alpha = 25\text{s} ))</td>
</tr>
</tbody>
</table>
For the routing step (described in Section 6.2), the number of routing trees to generate $k_1 = 15$ and the number of routing trees that are kept $k_2 = 5$. In where-scheduling (described in Section 6.3), the initial number of neighbours $k_3 = 10$, the number of consecutive values within a threshold before considering that a minimum value has been found $k_4 = 3$, the aforementioned threshold $k_5 = 5\%$, and the step for increasing the number of neighbours in the search $k_6 = 3$. These values have been chosen with the intention of avoiding an excessively high number of candidate QEPs from being considered in the search space.

### 7.2.2 Results

The aggregated results for 15 randomly generated scenarios, compiled with FG-SNEE and QoSA-SNEE using the QoS expectations listed in Tables 7.1 and 7.2 are presented using a separate graph for each QoS variable measured. In each graph, each x-axis point corresponds to a different QoS expectation, and the y-axis shows the average value of the QoS variable measured over all scenarios for the QoS expectation. To enable comparison between QoSA-SNEE and FG-SNEE, the result for the closest FG-SNEE QoS expectation (i.e., $QoS\{A–C\}$) is presented alongside the result for each QoSA-SNEE QoS expectation (i.e., $QoS\{1–10\}$). A breakdown of the results for each individual scenario for each graph in this section is given in Appendix G. Some example QEPs generated are presented in Appendix H and referred to as appropriate.

Figure 7.5 presents the results for the acquisition interval QoS variable. For FG-SNEE and QoSA-SNEE with $QoS\{2,3,9,10\}$, an equality constraint has been specified over $\alpha$, and therefore the when-scheduling step has no leeway with regards to adjusting its value. $QoS\{4,5,6\}$ involve an optimization goal related to being economical with energy, so the highest possible acquisition interval is selected by the when-scheduling step of QoSA-SNEE in each case. In $QoS\{1\}$, where the optimization goal is to minimize the acquisition interval, $\alpha$ is constrained by the amount of time required to execute the tasks in a single agenda episode. The shorter the amount of time required, the higher the throughput that may be supported by a QEP. The length of an agenda evaluation episode is reduced by when-scheduling setting $\beta = 1$, and also, the routing step selecting a tree which involves fewer (albeit longer) hops $\footnote{For example, compare the routing tree for scenario 6 in Figure H.6 optimized for $\alpha$, to the}$. For $QoS\{7,8\}$, when-scheduling
selects the smallest value of $\alpha$ that will meet the lifetime stipulated in the constraints over $\lambda$; as one would expect, a higher $\alpha$ is required to satisfy a more challenging lifetime expectation.

The delivery time variable, shown in Figure 7.6, is impacted, first and foremost, by the buffering factor selected during when-scheduling, and secondly, by decisions made in routing such as the number of hops required by the sources to reach the root of the routing tree. This is because increasing the buffering factor leads to an increase in the delivery time proportional to $\alpha$ (usually in the order of seconds, or possibly a larger unit), whereas selecting a routing tree with an additional edge increases $\delta$ proportionally to the time that it takes to send a packet of data between two nodes (typically in the order of tens of milliseconds [Bre09]). The higher the value of $\beta$ determined during when-scheduling, the longer the delivery time. The shortest values for $\delta$ are therefore obtained when $\beta = 1$. In the case of $QoS \{2,3\}$, having a minimal buffering factor is a direct result of specifying the $\min \delta$ goal, whereas in the case of $QoS \{1\}$, this is a side-effect resulting from reducing the length of an agenda evaluation episode, as discussed routing tree in Figure H.7, which is optimized for $\lambda$. 

Figure 7.5: Average $\alpha$ across scenarios, obtained for the various QoS specifications.
7.2. EVALUATION OF QOS-AWARENESS IN SNQPS

Figure 7.6: Average δ across scenarios, obtained for the various QoS specifications.

in the previous paragraph\(^5\). For QoS \{4,5\}, δ is higher than for QoS \{2,3\}, for which β is maximized insofar as there is available memory in order to preserve energy\(^6\). The delivery time is slightly higher for the optimization goal of \(\text{max } \lambda\) than \(\text{min } \epsilon\), as the former favours routing trees with a higher message complexity, i.e., there are more nodes in the routing tree to avoid hotspots. For QoS \{6,C\}, the delivery time is within the upper-bound specified in the QoS expectations.

In both cases, the optimization goal is concerned about maximizing lifetime, so the highest possible buffering factor that allows the delivery time constraint to be maintained is selected. QoS \{7,8\} illustrate the trade-off between lifetime and delivery time; the QoS expectation with a shorter lifetime results in QEPs with a shorter delivery time.

Figure 7.7 presents the performance obtained in terms of \(\epsilon\), the total network

\(^5\)Note, however, that as one would expect, although QoS \{2,3\} both favour QEPs that are similar in nature (i.e., agendas with a low buffering factor, and routing trees with as few hops as possible), given that \(\delta = \alpha(\beta - 1) + \pi\) (as defined in Section 4.6.4), when \(\beta = 1\) there is not any correlation between \(\alpha\) and \(\delta\) variables, as demonstrated by the use of two different values of \(\alpha\) in QoS \{2,3\} having exactly the same value for \(\delta\).

\(^6\)For example, see the agenda in Figure H.5 resulting from optimizing scenario 5 for QoS \{5\}, compared to the agenda for the same scenario for QoS \{2\} shown in Figure H.4.
energy consumption over a specific period of time (in this case 6 months). QoS $\{1\}$ has the highest $\epsilon$ associated with it, a result of having the shortest acquisition interval, and also, a buffering factor of 1. The $\epsilon$ value is significantly reduced by increasing $\alpha$ (with a fixed $\beta = 1$), as demonstrated by the energy consumption values for QoS $\{2,3\}$. For QoS $\{4,5\}$, both of which reflect concerns about preserving energy, both $\alpha$ and $\beta$ are maximized. QoS $\{4\}$, which is optimized specifically to minimize total network energy, has a slightly lower $\epsilon$ value than QoS $\{5\}$, a result of selecting a routing tree with fewer nodes involved in the QEP. In contrast, QoS $\{5\}$ selects a routing tree that aims to distribute energy consumption as evenly as possible, a goal that may involve energy being required for the network as a whole. For FG-SNEE, QoS $\{A\}$ has a higher $\epsilon$ value than QoS $\{2\}$, despite the fact that the latter optimizes for delivery time. This is because similar types of routing trees (i.e., those with fewer nodes) are favoured by both the $\delta$ and $\epsilon$ QoS variables. Furthermore, for comparable QoS expectations, i.e., QoS $\{B\}$ with QoS $\{4,5\}$, and QoS $\{C\}$ with QoS $\{6\}$, $\epsilon$ is slightly higher, and therefore worse, for FG-SNEE. This is because QoSA-SNEE is able to optimize specifically to minimize total network energy, e.g., during the routing step it aims to reduce the number of sites that will participate in the QEP compared
7.2. EVALUATION OF QOS-AWARENESS IN SNQPS

Figure 7.8: Average $\lambda$ across scenarios, obtained for the various QoS specifications.

to FG-SNEE which selects routing trees in a predetermined manner.

For lifetime, the results shown in Figure 7.8 broadly mirror the results for $\epsilon$. An important exception is that $QoS\{5\}$ now performs significantly better than $QoS\{4\}$, given that the query has been optimized specifically to maximize lifetime, causing that the routing step to select a tree which is less likely to have hotspots. For FG-SNEE, the comparable $QoS\{B\}$ has the worst performance in terms of $\lambda$. Again, this is because FG-SNEE does not optimize specifically for $\lambda$: its routing, where- and when-scheduling steps do not attempt to avoid hotspots so that energy consumption is as evenly distributed as possible.

The average energy per node $\bar{\epsilon}$ variable (Figure 7.9), not supported by QoSA-SNEE, shows a very similar trend as for $\epsilon$, with the exception that scenarios optimized for $QoS\{5\}$, which has a max $\lambda$ goal, result in a more favourable $\bar{\epsilon}$ than those optimized for $QoS\{4\}$ with the min $\epsilon$ goal. This is not surprising because the $\bar{\epsilon}$ variable does not penalize having additional nodes in the QEP as does the $\epsilon$ variable, and is therefore more similar to the $\lambda$ variable in this respect.\footnote{For example, compare the routing tree for scenario 5 in Figure H.1, optimized for $\delta$, to the routing tree in Figure H.2 which is optimized for $\lambda$.}

\footnote{Recall that sites that do not participate in the routing tree are excluded from the $\epsilon$, $\bar{\epsilon}$ and...}
Contrary to the expectations expressed at the start of this chapter, in terms of \( \bar{\epsilon} \), \( QoS \{4,5\} \) performs slightly better than \( QoS \{B\} \), and \( QoS \{6\} \) performs slightly better than \( QoS \{C\} \), i.e., \( QoSA-SNEE \) outperforms \( FG-SNEE \) at its own optimization goal. This result can be attributed to the fact that \( QoSA-SNEE \) considers several candidate QEPs and specifically takes QoS expectations into account, whereas \( FG-SNEE \) does not consider several alternatives, is consequently more deterministic, and is based on heuristics that are deemed to be beneficial for a single QoS variable.

7.3 Discussion

Overall, the scenarios result in stark trade offs between QoS variables related to time and energy, which in turn demonstrates that \( QoSA-SNEE \) makes optimization decisions in response to the user QoS expectations and is therefore QoS aware. Given the diverse nature of sensor network applications (see Section 2.3), it can be concluded that \( FG-SNEE \) is less of a general-purpose SNQP than \( QoSA-SNEE \) is. Furthermore, it is also interesting to note that \( QoSA-SNEE \) outperforms \( \lambda \) calculations.
FG-SNEE in terms of the QoS variables $\epsilon$, $\bar{\epsilon}$ and $\lambda$ even when the former is optimizing for a similar QoS expectation to the inbuilt one of FG-SNEE (i.e., for QoS $\{4,5\}$). For example, for QoS $\{5\}$, QoSA-SNEE achieves both a longer lifetime and shorter delivery time than FG-SNEE. This is because, in QoSA-SNEE, the routing and where-scheduling steps are less deterministic and various alternative solutions are explored in the search space, whereas FG-SNEE has a heuristic-based approach that does not consider various alternatives. This is a shortcoming of FG-SNEE as a software artifact, as generally, one would expect a SNQP specialized for a fixed QoS $q$ to equal or improve upon the performance, in terms of $q$, of a more generic QoS-aware SNQP optimizing for $q$.

For most QoS desiderata, the when-scheduling step has a more significant impact than other steps in the query processing stack. For example, when-scheduling decisions dominate with regards to delivery time, whereas the effect of routing decisions is less significant. Routing decisions do however have a significant impact on $\epsilon$ (as they can exclude nodes from participating in the QEP thus reducing its total network energy cost) and also $\lambda$ (as they may avoid the formation of hotspots in the routing tree, the main factor affecting network lifetime). Where-scheduling decisions do not appear to differ according to the QoS expectation; for both time- and energy-related QoS variables, it is beneficial to reduce the amount of data to be transmitted via the radio, as radio activity is both more time- and energy-intensive than CPU activity.

