Enhancing sensor network programming: Extending TinyDB with HAVING and aggregation, and investigating TinyDB reliability

Master thesis
60 credits

Leif Morten Kofoed

01. November 2007
Preface

The work with this master thesis commenced December 2006 and finished autumn 2007. I have written this thesis at the research group Distributed Multimedia Systems (DMMS).

First I would like to thank my previous employer Norges Bank, for giving me a paid leave of absence for 18 months to accomplish this study. Without it, this thesis would never have been written. Furthermore, I would like to thank my supervisors Associate Professor Ellen Munthe-Kaas and PhD student Jarle Søberg for their indispensable help. They gave me valuable advices, both on the technical and the linguistic part of this thesis. Finally, my wife Gerd and our two lovely children Espen and Emil have shown a great understanding and been patience throughout my work with this thesis.

Lillehammer 01. November 2007

Leif Morten Kofoed
Contents

PREFACE .......................................................................................................................................... III

CONTENTS ....................................................................................................................................... IV

LIST OF FIGURES............................................................................................................................. X

ABSTRACT .................................................................................................................................... XIII

1 INTRODUCTION ......................................................................................................................1

1.1 PROBLEM STATEMENT .............................................................................................................2

1.2 THESIS OUTLINE .......................................................................................................................3

2 BACKGROUND ........................................................................................................................4

2.1 WIRELESS SENSOR NETWORKS................................................................................................4

  2.1.1 Wireless Sensor Networks vs. Traditional Wired Networks ...................................................... 5

  2.1.2 The goal of Wireless Sensor Networks .............................................................................. 7

  2.1.3 Communicating in a Wireless Sensor Network .................................................................... 7

2.2 APPLICATION DOMAINS ...........................................................................................................8

  2.2.1 Military applications ........................................................................................................... 9

  2.2.2 Environmental applications ............................................................................................... 10

  2.2.3 Habitat applications .......................................................................................................... 11

  2.2.4 Industry and business applications ............................................................................... 11

  2.2.5 Health applications .......................................................................................................... 12

  2.2.6 Smart home/office/classroom applications .................................................................... 13

  2.2.7 Developing applications for WSNs ................................................................................ 14

2.3 MOTES ...................................................................................................................................14

  2.3.1 Sensor platforms ............................................................................................................ 15

  2.3.2 Mica2 motes ................................................................................................................... 16
2.4 OPERATING SYSTEMS.............................................................................................................18

2.5 TINYOS ....................................................................................................................................19

2.5.1 Design goals .....................................................................................................................20

2.5.2 The TinyOS kernel design and concurrency model ..........................................................21

2.5.3 The TinyOS basic constructs.............................................................................................22

2.5.4 The TinyOS programming model.......................................................................................25

2.5.5 The nesC programming language......................................................................................25

2.5.6 The TinySchema.............................................................................................................26

2.5.7 TinyOS versus other OS’s..................................................................................................27

2.6 DATA STREAMS......................................................................................................................28

3 TINYDB ....................................................................................................................................30

3.1 TINYDB FEATURES ................................................................................................................30

3.2 ARCHITECTURAL OVERVIEW................................................................................................32

3.2.1 Sensor network software...................................................................................................32

3.2.2 The TinyDB components.....................................................................................................34

3.2.3 Java based client interface..................................................................................................36

3.3 TOPOLOGY................................................................................................................................37

3.3.1 Tree-based routing.............................................................................................................38

3.3.2 Semantic Routing Trees (SRTs) ......................................................................................39

3.4 DATA MODEL..........................................................................................................................41

3.5 QUERY LANGUAGE (TINYSQ) ...............................................................................................42

3.5.1 TinySQL versus standard SQL..........................................................................................43

3.5.2 TinySQL query categories.................................................................................................44

3.5.3 Query syntax.......................................................................................................................45
### 3.6 Query Types

- **3.6.1 Event based queries** ................................................................. 46
- **3.6.2 Traditional relational queries** .................................................. 48
- **3.6.3 Aggregation queries** ............................................................... 49
- **3.6.4 Queries over stored data** ....................................................... 54

### 3.7 Tag: Tiny Aggregation ...................................................................... 55

### 3.8 Acquisitional Query Processing .................................................... 57

### 3.9 The TinyDB GUI ............................................................................ 58

### 4 Aggregations and Group By Issues .................................................. 61

- **4.1 Default Compile Options in TinyDB** ......................................... 61
  - **4.1.1 Investigating the kQUERY_SHARING option** ......................... 63
  - **4.1.2 Investigating the kSTATUS option** ....................................... 63
  - **4.1.3 Investigating other options** .................................................. 63
- **4.2 Features Not Implemented in TinySQL** ...................................... 64
- **4.3 Aggregation Issues** .................................................................. 65
  - **4.3.1 Challenges related to correction of aggregates** ..................... 65
  - **4.3.2 Deficiencies regarding aggregated functions** ...................... 66
  - **4.3.3 Solution sketch** ................................................................ 68
- **4.4 Group By and Having Issues** .................................................... 69
  - **4.4.1 The importance of GROUP BY and HAVING** ....................... 69
  - **4.4.2 Challenges related to implementation of the HAVING clause** ... 70
  - **4.4.3 Solution sketch** ................................................................ 71
- **4.5 Logging of Aggregated Queries to PostgreSQL** ......................... 71

### 5 Error Searching ................................................................................ 72
VII

5.1 INCORRECT AGGREGATES ......................................................................................................72

5.1.1 Investigating the software on the motes ......................................................................72

5.1.2 Investigating the networking software in the Java API ..............................................76

5.2 LOGGING AGGREGATE QUERIES TO POSTGRESQL .................................................................78

6 DETAILED DESIGN OF THE NEW IMPLEMENTATION .............................................79

6.1 THE HAVING CLAUSE ..........................................................................................................79

6.1.1 Making HAVING part of the TinySQL query language ..............................................79

6.1.2 Changes in the Java API ...................................................................................................83

6.1.3 Applying the HAVING clause to the resultset .............................................................90

6.1.4 Changes to the GUI ....................................................................................................91

6.2 THE AGGREGATES ..................................................................................................................97

6.3 THE LOGGING ......................................................................................................................100

7 CHALLENGES REGARDING TESTING AND DEBUGGING TINYDB IMPLEMENTATIONS....................................................................................................................102

7.1 MAKING TINYDB AN EXECUTABLE APPLICATION ..............................................................102

7.2 MAIN CHALLENGES IN TESTING AND DEBUGGING ........................................................103

7.3 SIMULATION IN WSNs ..........................................................................................................104

7.4 SIMULATORS ........................................................................................................................105

7.4.1 Ns-2 ..................................................................................................................................105

7.4.2 SensorSim ......................................................................................................................105

7.4.3 J-Sim ............................................................................................................................105

7.4.4 SENSE ..........................................................................................................................106

7.5 EMULATORS ........................................................................................................................106

7.5.1 VMNet ..........................................................................................................................106

7.5.2 TOSSIM .......................................................................................................................107
List of figures

Figure 2-1 Sensor network application development (Khemapech I et al., 2005) ................................................................................................................................. 9

Figure 2-2 Mica Architecture Diagram (Prabal Dutta, 2005)............................ 16

Figure 2-3 The component, interface, command and event connection (Schonwalder Jurgen and Harvan Matus, 2007)................................................................. 24

Figure 2-4 Configurations and components (Schonwalder Jurgen and Harvan Matus, 2007) ........................................................................................................... 24

Figure 2-5 The overall picture of the TinyOS building blocks (Yoo, 2005) ....... 25

Figure 2-6 The compilation process of a TinyOS application (Yoo, 2005) ..... 26

Figure 3-1 TinyDB component diagram (Madden Samuel et al., 2003 C) ...... 33

Figure 3-2 Sensor network topology with routing tree overlay (Madden Samuel, 2003 A). ............................................................................................................ 38

Figure 3-3 An SRT based on the x-location (Madden Samuel et al., 2005) .... 40

Figure 3-4 Oscilloscope plot of power consumption (Madden Samuel et al., 2005) ....................................................................................................................... 47

Figure 3-5 The aggregate count(*) propagates up the tree using a partial state record (Madden Samuel et al., 2005).................................................................... 50

Figure 3-6 Classes of aggregates (Madden Samuel, 2002 A)....................... 52

Figure 3-7 An Epoch with time slice for each node level (Madden Samuel, 2002 A).................................................................................................................... 56

Figure 3-8 Server based versus in-network aggregation (Madden Samuel et al., 2002 B).............................................................................................................. 57
Figure 3-9 TinyDB GUI the query construction and mote commands panel... 58
Figure 3-10 Maintenance panels in TinyDB..................................................... 59
Figure 3-11 The GUI result frame .................................................................... 60
Figure 4-1 The result of an aggregated query producing incorrect results...... 67
Figure 5-1 The TinyDB aggregation framework (Shvets, 2003)...................... 72
Figure 5-2 Construction of a QueryResult Object............................................ 76
Figure 6-1 GUI with HAVING implemented...................................................... 92
Figure 6-2 The new structure for holding information about queries............. 98
Figure 8-1 The ACM taxonomy of computer science (Akkök, 2004)............ 111
Figure 8-2 The 14 + 2 approaches to research in information systems (IS) (Akkök, 2004)........................................................................................................ 113
Figure 9-1 The SerialForwarder execution panel ........................................... 120
Figure 9-2 The three topologies used in the experiments............................... 122
Figure 9-3 Percentage of null values in the result set from the queries........... 124
Figure 9-4 Percentage of time when delivery is from one node only............. 126
Figure 9-5 Percentage of time with one or more null values....................... 127
Figure 9-6 Percentage of undelivered messages for non-aggregate queries... 128
Figure 9-7 Percentage of undelivered messages for aggregate queries........... 128
Figure 9-8 Percentage of undelivered results for aggregate queries when nulls are acceptable................................................................. 129
Figure 9-9 Percentage of undelivered messages when only one mote is used 130
Figure 9-10 Percentage of QueryResults delivered without null values. .... 132

Figure 9-11 Light readings from the motes .................................................... 134

Figure B-1 Flow of aggregated queries ......................................................... 147
Abstract

The recent years Wireless Sensor Networks (WSNs) have had a tremendous growth in interest, and advances in hardware and wireless network technologies have resulted in multifunctional and small low-cost and low-power sensor devices. These devices can be spread across a geographical area to collaborate among themselves to establish a sensor network. These devices have several distinguishing characteristics; they communicate with a low-range radio, they have very low processing power, and they have limited memory resources. Given their limited power supply, they permit very low energy consumption. Further, they perform limited and specific monitoring and sensing functions.

TinyDB is an experimental query processing middleware system, which runs on each of the nodes in a WSN. The primary goal of TinyDB is to make it easier to develop applications for WSNs. TinyDB frees the developer from the burden of writing low-level code for sensor devices like the sensor network interfaces. It offers a query language much like SQL for extracting information from the sensors. The current implementation of TinyDB has some weaknesses. First, queries containing aggregates produce incorrect values. Second, the HAVING clause in the query language is not implemented, and it is not possible to log aggregated queries to the PostgreSQL database. Finally, there is a considerable loss of messages in the network.

It is vital that aggregation and grouping techniques are correctly implemented in a WSN, since those techniques reduce the amount of data that must be sent in the network, and thereby extending the life of battery powered WSNs. In addition, the HAVING clause is the only way to filter aggregated values. When the HAVING clause is implemented and applied in the network, it will reduce the amount of data sent in the network.

In this thesis we develop a new implementation of TinyDB. It extends the query language with the HAVING clause, and the errors regarding aggregating functions and logging of aggregated data to the PostgreSQL database are corrected. We use the new implementation to investigate the reliability of TinyDB and perform an evaluation of the HAVING clause feature.
1 Introduction

In recent years, Wireless Sensor Networks (WSNs) and applications in WSNs have emerged as an important research area. The nodes in a WSN consist of small sensor devices called motes\(^1\), which use their sensing capabilities and a radio to cooperatively create a network and monitor physical conditions such as temperature, noise and light. A WSN can consist of a large number of motes, currently up to several thousand according to (Yao and Gehrke, 2002). Motes in a WSN usually use a multihop routing protocol to communicate with motes that are spatially distant. Each mote is capable of only a limited amount of processing. But when coordinated with the information from a large number of other motes, they have the ability to measure a given physical environment in great detail. Preservation of power is an important feature for all applications in a WSN. The motes have limited energy resources, and they mainly use standard batteries as a source of power. This implies that there is only a finite source of energy, which must be optimally used for processing and communication. Developing applications for a WSN is a challenging task. The motes have limited processing speed, storage capacity, and communication bandwidth. The necessary embedded code for communication and extraction of values from the motes is not straight forward to write without errors.

The WSN application TinyDB (Madden Samuel, 2003 B) is a query processing system for extracting information from a WSN consisting of motes running the embedded operating system TinyOS (Hill, 2007). TinyDB provides a simple declarative language, TinySQL, which is used to specify the data the user wants to extract from the motes in the network. In a declarative language like TinySQL the user specifies what he wants and the system figures out how to get it. The primary goal of TinyDB is to offer a middleware that makes it easier to develop applications for WSNs. TinyDB incorporates acquisitional techniques designed to minimize power consumption. In TinySQL you specify not only the data you want to extract, but also acquisitional parameters like the sample rate for the query. TinyDB collects data from the motes in the WSN, filters it, aggregates it and routes it to a PC.

\(^1\) Mote (Berkley Mote): Short for remote. Mote refers to a wireless sensor network node, developed at UC Berkeley. The name “mote” comes from the “Smart Dust” metaphor introduced by sci-fi author Neal Stephenson.
Research has noted that the most common operation in WSNs is probably aggregation (Yao and Gehrke, 2002, Intanagonwiwat et al., 2000). It is therefore of outmost importance that aggregation is correct and that the tools necessary for handling aggregates are correct. Aggregates are important for two reasons. First, when aggregating functions are implemented in a network, it is power saving because messages in the network can be combined, resulting in fewer messages to be sent over the network. Sending of messages in the network is very power consuming. Second, many times you are not interested in readings from a single node, but rather the aggregation of values from a group of nodes.