Turning to related work, recall, from Section 3.2, that TinyDB contains a LIFETIME clause in the query language, that enables the query processor to adjust $\alpha$ in order to ensure that the requested lifetime is met (and optionally, a minimum acquisition interval). This is equivalent to having a QoS constraint of the form $\{\lambda \geq l, \alpha \leq a\}$, where $l$ denotes the desired lifetime, and $a$ the longest acceptable acquisition interval, respectively. While Madden et al. [MFHH05] describe an approach to predict the lifetime remaining based on the current energy stock, no experimental evaluation is presented showing how $\alpha$ varies for different values of the requested lifetime, which would be interesting to reveal the trade-offs obtained between these two QoS variables for TinyDB.

For the Wave Scheduling approach [TYD+07], two routing heuristics, one intended to reduce $\delta$, and the other to reduce $\epsilon$, are evaluated. These heuristics exploit knowledge about the predefined wave schedules, which stipulate communication times between pairs of nodes. The $\delta$ heuristic may require more hops
CHAPTER 7. EVALUATION

for data to reach the destination node, which in turn implies the use of more energy, whereas the \( \epsilon \) heuristic aims to reduce the number of hops, possibly leading to an increase in \( \delta \). The approach therefore aims to trade off total energy and delivery time. While it is shown that the \( \delta \) heuristic indeed results in shorter delivery times, it is reported that using the \( \delta \) heuristic only incurs a small energy overhead, suggesting that it is not worth using the \( \epsilon \) heuristic. While this result may suggest that, in general, routing decisions have a limited scope for impact on QoS variables, recall from the example routing trees presented in Section 6.2.2 that the same routing trees are generated for the \( \min \epsilon \) and \( \min \delta \) optimization goals, because they both benefit from having fewer nodes in the QEP. In the case of the former goal it is because involving fewer nodes means that less energy is consumed overall, and for the latter goal, because having fewer hops reduces the message complexity of the QEP. Note, however, that the definition of a routing decision is broader for SNEE, as it can include or exclude a node from the QEP, thereby impacting \( \epsilon \). Furthermore, routing has an important impact on lifetime, as for \( \max \lambda \) it will attempt to generate a tree with the aim of avoiding hotspots in the network, and \( \lambda \) is a QoS variable that is not considered by Wave Scheduling.

The SNQL [BLM+07] approach involves trading off the acquisition interval against memory available, and the delivery time against the energy available on a node. The former is essentially a technique to reduce the need for load-shedding when the network becomes congested. Unfortunately no results are presented that show the relationship between energy consumption and delivery time.

In summary, for the related work in the SNQP area described in the preceding paragraphs, there are few directly comparable results. This is a consequence of the fact that, to the best of the author’s knowledge, no research to date has explored the issue of QoS-awareness in SNQP with the depth (i.e., for different types of query planning decisions) and breadth (i.e., for a range of different QoS variables) that is done in this dissertation.

7.4 Conclusion

This chapter has presented an experimental evaluation that firstly, argues that FG-SNEE can be taken to be a suitable representative for previous state-of-the-art SNQPs and therefore suitable to be used as a baseline for evaluating the benefits of QoS-awareness compared to optimizing for a fixed goal. Then, the evaluation
QoSA-SNEE is presented. It is shown that, according to the definition given of QoS-awareness at the outset of this dissertation, QoSA-SNEE is indeed QoS-aware as it responds to diverse QoS expectations and that this leads to performance improvements that are significant in magnitude and instrumental in terms of enabling a SNQP to support several classes of sensor network applications. Based on this, the behaviour of a QoS-aware SNQP has been compared to FG-SNEE which has limited QoS awareness, and it has been ascertained that the former has greater flexibility (i.e., it is able to support more diverse QoS expectations) and is therefore more generic, in the sense that it may be used for a broader range of sensor network applications. When optimizing for a specific QoS variable, QoSA-SNEE results in the QEPs with best performance for that QoS variable (all other factors, such as QoS constraints, being equal). Furthermore, it turns out that QoSA-SNEE outperforms FG-SNEE at its own optimization goal, a consequence of the former being more cautious about pruning QEPs at early steps in the stack.

Based on the QoS expectations and example applications given in Table 7.2, the findings suggest that a QoS-aware SNQP is useful for a broader range of sensor network application classes than a SNQP with a fixed-goal or limited QoS-awareness. For example, the type of applications that FG-SNEE would be well-suited for is restricted to applications with QoS expectations like the ZebraNet and Glacier Monitoring deployments in which network longevity is the prime concern, and a delay in result delivery is tolerable. For other types of applications cited (e.g., those with QoS expectations like the sniper detection application) FG-SNEE is less likely to generate QEPs that are satisfactory to the user. QoS-awareness is also likely to provide significant benefits for applications that involve different QoS expectations in their lifecycle, e.g., the Volcano monitoring application, during which data is collected at a much shorter acquisition interval when interesting events such as eruptions occur. In the next chapter, overall conclusions are drawn, and future work is proposed.
Chapter 8

Concluding Remarks

The purpose of this chapter is to review the research contributions made in this dissertation (Section 8.1), to discuss the significance of the results obtained (Section 8.2), and to propose future research directions that arise from these results (Section 8.3).

8.1 Research Contributions

This section summarizes the research contributions made in this dissertation, by considering each research objective set out in Section 1.4 in turn.

Objective O1 involved the design of a functional decomposition of the required steps to compile a declarative query into a sensor network QEP. The resulting research contribution was the SNEE query stack template, presented in Chapter 4. From the template, two instantiations of SNEE were designed and implemented as software prototypes. The existence of these two example instantiations that provide a top-to-bottom translation from a declarative query to executable code demonstrates that the proposed functional decomposition is valid. The resulting software architecture enables more query planning decisions to be made explicitly than with the SNQPs surveyed in Section 3.2. Being able to make decisions explicitly is beneficial, as it provides the optimizer with greater control over different aspects (such as the routing of tuples, timing of activities or placement of operators) of the QEP. This in turn enables QoS expectations to be reflected for more types of decisions during QEP generation, as is done during the design of QoSA-SNEE, and demonstrated by the experimental evidence provided in Chapter 7.
The contribution associated with Objective O2 was the design of an instantiation of O1 that optimizes queries according to a fixed optimization goal. The resulting software artifact was FG-SNEE, which is publicly available under a permissive open-source license at http://code.google.com/p/snee\textsuperscript{1}. Its decision-making policies are implemented using heuristics that explicitly aim to reduce average node energy consumption. Compared to previous SNQPs, which make most decisions associated with QEP generation in a predetermined or implicit manner, FG-SNEE makes more decisions explicitly and taking into account the inbuilt optimization goal. The advantages of this approach are shown experimentally in Section 7.1 where the performance of TDB-SNEE (which is assumed to behave in a sufficiently similar fashion to TinyDB for a rough contrast to be drawn) is compared to that of FG-SNEE. The experiments carried out indicate that FG-SNEE would outperform TinyDB for energy-related QoS variables, thereby suggesting the benefits of making more query planning decisions explicitly. Based on the analyses on the often rather limited and focussed empirical performance results in research papers, FG-SNEE is therefore considered to be a suitable representative for the previous state-of-the-art SNQPs with optimization goals that involve the conservation of energy and is used as the baseline for comparison with the QoS-aware instantiation of SNEE.

Objective O3 involved designing and implementing a QoS-aware instantiation of O1 that generates QEPs that reflect diverse optimization goals and constraints. The resulting software artifact was QoSA-SNEE. Its constituent decision-making algorithms use utility functions (composed of time and energy CEM expressions) defined for each QoS variable that reflect the desirability of a solution (i.e., a candidate intermediate QEP) in the search space with respect to that QoS variable. The approach taken is to either randomly choose a sample from a large space of solutions and assess them on the grounds of utility (e.g., as in the routing step), or to formulate a constrained optimization problem that is delegated to an external optimizer in order to obtain the solution associated with a quasi-optimal value for its corresponding utility (e.g., as in the where- and when-scheduling steps). This approach is successful in generating QEPs that exhibit significantly more appropriate properties depending on the QoS expectations given, as demonstrated in the experimental evaluation in Section 7.2.

\textsuperscript{1}The codebase is at the time of writing actively maintained and hence usable by other researchers. Unfortunately, this is true of no other SNQPs to the best knowledge of the author.
The algorithms proposed as part of O3 for QoSA-SNEE are generic in the sense that the same algorithm is used for diverse QoS expectations (a key desideratum stated at the outset of Chapter 6). The decision-making policies were customized for different QoS expectations by using QoS-dependent utility functions to assess the performance of alternative QEPs. In other words, the points in the algorithms where utility functions were employed are extensibility points whereby the behaviour of the algorithm can be significantly changed. For example, the routing step relies on QoS-dependent scoring functions used for ranking alternative routing trees, and the output of this step can be altered by using different scoring functions. Therefore, the algorithms presented are generic insofar as they can support any QoS variable that may be expressed as a utility function.

The final objective, O4, aimed at evaluating the performance benefits of QEPs generated by a QoS-aware SNQP compared to one with a fixed goal. This was done by comparing the performance (in terms of various QoS variables) of a SNQP representative of the previous state-of-the-art (i.e., FG-SNEE) to the only QoS-aware SNQP in existence (i.e., QoSA-SNEE). The experimental evaluation showed that QoSA-SNEE generated QEPs with properties corresponding to the QoS expectations provided. On the other hand, FG-SNEE could not generate QEPs with such a broad range of properties, as it is less flexible with respect to the range of QoS expectations that it is possible for a user to express. In terms of the QoS variables relevant to the QoS expectation, QoSA-SNEE always exceeded the performance obtained by FG-SNEE, even for the QoS variables that FG-SNEE was specialized for. This was because FG-SNEE, like many classical query optimizers deploys, as its optimization strategy, dynamic programming, whereby, greedily, only the best intermediate QEP is passed from one step of the stack to the next, whereas QoSA-SNEE has a more cautious pruning policy in order to avoid eliminating near-optimal QEPs too early on. In the next section, the overall significance of the research contributions made is discussed.

8.2 Significance of the Results

As mentioned at the start of the dissertation, software development in sensor networks is an expensive and time-consuming process. This is a result of software developers requiring specialist skills (i.e., typically knowledge of a variant of embedded C), and of distributed algorithms being inherently complex. This
8.2. SIGNIFICANCE OF THE RESULTS

is currently an important concern, as demonstrated by the amount of research carried out in recent years aimed at facilitating sensor network software development. Examples of approaches that have been proposed in the literature to insulate developers from the underlying complexities involved in programming sensor networks are query processors (e.g., TinyDB), virtual machines (e.g., SwissQM [MAK07]), logic programming languages (e.g., Snlog [CT+07]) and scripting languages (e.g., BASIC for sensor networks [MDD09]).

Given that the functional requirements of most sensor network applications involve data collection, query processing has been effectively used in real-world deployments to repurpose sensor networks on-the-fly (e.g., TinyDB was used in SMP+04 [TPS+05]). However, given that, as discussed in Section 2.3 sensor network applications have very diverse QoS expectations, previous sensor network query processors with a fixed goal and limited QoS awareness would not be able to generate appropriate QEPs for applications across the whole spectrum of QoS expectations. For example, the Glacier monitoring application described by Martinez et al. [MOH04] acquires readings every 15 minutes, and only sends them to the gateway once a day. TinyDB would not be suitable for this application as it streams results to the gateway with no or minimal buffering, meaning that nodes would be depleted of their energy prematurely in comparison with either FG-SNEE or QoSA-SNEE.

The research contributions in this dissertation demonstrate that SNQPs can be made QoS-aware, and that, in doing so, they will become more general purpose. By allowing more application requirements to be specified declaratively, and having their respective programs generated automatically, fewer applications will need to be hand-crafted from scratch. This is significant, given that the functionality associated with even a simple query may imply the need to write thousands of lines of source code in an embedded programming language (and that, in addition, the application would need to be thoroughly tested and debugged prior to being deployed). Another significant advantage of using query processing technology as opposed to hand-crafted applications is the ability to change the application requirements, on-the-fly, once the application has been deployed.

---

2For example, the number of lines of nesC code generated by QoSA-SNEE for the query 3 agendas in Figure 6.30 in Section 6.4 is, respectively, 10189, 12232, 17285. Even assuming that an expert human programmer might reduce the count by a substantial percentage, these figures suggest a significant amount of specialized development for this.
deployed. Therefore, given these advantages, the savings in software development costs mean that users looking to deploy sensor networks are likely to be more willing to forfeit the potential loss in performance associated with a generated program compared to a hand-crafted one. This trade-off occurs in every other mode of query processing, and is generally accepted to be worthwhile, as demonstrated by the widespread uptake of database technology.

In effect, the research contributions imply that a broader range of sensor network applications may benefit from query processing technology than was previously the case. Being able to express the requirements of more applications as queries would result in significant cost-savings in software development for those involved in deploying sensor networks. This, in turn, could contribute to the increase in uptake of this technology, meaning that more people would be able to reap the potential benefits of sensor networks.

Another significant contribution of this dissertation is a query processing approach that is generally applicable for other modes of query processing in which trade-offs between desiderata (not limited necessarily to QoS) may need to be considered, and alternative decision-making behaviours need to be exhibited in light of these. This approach, based on that proposed by Paton et al. [PAL+09], broadly involves the use of utility functions to represent desiderata (e.g., a QoS variable) to assess potential QEPs and/or find a QEP associated with the optimal (or quasi-optimal) solution of the utility function. This approach is generic so long as desiderata may be expressed in terms of utility functions, and that the solution space can be explored effectively. This approach may be applicable in cases where a performance metric such as monetary profit of a system needs to be traded-off against response time (e.g., as done in Mariposa [SAL+96]). In cases where resource availability is a concern, this approach could trade-off monetary profit against memory consumption or bandwidth. Similarly, streaming query processors may use this approach to trade-off delivery time and the accuracy of results (e.g., by varying the degree of load-shedding performed). Also, this approach may lend itself to trade-off data quality against response time or monetary cost in data integration systems. Therefore, this work is also an exploration into how alternative decision-making behaviours can be incorporated into query optimizers, and comes under the category of techniques surveyed in Section 3.4.
8.3 Future Work Directions

The contributions made in this dissertation give rise to several potential directions for future work. Broadly speaking, these may involve exploring how the expressiveness of the QoS expectations may be enhanced, and also how QoS-awareness may be supported in a multiple query setting, or in an adaptive manner at runtime.

**Support for Multiple Optimization Goals.** For some applications, more than one (mutually conflicting) QoS variable may be highly relevant. For example, the sniper detection application \[^{LNV+05}\] is challenging because it requires both a short delivery time and short acquisition interval, making it potentially very energy intensive. In situations such as these, it may be desirable to be able to express more than one optimization goal. This is not possible in QoSA-SNEE in which only one optimization goal is supported. Thus, incorporating support for multiple optimization goals is likely to be a useful area of future research. Ideally, this should be possible with no modifications to the generic, core algorithms proposed in QoSA-SNEE. Deriving composite utility functions that represent more than one QoS variable should suffice.

**Support for Weightings Associated with Constituents of a QoS Expectation.** There are cases, within a QoS expectation, where the user may want to associate a weighting to each optimization goal or constraint to reflect its relative importance. For example, if a user specifies the QoS expectation \(\langle \max \lambda : 1, \{\delta \leq 3s : 5 \} \rangle\), it would indicate that it is considered that having a delivery time of less than 3 seconds is considered to be five times more important to the user than maximizing lifetime. Naturally, it may not be clear exactly what this means, but the intuition is that the query optimizer should give priority, first and foremost, to achieving a delivery time within three seconds, and then, secondarily, to maximizing lifetime. Conversely, if the weightings are reversed, giving \(\langle \max \lambda : 5, \{\delta \leq 3s : 1 \} \rangle\), the query optimizer should primarily aim to achieve a long lifetime. Note, however, that any QEPs that the optimizer predicts will not meet the delivery time expectation are discarded, despite the lower weighting assigned to the constraint, as a constraint is effectively a stricter type of expectation than an optimization goal as it characterizes feasibility rather than comparative excellence. Again, it should be possible to incorporate this into the...
existing QoSA-SNEE architecture with minimal disruption to the core algorithms.

**Incorporation of Other Types of QoS Variables.** Other types of QoS variables are described in the literature that could be supported using the approach described in this dissertation. For example, a user may be interested in specifying the spatial resolution at which data acquisition from sensors should take place, particularly in a dense network, given that it may not be necessary to sample all the sensors at every acquisition time, e.g., as described by the region sampling approach by Lin *et al.* [LAGD08]. The maximum bandwidth that a QEP may use could also be an important concern [DKR08], for example, if applications in addition to the query are being executed concomitantly by the sensor network. Furthermore, data freshness, which considers how recent the data is when it is used[^3] may be a significant concern to the user [WSS04]. The above examples indicate that there is significant scope to extend the work in this dissertation with other types of QoS variables, and that doing so would yield tangible benefits.

**Incorporation of Quality-of-Data Awareness.** Incorporating expectations that relate to information quality, so that users may trade off aspects of Quality-of-Data (QoD) against QoS variables would be empowering to the user, given that sensed data is inherently inaccurate, and often incomplete. For example, a user may be willing to have a QEP that takes a higher number of sensor readings to obtain more accurate results, at the cost of higher energy (as done, e.g., in BBQ [DGM+05]). Similarly, a user may wish to obtain readings with the assurance that they lie within a certain confidence interval (e.g., as proposed by Faradjian *et al.* [FGB02]). Such concerns are relevant in the case of sensor networks, as demonstrated by the focus on approximate answers in this context [NGSA04, CLKB02, DGHM05, DKR06, CDHH06, TM06, WXT06]. Using the QoSA-SNEE approach, support for these concerns could be implemented by enabling utility functions to represent different aspects of information quality. For example, the approach by Naumann *et al.* [NLF99] involves deriving a composite information quality score (based on factors such as accuracy and completeness) which should be representable using a utility function of the nature used to support QoS awareness in SNEE. This would result in a SNQP[^SNQP] that is able to consider QoS and QoD in a unified manner.

[^3]: This is in contrast to delivery time, which considers how long it takes data to reach the gateway.
8.3. FUTURE WORK DIRECTIONS

Considering QoS-awareness in a Multiple Query Setting. The research contributions made in this dissertation assume the compilation of a single query. The ability to pose multiple queries against a sensor network is likely to be beneficial, e.g., in cases where there are multiple users. TinyDB and Cougar [TYD+05, TGS08] can handle multiple queries. Naja [Naj08] describes a technique to support the execution of multiple queries in FG-SNEE, and proposes some optimization opportunities. In a QoS-aware setting, QoS expectations would need to be considered for the entire workload, and different queries may have significantly different, possibly conflicting, QoS expectations that would need to be reconciled.

Enabling Adaptive QoS-aware Decision-making at Runtime. The research contributions made in this dissertation are focussed primarily on query planning decisions made statically, at compile time. However, sensor networks are an inherently fragile platform in which the initial conditions assumed by an SNQP optimizer (i.e., the metadata used by the optimizer) may change rapidly. As stated in Section 1.5, an assumption is made throughout that underlying network management software exists that is able to provide an abstraction of a stable, robust network. However, there are cases when network conditions may change to such an extent that the query needs to be recompiled and redeployed, a potentially energy intensive process as executable code will need to be disseminated to nodes in the sensor network. Adaptive query processing is a technique that has been adopted in other modes of query processing in which the QEP adjusts to changes in conditions during query evaluation. Given the unstable nature of sensor networks, the exploration of adaptivity in a QoS-aware manner is also likely to be a fruitful avenue for future research [FGGP].

The above proposals for future work directions indicate that the further exploration of QoS-awareness in SNQPs has the potential to yield significant research results.
Appendices
Appendix A

SNEEql Physical Algebra

This appendix describes the SNEEql physical algebra. Note that this is largely adapted from [GBJ+09], and not claimed as a research contribution of this dissertation. Table A.1 presents a subset of the SNEEql operators, with the respective input and output collection types. A signature has the form OPERATOR_NAME[Parameters](InputArgumentTypes):OutputArgumentTypes, where the argument types are denoted R, S and W, for relation, stream and window respectively, and a vertical bar indicates a choice of one of the types given. Figures A.1 and A.2 present example PAFs using these operators for the example queries Q1-Q5 from Figure 4.3 in Section 4.2.

Leaf and Root Operators. SP.ACQUIRE is a leaf operator, and denotes a data source for a sensed stream of the type defined in the schema. DELIVER denotes a data sink, and is always a root operator.