1.1 Problem statement

There are some unresolved issues with the current implementation of TinyDB. First, there is not implemented any form of filtering for aggregated values, like the HAVING clause in SQL. Second, there are errors concerning the aggregate function, with the result that it returns the same value for the different motes in an aggregated query. TinyDB can be set up to log the result of queries to a PostgreSQL database. This option works for non-aggregated queries, but it does not work for aggregated queries.

We know from (Madden Samuel, 2003 A) that the motes used with the work on this thesis probably will drop 20% to 30% of messages received at a range of about 16 - 32 feet. For TinyDB to be useful it is important that it is reliable, and we want to investigate the reliability of TinyDB. Since aggregation is an important issue, we are particularly interested in the handling of queries with aggregates.

For TinyDB to be useful it is important that it is reliable and equipped with the necessary tools to handle aggregates in a proper way. Based on this, the primary goals for this thesis are the following:

- Extend TinyDB with HAVING.
- Correct errors regarding the aggregate queries that exist in the current implementation.
• Correct errors regarding the logging of aggregate queries to the PostgreSQL database.

• Investigate the reliability of TinyDB. With reliability in this context we mean the ability to deliver correct and complete results for a query. Incomplete results occur as a result of messages lost in the WSN.

1.2 Thesis outline

The thesis is structured as follows. Chapter 2 provides the necessary background information for the area of Wireless Sensor Networks. In Chapter 3 we describe TinyDB. In Chapter 4 we look at aggregation and group by issues, and we look at the features in TinyDB which are not implemented, or incorrectly implemented. Chapter 5 discusses the search for errors in the current implementation of TinyDB. Chapter 6 gives a detailed design of the new implementation of TinyDB. In Chapter 7 we look at the challenges regarding testing and debugging of TinyDB implementations. The research methods used in this thesis are discussed in Chapter 8. Chapter 9 is about implementation and results, and finally in Chapter 10, we conclude and future work is discussed.
2 Background

This chapter provides the necessary background information for the area of WSNs. In Section 2.1 we give an introduction to WSNs and look at differences between a traditional network and a WSN. The usage and the communication in a WSN are also investigated in this section. The application domain for WSNs is presented in Section 2.2. Some of the sensors available and their platforms are looked at in Section 2.3. There are several operating systems available for WSNs and some of them are presented in Section 2.4. TinyOS, the operating system used by TinyDB, is presented in Section 2.5. Finally in Section 2.6 a short introduction to data streams is given.

2.1 Wireless Sensor Networks

The following definition is from (Schonwalder Jurgen and Harvan Matus, 2007). “A WSN is a wireless network consisting of spatially distributed autonomous devices using sensors to cooperatively monitor physical or environmental conditions such as temperature, noise, vibration, pressure, motion or pollutants”. Advances in technology have created low-cost multifunctional, miniature, low-power sensor devices. These devices collaborate among themselves to make up a WSN (Tubaishat and Madria, 2003). A WSN can consist of a large number of sensor devices, currently up to several thousands according to (Yao and Gehrke, 2002). Nodes in a WSN consist of a small sensor device named mote. The motes can sense a physical phenomenon, and then actuate and communicate over a wireless network. Individual motes are connected to each other through a wireless communication interface, and they use a multihop routing protocol to communicate with motes that are spatially distant. Each mote is capable of only a limited amount of processing. But when coordinated with the information from a large number of other motes, they have the ability to measure a given physical environment in great detail. Motes are discussed in Section 2.3. sensor networks is the key to gathering the information needed by smart environments (Diane Cook, 2004).

The development of WSNs was originally motivated by military applications such as battlefield surveillance, but WSNs are now used in many civilian application areas as
well. With large WSNs it would not be feasible to manually administrate the network, and a key advantage of WSN is that the network can be deployed on the fly and it can operate unattended, without the need for any pre-existing infrastructure and with little maintenance. Typically, sensor nodes are deployed randomly (e.g., via aerial deployment), and are expected to self-organize to form a multi-hop network (Tatiana Bokareva, 2006).

### 2.1.1 Wireless Sensor Networks vs. Traditional Wired Networks

Traditional wired sensor networks have existed for a long time, with sensors deployed throughout buildings, labs and equipment. They passively transmit signals based on some environmental parameter. They might even have the nodes connected to a local area network and be attached to permanent power sources. The advantage of WSNs is considerable, deploying and maintaining a wired network of thousands of nodes is impractical considering the thousands of miles of wire that would be needed for the connections (Sinopoli et al., 2003). WSNs opens up for a new area of applications and this is further investigated in Section 2.2. The major difference between WSNs and traditional networks, as well as working with them, addresses a new set of issues as follows.

The nodes in WSNs have limited energy resources, and they mainly use standard batteries as a source of power. This implies that there is only a finite source of energy, which must be optimally used for processing and communication. An interesting fact is that the communication dominates the processing in energy consumption. Thus, in order to make optimal use of energy, a shift from communication to processing is preferable and communication should be minimized as much as possible.

There are resource constrains in WSNs, and the nodes have:

- Limited processing speed. The microcontroller runs in MHz speed, compared to a PC where the standard is GHz.
- Limited storage capacity. A small amount (a few kilobytes) of SRAM (Static Random Access Memory) compared to a PC where there can be Gigabytes available.

- Limited communication bandwidth. The communication bandwidth is a measurement on how much data that can be sent through the network.

- The nodes in WSNs may have unattended operation. In most cases, once deployed, WSNs have little or no human intervention. Hence the nodes themselves are responsible for reconfiguration of the network, in case of any changes.

- WSNs use a broadcast communication paradigm whereas most traditional networks use a point-to-point communication.

- Nodes in WSNs are very prone to failures due to packet loss when communicating and because they eventually run out of power.

- Nodes in WSNs can be densely deployed in large numbers. This might cause problems with collision and congestion.

- Ad-hoc deployment. Most nodes are deployed in regions which have no infrastructure at all. In an extreme case a way of deployment might be tossing the sensor nodes from an airplane. In such a situation, it is up to the nodes to identify connectivity and distribution.

- Dynamic changes. Sensor nodes leaving, moving or connecting to the network require the system to adapt over time to changing connectivity.

- In traditional networks the focus is on maximizing channel throughput and minimizing node deployment. In contrast, the major consideration in WSNs is to extend the system lifetime as well as the system robustness (Nirupama Bulusu et al., 2001).
2.1.2 The goal of Wireless Sensor Networks

A WSN has two main goals (Van Dyck and Miller, 2001):

- Detect events of interest and collect parameters that characterize these events.
- Communicate the resulting information to a location outside the network.

A WSN is a sensing, computing and communication infrastructure that allows us to instrument, observe, and respond to phenomena in the natural environment, and in our physical infrastructure. Computer-based instrumentation has existed for a long time, but the density of the instrumentation made possible by a shift to mass produced intelligent sensors, gives WSNs a new kind of scope that can be applied to a wide range of uses (Culler et al., 2004).

2.1.3 Communicating in a Wireless Sensor Network

As mentioned in the prior sections, a WSN is a collection of sensor nodes forming a network often without the aid of any central administration or support services, i.e. a WSN must be power-aware. Most traditional network protocols are conservative only in their use of bandwidth. As pointed out earlier, the power consumption is the main challenge with a WSN, and in a WSN all communication, including passive listening, will have significant effect on the node’s limited energy resource. WSNs are highly dynamic, and over time sensors will be moved, run out of battery power or otherwise fail and new sensors may be added. Therefore, algorithms for WSNs should be robust and stable, and work also in case of node failure. When mobility is introduced in the sensor nodes, maintaining the robustness and consistent topology discovery becomes even more difficult.

The sensor nodes use radio frequency (RF) transceivers as their network interface and they communicate with each other using multi-hop wireless links. Each node in the WSN acts as a router, forwarding messages for its neighbors. The typical communication distance for nodes in a WSN range from a few feet up to around 100 feet, depending on the environmental conditions and the transmission power (Madden Samuel, 2003 A). Nodes receive per-message, link-level acknowledgements
indicating whether or not a message was received by the intended node. Normally no end-to-end acknowledgements are provided. The nodes in the WSN maintain a list of neighbor nodes, and associate a link quality with each of them. This list is used to help make intelligent routing decisions. The Mica2 motes, explained in Section 2.3.2, will drop 20% to 30% of packets received. Because of these high loss rates, as packets are forwarded the network management layer is constantly engaged in topology maintenance. This maintenance consists of trying to determine if there is a better choice of parent in the routing algorithm (Madden Samuel, 2003 A).

2.2 Application domains

In 1999, Business week chose WSNs as one of the 21 most important technologies for the 21st century, and Technology Review holds WSNs as “one of the 10 Emerging Technologies That Will Change the World” (Roush, 2003). The research on WSNs started as specific research objectives contributing to military applications like battlefield surveillance. This has later evolved to applications for different domains. The use of WSNs can roughly be classified into the following groups of applications.

- Space monitoring. This is environmental and habitat monitoring, precision agriculture, indoor climate control, surveillance, treaty verification, and intelligent alarms.

- Monitoring of objects. Structural monitoring, condition based equipment maintenance, medical diagnostics, and urban terrain mapping.

- Monitoring the interactions of objects with each other and the encompassing space.

A classification of the application domains for WSNs is shown in Figure 2-1.
The overview of the application domain is not based on the development timeline but it is more an objective oriented classification of the different areas of applications. We have that components in applications developed for a specific domain might be reused when applications for another domain is developed. Following is a description of the application domains.

### 2.2.1 Military applications

It is with WSNs like many other technologies, military applications have been a driving force for research and development. Modern research on sensor networks started around 1980 with the Distributed Sensor Networks (DSN) program at the Defense Advanced Research Projects Agency (DARPA) (Chee-Yee and Kumar, 2003). WSNs provide the technology for a broad spectrum of systems in the defense arena, and can be an integral part of defense/offensive military systems. They can be used in control, communication, computing intelligence, surveillance and reconnaissance and targeting systems. Examples of applications are:
• Monitoring friendly forces. WSN can be used to continuously monitor the placement of our own troops. Every soldier and vehicle can be provided with a small sensor that gathers necessary information.

• Monitoring equipment and ammunition. Equipments and ammunition can be attached with small sensors that report the status.

• Battlefield surveillance. Critical terrains, approach routes, paths and straits can be rapidly covered with WSNs and watched for activities from non-friendly forces.

• Reconnaissance of opposing forces and terrain. The nature of WSN makes it possible to deploy them in terrains occupied by non-friendly forces. The WSN can then be used to collect valuable detailed information about opposing non-friendly forces.

• Nuclear, biological and chemical (NBC) attack detection and reconnaissance. WSNs can be used as a warning system, given an alert if an NBC attack is detected. This can provide forces with the necessary time to take appropriate actions. It can also be used after an NBC attack to reconnaissance an area without exposing humans for radiation for instance.

2.2.2 Environmental applications

WSN can be used to monitor condition that affects crops and livestock, chemical or biological detection and environmental monitoring in marine, soil and atmospheric contexts. Examples of systems are:

• Flood detection. WSNs can give early warnings on flooding based on sensing data and algorithms calculating effects of rainfall etc.

• Forest fire detection. Detection of a forest fire and information about the extent of the fire can be given using a WSN.
• Precision Agriculture. In agricultural areas WSNs can monitor parameters like bacterium levels in the drinking water, level of soil erosion and air pollution.

• Volcanic eruption. In 2004 a WSN was deployed on the Tungurahua volcano\(^2\) to monitor volcanic eruption. 16 Tmotes\(^3\) were deployed for 2 months (Werner-Allen Geoff et al., 2004).

2.2.3 Habitat applications

Habitat applications are applications like tracking the movements of birds, animals or even insects. Together with military applications, Habitat monitoring applications are described as a driver application for WSNs (Cerpa et al., 2001). In August 2002, researchers from UC Berkeley/Intel Research Laboratory deployed a WSN on Great Duck Island, Maine, to monitor the behavior of the shy seabird Leach’s Storm Petrel. Using a sensor network allowed scientists to continually measure environmental data in and around the Petrel's nesting burrows without disturbing the nesting birds. The researchers had questions like what factors make a good nest. What are the occupancy patterns during incubation and breeding season? Over a three-month period during summer 2002, the sensor network delivered ~1.8 million measurement packets to its base station, with individual motes delivering up to 50,000 packets (Haman, 2003). In the period from April to June 2004 researchers from UC Berkeley/Intel Research Laboratory had a WSN in operation, capturing a detailed picture of the complex spatial variation and temporal dynamics of the microclimate surrounding a coastal redwood tree (Tolle et al., 2005).

2.2.4 Industry and business applications

For the commercial marked, so far no “killer” application has been made. Inventory systems and machine monitoring is only two of possible many areas where

\(^2\) The Tungurahua volcano is in central Ecuador 5018 m.

\(^3\) Tmote is from the Moteiv corp. described further in Section 2.3.1.
applications are already made, or can be made. Applications for machine monitoring have been discussed in (Tiwari et al., 2004, Tiwari et al., 2007). Another area where it has been predicted that we will see WSN in production, is in the area of building environment control, where heating, cooling, lighting and security are controlled by the WSN.

2.2.5 Health applications

This section is mostly based on the information from the 2006 International Workshop on Wearable and Implantable Body Sensor Networks (Paradiso Joseph, 2006). In the last decade there has been a rapid surge of interest in new monitoring and sensing devices for healthcare. In this domain it has been done much research on wearable/wireless devices. One key development in this area has been implantable, *in-vivo*, monitoring and intervention devices. Application and sensor prototypes for patients with diseases like acute diabetes, chronic cardiac diseases, epilepsy and other debilitating neurological disorders are starting to emerge. Applications in this category include applications for:

- Body Sensor Networks as described above.
- Tracking and monitoring doctors and patients inside a hospital.
- Tracking and monitoring patients and rescue personnel during rescue operations.
- Telemonitoring of human bioelectrical, biochemical, biophysical data.
- Providing interfaces for disabled.
- Drug administration in hospitals.
2.2.6 Smart home/office/classroom applications

A particular niche in which sensors would play an important role is what researchers call smart environments (Abowd and Sterbenz, 2000). A smart environment is able to identify people, interpret their actions and react accordingly. Examples of smart home applications can be found in (Huan et al., 2006) they mention applications like:

- Gate reminder, reminds you what you need before you leave home.
- Smart wardrobe, tells you today’s weather forecast.
- Smart bed and pillow, smart picture frame, smart dressing table provides comfortable and convenient housing lives.
- Smart home vacuum system.