Stream-to-Window Operators. Stream-to-window operators take a subset of the tuples in a stream to derive a window from it. TIME_WINDOW and ROW_WINDOW specify the window dimensions and the gaps between the starts of successive windows, using either time, or number of tuples, respectively. A time window is represented in the algebra in milliseconds, e.g., as can be seen in the PAF for Q2 shown in Figure A.1(b). TIME_WINDOW[t-3600000,t,1800000] denotes a window which outputs tuples with timestamps ranging from 1 hour ago to the current time, re-evaluated every 30 minutes, and is relative to t, which is bound in turn to the time in which each evaluation episode of the query starts. Note that in Figure 4.3, windows are time-based, and refer to points or intervals
### Leaf and root operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP_ACQUIRE<a href="">extentName, attrSenseList, predExpr, projExprList</a> : S</td>
<td>Take readings from sensors in attrSenseList and apply SELECT[predExpr] and PROJECT[projExprList].</td>
</tr>
<tr>
<td>DELIVER<a href="S"> </a> : S</td>
<td>Deliver the query results. LocSen. TimeSen.</td>
</tr>
</tbody>
</table>

### Stream-to-Window Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME_WINDOW<a href="S">startTime, endTime, slide</a> : W</td>
<td>Define a time-based window on stream S from startTime to endTime inclusive and re-evaluate every slide time units.</td>
</tr>
<tr>
<td>ROW_WINDOW<a href="S">startRow, endRow, slide</a> : W</td>
<td>Define a tuple-based window on stream S from startRow to endRow inclusive and re-evaluate every slide rows. AttrSen.</td>
</tr>
</tbody>
</table>

### Window-to-Stream Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSTREAM<a href="W"> </a> : S</td>
<td>Emit all the tuples in window W.</td>
</tr>
<tr>
<td>ISTREAM<a href="W"> </a> : S</td>
<td>Emit the newly-inserted tuples in window W since the previous window evaluation.</td>
</tr>
<tr>
<td>DSTREAM<a href="W"> </a> : S</td>
<td>Emit the newly-deleted tuples in window W since the previous window evaluation.</td>
</tr>
</tbody>
</table>

### Operators over windows or relations

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGGR_INIT[aggrFunction, aggrList, projList][R</td>
<td>W] : R</td>
</tr>
<tr>
<td>AGGR_MERGE[aggrFunction, aggrList, projList][R</td>
<td>W] : R</td>
</tr>
<tr>
<td>AGGR_EVAL[aggrFunction, aggrList, projList][R</td>
<td>W] : R</td>
</tr>
</tbody>
</table>

### Operators over windows, relations or streams

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJECT[projExprList] (R</td>
<td>S</td>
</tr>
</tbody>
</table>

Table A.1: The SNEEql physical algebra, adapted from [GBJ'09].
Figure A.1: Physical-algebraic Form for Q1, Q2 and Q3.
in time, rather than number of tuples.

**Window-to-Stream Operators.** Window-to-stream operators convert windows into streams by comparing the contents of the window at the current and previous evaluation episodes. These are based on the window-to-stream operators from CQL, namely **ISTREAM**, **DSTREAM** and **RSTREAM**, denoting **inserted-only**, **deleted-only** and **all-tuples**, respectively. Note that, in cases where there is a **NOW** window followed by an **RSTREAM**, both these operators may be removed, e.g.,
in Q3.

**Operators over Windows or Relations.** Join and aggregation typically operate over extents of definite cardinality, and therefore are performed over windows or relations. Aggregates (such as average) are computed in three phases (*initialise, iterate and evaluate*), each implemented as a separate physical operator, based on the incremental aggregation techniques proposed by Gray *et al.* [GCB+97]. In the TinyDB project [MFHH05], such an approach is shown to be effective in the context of sensor networks as it helps reduce radio traffic and allows the computation of different operands in the algebraic expression to be evaluated separately.

**Operators over Windows, Relations or Streams.** The SELECT and PROJECT operators can be performed over any collection type.

**Operator Properties.** In some cases, properties are associated with certain operators and used by the optimizer to make distributed query planning decisions. These were defined in Section 4.6.1.
Appendix B

Deriving Cost Estimation Models from Operator Implementations

This appendix describes the approach used to derive CEMs used by SNEE, based on the structure of algorithms used to implement QEP operators, summarized from [BGFP09, Bre09]. CEMs are used to estimate the memory, time and energy, three metrics to consider during query optimization. The approach is illustrated using two QEP components as an example, viz., the SP.ACQUIRE operator, and transmit (used by the producer part of the EXCHANGE operator, in the distributed algebra, that encapsulates the transmission of tuples over the radio). The same approach is used for the other operators in the physical and distributed algebra.

Figure B.1 illustrates the algorithm for the SP.ACQUIRE operator in the physical algebra. This operator first senses the attributes in attrSenseList which are to be acquired (according to the schema definition). For example, for the tree stream defined in Figure 4.2, these are the smoke, temperature and rhumidity attributes, respectively. Then, a (possibly complex) predicate expression is applied, and the tuple is discarded if it evaluates to false. Finally, a tuple is created according to projExprList, which may involve removing attributes that are projected out and/or adding new attributes to the tuple, and the tuple is placed in the output queue.

The algorithm for transmit is shown in Figure B.2. In the pseudocode, as many tuples are placed into a radio packet prior to sending as fit in the maximum packet size. If there are insufficient tuples to fill the last packet sent in the evaluation episode, it is padded with NULL values.

The calculations for deriving cost models for SP.ACQUIRE and transmit are
SP_ACQUIRE[attrSenseList, predExpr, projExprList](tick)

dependencies▷CPU, Sensor Board: on; Radio: off

state
sensedValues: array of float size length(attrSenseList)
result: array of float size length(projExprList)+1

begin
  for i=1 to length(attrSenseList):
    sensedValues[i] ← SENSE(typeof(attrSenseList[i]))
  for j=1 to length(projExprList):
    result[j] ← apply(projExprList[j], sensedValues)
  return bagof(result)
end

Figure B.1: Pseudocode for SP_ACQUIRE from SNEEql physical algebra, taken from [BGFP09].

shown in Figures B.3 and B.4 respectively. Memory cost models, denoted by $M_{OP\_NAME}$, are derived from the list of variables in the state clause, which comprise any input and output queues for tuples, as well as other variables, used by the algorithm. The size of each variable, based on its type, is directly summed to estimate total operator memory consumption, inclusive of a fixed overhead. For example, as described in Figure B.3, for the SP_ACQUIRE operator, the memory cost sums up the sizes of the tick variable, the sensed attribute list and the projection expression list.

Time cost models, denoted by $T_{OP\_NAME}$, are derived from the duration of each activity in the algorithm, as reflected in its computational structure (e.g., loops result in parts of the algorithm being repeated several times implying a multiplier on the time taken to execute one pass through the body of the loop). For the SP_ACQUIRE operator, this comprises the duration of taking a sensor reading for each sensed attribute, applying the predicate on the latter, and applying the projection expression for each projected attribute.

Energy cost models, denoted by $E_{OP\_NAME}$, are derived from the time cost models. The duration of each step in the algorithm is multiplied by the power
APPENDIX B. DERIVING OPERATOR COST MODELS

TRANSMIT[ ](tick, child)
1 dependencies ▷ CPU, Radio: on; Sensor Board: off
2 state
3   resultsFromChild ▷ a pointer to
4   block: array of tuple size
5       ⌊(MAX_PACKET_SIZE/sizeof(tuple))⌋
6   packet: array of byte size MAX_PACKET_SIZE
7   i: int
8 begin
9   resultsFromChild ← child.getNext(tick):
10  i ← 1
11  for tuple ∈ resultsFromChild:
12     block[i] ← tuple
13     i ← i + 1
14     if i = length(block)+1:
15        then ▷ we have a full block
16           packet ← convert(block)
17           SEND(packet, sizeof(block))
18           i ← 1
19     if i > 1:
20        then ▷ the last block is not full, so pad it
21           for j=i to length(block):
22              block[j] ← NULL
23           packet ← convert(block)
24           SEND(packet, sizeof(block))
25 return end

Figure B.2: Pseudocode for Transmit method, used for sending tuples over the radio, taken from [BGFP09].

consumption associated with the activity. For example, for the SP_ACQUIRE operator, this involves powering up the sensor board and taking sensor readings, followed by the execution of some CPU instructions. transmit involves switching the radio on and transmitting one or more packet(s) using the required power setting, depending on the distance to be covered. Each addend for the energy consumption estimation, is computed by making reference to terms in the time expressions using the form Addend(termId), which denotes that the expression should be textually substituted with the identified addend in the time expression.

Throughout the dissertation, Memory(op), Time(op) and Energy(op) are used to denote the memory, time and energy costs of a given operator op respectively.
\( M_{\text{ACQUIRE}}(\text{attrSenseList}, \text{predExpr}, \text{projExpList}) = \) (B.1)
\[
M_{\text{ACQ\_OVERHEAD}} + \text{sizeof}(\text{tick}) + \\
\text{sum}\{\text{sizeof}(s) \mid s \in \text{attrSenseList}\} + \\
\text{sum}\{\text{sizeof}(a) \mid a \in \text{projExpList}\}
\]

\( T_{\text{ACQUIRE}}(\text{attrSenseList}, \text{predExpr}, \text{projExpList}, \text{sensedValues}) = \) (B.2)
\[
T_{\text{ACQ\_OVERHEAD}} + \\
(\text{sum}\{T_{\text{SENSE}}(\text{typeof}(s)) \mid s \in \text{attrSenseList}\}) + \\
(T_{\text{APPLY}}(\text{A\_P}, \text{sensedValues}) \ast \\
\text{count}\{p \mid p \in \text{predExpr} \land \text{atomic}(p)\}) + \\
(\text{sum}\{T_{\text{APPLY}}(\text{A\_E}, \text{sensedValues}) \ast \\
\text{count}\{e \mid a \in \text{projExpList} \land e \in a \land \text{atomic}(e)\}\})
\]

\( E_{\text{ACQUIRE}}(\text{attrSenseList}, \text{predExpr}, \text{projExpList}) = \) (B.7)
\[
(\text{E\_SENSE} \ast \text{Addend}[B.4]) + \\
E_{\text{PROCESS}} \ast \\
(\text{Addend}[B.3] + \\
\text{Addend}[B.5] + \\
\text{Addend}[B.6] \ast \text{selectivity(\text{predExpr})})
\]

Figure B.3: Cost estimation models for SP\_ACQUIRE [BGFP09]

A complete set of CEMs for the SNEEq\_ distributed algebra that is used in in subsequent chapters is available in [Bre09].
\[ M_{\text{transmit}(\text{tick}, \text{child})} = \]
\[ M_{\text{TRANS\_OVERHEAD}} + \text{sizeof(pointer)} + \]
\[ (\text{sizeof}(\text{tuple}) \times \lceil \text{MAX\_PACKET\_SIZE/\text{sizeof}(\text{tuple})} \rceil) + \]
\[ \text{MAX\_PACKET\_SIZE} + \text{sizeof}(i) \]

\[ T_{\text{transmit}(\text{tick}, \text{child})} = \]
\[ T_{\text{TRANS\_OVERHEAD}} + \]
\[ (T_{\text{child}.\text{getNext}(\text{tick})}) + (\]
\[ T_{\text{RX\_OVERHEAD}} + \]
\[ T_{\text{TX\_OVERHEAD}} + \]
\[ (T_{\text{TX\_BYTE}} \times \text{sizeof(block)}) \times \]
\[ \lceil \text{count}\{\text{tuple} \in \text{resultsFromChild}\}/\text{length(block)} \rceil \]

\[ E_{\text{transmit}(\text{tick}, \text{child})} = \]
\[ (E_{\text{PROCESS}} \times \text{Addend}[B.10]) + \]
\[ (E_{\text{PROCESS}} \times \text{Addend}[B.11]) + ((\]
\[ (E_{\text{PROCESS}} + E_{\text{RX}}) \times \text{Addend}[B.12]) + \]
\[ (E_{\text{IDLE}} + E_{\text{TX}}) \times \text{Addend}[B.13]) + \]
\[ ((E_{\text{IDLE}} + E_{\text{TX}}) \times \text{Addend}[B.14])) \times \]
\[ \lceil \text{count}\{\text{tuple} \in \text{resultsFromChild}\}/\text{length(block)} \rceil \]

Figure B.4: Cost estimation models for transmit [BGFP09]
Appendix C

QoS-aware Where-scheduling: Supplementary Material

This appendix contains supplementary material for the QoSA-SNEE where-scheduling step, described in Section 6.3. Section C.1 presents an illustrated example to show how the neighbourhood function used by NOMADm may move an operator instance to generate a neighbouring DAF (and also, the impact on other operator instances in the DAF). Section C.2 gives more details about the post-processing stage of QoSA-SNEE where-scheduling.

C.1 Neighbour Generation Example

Figure C.1 illustrates how moving an operator instance may indirectly result in other operator instances being moved as well, based on the example partial DAF for Q3 in Figure 6.19. The thick dotted lines denote routing tree nodes, upon which the operator instance tree of the DAF is overlaid. In the example, AGGR\_MERGE\_10 is a non-location-sensitive operator, selected to be moved to a random valid confluence site, resulting in it being moved from site 0 to site 10. As a side-effect, AGGR\_INIT\_22 and AGGR\_INIT\_10 are also shifted down, in order to maintain the confluence constraint.

C.2 Post-processing Example

The post-processing phase of QoSA-SNEE where-scheduling is responsible for, firstly, removing redundant operator instances, and subsequently, inserting the
Figure C.1: Neighbour generation: Moving an operator instance to a different routing tree site.
EXCHANGE constituents (consumer, producer and relay) between operator instances in the DAF where radio communication needs to take place.

Operator instances are deemed to be redundant if there is more than one instance of the same operator situated at the same site that has the same parent operator instance. In this case, they are merged into a single operator instance. For example, in Figure C.1(b), the AGGR_INIT_22 and AGGR_INIT_10 on site 10 are merged into a single operator instance, as AGGR_INIT_24 and AGGR_INIT_4 on site 0. If an AGGR_MERGE operator instance and its parent operator instance are on the same site, the AGGR_MERGE operator instance is also removed. This is the case with AGGR_MERGE_12 in Figure C.1(b). The resulting DAF is shown in Figure C.2.

The final step in QoS-SNEE where-scheduling is to insert exchange components into the DAF. This involves identifying edges in the DAF operator instance tree where the source sourceOpInst and destination destOpInst operator instances are on different routing tree sites, e.g., the edge between SP_ACQUIRE_24 and AGGR_INIT_0. A producer is inserted as a parent of sourceOpInst at sourceOpInst.Site, a consumer is inserted as child of destOpInst at destOpInst.Site,
Figure C.3: Post processing: DAF in Figure C.2 with EXCHANGE constituents inserted and fragments demarcated.
and relays are placed between the producer and consumer at any intermediate routing tree sites. Figure C.3 shows the final result of the post-processing phase for the example in Figure C.1(b).

Note that the DAF in Figure C.3 does not yield such a favourable value for the QoS objective functions as the one shown in Figure 6.24. Less data is transmitted over the radio in the optimal DAF as the AGGR.MERGE operator instances, which are data reducing as they merge two tuples into one, are at the deepest confluence sites possible. The AGGR.INIT operator instances are data-increasing, as the output tuple type has one more attribute than the input tuple type, as the sensor readings are decomposed into \( \langle \text{count}, \text{sum} \rangle \) pairs. In each case, these are placed on the same site as the AGGR.MERGE, leading to an overall decrease in data transmitted over the radio.
Appendix D

QoS-aware When-scheduling: Supplementary Material

This appendix contains supplementary material for the QoSA-SNEE when-scheduling step, described in Section 6.4. Section D.1 presents some background on geometric programs [BKVH07], as these are the form of the when-scheduling constrained optimization problems that are generated, and cvx [GB09], the external optimizer used to solve these problems. Section D.2 gives examples of how expressions that constitute the constrained optimization problem are derived from an example DAF, and also how, in some cases, expressions are transformed into a form suitable for the cvx optimizer.

D.1 Geometric Programming

The constrained optimization problems generated during the QoSA-SNEE step were found to be representable (in some cases with minor adaptations) as geometric programs [GP] [BKVH07], a property that informed the selection of the external solver to be used. A [GP] is a mathematical constrained optimization problem with a specific form. Let a monomial be a function of the form $f(x) = cx_1^{a_1}x_2^{a_2}...x_n^{a_n}$ where $c > 0$ and $a_i \in \mathbb{R}$, and a posynomial be the sum of one or more monomials [BKVH07]. A standard form [GP] has the form:

$$
\begin{align*}
\text{minimize} & \quad f_0(x) \\
\text{subject to} & \quad f_i(x) \leq 1, i = 1, \ldots, m, \\
& \quad g_i(x) = 1, i = 1, \ldots, p,
\end{align*}
$$

(D.1)
where $f_i$ is a posynomial function, $g_i$ is a monomial function, and $x_i$ are the optimization variables, $x_i > 0$ \cite{BKVH07}. $f_0$ is the objective function (i.e., the mathematical expression whose value is to be minimized), and $f_i$ and $g_i$ are the constraint functions (i.e., mathematical expressions that need to meet certain predicates for the values of the variables selected). A GP may be converted to a convex optimization problem by means of variable substitution and transforming the objective and constraint functions \cite{GB09}. The cvx \cite{GB09} solver is a convex programming solver capable of solving GPs and is therefore used by QoSA-SNEE when-scheduling to obtain values for the optimization variables $\alpha$ and $\beta$.

Note that the form of a GP somewhat resembles a SNEE QoS expectation comprising an optimization goal and constraints. When modelling the when-scheduling problem, the optimization goal of the QoS expectations corresponds to $f_0$, and the QoS constraints correspond to some of $f_i$ and $g_i$. There are, however, other constraints that are not specified by the user QoS, that relate to system properties and consistency conditions. These constraints are necessary to ensure that the resulting agenda is valid.

The optimization variables are $\alpha$, which represents the acquisition interval time units (milliseconds are the finest granularity that can be represented in an agenda in the SNEE implementation), and $\beta$, the buffering factor. In GPs, the optimization variables are assumed to be continuous. However, in this case, integer solutions are required for the optimization variables. It is not possible to merely round the solutions obtained to the nearest integer, as this may lead to an infeasible assignment of variables. Therefore, more specifically, the problem to be solved is a mixed-integer geometric program (MIGP) \cite{BKVH07}, which is essentially a GP that contains variables that have integer solutions.

The approach taken to solve the MIGP is to implement a wrapper over cvx that first solves the GP relaxation of the MIGP \cite{BKVH07}, which is the original GP without the integer constraints, thus obtaining real values for each variable, denoted $\bar{\alpha}$ and $\bar{\beta}$. Then, the value of each variable is fixed by adding an equality constraint in which the value is rounded to the nearest integer. Four instances of
the original [GP] problem are spawned, with additional equality constraints

\[ \alpha = \text{RoundDown}(\bar{\alpha}); \beta = \text{RoundDown}(\bar{\beta}) \] (D.2)

\[ \alpha = \text{RoundDown}(\bar{\alpha}); \beta = \text{RoundUp}(\bar{\beta}) \] (D.3)

\[ \alpha = \text{RoundUp}(\bar{\alpha}); \beta = \text{RoundDown}(\bar{\beta}) \] (D.4)

\[ \alpha = \text{RoundUp}(\bar{\alpha}); \beta = \text{RoundUp}(\bar{\beta}) \] (D.5)

The four arising [GPs] are solved, and the feasible solution that yields the best objective value function is the one used. This simple approach is adequate in this context because there are only two variables in the [MIGPs] that are generated, and both are required to have integer solutions.

### D.2 Modelling the when-scheduling problem

This section gives more details about how expressions are derived to model certain system properties and QoS metrics. The example used is based on the DAF shown in Figure 6.24.

#### D.2.1 Modelling the Memory Constraint (C1)

To give an example, for the DAF in Figure 6.24, for site 22, given that according to the output of the CEMs, \( \text{Memory}_{\beta}(F2,22) = 16 + 16\beta \) bytes and \( \text{Memory}_{\beta}(F2\text{-}\text{producer},22) = 32 + 16\beta \) bytes, the total memory consumption at the site will be \( m_i = 48 + 32\beta \) bytes. For Mica2 mote hardware with 4K RAM [Tec09b], \( M_i = 4096 \) bytes. The constraint that would be added to the constrained optimization problem to check that memory is not exceeded for this site is therefore:

\[ 48 + 32\beta \leq 4096 \] (D.6)

It is noted that it is straightforward to rewrite this constraint in the form \( f_i(x) \leq 1 \), where \( f_i(x) \) is a posynomial expression, so that it is compliant with the standard form of inequality constraints for [MIGPs] i.e.,

\[ -4047 + 32\beta \leq 1 \] (D.7)

Given that for checking memory, a constraint is generated for each site, there
may be cases when constraints are either duplicated or subsumed by one another. In this case, the constraint is redundant, and can be omitted from the MIGP. For example, suppose that for another site the constraint $48 + 22\beta \leq 4096$ was generated; since $48 + 22\beta < 48 + 32\beta$, it would not need to be included in the MIGP.

**D.2.2 Modelling the Processing time constraint (C2)**

The processing time $\pi$ has two constituents, the leaf fragment processing time, and the non-leaf fragment processing time, as described in Equation (6.3).

The instances of the leaf fragments of the DAF are assumed to execute concurrently, as is scheduled by the Build-Agenda method described in Figure 5.10. If instances of different leaf fragments are assigned to the same site, the leaf fragment instances are scheduled one after the other. The duration of the leaf fragments $\pi_L$ is therefore assumed to be the maximum duration of all the leaf fragment instances assigned to any site, i.e.,

$$\pi^L = \max_{i \in R_Q \cdot \text{sites}} (\pi^L_i) \quad (D.8)$$

where the total time cost of the leaf fragments at a site $i$ is equal to

$$\pi^L_i = \sum_{\text{fragInst} \in i \cdot \text{leafFragInstances}} \text{Time}_\beta(\text{fragInst}) \quad (D.9)$$

Tasks corresponding to non-leaf fragments and exchange operator instances in the DAF are scheduled according to the Build-Agenda method using the precedence constraints implied by the shape of the DAF fragment tree, which dictate, e.g., that the children of a fragment $F$ need to be scheduled earlier in the agenda than the task corresponding to $F$ itself. As a consequence, the Build-Agenda method generally schedules most of the non-leaf fragment tasks in a sequential manner (although, in less common cases, when precedence constraints allow, they may be scheduled concurrently). As an approximation, $\pi^N$ is assumed to be the sum of the duration of all the non-leaf fragment and communication
APPENDIX D. QOS-AWARE WHEN-SCHEDULING

tasks in the DAF, i.e.,

\[
\pi_N = \sum_{\text{fragInst} \in \mathcal{D}_Q \text{.nonLeafFragInstances}} \text{Time}_\beta(\text{fragInst}) + \sum_{\text{exchOpInst} \in \mathcal{D}_Q \text{.exchOpInstances}} \text{Time}_\beta(\text{exchOpInst})
\]

(D.10)

This approach may result in a slight over-approximation of \(\pi\), but this is acceptable as it avoids having to construct an agenda to compute \(\pi\) (as done in \text{FG-SNEE} when-scheduling). With this approach, there is no risk of underestimating \(\pi\), which would lead to an invalid agenda.