Several research projects for smart office applications exist. Among them are the Smart Doorplate which is described in (Wolfgang Trumler, 2003). The Smart Doorplates are able to display current situational information about the office owner. If the office owner is absent, the visitor can be directed to the current location of the office owner based on a location-tracking system. Another smart office application is the Active Badge system described in (Want et al., 1992). This application is an infrared-based location tracking system. It is used in an office scenario to redirect phone calls to the current location of a user absent from his office desk.

Research projects have also looked at how classrooms could benefit from the use of sensor networks. These intelligent classrooms are able to track and respond to the needs of students and teachers. Intelligent environments make use of sensors, cameras, microphones/speakers and actuators.
2.2.7 Developing applications for WSNs

WSNs make it possible to build radically new kinds of applications. These applications are built on top of a large number of low-powered sensor nodes. Developing software for this kind of network can be a challenging task. The sensor nodes are often located in remote and largely inaccessible regions, or distributed over wide geographical areas. The nodes might range from tiny resource-constrained devices (motes) to PDA-class computing devices. In Section 2.1.1 we mention some of the differences between WSNs and traditional networks. There are some additional challenges when developing applications for WSNs. In addition to optimizing battery time and amount of information there are challenges with the event-driven, concurrent programming model that makes the application hard to debug. WSN applications can have real-time requirements such as time-critical tasks like sensor acquisition and radio timing.

The sensor nodes are usually not equipped with standard input-output devices, like key-board and monitor. This together with the event-driven nature of the applications makes it hard to debug code which is running on the motes. In this thesis we do not discuss formal approaches to concurrent programming. Several simulator and emulators exist and are discussed in Chapter 7. Use of emulators help the programmer in the challenging task of developing applications for WSNs, and the use of emulators makes it possible to run the application in a controlled environment. In Section 7.2 more about the challenges related to developing applications for WSNs, and the TinyDB application in particular, is discussed.

2.3 Motes

The research on sensor technology has been active for several years. In fundamental electronics, sensors have existed for decades, and have been used in different scenarios. The innovation is that sensors have been equipped with new capabilities in terms of computation and communication. This new device called a mote consists of five basic components:
1. A sensing unit, which is usually composed of sensors and analogue to digital converters (ADCs).

2. Processing units which make the sensor collaborate with other sensors to carry out the assigned sensing tasks.

3. Memory. Static Random Access Memory and some motes also have flash based memory available.

4. A transceiver unit which connects the node to the wireless network.

5. A power unit usually consisting of small standard batteries.

The power consumption of each mote is dominated by the cost of transmitting and receiving messages. In terms of power consumption, transmitting a single bit of data is equivalent to 200 - 800 instructions (Madden Samuel, 2002 A). Since sensors have the ability to perform local computation, part of the computation may be moved from the central node and pushed into the sensor network. Then sensors can aggregate records, or eliminate irrelevant records. Compared to traditional centralized data extraction and analysis, in-network processing can reduce energy and bandwidth consumption by replacing more expensive communication operations with relatively cheaper computation operations. This can extend the lifetime of the sensor network significantly (Yao and Gehrke, 2002).

2.3.1 Sensor platforms

The sensors themselves can range from small passive micro sensors (e.g., "smart dust" (Pister, 2001)) to larger scale, controllable weather-sensing platforms. Their computation and communication infrastructure is radically different from that found in today's Internet-based systems, reflecting the device and application-driven nature of these systems. There are several commercial vendors of motes and the two best known are:

- Moteiv Corporation, producer of the Tmote platform of sensors.
Crossbow Technologies Inc, producer of the Mica, TelosB, IRIS, Imote2 and Cricket platform of sensors.

2.3.2 Mica2 motes

The company that first started up producing motes for commercial use is Crossbow Technologies Inc. founded in 1995. They produce the Mica series of motes. This series is much used in research worldwide. The one used for the experiments in this thesis is the Mica2 mote, which is the third generation mote module used for enabling low-power, wireless sensor networks. In Figure 2-2 we see a picture of the Mica2 mote, and an architecture diagram for the mote.

![Mica Architecture Diagram](Prabal Dutta, 2005)

Here only the main components of the Mica motes are explained. Thorough description can be obtained from (Crossbow Technology Inc, 2007). The core of the mote is the AVR microcontroller. The Mica2 motes are equipped with:

- One 7.3827 MHz microcontroller.
- 4 KB of static RAM (SRAM).
- 128 KB of flash program memory.
• One external universal asynchronous receiver/transmitter (UART).

• One serial (SPI\textsuperscript{4}) port.

The Mica2 motes use a 917 MHz RFM radio running at 38.4 kb/s with a range up to 32 feet, depending on the environment it is placed in. The operating system used controls radio transmission strength and it can sense the strength of the received signal. The radio is single channel half duplex, which means that the motes can send and receive, but not at the same time. A CSMA\textsuperscript{5}-like media access protocol is used. The battery power for the Mica2 motes are two standard 1.5 volt AA batteries.

The Mica2 motes are equipped with a 51-pin connector for attaching sensor boards to increase sensing capabilities. The most common sensors are:

• Temperature.

• Barometric pressure.

• Light.

• Noise (Acoustic).

• Magnetic fields.

• Passive infrared frequency (Motion).

The Mica2 motes on the DMMS lab, used in this thesis, are capable of sensing light, temperature and noise.

\textsuperscript{4} Serial Peripheral Interface.

\textsuperscript{5} CSMA Carrier Sense Multiple Access. When a station has data to send, it first listens on the network to see if anyone else is sending and if so it waits a random time and then repeats the process. If no one is sending, it sends its message. This means that two nodes might have a collision during send, and none of them will discover it. (TANENBAUM, A. S. (2003) Computer Networks, Pearson Education Inc. page 255.)
2.4 Operating systems

Operating systems for WSNs typically are less complex than general-purpose operating systems. This is both because of the special requirements of sensor network applications and because of the resource constraints in sensor network hardware platforms. For example, sensor network applications are usually not interactive in the same way as applications for PCs. Thus, the operating system does not need to include support for user interfaces. Furthermore, the resource constraints in terms of memory and memory mapping hardware support make mechanisms such as virtual memory either unneeded or impossible to implement. Features of all operating systems are abstraction and managing of system resources, handling of concurrency and the ability to run applications. Embedded operating systems are more application specific and just use the features that they need. This indicates that they thereby can be smaller in size and save memory.

There are several different operating systems for motes. The system that is most used is TinyOS. It is described in detail in Section 2.5. Other operating systems are:

- Contiki (Dunkels, 2007, Prabal Dutta, 2005) is a light weight open source operating system for memory-constrained networked embedded systems. It has an event-driven kernel on top of which application programs can be dynamically loaded and unloaded at runtime. Contiki supports pre-emptive multi-threading per-process. There is no memory protection between applications. Contiki has a dynamic linking of libraries and offers two communication stacks uIP and Rime. Rime is a lightweight communication stack designed for low power radios. Rime provides a wide range of communication primitives, from best-effort local area broadcast, to reliable multihop bulk data flooding. uIP\(^6\) makes it possible for Contiki to communicate over the Internet using the small RFC-compliant TCP/IP stack. Contiki runs on a variety of platforms among them the Mica motes. Contiki is larger in size than TinyOS and smaller than MANTIS (which is mentioned below).

\(^6\) uIP is an implementation of the TCP/IP protocol stack which provides the necessary protocols for Internet communication for small 8 and 16 bit microcontrollers. (DUNKEL, A. (2006) The uIP Embedded TCP/IP Stack.)
• SOS (Prabal Dutta, 2005, SOS Embedded Operating System, 2007) is an operating system for motes developed at UCLA. The design goal of SOS was to develop an application independent sensor operating system. It uses a common kernel that implements messaging, dynamic memory, module loading and unloading, together with other services. The primary motivation for SOS is dynamic reconfigurability. What this means is the ability to perform modifications to the software on an individual node in the network, after the software have been deployed and initialized on the motes. SOS is ported to several platforms; among them are Mica2, MicaZ and Telos motes. According to (Prabal Dutta, 2005) the SOS and TinyOS have about the same performance.

• MANTIS OS (Bhatti et al., 2005, Prabal Dutta, 2005) provides a new pre-emptive multithreaded cross-platform embedded operating system for WSNs. It is developed at the computer science Department at the University of Colorado, Boulder. MANTIS is user friendly in that it offers a familiar API and language (the ANSI C API). It is designed to provide cross-platform support across PC’s, PDA’s as well as diverse micro sensor hardware platforms. Hence the application running on MANTIS will be cross platform portable. The goal for the developers of MANTIS was to make a general-purpose software/hardware platform for sensor networks. They wanted to simplify sensor networks for novices, and make a flexible OS for advanced research. MANTIS boot loader can re-flash the entire OS, or load stored code image from the EEPROM, to a deployed node. It offers a remote “login” to nodes. A remote Shell/Command server can be used for debugging functions, peek/poke and to get kernel status information.

2.5 TinyOS

The content of this Section is based on (Hill, 2007, Schonwalder Jurgen and Harvan Matus, 2007, Yoo, 2005). TinyOS was developed in collaboration between UC Berkeley and Intel Research, led by Jason Hill, UC Berkeley. TinyOS is to day further developed by an international consortium, the TinyOS Alliance.
The first version, 0.4 - based on Perl scripts and C, was developed in 2000. In 2002 TinyOS 1.0 based on nesC\textsuperscript{7} 1.0, was developed for Rene and Mica motes. In 2003 TinyOS 1.1 based on nesC 1.1 was released. It can be deployed to more than 20 different architectures. In 2006 TinyOS 2.0 based on nesC 1.2 was released. TinyOS 2.0 can be deployed to nine platforms. TinyOS 2.0 is not backward compatible with TinyOS 1.1. In this thesis only the TinyOS 1.1 version is used because this is the latest version compatible with TinyDB. If not explicitly mentioned a reference to TinyOS is to TinyOS 1.1.

2.5.1 Design goals

TinyOS is an open source operating system for memory-constrained networked embedded systems. The design goal of TinyOS is to make an OS which have the following characteristics:

- Support network embedded systems by being asleep most of the time, but remain vigilant to stimuli.
- Allow for high concurrency by handling bursts of events and operations.
- Operate with limited resources concerning processing speed, storage capacity, and communication bandwidth.
- Adapt to hardware evolution by keep scaling down and use less power.
- Support a wide range of applications.
- Be robust to changes, and adapt to the evolution of hardware.
- Support a diverse set of platforms and be able to run on different motes.

\textsuperscript{7}NesC is an extension to the C programming language and is explained in Section 2.5.5
2.5.2 The TinyOS kernel design and concurrency model

The TinyOS kernel is based on a two-level concurrency model with *interrupts* and *tasks*. The code which is reachable from interrupts is asynchronous code (AC). Synchronous code (SC) is code that is reachable only from a task. Events generate interrupts and they pre-empt tasks.

Only one application can be linked to TinyOS, deployed on the motes and executed at the time. The TinyOS kernel is event driven and consists of tasks that run to completion before other tasks can start, and interrupt handlers that are signaled asynchronously by hardware.

Tasks are placed in a FIFO queue and they can be pre-empted by interrupts, but not by other tasks. Tasks are not time critical and are used for larger amounts of processing like computing the average of a set of readings in an array. As there is no interleaving between tasks, the kernel needs only a single stack for the tasks.

Interrupts pre-empt tasks, and they can post new tasks to the queue. An interrupt causes an *event* to happen. Events are used for small amounts of processing which is to be done in a timely manner. Examples of events are handling of ADC\(^8\) interrupts.

There might be concurrency conflicts between interrupts and tasks. Potential race conditions might occur between AC and SC, and between AC and AC. Non pre-emption eliminates data race conditions among tasks. Code can be protected by use of the *atomic* statement.

```
Atomic {
    ...
}
```

As opposed to synchronous tasks, asynchronous service routines can be used.

```
Async result_t interface_name.cmd_or_event_name {
    ...
}
```

Race conditions are normally detected by the compiler, but the compiler can generate false positives. The keyword “norace” can be used to stop warnings about race

---

\(^8\) Analog to Digital Converter (ADC) are devices that convert analog input signals to discrete digital output signals, typically voltage to a binary number.
conditions. This keyword must however be used with care. If not, you might end up with race conditions that are not reported by the compiler, and you get unpredictable behavior of the application.

2.5.3 The TinyOS basic constructs

TinyOS is a component-based operating system with some basic building blocks. These building blocks are linked together with the application to form an executable application. The building blocks are:

- **Command** causes an action to be initiated. Commands are issued with a call statement. Commands are similar to a stored procedure in a traditional database system. Commands do not block and the execution of a command is immediate. Commands are issued with “call”.

  Call Timer.start(TIMER_REPEAT, 1000);

- **Event** is called with the signal statement. Events are used to notify a component (explained later in this section) that an action has occurred. Events are on the lowest software level and are triggered by hardware interrupts. Events do not block. Events are similar to a function call and execution is immediate. Events are used as call back to provide results from a previous command. Events are raised with “signal”.

  Signal ByteComm.txtByteReady(SUCCESS);

- **Task** is queued with the post statement which places the task at the end of the execution queue. Tasks are generally not time critical and are used for longer running operations. As mentioned before, tasks are pre-empted by events, but not by other tasks. Tasks run to completion before the next task is allowed to run, and thereby tasks provide concurrency internal to a component. Tasks may call commands and signal events. Example of tasks might be:

  - Calculating of aggregates from sensor readings.
  
  - Encoding of radio packet for transmission.
  
  - Calculating of the CRC for a radio packet.
**Component.** There are two types of components: module and configuration. A component provides and uses interfaces.

**Module** contains C-like code. It implements the “provides” and “uses” interfaces and the code for the signal and commands it provides. A module implements the code for the events it needs to handle and it call commands when needed. Modules can post tasks. If a Module has a very long computation to do, it should divide it up into multiple tasks.

**Configuration** is a component that “wires” together other components. Configurations connect interfaces used by components to interfaces provided by others. Configurations are used to assemble other components together.

**Interface** is a bi-directional multi-functional interaction channel between two components. It specifies a set of functions to be implemented by the provider of the interface (commands), and a set of functions to be implemented by the users of the interface (events). This allows a single interface to represent complex interactions between components. A component may register interest in some event, followed by a call-back when the event happens. Typically commands call downwards to components closer to the hardware, while events go the opposite direction. An interface specifies functionality to the outside world. The functionality is which commands can be handled and what events need handling. There are two sides of an interface: *provides* and *uses*.

**Provides** interface represents the functionality that the component provides to its users. It is used for handling the implemented command functions, and to issue the events it shall signal.

**Uses** interface represents the functionality that the component needs from a provider. It is used for sending commands to ask the provider to do something, and to handle the events it has implemented.

---

9 Components are like Lego bricks. They can be assembled together in many different ways. This assembling of components is called wiring.
Application consists of one or more components wired together to form a runnable program. An application is the single top-level configuration which specifies the set of components in the application and how they are connected to each other. An application has to implement the init, start and stop commands, and it is wired to the main component to start execution.

The relation of the building blocks can be seen in the two following figures.

![Diagram 1](image1.png)

**Figure 2-3 The component, interface, command and event connection (Schonwalder Jurgen and Harvan Matus, 2007)**

We see from Figure 2-3 that a component uses and/or provides interfaces, and that an interface consists of the command and event part.

![Diagram 2](image2.png)

**Figure 2-4 Configurations and components (Schonwalder Jurgen and Harvan Matus, 2007)**

In Figure 2-4 we see the wiring of components into a configuration. Configurations connect modules together (wiring) via their interfaces.
In Figure 2-5 we see how commands cause actions to be initiated. Events notify that an action has occurred. The lowest-level events are triggered by hardware interrupts. Events call back to provide results from previous command. As mentioned before, the task are not time critical and run in the background.

![Figure 2-5 The overall picture of the TinyOS building blocks (Yoo, 2005)](image)

### 2.5.4 The TinyOS programming model

The programming model of TinyOS is based on the following principles. Construction and composition are separated. Programs are built out of components which behavior is specified in a set of interfaces. The components specify the interfaces they use and provide. Components are statically wired to each other via their interfaces. The statically wiring increases runtime efficiency by enabling compiler optimizations. The thread of control passes into a component through its interfaces to other components.

### 2.5.5 The nesC programming language

The nesC (network of embedded sensors C) programming language is a dialect or extension and limitation to the Kernighan and Ritchie C programming language (Gay et al., 2003). Everything in nesC is Static. There is no heap, no function pointers and no dynamic memory allocation. The nesC language is designed to support reuse of code by the use of components and wiring. The advantage of eliminating monolithic programs is that code (components) can be easily reused, and because of that the
number of errors should decrease. Reused components that have been exercised in working systems generally should be more reliable than new components or programs. Reused components are known to be correct due to that they have been tried and tested before. This reuse of a component will reduce the number of failures in an application (Sommerville, 2001).

![Compilation process of a TinyOS application](Yoo, 2005)

In Figure 2-6 we see how the application and TinyOS are compiled and linked together to form an executable application. The nesC application is compiled by a nesC pre-compiler which transform the nesC constructs to ordinary C code. When the application is loaded to the motes the executable code and the program constants is placed in the 128K flash program memory. The variables are placed in the 4K Static RAM. The free-space is fixed and there is no dynamic memory allocation. The stack grows into the available free space.

**2.5.6 The TinySchema**

TinySchema is a collection of TinyOS components. This collection of components is used for handling a small repository of named attributes, commands and events that can be queried, invoked or signaled from inside or outside a network of motes.

An attribute is equivalent with a column in a traditional database system. It has an identifier (name) and a data type. In addition to this, TinySchema allows the creation of methods to get and set the value of the attribute. When an attribute is created it can be retrieved through a unified interface provided by TinySchema. TinyDB (described in the next chapter) is one of the applications built on top of this interface.
A TinySchema command consists of a name, a return type and a list of arguments. It is much like a stored procedure in a traditional database system. Arbitrary TinyOS code can be associated to a command.

A TinySchema event is an asynchronous happening in the sensor network. Examples might be that a button is pushed, or a bird is detected. TinySchema provides interfaces for registering and invocation of events. Commands might be associated with events as callbacks when an event is signaled.

2.5.7 TinyOS versus other OS’s

A main difference between TinyOS and the OS’s discussed in Section 2.4 is the linking of applications. In TinyOS there is a static linking of applications. The OS, the application and the drivers, which are all NesC components, are linked together to a single runnable block of code. This in contrast to the OS’s discussed, which all have dynamic linking of applications.

The pros and cons of TinyOS compared to the other OS mentioned is:

- **Pros.** Static memory allocation guarantees that the resource is present when needed. The non pre-emptive scheduling minimizes the memory needs. The modularized language allows for independent software development.

- **Cons.** TinyOS is written in a brand new programming language, nesC. The static memory allocation in TinyOS may result in an over subscription of the memory resource. TinyOS is not really an OS, it is more a library of modules which can be wired together with your program, and there is no resource management. The simple concurrency model of TinyOS leads to the use of a lot of critical sections. Only one TinyOS application at a time may run on the motes.
2.6 Data streams

When data is received from a WSN, the data is delivered in form of a continuous ongoing stream, called a data stream. A data stream is an infinite sequence of <timestamp, tuple> pairs. The stream tuples can be externally time stamped, or they can be time stamped by the system. Most traditional database management systems (DBMS) are optimized for one-time queries on relatively static data. Hence, we have long-lived (persistent) data and short-lived (transient) queries. This matches badly applications that want to manipulate real-time unbounded streams of data. Here we have the opposite, long-lived queries and short lived data. An example on this type of application is real-time analysis of financial data like stock trades. sensor network data also comes in this category. TinyDB is an example of a system that was created for handling data streams in sensor network.

Data streams differ from traditional DBMSs in several ways. They may be unbounded in size, they arrive online and are pushed to the query processor. This in contrast to when a query processor in a DBMS pulls persistent data from the disk. When handling data streams, the query processor has no control over the order in which elements are pushed to it, and once an element has been processed it is discarded or archived. It is generally not easy to retrieve this information again unless it is stored in memory. The memory is typically small relative to the size of data streams (Babcock et al., 2002).

Queries over data streams have much in common with queries in traditional DBMS. There are however differences and the most important difference between a traditional DBMS and the data stream model is the difference between the traditional one-time-query and the continuous-query. A one-time-query is evaluated once over a point in time snapshot of the data set, and the result is delivered to the user as a finite set of rows. The continuous-query is evaluated continuously as the data stream arrives. The answer to a continuous query is produced over time, and it always reflects the stream of data seemed so far. It is common to use facilities like a sliding window to be able to analyze a portion of a data stream.
In general it is not feasible to simply load the stream of data that arrives to a traditional DBMS for further investigation later. DBMSs are normally not designed for rapid and continuous loading of unbounded streams of data items. A new management system, a Data Stream Management System (DSMS) is therefore needed to handle this new kind of data and applications. There have been several projects on the subjects of data streams, and examples are the TelegraphCQ project (Reiss, 2006) from UC Berkeley and STREAM (The stream group, 2006) from Stanford.
3 TinyDB

In this chapter we describe TinyDB, which is a query processing system for extracting information from a wireless network of motes running on TinyOS. According to (Madden Samuel, 2003 B) the primary goal of TinyDB is to make the life as an application programmer easier. This is done in TinyDB by offering a declarative query language called TinySQL, which can be used to extract data from the sensor network much the same way data is extracted from a relational database. When TinyDB is used to extract information from the motes in a sensor network, there is no need for the programmer to write embedded code and deploy it on the motes. This allows for a much faster development of data driven applications, than without the use of TinyDB. TinyDB frees the programmer from writing low level code for sensor devices, including the complex sensor network interfaces.

In Section 3.1 we start by describing the features of TinyDB. In Section 3.2 we give an architectural overview of TinyDB where sensor network software, TinyDB components and the Java API are explained. In Section 3.3 the topology and routing used in TinyDB is presented. TinyDB’s data model is explained in Section 3.4. TinySQL, the query language of TinyDB, is described in Section 3.5. The different query types are described in Section 3.6. Since aggregated queries are an important issue in this thesis, they are described in detail in this section. In Section 3.7 the Tiny Aggregation Approach (TAG) is described. Acquisitional query processing is described in Section 3.8 before we end the chapter with Section 3.9 where we look at the GUI of TinyDB.

3.1 TinyDB features

TinyDB offers several features that make it easier to write programs that query and obtain data from a sensor network. The content in this section is based on information from the TinyDB homepage overview pages (Madden Samuel, 2003 B). Features offered by TinyDB include:
• Metadata management: A metadata catalog which describes all the attributes and commands available for querying and invoking in the WSN is provided by TinyDB. The attributes are typically sensor readings like temperature and light, and parameters local to each mote, like the identity of the mote. Commands can range from physical actuations to parameter tuning. The creation of attributes and commands is done through the TinySchema components in TinyOS, which are described in Section 2.5.6.

• TinyDB provides TinySQL, a declarative query language used to launch queries into the sensor network.

• The network topology is maintained by TinyDB without any administration from the users. TinyDB manages the underlying radio network by tracking neighbors, maintaining routing tables and synchronizing clocks. The network topology is further explained in Section 3.3.

• TinyDB allows for multiple queries to be run on the motes simultaneously. The queries can access different sensors and have different sample rates. (Sample rate is explained in Section 3.5.1) TinyDB efficiently share work between queries whenever possible.

• Incremental deployment via Query Sharing. A sensor network can be expanded by simply adding new motes with TinyDB deployed on them. TinyDB then automatically sets up routing tables, synchronizes clocks, and starts running active queries on the new mote.

• Java API. TinyDB comes with a Java API for developing applications that can query and obtain data from the WSN.

• GUI. TinyDB provides a graphical query-builder and result display which both are built using the Java API.
3.2 Architectural overview

TinyDB can be broadly classified into sensor network software which runs on the motes in a WSN, and a Java based client interface which is on the PC or another device capable of running the API, attached to the sensor network. First we look at the sensor network software and then the Java based client interface.

3.2.1 Sensor network software

This is the core of the TinyDB system, which runs on each mote in the network. It is written in the NesC language which is discussed in Section 2.5, and should in general not have to be changed by the user of TinyDB (Gay et al., 2003). The sensor network software consists of four modules.

Sensor catalog and schema manager. The sensor catalog is responsible for holding information of what type of sensing capabilities this sensor has (light, temperature e.g.) and other properties like nodeid, parent in the routing tree etc. This list is unique for each mote. Sensor networks may consist of a collection of heterogeneous motes which report from different sensing devices. The Schema Manager handles the TinySchema components.

Query engine. The main component in TinyDB is a declarative query processor. It uses the sensor catalog to obtain the values of local attributes, and it receives sensor readings from its child nodes (motes) over the radio. The query processor combines and aggregates its own readings with readings received from its child nodes. The query processor filters out unwanted data based on the values in the WHERE predicate, and outputs values to its parent node in the routing tree. The topology of the network is discussed in Section 3.3.

Memory manager. TinyDB extends TinyOS with a small, general purpose component for dynamic memory management. This component might also be used by other applications than TinyDB. The component is a simple handle-based\textsuperscript{10} compacting

\textsuperscript{10} A handle is an object that represents a resource. Examples of a handle can be a handle to a file or as in this case a handle to a chunk of memory. In relevance to memory management handles are used because the object self (the memory chunk) might need to be moved inside the frame, the application therefore cannot know the real address of the memory chunk.
memory manager. It allocates bytes from a fixed size frame and returns handles into that frame. The use of handlers makes it possible to move memory around in the frame without changing all the external references. This is an advantage because it allows frame compacting and tends to reduce wasted space.

Network topology manager. TinyDB manages all the network traffic, and effectively routes data and query sub-results through the network. All the necessary efforts to set up routing tables are managed by the Network Topology Manager.

![Figure 3-1 TinyDB component diagram (Madden Samuel et al., 2003 C)](image)

The components that make up TinyDB are shown in Figure 3-1. The TinyDB components are inside the blue “circle”, and the TinyOS components are outside the “circle”. A description of the TinyDB components is given in the following section.
3.2.2 The TinyDB components

The descriptions of the TinyDB components are based on (Madden Samuel et al., 2003 C), and the source code of the components. See Figure 3-1 for additional information about the relation between the components.

TinyDBAttr is the component that wires together all components that implement the built-in attributes of TinyDB. This is the component that must be updated when you add a new component that implements new attributes to TinyDB.

TupleRouter is the heart of the TinyDB system and it provides the main query processing functionality on a mote. The reason it is called a router is because it routes tuples through a variety of local query processing components. The network routing is however not done by this component, but by the NetworkC component which is described later in this section. The TupleRouter has three main execution paths.

1. Handling of new queries.
2. Handling results from neighboring nodes.
3. Generating local results and deliver data to parent nodes.

TinyAlloc. This component is the TinyDB memory manager described in Section 3.2.1.

NetworkC. This is the component responsible for the routing in TinyDB. It receives and sends data and query messages between motes. The data messages are tuples or aggregate values.

Tuple. This component provides a set of utilities to handle the tuple data structure. A tuple is a typed vector of values.

QueryResult. In TinyDB data are represented as tuples, byte-strings or QueryResult objects. This component converts between these structures. A QueryResult holds
some metadata including the query ID and Epoch number (Epochs are explained in Section 3.6.2), a tuple, and an index into the result set.

**SelOperator.** This component is responsible for the relational selection. It tests to see that the tuples match the predicates given in the query.

**AggOperator.** This component performs two features regarding SQL: The aggregation of queries and the GROUP BY. The GROUP BY is an optional feature used to partition the data over some attribute(s). When used, the aggregate functions are computed for each partition. The aggregations of a query are done by taking the readings from this node and merge those readings with aggregates from the child nodes and propagate this merged result up the tree to this node’s parent node.