Note that a distinction between leaf fragments and tasks corresponding to non-leaf fragments, is that for the leaf fragments, only a single epoch’s worth of tuples is processed in the last epoch of the agenda evaluation episode, so \(\pi_L\) is a constant and is not dependent on \(\beta\). In contrast, for non-leaf fragments and exchange operator instances, tuples for \(\beta\) epochs are processed per agenda evaluation episode, so \(\pi_N\) is a linear expression in terms of \(\beta\).

For example, for the DAF in Figure 6.24 the sole leaf fragment is F2. Therefore, for the leaf fragments in the DAF

\[
\pi^L = \min(\text{Time}_\beta(F2,22), \text{Time}_\beta(F2,10), \text{Time}_\beta(F2,24), \text{Time}_\beta(F2,4))
\]

\[
= \min(40, 40, 40, 40)
\]

\[
= 40
\]

(D.11)
The expression corresponding to the non-leaf fragments, $\pi^N$, is:

$$\pi^N = \text{Time}_\beta(\text{comm22-10}) + \text{Time}_\beta(\text{F1,10}) + \text{Time}_\beta(\text{comm24-12}) + \text{Time}_\beta(\text{comm4-12}) + \text{Time}_\beta(\text{comm12-0}) + \text{Time}_\beta(\text{F0,0})$$

$$= (25 + 5\beta) + (5 + \beta) + (25 + 5\beta) + (25 + 5\beta) + (25 + 5\beta)$$

$$= 135 + 27\beta$$

(D.12)

Therefore, the total processing time $\pi$ is:

$$\pi = \pi^L + \pi^N$$

$$= (40) + (135 + 27\beta)$$

$$= 175 + 57\beta$$

(D.13)

Therefore, for the DAF in question, the acquisition interval must be greater than or equal to $145 + 57\beta$ for the length of the last epoch in an agenda evaluation episode not to exceed $\alpha$ (i.e., to ensure that the agenda is valid).

### D.2.3 Modelling the Total Energy QoS Metric

For example, for the DAF in Figure 6.24, the energy consumption of site 22 over a agenda evaluation episode, measured in milliJoules, is expressed as:

$$\hat{e}_{22} = \text{Energy}_\beta(\text{F1,22}) + \text{Energy}_\beta(\text{tx22-10})$$

$$= (0.01\beta + 0.03) + (0.09\beta + 0.5)$$

$$= 0.1\beta + 0.53$$

(D.14)
For a single unit of time, this becomes:

\[ e_{22} = \frac{e_i}{\alpha \beta} = 0.1\beta + 0.53 \]

Summing the energy consumption for all the sites (excluding site 0, which is assumed to be tethered) to obtain total network energy \( e_{\text{ins}} \):

\[ e_{\text{ins}} = e_{22} + e_{10} + e_{24} + e_4 + e_{12} = \frac{0.1\beta + 0.53}{\alpha \beta} + \ldots + \frac{0.18\beta + 0.64}{\alpha \beta} = 0.67\beta + 2.22 \]

\[ = \frac{0.67}{\alpha} + \frac{2.22}{\alpha \beta} = 0.67\alpha^{-1} + 2.22\alpha^{-1}\beta^{-1} \]

(D.16)

It is noted that the resulting expression is of posynomial form.

### D.2.4 Modelling the Lifetime QoS Metric

The lifetime QoS metric differs from the other metrics in that, rather than the a low value being desirable (as is the case with delivery time), it is a metric for which the higher the value, the better the performance is considered to be.

Therefore, in order to map it to a standard form \([\text{GP}]\) for \(\text{cvx}\), which is a minimization problem over a posynomial expression, the reciprocal of the lifetime, denoted \(\lambda^{-1}\), is used as the objective function to be minimized. The \(\lambda_i^{-1}\) for a single site is therefore

\[ \lambda_i^{-1} = \frac{e_i}{E_i} \]

(D.17)

and for the whole network, this becomes

\[ \lambda^{-1} = \max_{i \in RQ\text{-sites}}(\lambda_i^{-1}) \]

(D.18)

This is the reciprocal of the network lifetime.

For example, for the [DAP] in Figure 6.24, assuming a battery of 31320 J (two
Lithium AA batteries) per site, for site 22 the $\lambda^{-1}_{22}$ expression is derived as follows:

$$
\lambda^{-1}_{22} = \frac{v_{22}}{E_{22}} \\
= \frac{(0.1\beta + 0.53)}{31320000} \\
= 0.1\beta + 0.53 \\
= \frac{0.1\beta}{31320000} + \frac{0.53}{31320000} \alpha \beta^{-1} \\
= \frac{0.1}{31320000} \alpha^{-1} + \frac{0.53}{31320000} \alpha^{-1} \beta^{-1}
$$

(D.19)

The resulting expression, to be minimized by \texttt{cvx}, is an expression in posynomial form.

In order to obtain the overall network lifetime, the expression becomes:

$$
\lambda^{-1} = \text{Max}_{i \in RQ \text{-sites}}(\lambda^{-1}_i) \\
= \text{Max}(\lambda^{-1}_{22}, \lambda^{-1}_{10}, \lambda^{-1}_{24}, \lambda^{-1}_4, \lambda^{-1}_{12}) \\
= \text{Max}\left(\frac{0.1}{31320000} \alpha^{-1} + \frac{0.53}{31320000} \alpha^{-1} \beta^{-1}, \ldots, \frac{0.18}{31320000} \alpha^{-1} + \frac{0.64}{31320000} \alpha^{-1} \beta^{-1}\right)
$$

(D.20)

Note that because there is limited support in \texttt{cvx} for \texttt{Min} and \texttt{Max} functions, it is necessary to make separate invocations to the solver and the coalesce the solutions obtained. In other words, for the example above, when specifying lifetime as the optimization goal, rather than solving

$$
\text{Minimize } \text{Max}\left(\frac{0.1}{31320000} \alpha^{-1} + \frac{0.53}{31320000} \alpha^{-1} \beta^{-1}, \ldots, \frac{0.18}{31320000} \alpha^{-1} + \frac{0.64}{31320000} \alpha^{-1} \beta^{-1}\right)
$$

(D.21)

the problem is decomposed into separate MIGPs as follows:

$$
\text{Max}\left(\text{Minimize } \frac{0.1}{31320000} \alpha^{-1} + \frac{0.53}{31320000} \alpha^{-1} \beta^{-1}, \ldots, \frac{0.18}{31320000} \alpha^{-1} + \frac{0.64}{31320000} \alpha^{-1} \beta^{-1}\right)
$$

(D.22)
Appendix E

Evaluation: Scenarios Used for Experiments

Scenario 1

Query

```
SELECT RSTREAM AVG(anow.x) as qx
    FROM A[NOW] anow,
        B[NOW] bnow
    WHERE anow.x=bnow.x;
```

Network

![Network Diagram]

rValue=50.
Physical Schema
A \{19, 22, 2, 0, 21, 10, 18\}
B \{18, 26, 9, 15, 4, 17\}

39% of nodes in network are sources.

Scenario 2
Query

\[
\text{SELECT RSTREAM anow.x as qx} \\
\text{FROM A[NOW] anow;}
\]

Network

\[
\begin{array}{cccccccc}
3 & 2 & 1 & 0 & -1 & -2 & -3 & -4 \\
\hline
+ & + & + & + & + & + & + & + \\
-1 & + & + & + & + & + & + & + \\
-2 & + & + & + & + & + & + & + \\
-3 & + & + & + & + & + & + & + \\
-4 & + & + & + & + & + & + & + \\
\end{array}
\]

rValue=54.

Physical Schema
A \{29, 13, 25, 27, 26, 20, 22, 21, 17, 2, 10, 19\}

40% of nodes in network are sources.

Scenario 3
Query

\[
\text{SELECT RSTREAM AVG(sq1.sq1x) as qx} \\
\text{FROM (SELECT AVG(anow.x) as sq1x} \\
\text{FROM A[NOW] anow,}
\]

\[
\text{...}
\]
APPENDIX E. EVALUATION: SCENARIOS USED FOR EXPERIMENTS

\[
\begin{aligned}
\text{B} \[\text{NOW}\] & \text{bnow} \\
& \text{WHERE } \text{anow.x=bnow.x} \text{ sq1,} \\
\text{C} \[\text{NOW}\] & \text{cnow} \\
& \text{WHERE } \text{sq1.sq1x=cnow.x;}
\end{aligned}
\]

Network

\[
\begin{array}{|c|}
\hline
\text{rValue}=25. \\
\hline
\end{array}
\]

Physical Schema

A \{2,13,15,23\}
B \{23,17,4,14\}
C \{14,18,5\}

30\% of nodes in network are sources.

Scenario 4

Query

\[
\begin{aligned}
\text{SELECT RSTREAM AVG(\text{anow.x}) as qx} \\
& \text{FROM A} \[\text{NOW}\] \text{anow,} \\
& (\text{SELECT sq2.sq2x as sq1x} \\
& \text{FROM (SELECT AVG(sq3.sq3x) as sq2x} \\
& \text{FROM (SELECT bnow.x as sq3x} \\
& \text{FROM B} \[\text{NOW}\] \text{bnow,} \\
& \text{C} \[\text{NOW}\] \text{cnow} \\
& \text{WHERE bnow.x=cnow.x}) \text{ sq3,}
\end{aligned}
\]
Network

```
\begin{align*}
\text{(SELECT dnow.x as sq4x} \\
\text{FROM D\([\text{NOW}]\) dnow ) sq4} \\
\text{WHERE sq3.sq3x=sq4.sq4x) sq2 ) sq1} \\
\text{WHERE anow.x=sq1.sq1x;}
\end{align*}
```

\textbf{Physical Schema}

\begin{align*}
\text{A \{19,26,16,13\}} \\
\text{B \{13,23,3,21,14\}} \\
\text{C \{14,5,27,24\}} \\
\text{D \{24,11,8,1\}}
\end{align*}

44\% of nodes in network are sources.