**DBBufferC.** This module is an output buffer abstraction for TinyDB. It uses the MemAlloc module to allocate memory.

**Query.** This component provides an interface to unparsed queries. Unparsed queries are queries arriving over the network that have not yet been translated into **ParsedQueries.** A query is identified by a global unique ID, and contains a part of a generated query. This is either a single field to retrieve, a single selection predicate to apply, or a single aggregation function to apply. All parts must arrive to the **TuplerRouter** before the query is converted to a parsed query. **Query** provides basic methods to determine if a query contains all the records needed to be converted into **ParsedQueries.**

**ParsedQuery.** This component provides the necessary interfaces for interacting with an ongoing query. All the fields in the query have been converted into schema indices.

The TinyDB components on the motes are not modified in the implementation made from this thesis. However much research has been done on them and it is necessary to describe them to get an overall picture of the TinyDB application.
3.2.3 Java based client interface

TinyDB provides a client interface which consists of a set of Java classes and applications. This API is used when a network of TinyDB motes is accessed from a connected device running the Java API, and in most cases this will be a PC. The Java API consists of several classes, but we only mention the most important ones in this section:

TinyDBNetwork is the class that is responsible for sending and receiving queries and results over the network. It is a network interface class that allows applications to inject queries (sendQuery()), cancel queries (abortQuery()), and listen for results by adding the resultListeners\(^\text{11}\) (addResultListener()). For a network, only one instance of the TinyDBNetwork object needs to be allocated. That particular instance can handle multiple ongoing queries simultaneously. Each output from a query can be assigned to multiple listeners, and a listener can listen for a specific query or for all queries. The TinyDBNetwork object handles the incoming AM (Active Messages) from the serial port on the PC, and dispatches copies of them to the various listeners.

TinyDBQuery is the class used to build and transmit queries. Information about a query running on the motes is kept in the TinyDBQuery object. This object is generated in the SensorQueryer object where an SQL string is converted to a TinyDBQuery object. The TinyDBQuery object consists of:

- A list of the attributes which are to be selected.
- A list of expressions over these attributes. An expression is either a filter that discards values that do not match a boolean expression, or an aggregate (SUM, COUNT, MIN, MAX and AVG) that combines local values with values from child nodes and optionally includes a grouping.
- The SQL string that corresponds to the list of attributes and the expressions in the query.

\(^\text{11}\) ResultListener is an interface to deliver query results to the various UI panels that are a part of the TinyDB Java tools.
**QueryResult** is a class to receive and parse query results. The **QueryResult** object accepts a query result from the network in the form of an array of bytes. Aggregates are finalized in this object. It provides a number of utility routines to read the result data.

**Catalog** is a class used to extract information about the attributes and capabilities of the devices. The **Catalog** object provides a simple parser for a catalog file which it reads, and after parsing it provides a list of attributes.

The **MainFrame** class is the main GUI for generating queries with TinyDB. It provides a simple API for handling new queries for processing of keyboard input. It is further discussed in Section 3.9.

The **ResultFrame** class displays the result of a query in a scrollable list and a graph showing the development of readings over time. For each query it adds a processedListener object to the **TinyDBNetwork** in order to receive the final results for this specific query. It is further discussed in Section 3.9.

The **Topology** is a set of classes for visualization of the dynamic network topology.

The TinyDB demo application serves as an example application. It allows users to interactively communicate with the sensor network by constructing and launching queries to the sensor network, and later viewing the result from the queries in the provided **ResultFrame**.

### 3.3 Topology

The network topology is maintained by TinyDB as a routing tree. Motes are identified by a nodeid with nodeid zero as the root node of the tree. In general, queries flood down the tree, and data messages flow back up the tree. The root node of the tree (mote #0) passes the result data to the front-end via its serial interface. There is only one exception to this rule. The root node itself sends out periodic data messages as a “heartbeat” so that its communication abilities can continue to be measured also while queries are running.
3.3.1 Tree-based routing

TinyDB uses a tree-based routing protocol. The set of optimal routes from all nodes to the base station forms a tree. Such a tree is called a sink tree (Tanenbaum, 2003). A sink tree is not necessarily unique, and other trees with the same path length can be created. The tree-based routing is used in Query delivery, data collection and the in-network aggregation. According to (Buragohain et al., 2005) the development of In-network aggregation has resulted in routing trees playing a central role in WSN applications\textsuperscript{12}. In the tree-based routing scheme, one node is chosen to be the root node. The user interface connects to the network through the root node. In TinyDB this node is the one connected to the PC. Queries from the user are sent from the PC through the root node down the tree. In Figure 3-2 solid arrows indicate child-parent relationships, and dotted lines indicate nodes that hear each other but they do not route through each other.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{sensor_network_topology_with_routing_tree_overlay.png}
\caption{Sensor network topology with routing tree overlay (Madden Samuel, 2003 A).}
\end{figure}

When a rooting tree is generated it starts with the root node which send a broadcast message containing its own id and level (distance from the root). In the case of the root node, the level is zero. Using this message the root asks the motes to organize a

\textsuperscript{12} Chiranjeeb Buragohain et al. speak of TinyDB as an example of a sensor network database application.
routing tree. The motes without an assigned level that hear this message, first assign their own level to be the level in the message plus one, and then rebroadcast this updated routing message again. This will be done until all nodes have been assigned a level and a parent. The motes choose the sender of the message as their parent. The motes will route messages back to the root by sending it to their parents. These routing messages are periodically broadcasted from the root, so that the coming, leaving or moving nodes continuously can be updated in the new topology.

### 3.3.2 Semantic Routing Trees (SRTs)

Many queries are such that they only need results from a subset of the nodes, e.g. if they select on nodeid. An SRT maintains information about constant (and changing) attributes of their child nodes. An SRT is a distributed index to locate relevant nodes for a particular query. In an SRT each node stores value intervals of itself and its child nodes. The purpose with an SRT is to determine if a node or any of its children are to produce results for a given query. With SRT (Madden Samuel et al., 2005) argue that the choice of parent should be based on some semantic properties as well. In general, SRTs are most applicable when a node has more than one candidate as its parent node in the tree. When a query arrives on a node, the node checks to see if any of its child nodes have relevant data for it. If this is the case the query is forwarded to its child nodes. If the query applies locally the node starts executing it. The query is discarded if it does not apply either locally or to any of its child nodes. This functionally reduces much of the unnecessary sending of messages to nodes that we know do not have relevant information. Figure 3-3 is an example of a SRT in use. Here we see that nodes 2 and 5 do not have relevant data for the query, and they should not produce any results for the query. Only the nodes colored gray must produce and forward results in the query.
Building an SRT is a two phase process: First the SRT build request is flooded down the network. This request has the identity of the attribute Att on which the SRT is to be built up on as a parameter. When the request is flooded down the network, a node might end up with several possible candidates as its parent.

The second phase starts when all the nodes have received the SRT build request. Nodes without child nodes report the value of Att to the node they choose as their parent node. A parent node waits until it has received information from its child nodes and reports to its parent node the range of values of Att it covers. This way the information is sent up the tree and when the information reaches the root node, the SRT is constructed. Even though SRTs are built up on constant values, some maintenance is required. New nodes can appear, link qualities can change, and existing nodes can fail. All this may for instance require that a node has to change parent. This is done with a parent selection message. When a node receives a parent selection message, it checks to see if this new message is within the range it already covers. If not, it alters the range it is covering, and sends a new parent selection message to its parent, and the information propagates up the tree. To handle the disappearance of a child, every node with child nodes associate an active queryid and
a last Epoch with every child in the SRT. If a parent node does not hear anything from one of its child nodes for some number of Epochs, it assumes it has moved away and removes its entry from the SRT table. Then it sends a message to its remaining children and asks them to retransmit their interval for Att. If this new interval differs from what is stored in the SRT, a new SRT is built and information is propagated up the tree.

The benefit of an SRT is depending on the clustering of the values that the SRT is to be built upon. Location attributes are often a good candidate for SRT because of the natural cluster of child and parent values. The benefit of using SRT can be substantial, but there are maintenance and construction costs associated with the SRTs (Madden Samuel et al., 2005).

### 3.4 Data model

In TinyDB the entire sensor network is like one single, infinitely-long logical table named SENSORS. The columns of this table are all the attributes defined in the network. There are three types of attributes.

1. Sensor readings like light, temperature and noise.

2. Meta-data, each node can be provided with some constant values such as the identity of the node, the location where it is (like which floor in a building it is placed) and others.

3. The third type is internal states such as routing tree parent, timestamp, and queue length.

The data type of an attribute is one of int8, uint8, int16, uint16, int32, uint32 or string where u denotes unsigned (Madden Samuel et al., 2003 C).

---

13 Epoch is a discrete window of time. In TinySQL new answers to a query (tuples of data) are produced every Epoch. The duration of an Epoch can be specified in the query. A further explanation of Epoch can be found in Section 3.6.2.
The sensor table logically has one row per node per instance in time, with one column per attribute that any device can produce. Even when a node is missing the sensing device that is specified in a query, the node will produce results for this query; it will simply return a null value for each unknown attribute in the query.

Records in the SENSORS table are materialized only as needed to satisfy the running queries, and they are usually stored only for a short time, or distributed directly out on the network. The SENSORS table is an unbounded continuous stream of data and therefore some operations, which require a finite set of rows, like sort and symmetric joins, are not allowed. Data from the SENSORS table might however be stored in materialization points on the motes. A materialization point is a buffer where a bounded subset of the stream, a window, is specified. In TinyDB windows can be defined via materialization points over the stream of data from sensors. Materialization points are further explained in Section 3.6.4.

3.5 Query language (TinySQL)

TinyDB provides a declarative SQL like language for specifying queries to be run in a network of sensors running TinyOS. In a declarative language the user describes what he wants, but not how to get it. A procedural language, in contrast, is a detailed description of how a task is to be accomplished. A declarative language operates on sets of data, a procedural language operates on one record or unit of data at a time. Examples of procedural languages are FORTRAN, COBOL, Java and Pascal. Declarative languages have two advantages over procedural languages. First it is easier to learn to read and write declarative languages like SQL than it is to program in procedural languages like COBOL. The second and most important advantage is that with a declarative language the underlying system might change the way it runs a query, without the need for change in the query language. The last is important in a volatile environment like a sensor network, where the underlying implementation might need to change frequently when motes move, leave or join the network.
3.5.1 TinySQL versus standard SQL

TinySQL is based on SQL (Madden Samuel, 2003 A). As in SQL a basic TinySQL query consists of a set of attributes to project from the motes, a set of aggregation expressions for forming aggregate results, a set of selection predicates for filtering on rows, and optionally a grouping expression. TinySQL has limitations and extensions compared to SQL and we mention some of the most important here:

- The FROM clause is optional and must list exactly one table, SENSORS. An exception from the rule is when materialization points are used. Materialization points are discussed in 3.6.2 and 3.6.4.

- The WHERE clause can contain only simple conjunctions over arithmetic comparison operators. There is no support for the boolean operators OR and NOT, or string matching comparisons (SQL’s LIKE and SIMILAR constructs).

- Nested queries can only be used in conjunction with materialization points or events as discussed in the next Section.

- There is no support for column renaming (SQL’s AS construct) in the group by list.

- No column alias in the SELECT clause is allowed.

- Arithmetic expressions are currently limited to the form column op constant, where column is an attribute from the mote, and op is one of the elements in the set [+,-,\*,/]. The constant is an integer.

- TinyDB has some functions you do not find in standard SQL. The trigger action clause specifies an (optional) trigger action that should be executed whenever a query result is produced. This is further discussed in Section 3.6.

- The ON EVENT clause gives us an opportunity to execute a query when a certain event happens.
- The SAMPLE PERIOD clause is specific to TinyDB and is used to specify how
  often a sample is to be taken. If not specified, a default value of 2048
  milliseconds is used.

### 3.5.2 TinySQL query categories

The queries supported by TinyDB can be divided into the following categories.

- Monitoring queries are queries that request the value of one or more attributes
  from motes continuously and periodically.

- Network health queries are meta-queries over the network itself. Examples
  include selecting parents and neighbors in the network topology, or nodes with
  voltage less than some threshold.

- Exploratory queries: One-shot queries examining the status of a particular
  node or set of nodes at a point in time. e.g.

  ```
  SELECT light, Temperature
  Where nodeid = 2
  ONCE
  ```

  The use of the keyword ONCE in this query prevents the motes from producing
  a stream of data. Instead this query will produce a single reading from the
  node with nodeid = 2.

- Nested queries: A nested query is a query inside a query. Example of a nested
  query is

  ```
  SELECT ...
  FROM ...
  WHERE attribute in (SELECT some attribute FROM
  someTab WHERE x < 10)
  ```

  This kind of a nested query can not be used in TinySQL. However both events
  and materialization points can provide a form of a nested query. Users can
  build a materialization point and run a query over that buffer, and this way
  emulate the same effect as a nested query.
- Actuation queries: Used if one wants to take some physical action in response to a query. TinyDB has a special OUTPUT ACTION clause for this purpose. It specifies an external COMMAND that should be invoked in response to a tuple satisfying the query.

- Offline delivery: There are times when users want to log some phenomenon that happens faster than the data can be transmitted over the radio. TinyDB supports the logging of results to EEPROM for offline, non-real time delivery.

### 3.5.3 Query syntax

The query syntax in TinySQL follows SQL, and the semantics of SELECT, FROM, WHERE and GROUP BY clauses are as in SQL. In addition semantics for acquisitional query processing clauses like SAMPLE PERIOD are described. In the specification of the TinySQL syntax below the {} is used to denote a set, the [] is used to denote an optional clause, and the <> is used to denote an expression. Text in Italics is used to denote user specified tokens such as aggregate names, commands and arithmetic operators. The “|” is used to indicate that one or the other of the surrounding tokens must appear, but not both. Parentheses (...) indicate a repeating set of tokens, such as fields in the SELECT list. The syntax and use of symbols complies to what is in the TinyDB article (Madden Samuel et al., 2005)\(^\text{14}\):

```
[ON [ALIGNED] EVENT event-type[\{paramlist\}] ]
[ boolop event-type\{paramlist\} ... ]]
SELECT [NO INTERLEAVE] <expr> | agg(<expr>) | temporal agg(<expr>), ...
FROM [sensors | buffer], ...
[WHERE {<pred}>]
[GROUP BY {<expr>}]
[HAVING {<pred>}]
[OUTPUT ACTION [ command | SIGNAL event(\{paramlist\}) | (SELECT ... ) ] |
[INTO bufname]]
```

\(^{14}\) The syntax of TinySQL is taken from Appendix B.1 in the article.
3.6 Query types

Within each query category (Section 3.5.2) there are different query types. In Section 3.6.1 we describe the event based queries. Relational queries much like the one you find in traditional database systems is described in Section 3.6.2. The aggregated queries are an important issue in WSNs and they are described in Section 3.6.3. Finally, queries over stored data is described in Section 3.6.4

3.6.1 Event based queries

TinySQL supports events as a mechanism for initiating data collection in response to some external stimuli. Events are described in Section 2.5.3 and they are generated explicitly by different sources:

- Another running query can generate an event.
- Software in the operating system can generate an event.
- An event can also be generated by special hardware on the mote that signals the operating system.