\textbf{Scenario 5}

\textbf{Query}

```
\begin{align*}
\text{SELECT RSTREAM sq1.sq1x as qx} \\
\text{FROM (SELECT AVG(anow.x) as sq1x} \\
\text{FROM A\([\text{NOW}]\) anow,} \\
\text{B\([\text{NOW}]\) bnow} \\
\text{WHERE anow.x=bnow.x) sq1,} \\
\text{(SELECT AVG(sq3.sq3x) as sq2x} \\
\text{FROM (SELECT AVG(sq4.sq4x) as sq3x}\}
\end{align*}

\text{rValue=5.}
APPENDIX E. EVALUATION: SCENARIOS USED FOR EXPERIMENTS

FROM (SELECT cnow.x as sq4x
FROM C[NOW] cnow,
D[NOW] dnow
WHERE cnow.x=dnow.x) sq4 ) sq3,
E[NOW] enow
WHERE sq3.sq3x=enow.x) sq2
WHERE sq1.sq1x=sq2.sq2x;

Network

\[ rValue = 21. \]

Physical Schema

A \{18,27,26,22\}
B \{22,10,17,11,23\}
C \{23,7,20,15\}
D \{15,25,0,5,13\}
E \{13,1,12,16\}

57% of nodes in network are sources.

Scenario 6

Query

SELECT RSTREAM AVG(sq1.sq1x) as qx
FROM (SELECT anow.x as sq1x
FROM A[NOW] anow ) sq1,
(SELECT AVG(bnow.x) as sq2x
FROM B[NOW] bnow,
C[NOW] cnow
WHERE bnow.x=cnow.x) sq2
WHERE sq1.sq1x=sq2.sq2x;

Network

rValue=1.

Physical Schema

A \{0,8,21,13,10,19\}
B \{19,4,23,28,5,18,24\}
C \{24,9,7,2,14,22\}

56\% of nodes in network are sources.

Scenario 7

Query

SELECT RSTREAM sq1.sq1x as qx
FROM (SELECT AVG(sq2.sq2x) as sq1x
FROM (SELECT AVG(sq3.sq3x) as sq2x
FROM (SELECT AVG(anow.x) as sq3x
FROM A[NOW] anow ) sq3 ) sq2 ) sq1,
B[NOW] bnow
WHERE sq1.sq1x=bnow.x;
Network

\[ rValue=20. \]

Physical Schema

A \{29, 21, 4, 15, 28, 17, 11, 16, 6, 3, 13, 10, 19, 24\}
B \{24, 12, 25, 20, 26, 27, 18, 22, 14, 2, 23, 9, 5, 0\}

89% of nodes in network are sources.

Scenario 8

Query

\[
\text{SELECT RSTREAM AVG(anow.x) as qx}
\text{FROM A[NOW] anow,}
\text{B[NOW] bnow}
\text{WHERE anow.x=bnow.x;}
\]
rValue=9.

Physical Schema

A \{25, 24, 5, 17, 19, 3, 18, 10, 23, 12\}
B \{12, 9, 28, 6, 1, 27, 22, 4, 26, 21\}

63% of nodes in network are sources.

Scenario 9

Query

SELECT RSTREAM anow.x as qx
FROM A[NOW] anow;
APPENDIX E. EVALUATION: SCENARIOS USED FOR EXPERIMENTS

rValue=32.

Physical Schema

A \{0,4,26,8,22,16,1,24,11,3,19,28,25,12,6,21,18,2,14,10,7,9\}

73\% of nodes in network are sources.

Scenario 10

Query

\[
\begin{align*}
\text{SELECT} & \quad \text{RSTREAM} \quad \text{AVG}(\text{sq1.sq1x}) \quad \text{as} \quad qx \\
\text{FROM} & \quad (\text{SELECT} \quad \text{anow.x} \quad \text{as} \quad \text{sq1x} \\
& \quad \text{FROM} \quad \text{A[NOW]} \quad \text{anow}, \\
& \quad \text{B[NOW]} \quad \text{bnow} \\
& \quad \text{WHERE} \quad \text{anow.x}=\text{bnow.x}) \quad \text{sq1,} \\
& \quad (\text{SELECT} \quad \text{AVG}(\text{cnow.x}) \quad \text{as} \quad \text{sq2x} \\
& \quad \text{FROM} \quad \text{C[NOW]} \quad \text{cnow}, \\
& \quad (\text{SELECT} \quad \text{AVG}(\text{sq4.sq4x}) \quad \text{as} \quad \text{sq3x} \\
& \quad \text{FROM} \quad (\text{SELECT} \quad \text{dnow.x} \quad \text{as} \quad \text{sq4x} \\
& \quad \text{FROM} \quad \text{D[NOW]} \quad \text{dnow}, \\
& \quad \text{E[NOW]} \quad \text{enow} \\
& \quad \text{WHERE} \quad \text{dnow.x}=\text{enow.x}) \quad \text{sq4 } ) \quad \text{sq3} \\
& \quad \text{WHERE} \quad \text{cnow.x}=\text{sq3.sq3x}) \quad \text{sq2} \\
& \quad \text{WHERE} \quad \text{sq1.sq1x}=\text{sq2.sq2x};
\end{align*}
\]

Network
rValue=11.

Physical Schema
A {8, 16, 29}
B {29, 7, 12}
C {12, 13, 6}
D {6, 11, 14}
E {14, 23, 21}

36% of nodes in network are sources.

Scenario 11

Query
SELECT RSTREAM AVG(sq1.sq1x) as qx
FROM (SELECT AVG(anow.x) as sq1x
    FROM A[NOW] anow,
    B[NOW] bnow
    WHERE anow.x=bnow.x) sq1;

Network

rValue=23.

Physical Schema
A {3, 8, 19, 0, 22, 11, 23, 17, 15}
B {15, 26, 10, 9, 25, 28, 7, 5, 12}
56% of nodes in network are sources.

**Scenario 12**

**Query**

```sql
SELECT RSTREAM anow.x as qx
FROM A[NOW] anow;
```

**Network**

![Network Diagram]

rValue=8.

**Physical Schema**

A {27, 25, 22, 10, 7, 26, 6, 5, 3, 16, 18, 13, 0, 28, 15, 17, 24}

56% of nodes in network are sources.

**Scenario 13**

**Query**