Events make it possible to have a query dormant until some external event happens instead of continually polling or blocking on an iterator waiting for some data to arrive.
Figure 3-4 Oscilloscope plot of power consumption (Madden Samuel et al., 2005)

Figure 3-4 shows an oscilloscope plot of different power consumption from a device running an event-based query. On top the triggering is done by toggling a switch connected to an external interrupt line that causes the device to wake from sleep. The second is an event-based query triggered by another query that polls for some condition to be true. It is obvious that the interrupt driven query consumes less energy than the polling driven query (Madden Samuel et al., 2005). Events can also be used as a stopping condition for queries. This is done by appending a clause on the form STOP ON EVENT (param) WHERE cond (param). This will stop a running query when the given event occurs and the given condition is true. An event might also be signaled by a query. The following query might be used to signal that the temperature has reached a given threshold. For instance, this can be used to start some cooling device.

```sql
SELECT nodeid, temp
WHERE temp > threshold
OUTPUT ACTION SIGNAL hot(nodeid, temp)
SAMPLE PERIOD 10s
```

In this query the ACTION hot is SIGNALED when the temperature reaches some threshold. TinyDB only supports signaling of events on the local node. The signals are not propagated into the network.
3.6.2 Traditional relational queries

Traditional relational queries are of the type you find in traditional database systems like Oracle, DB2 and others. In TinySQL this type of query consists of a SELECT-FROM-WHERE-GROUPBY clause, supporting selection, join, projection and aggregation. As mentioned in Section 3.5.1, the FROM clause may refer to the sensor table or stored tables, which in TinySQL is called materialization points. Tuples are produced at a well-defined sample interval called an Epoch. The SAMPLE PERIOD clause is used to specify the length of the Epoch. An Epoch provides a mechanism for structuring computation. Each Epoch is divided into timeslots in such a way that for every Epoch, reading from all the nodes manages to be propagated up the tree. The maximum depth allowed for the tree in TinyDB, is 5 levels. In Section 3.7 the dividing of an Epoch into time slices is further explained. The following query specifies that each mote in the network shall report its own nodeid, light and temp once per second for 10 seconds.

\[
\begin{align*}
\text{SELECT nodeid, light, temp} \\
\text{FROM SENSORS} \\
\text{SAMPLE PERIOD 1 FOR 10}
\end{align*}
\]

The result tuples from this query is clustered into one second time intervals, and they flow to the root of the network where they are output to the user. Each tuple contains a timestamp corresponding to the time it was produced. The motes initiate data collection at the beginning of each Epoch as specified in the SAMPLE PERIOD clause.

As noted earlier TinyDB has a simple time synchronization protocol to agree on a global time. This allows for the motes to start the Epoch at the same time. The protocol is described in work by (Ganeriwal et al., 2003). The basic concept is as follows:

- Level discovery phase:
  - Create a hierarchical topology in the network. Every node is assigned a level in this structure.
- Ensure that every node belonging to level $i$ can communicate with at least one node on level $i-1$ (a node closer to the root node).

- Assign only one node to level 0, which is then called the root node.

- Synchronization phase: Once the hierarchical structure has been established, the root node instantiates the second stage of the algorithm. In this phase the nodes belonging to level $i$ synchronize with the node belonging to level $i+1$. Eventually every node is synchronized with the root node and we have a network wide global time.

Note that because the SENSORS table is an unbounded continuous stream of data, we need to define materialization points to create windows in order to save data that can be used for joins in other queries. This is done with the CREATE BUFFER statement.

```sql
CREATE BUFFER buffername SIZE 8
AS (SELECT nodeid, light FROM SENSORS SAMPLE PERIOD 10s)
```

This statement provides us with a window which can hold eight occurrences of the tuple \{nodeid, light\}. This data might be used in a join with another storage point on the same node or in a join with the SENSORS table. This feature is described under 3.6.4.

### 3.6.3 Aggregation queries

According to (Gao et al., 2007) in-network aggregation is an important topic addressed by the WSN community over the last several years. Other sensor network research has noted that the most common operation in sensor networks is probably aggregation (Intanagonwiwat et al., 2000, Yao and Gehrke, 2002). There are two different approaches to the aggregation of queries, a server based approach and an in-network approach. In a server based approach all sensor readings are sent to the base station where the computation of aggregates takes place. TinyDB has taken a different approach and supports in-network aggregation queries. Aggregates are computed on the motes whenever possible, and they thereby have the attractive property that it reduces the quantity of data that must be transmitted through the network. For
aggregates to be computed in-network, a record holding the partial state of the computation, is needed. This partial state record flows up the tree, and the content is merged with values from the current node all the way up to the root node. In Figure 3-5 we can see how the partial state record is used to compute the aggregate count(*)

We can see that from each node only one record is sent to its parent representing the partial value computed until this node.

![Figure 3-5 The aggregate count(*) propagates up the tree using a partial state record (Madden Samuel et al., 2005)](image)

Aggregates in TinyDB are implemented via three functions.

1. An initializer $i$.

2. A merging function $f$.

3. An evaluator $e$.

If we use the implementation of average as an example the functions has the following structure.
The initializer \( i \) is needed when we instantiate a single value from the sensors. When we instantiate the partial state record with the value \( n \), the initializer will return the tuple \(< n, 1 >\). The tuple consists of a sum with the value of \( n \) and a count with the value of 1.

The merging function \( f \) merges two partial state records and returns a new partial state record. The structure is as follows:

\(<z> = f(<x>,<y>) \) where \(<x> \) and \(<y> \) are multi-valued partial state records. They represent the intermediate state over the values that will be required to compute an aggregate. In this example \(<x> \) and \(<y> \) both consist of a pair of values representing the sum and count.

Given two partial state records \(<\text{Sum}_1, \text{Count}_1>\) and \(<\text{Sum}_2, \text{Count}_2>\) the merging function \( f \) will then be \( f(<\text{Sum}_1, \text{Count}_1>, <\text{Sum}_2, \text{Count}_2>) = <\text{Sum}_1 + \text{Sum}_2, \text{Count}_1 + \text{Count}_2>\).

The third function, the evaluator \( e \), finalizes the value for the aggregate. For the average aggregate, the evaluator \( e \) simply returns the sum divided by the count – \( e(<\text{Sum}, \text{Count}>) = \text{Sum}/\text{Count}. \)

If we take a closer look at the aggregates MAX, MIN, COUNT, SUM and AVG (average) we find that they might be classified according to four functional properties (Madden Samuel, 2002 A) as shown in Figure 3-6.
Duplicate sensitive aggregates are affected by duplicate readings from the sensors, while duplicate insensitive aggregates are unaffected by duplicate readings from the sensors. For instance the MAX aggregate are unaffected by duplicates, while the Average aggregate will produce wrong results when there are duplicates.

Exemplary aggregates return one of possibly many equal candidate values representing the set of values read. For instance MIN will return one value representing possibly many equal minimum values.

Summary aggregates compute some property over all values and return one value representing the set of values read. The difference between exemplary and summary is important because exemplary aggregates behave unpredictably in the case of data loss, and for the same reason are not probable to give a correct answer based on a subset of data. Conversely, for the summary aggregates, the aggregate applied to a subset can be treated as a robust approximation of the true aggregate value, assuming that the subset is chosen randomly.

Monotonic. When two partial state records s1 and s2 are combined to s, a monotonic aggregate has the property that for the resulting state record s one of the following will be true.

\[ e(s) \geq MAX\{e(s1), e(s2)\} \quad \text{Or} \quad e(s) \leq MIN\{e(s1), e(s2)\} \]
This is important when determining whether predicates like those in a HAVING clause can be applied in the network, before the final value of the aggregate is known. If a predicate is to be applied in the network it has to be monotonic. For aggregates like MAX and MIN, the HAVING predicate can be applied on each mote in the network to reduce the communication cost. For aggregates like AVG, the HAVING can not be applied before we have the finalized result. An example will clarify this.

```
SELECT MAX(light)
FROM SENSORS
HAVING(light) > 100
```

In this query the HAVING predicate can be applied in the network. The value of light can be tested against the value given in the HAVING predicate. If the test fails, the node simply does not send the result to its parent. In contrast we can look at the following query.

```
SELECT AVG(light)
FROM SENSORS
HAVING AVG(light) > 100
```

In this query the HAVING predicate can not be applied in the network. The reason is that all the values in the network are needed to compute the correct average. We simply can not discard the readings so far even if they do not pass the HAVING test, since readings from other nodes might change the value. Thus this might lead to passing the test.

Partial state relates to the amount of states required for each partial state record. For example the MAX partial state record consists of only a single value, while the AVG partial state record consists of a pair of values. In distributive aggregates, the partial state is simply the aggregate for the partition of data over which they are computed. The size of the partial state record has the same size as the size of the final aggregate. In algebraic aggregates, the partial state records are not themselves aggregates for the partitions, but they are of constant size.

Example of an aggregate query that find the average light pr Epoch over all nodes:

```
SELECT AVG(light)
FROM SENSORS
```
Example of an aggregate query that finds the MAX light, grouped by each floor: (In this case floor is a constant attribute on each mote, determining which floor it is placed in).

```
SELECT MAX(light), floor
FROM SENSORS
GROUP BY floor
```

### 3.6.4 Queries over stored data

As mentioned before, TinySQL includes the ability to write data into the memory of the motes. This is done by first declaring a materialization point with the `CREATE BUFFER` statement, and then run queries that extract data from the mote and write it into the buffer. In TinyDB both the field and table (buffer) name can be up to eight characters long, and the file type has to be, as mentioned in Section 3.4, one of `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32` or `string`. The `u` indicates that the field is of type unsigned. There are two ways to fill the buffer with data. The `CREATE BUFFER` statement described under 3.6.2, both create the buffer and start filling it with data. The second method is to first create the buffer and later fill it with data by launching a query like the following.

```
SELECT field1, field2
FROM SENSORS
INTO buffername
```

This will fill up the table (buffername). The number of fields and the field type must match the description in the `CREATE BUFFER` statement.

The data in the table can be accessed through ordinary queries. It is also possible to join data from this table with data from the sensors. You may not have queries reading and writing to the table running simultaneously. Before any reading from the table takes place the writing has to be stopped by stopping the query that writes into the table (Madden Samuel et al., 2003 D). This in contrast to how a sliding window works in data streams systems like TelegraphCQ (Reiss, 2006), where reading and writing to a sliding window can be done simultaneously. The statement

```
SELECT MAX(light)
FROM buffername
```
will find the maximum value of the field light in the defined table. The selected attribute (light) must be the same name as used in the CREATE BUFFER statement. Finally, the tables in memory are dropped and the internal states associated with them are reset using the statement:

```
DROP ALL
```

**3.7 TAG: Tiny aggregation**

From Section 3.6.3 we know that there are two different approaches on how to perform aggregations. TinyDB uses the distributed approach. This means that aggregates are computed in-network whenever it is possible. In this way there are a lower number of transmissions, latency and power consumption. This is formalized by (Madden Samuel, 2002 A), and Tiny Aggregation – TAG, consists of two phases:

*Distribution phase.* In which aggregates are pushed down into the network.

*Collection phase.* Where the aggregate values are continually routed up from children to parents (during an Epoch every device produces a single aggregate).

The collection phase is divided into small timeslots making sure that parents in the routing tree wait until they have received data from their children before they propagate an aggregated value for the current Epoch. Every aggregate request therefore has to include the interval when the parent will be waiting for the partial state record from its children. Intervals are selected so that there is enough time for the parents to combine partial state records and then in turn propagate their own record to their parent. This requires the clock on the motes to be synchronized as explained in Section 3.6.2. A mote goes through four phases when it receives an aggregated query request.

1. Awaken.

2. Synchronize its clock with its parent.

3. Choose the sender of the message as its parent.
4. Forward the query by setting the delivery interval for children to be slightly before the time its parent expects to see the partial state record.

The Epoch duration has to be long enough so that all nodes can report within the same Epoch. The interval available will be calculated as Epoch-duration / tree-depth. An illustration of the process is given in Figure 3-7.

![Figure 3-7 An Epoch with time slice for each node level (Madden Samuel, 2002 A).](image)

There are differences between a centralized approach and the TAG approach. Centralized aggregates for example have the same communication cost irrespective of the aggregate function, since all data must be routed to the root. The principal advantage of TAG compared to a centralized solution is its ability to decrease the communication considerably when computing aggregates and thereby saving power, which in turn will extend the lifetime of the WSN. The dividing of time into Epochs, and the dividing of Epochs, is a convenient mechanism for idling the processor. TAG has the ability to tolerate disconnections and loss. Lost nodes can reconnect by listening to other node’s state records as they flow up the tree. Another advantage of the TAG approach compared to a centralized approach is that in a centralized approach the nodes on top of the tree are required to transmit significantly more data than the leaf nodes. In TAG, using partial state records, each node is required to transmit only a single message per Epoch regardless of its depth in the routing tree.
As we can see in Figure 3-8, in the server based approach (a) each node is labeled with a number showing the number of messages required to get data from the node to the connected PC. For this little network a total number of 16 messages are required to send all the readings to the PC. In the in-network approach (b) only one message is sent along each edge as the aggregation is performed by the nodes themselves. For the same network a total number of 6 messages are required to send the result to the PC.

The constant maintenance of the topology makes it relatively easy to adapt to changes in the network caused by nodes disconnecting and reconnecting. The cost of the constant maintenance is that it is very power consuming (Madden Samuel, 2002 A).

3.8 Acquisitional query processing

An acquisitional query processor controls when, where and with what frequency data is collected. Further on it decides if sampling should be interleaved with processing and which nodes that have relevant data for a particular query (Madden Samuel et al., 2005). Acquisitional query processing focuses on the new query process opportunity that arises in WSNs. Traditional databases assume that data is provided a priori. In a WSN different attributes can have different costs to sample. When we for instance have sensors for light and magnetometer (mag), the sampling of the mag attribute might be many times as expensive as the sampling of the light attribute. This leads to different ways of handling a query in traditional DBMSs and WSNs. The following query will illustrate this.
SELECT light, mag
WHERE pred1(light)
AND pred2(mag)

In a traditional DBMS the light and the mag attributes are selected and the filters in the where predicate are applied. (Indexes are used when appropriate). An acquisitional query processor can order the attributes and interleave the sampling and the processing in such a way that the cheapest attribute is sampled and checked first, and if that predicate holds, the most expensive one is sampled and checked. In this way, interleaving the sampling and the processing is very power saving when the costs of sampling between the attributes are very different. TinyDB incorporates acquisitional techniques by providing necessary features for deciding where, when and at what frequency data should be acquired from the motes. The syntax for this type of queries is explained in the Section 3.5

### 3.9 The TinyDB GUI

TinyDB comes with a predefined Java GUI for launching queries into the sensor network. When the GUI is started up it starts up three frames, and when a query is launched a forth frame is started up. In Figure 3-9 we see the frames started up when the GUI is started.

![Query Construction](image)

Figure 3-9 TinyDB GUI the query construction and mote commands panel
The Query Constructor frame has two panels, one text frame for writing queries and one graphical interface for constructing queries. After the queries are constructed they are sent into the network by pushing the Send Query button. Multiple queries can be constructed and launched, and thereby run simultaneously on the motes. The result of queries can be logged to a PostgreSQL database by selecting the Log to Database checkbox in the Query Constructor panel. To the upper right in Figure 3-9 we see the Mote Commands frame. This frame can be used for resetting of the motes, setting of radio strength, adding attributes and other. To the lower right we see a frame containing information about which queries that are currently running.

From the Mote Commands frame the three frames in Figure 3-10 can be started. The first is for setting the signal strength on the motes radio. The second is for setting the time interval of the broadcast messages from the rot node. The third is for adding constant attributes to the nodes in the network.

![Maintenance panels in TinyDB](image)

**Figure 3-10 Maintenance panels in TinyDB**

After a query has been launched, the query result frame appears. This frame contains the result of the query.
In Figure 3-11 we see the result frame, where the results are presented in a scrollable table and a plot showing the time vs. the selected attributes. In this frame you can stop a running query or resend the query and even change lifetime or sample rate of a running query. The plot can be changed to show any of the selected items vs. time. The graph can be reset or cleared.
4 Aggregations and GROUP BY issues

In (Madden Samuel, 2003 A) several TinyDB features are described, however the constraints in RAM on the motes prevent TinyDB from having all features implemented at the same time. The total usage of RAM for TinyDB should not exceed 3100 bytes (Madden Samuel et al., 2003 C). In Section 4.1 we look at what options are by default enabled or disabled in TinyDB. Not all parts of TinySQL are implemented and working correctly, and in Section 4.2 we discuss these parts, and argue for the need of their implementation. Section 4.3 discusses the issues related to the aggregation of queries, and in Section 4.4 we discuss the issues related to GROUP BY and HAVING. Finally in Section 4.5 we look at the logging of aggregated queries to the PostgreSQL database.

4.1 Default compile options in TinyDB

The following table is a description of the compile options in TinyDB. As we can see only two options are by default enabled, the kQUERY_SHARING and the kSTATUS. This means that we do not have to enable any options to run simple queries with or without ordinary aggregates.

<table>
<thead>
<tr>
<th>Option</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>kUSE_MAGNETOMETER</td>
<td>Disabled</td>
<td>Only needed when a sensor board with a magnetometer is used.</td>
</tr>
<tr>
<td>kQUERY_SHARING</td>
<td>Enabled</td>
<td>Enable the “query sharing” feature of TinyDB. This allows motes to exchange queries with their neighbors without explicit retransmission of the query.</td>
</tr>
<tr>
<td>kFANCY_AGGS</td>
<td>Disabled</td>
<td>Enables a number of fancy aggregates like the WINAVG which takes the average of data in a buffer created with the CREATE</td>
</tr>
<tr>
<td>Option</td>
<td>Status</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>KEEPROM_ATTR</td>
<td>Disabled</td>
<td>Enable the EEPROM logging attribute that allows a signal to be captured to the EEPROM. The use of this option is not documented in TinyDB.</td>
</tr>
<tr>
<td>CONTENT_ATTR</td>
<td>Disabled</td>
<td>Enable the contention attribute which provides a 16 bit integer indicating the amount of radio contention currently on the radio.</td>
</tr>
<tr>
<td>RAW_MIC_ATTRS</td>
<td>Disabled</td>
<td>This option enables raw microphone or tone detector attributes.</td>
</tr>
<tr>
<td>LIFE_CMD</td>
<td>Disabled</td>
<td>Enable the use of the lifetime option in a query.</td>
</tr>
<tr>
<td>SUPPORT_EVENTS</td>
<td>Disabled</td>
<td>Enable the use of event based queries, as described in Section 3.6.1.</td>
</tr>
<tr>
<td>STATUS</td>
<td>Enabled</td>
<td>Enable the use of the status command, which returns a list of running queries.</td>
</tr>
<tr>
<td>UART_DEBUGGER</td>
<td>Disabled</td>
<td>Enable the use of a debugging display connected over the UART. When this option is used the serial display cannot be used on the base station. It will force the UART to run at 9600 bits per second on all motes except mote id 0.</td>
</tr>
<tr>
<td>MATCHBOX</td>
<td>Disabled</td>
<td>Enable support for the MATCHBOX file</td>
</tr>
</tbody>
</table>
There are other undocumented options, but there is a warning saying: *Probably don’t want to mess with the options below – they control parameters that are set by the options above.*

### 4.1.1 Investigating the kQUERY_SHARING option

This option enables the “query sharing” feature of TinyDB. This allows motes to exchange queries with their neighbors without an explicit retransmission of the query. To investigate this option it is necessary to set up a parent-child relationship between Mote 1 and 2. This means that Mote 1 is the only mote that is within the range of the radio messages from the root node and from Mote 2. Mote 2 is within the range of radio messages from Mote 1 only. Then the power is switched off for the child mote, and a query is sent into the WSN. After the parent mote starts generating results for the query, the power to the child mote is switched on. When also the child mote starts generating results for the query, even without a resend of it, we know that this option works as it is supposed to do. This test was performed without errors.

### 4.1.2 Investigating the kSTATUS option

The kSTATUS option is used to enable the use of the status command. When we look at Figure 3-9 the lower right panel is used for generation of a list of running queries. By pushing the *refresh* button in this panel we get an updated list of queries currently running on the motes. The panel is updated according to the description.

### 4.1.3 Investigating other options

The kLIFE_CMD option was enabled to test the lifetime options in a query like the ONCE keyword. As explained in Section 4.2 we did not manage to get this option to work properly. The kFANCY_AGGS along with the kEEPROM_ATTR were enabled.
when we tried to use the materialization point. We did not manage to get these options to work properly. Many efforts have been done trying to get debugging messages from the motes. The kUART_DEBUGGER and the kMATCHBOX options were enabled with no success in this case.

4.2 Features not implemented in TinySQL

In Section 3.1 there is a list of the features in TinyDB. They are all implemented in the TinyDB application, but we have experienced some challenges with the implementation of TinySQL. Some features in TinySQL are not implemented, and others are not working. The following is a list of features in TinySQL that we have found either not implemented or not working:

- The ONCE keyword in TinySQL does not provide you with a query that produces only result for a single Epoch. The ONCE keyword is intended to give you a single result, but it has no effect on the query.

- The SAMPLE PERIOD 1 FOR 10; is meant to give you a sample once per second for ten seconds. It does not give you one sample per second, and it does not sample for ten seconds. The query runs as if the SAMPLE PERIOD option is not present.

- The SAMPLE PERIOD 10. Is meant to give you samples for ten seconds. The query continues to deliver results also after 10 seconds.

- CREATE BUFFER is a statement used for the creation of a buffer to load the sampled values into, like a sliding window. It looks like the buffer is created but it is not possible to obtain any data out of it.

- Queries with aggregates always produce incorrect results. For the aggregated values the results are always the same for the two motes within the same Epoch.
• The HAVING clause is not implemented at all. A syntax error is given when the HAVING clause is used in a query.

• It is not possible to load data from queries with aggregates to the PostgreSQL database. The necessary tables are not possible to create and the table creation aborts.

TinyDB is not a commercial product and some minor deficiencies are expected. However, the functionality regarding aggregation and HAVING is essential for research in WSNs, and it is vital that this functionality is implemented correctly. The reason why aggregation is essential is twofold. First aggregation is a very important technique to reduce the communication overhead by accumulating data and hence reduce the network traffic. Second we are often not interested in readings from a single node, but in data from groups of nodes. On top of this, HAVING is the only way to do filtering of aggregated data.

It is very difficult to do any form of analysis of the data if we can not first log it to a database. It is therefore important to also correct the error regarding the logging of queries with aggregates to the PostgreSQL database.

4.3 Aggregation issues

4.3.1 Challenges related to correction of aggregates

The correction of aggregates is a traditional maintenance task: Find the error and correct it. The environment in which we are to perform this task, however, is not a traditional environment, but a WSN environment. In Section 2.2.7 we mention some of the extra challenges that exist when developing applications for WSNs, the same challenges exist when maintenance is done.

The main challenge with correction of aggregates is related to finding where the error arises. There are two main areas where the error might arise, on the motes, or on the PC (i.e. the Java API). These two domains are totally different. Their programming
languages are different, the environments they execute in are different, and the available tools for development and debugging of code are different.

For debugging of the Java API we can use debugging messages and run the code inside an IDE like NetBeans. For the code running on the motes there is no such IDE available, and as mentioned before the sensor nodes are usually not equipped with standard input-output devices. These together with the event-driven nature of the application make it hard to debug code which is running on the motes.

To our knowledge there is no established way to debug code on the motes, and without input and output devices which can display messages or give you the opportunity to execute code step by step, debugging is a challenging task. A suitable emulator or simulator could help, and the use of emulators is discussed in Chapter 7.

TinyDB is not well documented and the absence of good system documentation also represents a challenge.

**4.3.2 Deficiencies regarding aggregated functions**

As mentioned, TinyDB produces incorrect results for aggregate queries. In Figure 4-1 we can see the result from an aggregated query launched in TinyDB before any changes have been done to the software. The data selected are nodeid, MIN, MAX and AVG light from sensors, grouped by nodeid.
From Figure 4-1 we see that for the same Epoch both nodes report the same value (or a null value) for the aggregates. Further on there is no correlation between the values generated.

In Figure 4-1, Mote 1 (= node .. 1) should give a value close to 277 each time, and Mote 2 should give a value close to 435 each time. This applies to MIN, MAX and AVG. The blank field occurs when no value is received for that field in that Epoch. This probably happens because of packet loss, which is discussed in the experiments conducted. The results, as we can see, are obviously wrong. There are examples of minimum values larger than maximum values, and average values that do not correspond to the minimum and maximum values. At Epoch 28 both values for MIN seems to come from Mote 2, and at Epoch 30 both values for MIN seems to come from Mote 1. At Epoch 35 the minimum value is larger than the maximum value, and at Epoch 40 the average value is much larger than both the MIN and MAX value.

The source for this error has to be located, and the error corrected, before the aggregate functions can be used properly.
4.3.3 Solution sketch

This section is an overall sketch of the solution design for correction of the error regarding the aggregates. A detailed design is given in Chapter 6.

The main problem with the errors regarding the aggregates is to locate them. In the Java API it is possible to use debugging messages. Debugging messages from the resultVector() method in the QueryResult class show that the result tuples delivered are incorrect. Initially, it is therefore assumed that the error regarding the aggregates is on the motes side. Three possible sources of error on the motes, and one in the Java API, seemed likely:

1. In the aggregating function on the motes. This because an error with the aggregating functions might produce erroneous results like the ones we have.

2. In the GROUP BY function on the motes. This can also be the case because if we have an error in the grouping function on the motes, we might end up with the same value reported from both of the motes.

3. In the TupleRouter on the motes. The TupleRouter is the heart of the TinyDB application. This component is very complicated and this is where the messages that are to be sent out on the network is generated.

4. In the TinyDBNetwork component in the Java API. The TinyDBNetwork component is where messages are received from the network. It is here the network messages are combined to form a query result.

The strategy to use for location of the error is evaluation of the code, and to extend the software with code in such a way that it is possible for us to be certain that the right part of the software has been reached.

By using an emulator we can make certain that the motes produce known fixed values. By altering the values on different places during the execution path of the code we can verify that the right code is reached.
For code that is part of the Java API, the use of debug messages and running the code inside an IDE is helpful for locating errors.

4.4 GROUP BY and HAVING issues

4.4.1 The importance of GROUP BY and HAVING

As mentioned before handling of aggregates is a very important feature for WSNs. When there is an aggregate in a query there is always a GROUP BY clause associated with it. The GROUP BY clause can be given explicit like in:

```
SELECT nodeid, AVG(light)
from sensors
GROUP BY nodeid
```

It can also be given implicitly like in:

```
SELECT AVG(light)
from sensors
```

In the last case the grouping is performed with an assumption that all results belong to the same group. The GROUP BY clause is important because it is the only way to group data in a query.