```sql
SELECT RSTREAM anow.x as qx
FROM A[NOW] anow;
```
Network

\[
\begin{array}{ccccccccccccc}
-12 & -10 & -8 & -6 & -4 & -2 & 0 & 2 & 4 & 6 & 8 & 10 & 12 \\
\end{array}
\]

rValue=18.

Physical Schema

A \{19,12,4,7,21,26,8,23,25,16,11,18,10\}

43\% of nodes in network are sources.

Scenario 14

Query

SELECT RSTREAM AVG(anow.x) as qx
FROM A[NOW] anow,
    (SELECT AVG(sq2.sq2x) as sq1x
    FROM (SELECT AVG(bnow.x) as sq2x
    FROM B[NOW] bnow ) sq2,
    C[NOW] cnow
    WHERE sq2.sq2x=cnow.x) sq1
WHERE anow.x=sq1.sq1x;
rValue=19.

Physical Schema

A \{0,17,29,22,10,16,7,2,11,26\}
B \{26,12,5,8,19,21,9,28,25,15\}
C \{15,4,18,20,1,23,14,13,6,27\}

93% of nodes in network are sources.

Scenario 15

Query

\[
\text{SELECT RSTREAM sq1.sq1x as qx}
\text{FROM (SELECT AVG(sq2.sq2x) as sq1x}
\text{FROM (SELECT AVG(anow.x) as sq2x}
\text{FROM A[NOW] anow ) sq2,}
\text{(SELECT AVG(sq4.sq4x) as sq3x}
\text{FROM (SELECT AVG(bnow.x) as sq4x}
\text{FROM B[NOW] bnow,}
\text{C[NOW] cnow}
\text{WHERE bnow.x=cnow.x) sq4,}
\text{(SELECT AVG(dnow.x) as sq5x}
\text{FROM D[NOW] dnow,}
\text{E[NOW] enow}
\text{WHERE dnow.x=enow.x) sq5}
\]
`WHERE sq4.sq4x=sq5.sq5x) sq3
WHERE sq2.sq2x=sq3.sq3x) sq1,
F[NOW] fnow
WHERE sq1.sq1x=fnow.x;

Network

\[\text{rValue}=45.\]

Physical Schema

\[\text{A } \{24,7,26\}\]
\[\text{B } \{26,23,16\}\]
\[\text{C } \{16,12,13\}\]
\[\text{D } \{13,18,19\}\]
\[\text{E } \{19,27,10\}\]
\[\text{F } \{10,25,8\}\]

43% of nodes in network are sources.
Appendix F

Fixed-Goal SNQP Evaluation: Breakdown of Results

F.1 6-month Energy Consumption per Node (J)

Results

Figure F.1: 6-month Energy Consumption per Node (J) obtained for each scenario compiled with \( \alpha = 3s \).
**F.1. 6-MONTH ENERGY CONSUMPTION PER NODE (J) RESULTS**

![Graph 1](image1.png)

Figure F.2: 6-month Energy Consumption per Node (J) obtained for each scenario compiled with $\alpha=15s$.

![Graph 2](image2.png)

Figure F.3: 6-month Energy Consumption per Node (J) obtained for each scenario compiled with $\alpha=30s$. 
Figure F.4: 6-month Energy Consumption per Node (J) obtained for each scenario compiled with $\alpha=60s$.

Figure F.5: 6-month Energy Consumption per Node (J) obtained for each scenario compiled with $\alpha=150s$. 
F.1. 6-MONTH ENERGY CONSUMPTION PER NODE (J) RESULTS

Figure F.6: 6-month Energy Consumption per Node (J) obtained for each scenario compiled with $\alpha=300$ s.

Figure F.7: 6-month Energy Consumption per Node (J) obtained for each scenario compiled with $\alpha=600$ s.
Figure F.8: 6-month Energy Consumption per Node (J) obtained for each scenario compiled with $\alpha=1000s$.

Figure F.9: 6-month Energy Consumption per Node (J) obtained for each scenario compiled with $\alpha=3000s$. 
F.2 Lifetime (days) Results

Figure F.10: Lifetime (days) obtained for each scenario compiled with $\alpha=3s$.

Figure F.11: Lifetime (days) obtained for each scenario compiled with $\alpha=15s$. 
Figure F.12: Lifetime (days) obtained for each scenario compiled with $\alpha=30s$.

Figure F.13: Lifetime (days) obtained for each scenario compiled with $\alpha=60s$. 
Figure F.14: Lifetime (days) obtained for each scenario compiled with $\alpha=150s$.

Figure F.15: Lifetime (days) obtained for each scenario compiled with $\alpha=300s$. 
Figure F.16: Lifetime (days) obtained for each scenario compiled with $\alpha=600$s.

Figure F.17: Lifetime (days) obtained for each scenario compiled with $\alpha=1000$s.
F.3. BUFFERING FACTOR RESULTS

Figure F.18: Lifetime (days) obtained for each scenario compiled with $\alpha=3000s$.

F.3 Buffering factor Results

Figure F.19: Buffering factor obtained for each scenario compiled with $\alpha=3s$. 
Figure F.20: Buffering factor obtained for each scenario compiled with $\alpha=15s$.

Figure F.21: Buffering factor obtained for each scenario compiled with $\alpha=30s$. 
F.3. BUFFERING FACTOR RESULTS

Figure F.22: Buffering factor obtained for each scenario compiled with $\alpha=60s$.

Figure F.23: Buffering factor obtained for each scenario compiled with $\alpha=150s$. 
Figure F.24: Buffering factor obtained for each scenario compiled with $\alpha=300s$.

Figure F.25: Buffering factor obtained for each scenario compiled with $\alpha=600s$. 
F.3. BUFFERING FACTOR RESULTS

Figure F.26: Buffering factor obtained for each scenario compiled with $\alpha=1000s$.

Figure F.27: Buffering factor obtained for each scenario compiled with $\alpha=3000s$. 
APPENDIX F. FIXED-GOAL SNQP EVALUATION: BREAKDOWN OF RESULTS

F.4 Delivery Time (s) Results

Figure F.28: Delivery Time (s) obtained for each scenario compiled with $\alpha=3s$.

Figure F.29: Delivery Time (s) obtained for each scenario compiled with $\alpha=15s$. 


F.4. DELIVERY TIME (S) RESULTS

Figure F.30: Delivery Time (s) obtained for each scenario compiled with $\alpha=30s$.

Figure F.31: Delivery Time (s) obtained for each scenario compiled with $\alpha=60s$. 
Figure F.32: Delivery Time (s) obtained for each scenario compiled with $\alpha=150s$.

Figure F.33: Delivery Time (s) obtained for each scenario compiled with $\alpha=300s$. 
Figure F.34: Delivery Time (s) obtained for each scenario compiled with $\alpha=600\text{s}$.

Figure F.35: Delivery Time (s) obtained for each scenario compiled with $\alpha=1000\text{s}$.
Figure F.36: Delivery Time (s) obtained for each scenario compiled with $\alpha=3000s$. 
Appendix G

QoS-aware SNQP Evaluation: Breakdown of Results

G.1 Acquisition interval (s) Results

Graphs for QoS expectations 2, 3, 9, 10, A, B and C have an equality constraint over $\alpha$, i.e., have a fixed acquisition interval, and have been omitted from this section.

Figure G.1: Acquisition interval (s) obtained for each scenario compiled against QoS 1.
### APPENDIX G. QoS-AWARE SNQP EVALUATION: BREAKDOWN OF RESULTS

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Acquisition Interval (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Avg</td>
<td></td>
</tr>
</tbody>
</table>

#### Figure G.2: Acquisition interval (s) obtained for each scenario compiled against QoS 4.

#### Figure G.3: Acquisition interval (s) obtained for each scenario compiled against QoS 5.
G.1. Acquisition Interval (s) Results

Figure G.4: Acquisition interval (s) obtained for each scenario compiled against QoS 6.

Figure G.5: Acquisition interval (s) obtained for each scenario compiled against QoS 7.
G.2 Delivery Time (s) Results

Figure G.7: Delivery Time (s) obtained for each scenario compiled against QoS 1.
G.2. DELIVERY TIME (S) RESULTS

Figure G.8: Delivery Time (s) obtained for each scenario compiled against QoS 2.

Figure G.9: Delivery Time (s) obtained for each scenario compiled against QoS 3.
Figure G.10: Delivery Time (s) obtained for each scenario compiled against QoS 4.

Figure G.11: Delivery Time (s) obtained for each scenario compiled against QoS 5.
Figure G.12: Delivery Time (s) obtained for each scenario compiled against QoS 6.

Figure G.13: Delivery Time (s) obtained for each scenario compiled against QoS 7.
APPENDIX G. QOS-AWARE SNQP EVALUATION: BREAKDOWN OF RESULTS

Figure G.14: Delivery Time (s) obtained for each scenario compiled against QoS 8.

Figure G.15: Delivery Time (s) obtained for each scenario compiled against QoS 9.
G.2. DELIVERY TIME (S) RESULTS

Figure G.16: Delivery Time (s) obtained for each scenario compiled against QoS 10.

Figure G.17: Delivery Time (s) obtained for each scenario compiled against QoS A.
Figure G.18: Delivery Time (s) obtained for each scenario compiled against QoS B.

Figure G.19: Delivery Time (s) obtained for each scenario compiled against QoS C.
G.3 Buffering factor Results

Figure G.20: Buffering factor obtained for each scenario compiled against QoS 1.

Figure G.21: Buffering factor obtained for each scenario compiled against QoS 2.
Figure G.22: Buffering factor obtained for each scenario compiled against QoS 3.

Figure G.23: Buffering factor obtained for each scenario compiled against QoS 4.
Figure G.24: Buffering factor obtained for each scenario compiled against QoS 5.

Figure G.25: Buffering factor obtained for each scenario compiled against QoS 6.
Figure G.26: Buffering factor obtained for each scenario compiled against QoS 7.

Figure G.27: Buffering factor obtained for each scenario compiled against QoS 8.
Figure G.28: Buffering factor obtained for each scenario compiled against QoS 9.

Figure G.29: Buffering factor obtained for each scenario compiled against QoS 10.
Figure G.30: Buffering factor obtained for each scenario compiled against QoS A.

Figure G.31: Buffering factor obtained for each scenario compiled against QoS B.
G.4 6-MONTH TOTAL NETWORK ENERGY CONSUMPTION (J) RESULTS

Figure G.32: Buffering factor obtained for each scenario compiled against QoS C.

G.4 6-month Total Network Energy Consumption (J) Results

Figure G.33: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS 1.
Figure G.34: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS 2.

Figure G.35: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS 3.
Figure G.36: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS 4.

Figure G.37: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS 5.
Figure G.38: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS 6.

Figure G.39: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS 7.
G.4. **6-MONTH TOTAL NETWORK ENERGY CONSUMPTION (J) RESULTS**

Figure G.40: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS 8.

Figure G.41: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS 9.
APPENDIX G. QOS-AWARE SNQP EVALUATION: BREAKDOWN OF RESULTS

Figure G.42: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS 10.

Figure G.43: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS A.
G.4. 6-MONTH TOTAL NETWORK ENERGY CONSUMPTION (J) RESULTS

Figure G.44: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS B.

Figure G.45: 6-month Total Network Energy Consumption (J) obtained for each scenario compiled against QoS C.
G.5 Lifetime (days) Results

Figure G.46: Lifetime (days) obtained for each scenario compiled against QoS 1.

Figure G.47: Lifetime (days) obtained for each scenario compiled against QoS 2.
G.5. LIFETIME (DAYS) RESULTS

Figure G.48: Lifetime (days) obtained for each scenario compiled against QoS 3.

Figure G.49: Lifetime (days) obtained for each scenario compiled against QoS 4.
Figure G.50: Lifetime (days) obtained for each scenario compiled against QoS 5.

Figure G.51: Lifetime (days) obtained for each scenario compiled against QoS 6.
Figure G.52: Lifetime (days) obtained for each scenario compiled against QoS 7.

Figure G.53: Lifetime (days) obtained for each scenario compiled against QoS 8.
Figure G.54: Lifetime (days) obtained for each scenario compiled against QoS 9.

Figure G.55: Lifetime (days) obtained for each scenario compiled against QoS 10.
Figure G.56: Lifetime (days) obtained for each scenario compiled against QoS A.

Figure G.57: Lifetime (days) obtained for each scenario compiled against QoS B.
APPENDIX G. QoS-AWARE SNQP EVALUATION: BREAKDOWN OF RESULTS

Figure G.58: Lifetime (days) obtained for each scenario compiled against QoS C.

G.6 6-month Energy Consumption per Node (J) Results

Figure G.59: 6-month Energy Consumption per Node (J) obtained for each scenario compiled against QoS 1.
G.6. 6-MONTH ENERGY CONSUMPTION PER NODE (J) RESULTS

Figure G.60: 6-month Energy Consumption per Node (J) obtained for each scenario compiled against QoS 2.

Figure G.61: 6-month Energy Consumption per Node (J) obtained for each scenario compiled against QoS 3.
Figure G.62: 6-month Energy Consumption per Node (J) obtained for each scenario compiled against QoS 4.

Figure G.63: 6-month Energy Consumption per Node (J) obtained for each scenario compiled against QoS 5.
G.6. 6-MONTH ENERGY CONSUMPTION PER NODE \((J)\) RESULTS

Figure G.64: 6-month Energy Consumption per Node \((J)\) obtained for each scenario compiled against QoS 6.

Figure G.65: 6-month Energy Consumption per Node \((J)\) obtained for each scenario compiled against QoS 7.
Figure G.66: 6-month Energy Consumption per Node (J) obtained for each scenario compiled against QoS 8.

Figure G.67: 6-month Energy Consumption per Node (J) obtained for each scenario compiled against QoS 9.
G.6. 6-MONTH ENERGY CONSUMPTION PER NODE (J) RESULTS

Figure G.68: 6-month Energy Consumption per Node (J) obtained for each scenario compiled against QoS 10.

Figure G.69: 6-month Energy Consumption per Node (J) obtained for each scenario compiled against QoS A.
Figure G.70: 6-month Energy Consumption per Node (J) obtained for each scenario compiled against QoS B.

Figure G.71: 6-month Energy Consumption per Node (J) obtained for each scenario compiled against QoS C.
Appendix H

QoS-aware SNQP Evaluation: Example QEPs

This section presents routing trees, DAFs and agendas generated by QoSA-SNEE for some of the scenarios and QoS expectations used in the experiments from Section 7.2. Section H.1 presents QEPs for scenario 5, which is characterized by having a relatively dense network with nodes in close proximity to each other, whereas Section H.2 presents QEPs for scenario 6, which is characterized by having a network with nodes distributed relatively sparsely.

H.1 Scenario 5

Figure H.1: Routing tree for scenario 5 compiled against QoS 2.
Figure H.2: Routing tree for scenario 5 compiled against QoS 5.

Figure H.3: DAF for scenario 5.
Figure H.5: Agenda for scenario 5 compiled against QoS 5.
Figure H.4: Agenda for scenario 5 compiled against QoS 2.
H.2 Scenario 6

Figure H.6: Routing tree for scenario 6 compiled against QoS 1 and QoS 10.
Figure H.7: Routing tree for scenario 6 compiled against QoS 6.
Figure H.8: DAF for scenario 6 compiled against QoS 1 and QoS 10.
Figure H.9: DAF for scenario 6 compiled against QoS 6.
Figure H.10: Agenda for scenario 6 compiled against QoS 1.
Figure H.11: Agenda for scenario 6 compiled against QoS 6.
Figure H.12: Agenda for scenario 6 compiled against QoS 10.
Bibliography


BIBLIOGRAPHY


[SHBIN06] Bianca Schroeder, Mor Harchol-Balter, Arun Iyengar, and Erich M. Nahum. Achieving class-based qos for transactional workloads. In Liu et al. [LRWZ06], page 153.