In TinySQL the HAVING clause is specified as part of the language (see Section 3.5.3), and HAVING is used in several of the examples in (Madden Samuel et al., 2005). However, we know that HAVING is not implemented in TinyDB. Samuel Madden has informed us that none of the public released versions of TinyDB had HAVING implemented. According to him they had implemented enough to do some measurements, but the code has never been released and it is no longer available. As mentioned, aggregation is an important topic in WSNs and filtering on aggregated values can only be done by use of the HAVING clause. It is important to understand the difference between the HAVING and WHERE clause in SQL. (Mullins, 1997) gives a good description of this difference. They are similar in terms of functionality, but they operate on different types of data.
In a query, the WHERE clause is used as a filter. This filter indicates which rows of the data that should be returned, and it operates on detail data rows from tables, views, synonyms and aliases.

As a contrast, the HAVING clause operates on aggregated groups of information. This means that HAVING operates on the result set and not on the detail data rows. The HAVING clause is applied after the GROUP BY is applied on the data. If both a WHERE clause and a HAVING clause are coded in a query the following occurs:

First, the WHERE clause is applied to detailed data rows. Second, the GROUP BY is applied in order to aggregate the data. Finally, the HAVING clause is applied to filter the aggregated rows.

From this we can see that the only way to do filtering on the aggregated rows is by use of the HAVING clause.

The reason we want to implement the HAVING clause is twofold. First, it enhances the TinySQL language and makes it possible to write queries which do filtering on aggregated rows. Second, when HAVING is part of TinySQL it opens up for implementation of HAVING on the motes. We know from Section 3.6.3, that HAVING implemented on the motes is power saving for monotonic aggregates.

### 4.4.2 Challenges related to implementation of the HAVING clause

There are a number of different challenges related to the task of extending the TinySQL query language with a new option. The challenges can be divided in three separated areas.

The first challenge is the modification of the parser for the language. The parser has to be modified to make the HAVING clause a legal option in the TinySQL query language.

The second is the choice between the centralized and distributed approach for the implementation of the handling of the HAVING clause.
The third challenge is related to extending of the GUI which comes with TinyDB. The GUI has to be modified in such a way that construction of queries, that uses HAVING, always generates a legal query.

4.4.3 Solution sketch

This section is an overall sketch of the solution design for implementation of the HAVING clause. A detailed design is given in Chapter 6.

The first step in the task of implementing HAVING in the TinySQL query language is to make the HAVING clause a part of TinySQL. This is done by making changes to the files that define the syntax and vocabulary. The TinySQL syntax and vocabulary are defined in the senseParser.cup and senseParser.lex files.

When these files are changed the HAVING clause is implemented in the language. The next step is to actually handle the option and apply the HAVING clause to the resultset of a query. There exist two approaches to how a HAVING clause might be implemented; a centralized and a distributed approach.

The last step in the implementation is to extend the TinyDB GUI with the option of using the HAVING clause. This is done by adding a checkbox for the HAVING clause to the query construction panel.

4.5 Logging of aggregated queries to PostgreSQL

It is important that logging of results from queries with aggregates into PostgreSQL is working. When this feature is working we can use the tools available in PostgreSQL to analyze data from the queries. We know that logging of non-aggregate queries performs ok. The challenge here is to ascertain why the error arises when we try to log aggregated queries. This is a maintenance task much like the one described in Section 4.3, only with the assumption that this one is easier to find and correct.

When we try to log data from aggregated queries into PostgreSQL, we get an exception in one of the Java components. The task will be to check the code in the methods mentioned in the stack trace that is given when we get the abort.
5 Error searching

This chapter discusses the search for errors regarding aggregates and the logging of data to PostgreSQL. We start in Section 5.1 by discussing the searching for errors regarding the aggregates, and continue in Section 5.2 with search for the error regarding the logging of data to PostgreSQL.

5.1 Incorrect aggregates

In this section we first look at how the search for the error is conducted on the software running on the motes. Then we look at the software in the Java API.

5.1.1 Investigating the software on the motes

From the start the most likely place to find the error seems to be in the aggregating functions on the motes, since the debugging messages from the Java API show us that the result tuples have incorrect data. In TinyDB all aggregates are nesC components that implement the aggregate interface described in Section 3.6.3. The AggregateUseM component uses the provided methods in this interface to control an aggregate throughout its lifetime.
In Figure 5-1 we can see how the aggregation-related tasks in TinyDB are connected to each other. We can here see that all aggregates have to provide the methods shown in the \textit{AggregateUseM.nc} box in the figure above. The ID parameter relates to the type of aggregate. SUM has id equals to 1, MIN has 2, MAX has 3, and in this way all aggregates are given a unique id. In the \textit{AggOperatorConf.nc} aggregates are, from their ID, connected to the appropriate aggregate function like \textit{AvgM.nc} for the AVG aggregate.

From the information gathered from the query used in Figure 4-1 we can assume that errors are not related to one specific aggregate, error occurs to MAX, MIN and AVG aggregates. The assumption is that if the error is found for one of the aggregates the rest would probably have the same type of error. The following is a list of the NesC source code for the \textit{AvgM.nc}. The leading comments and disclaimers are excluded from the code. All methods are shown, but only the methods concerning calculation of the aggregate are commented.

```nescc
includes Aggregates;

module AvgM {
    provides {
        interface Aggregate;
    }
}

implementation {

    command result_t Aggregate.merge(char *destdata, char *mergedata,
                                      ParamList *params, ParamVals *paramValues) {
        AverageData *dest = (AverageData *)destdata;
        AverageData *merge = (AverageData *)mergedata;
        dest->sum += merge->sum;
        dest->count += merge->count;
        return SUCCESS;
    }

    command result_t Aggregate.update(char *destdata, char* value,
                                       ParamList *params, ParamVals *paramValues) {
        AverageData *dest = (AverageData *)destdata;
        int16_t val = *(int16_t *)value;
        dest->sum += val;
        dest->count++;
        return SUCCESS;
    }

    command result_t Aggregate.init(char *data, ParamList *params,
                                     ParamVals *paramValues, bool isFirstTime) {
        AverageData *mydata = (AverageData *)data;
        mydata->sum = 0;
        mydata->count = 0;
    }
```
return SUCCESS;
}

command uint16_t Aggregate.stateSize(ParamList *params, ParamVals *paramValues) { return sizeof(AverageData); }

command bool Aggregate.hasData(char *data, ParamList *params, ParamVals *paramValues) { return TRUE; }

command TinyDBError Aggregate.finalize(char *data, char *result_buf, ParamList *params, ParamVals *paramValues) {
    AverageData *mydata = (AverageData *)data;
    *(int16_t *)result_buf = (mydata->count == 0 ? 0 : mydata->sum / mydata->count);
    return err_NoError;
}

command AggregateProperties Aggregate.getProperties() { return 0; }
}

In Section 3.6.3 we explained, in general, how an aggregate function is implemented with an initializer, a merge function and an evaluator. In the source code these modules have the names Aggregate.init, Aggregate.merge and Aggregate.finalize. In addition we must look at the Aggregate.update.

In Figure 5-1 we see that these functions have to be provided by all aggregates.

The Aggregate.init function is called once to initialize the aggregate. As we can see, it only sets the sum and count variable to zero.

The Aggregate.update function is used for merging new readings from the current node with the values already saved. As we can see it increments the count by 1 and adds the new value to the sum variable.

The Aggregate.merge function is used for merging readings from its child nodes with the values for the current node. As we can see it increments the count by the count value received and updates the sum value in the same way.

Finally we see that the Aggregate.finalize function returns either zero or sum divided by count, depending on the number of readings captured.
Investigation of the code for the *AvgM.nc* does not reveal any errors. The next step is to make sure that the code is reached at all. Remember there are no input output devices attached to the motes where we could have sent debug messages.

The solution to this problem is to add some lines of code inside the `Aggregate.update` and `Aggregate.merge` functions. These few lines of code test to see if the readings are from Mote 1, and if so change the value with a constant value of our knowledge. This way it is easier to follow the generated message on its way through the system. Remember that the search for errors is performed with an emulator, and we can test with known values for the motes.

The next step is to investigate the grouping mechanism in TinyDB. When a query contains an aggregate it will always be grouped. Grouping computes aggregates over partitions of sensor readings, if no GROUP BY clause is specified it is simply assumed that all sensor readings for this query belongs to the same partition.

The grouping mechanism functions as follows: a leaf node simply marks the sensed value with the group number and forwards the readings to its parent. When a parent node receives a message from a child it checks the group number to see if it is in the same group, if so it combines the readings using the merge function. If they are in different groups, the parent node stores the values of the child group along with its own value for forwarding in the next interval. If a message from another child arrives with a value in either group, it is merged with the appropriate aggregate. During the next interval the node will send out the value for all groups it collected data from during the previous interval. The information about the readings is combined into a single message as long as the message size permits it.

The last step is to investigate the TupleRouter to see if the messages are somehow corrupted here before they are sent out on the network. As mentioned in Section 3.2.2 the TupleRouter is the heart of the TinyDB system. It is the largest component in the system and the most complex one. The task is to try to follow the messages that are generated and try to find out if they are ok or not. A roadmap for this examination is

---

15 The TupleRouterM.nc consists of 4000 lines of code.
the component diagram shown in Figure 3-1. A description of the message flow for aggregated queries is found in the Appendix B.

If no errors are detected it seems as the code on the motes is correct. The next step then is to continue the search along the path the messages are floating. The focus has to shift from the code on the motes to the code in the Java API.

5.1.2 Investigating the networking software in the Java API

We know from the debugging messages in the resultVector() method in the QueryResult class, that the result tuples them selves are incorrect. When we are looking for the error in the Java API, we have to look at the generation of the QueryResult object.

The TinyDBNetwork object handles all incoming active messages from the serial port, and the most likely place to start searching for the error is here. By analyzing the network message received by TinyDBNetwork we can see if the data received is correct.

TinyDBNetwork uses the constructor in the QueryResult class to construct new results. This class requires a TinyDBQuery and a QueryResultMessage as parameters for the construction of a new QueryResult. The TinyDBQuery is unique for this query, and we have only one object representing it. The QueryResultMsg is an object with information from a single reading belonging to the query.

![Figure 5-2 Construction of a QueryResult Object](image)
The TinyDBNetwork might receive many messages regarding the same Epoch for a query. The logic regarding the aggregate queries in TinyDBNetwork is as follows:

When a message is received from the network; check if there already exists a QueryResult for this query and Epoch. If so, merge this result with the existing QueryResult, and wait for the next message to arrive. If not, construct a new QueryResult with this message and the TinyDBQuery as parameters. Thereafter, send out the previous QueryResult to the listeners, and wait for the next message to arrive.

As we can see when the query is an aggregated query new QueryResultMsg are merged into the QueryResult. This is done until a message for a later Epoch arrives. The QueryResult is then sent to the listeners for that query.

The main difference between an aggregated and a non-aggregated query is that the non-aggregated queries are sent to the listeners for that query before the next messages arrives from the network. This can not be done for aggregated queries because they are not finalized on the motes. Average is finalized in the Java API. This is why we have to merge new arriving messages for aggregated queries with the existing QueryResult.

As noted earlier, this merging of results is done until messages for the next Epoch arrives. When this happens the error arises.

When a message for the next Epoch arrives, some of the data in the TinyDBQuery regarding aggregated queries are updated with information from the QueryResultMsg. The result of this is that the TinyDBQuery contains information from the message belonging to the next Epoch.

The problem comes down to the fact that a new QueryResult object can not be generated from the TinyDBQuery and the QueryResultMsg before the old QueryResult object is released and sent to the listeners. This is because some of the fields are then overwritten with new values, and the error arises.
The solution to this problem is described in Chapter 6 where we discuss the design for the new implementation of TinyDB.

### 5.2 Logging aggregate queries to PostgreSQL

In Section 4.5 we mentioned that logging of non-aggregated queries works fine and that logging of queries with aggregates result in an exception in Java. From the stack trace we can see that the error arises in the `DBLogger` object when we try to create the table for holding information about the values in a query.

The reason for this is that for aggregate queries, the column heading is generated with a space in its name. It is not legal to have a column name in a table that contains one or more spaces. The problem is fixed by changing the `getColumnHeadings()` method in the `TinyDBQuery` class. This method has to handle aggregate columns separately by building up their names without spaces. This will be explained in more detail in Chapter 6 where we look at the detailed design.
6 Detailed design of the new implementation

In this chapter we look at the design of the new implementation of TinyDB in more detail. The first part is where we extend the language with a feature that was not implemented, and the second part is where we look at how we can correct the errors regarding aggregates and logging to PostgreSQL. In Section 6.1 we take a closer look at the implementation of the HAVING clause. In Section 6.2 we present the design for correct handling of aggregates before we present the solution for the problem with logging of aggregated queries in Section 6.3.

6.1 The HAVING clause

Implementation of HAVING consists of four steps. The first is to make the HAVING clause a part of the TinySQL query language. The second is to make changes to the Java API. The third step is to apply the HAVING clause to the resultset. The fourth step is to change the GUI accordingly.

6.1.1 Making HAVING part of the TinySQL query language

The syntax of the TinySQL language is described in Section 3.5.3. TinySQL is formally defined using the Backus–Naur Form known as BNF. BNF is widely used to describe the syntax and grammar of computer programming languages. Two files have to be changed to extend the TinySQL with the HAVING expression, the senseParser.cup and the senseParser.lex. In addition the TinyDBQuery class has to be changed. The changes to the TinyDBQuery class are explained in Section 6.1.2.

To construct a parser for TinySQL a LALR\(^\text{16}\) parser generator for Java named CUP (Constructor of Useful Parsers) is used. CUP is a system which serves the same role as the widely used program YACC\(^\text{17}\). CUP is written in Java, uses specifications

\(^{16}\) LALR – Look-ahead LR parser. A LR parser is a parser for context-free grammars that reads input from Left to Right and produces a rightmost derivation. A LR parser is said to perform bottom-up parsing because it attempts to deduce the top level grammar productions by building up from the leaves.

\(^{17}\) YACC is a program that serves as the standard parser generator on many Unix systems. The name is an acronym for “Yet Another Compiler Compiler”.

including embedded Java code, and produces parsers which are implemented in Java. CUP is created by Scott Hudson, but it is now maintained and improved at the Technical University of Munich.

In `senseParser.cup` the syntax (i.e. the structure) of TinySQL is defined using BNF. In this file the necessary code for making HAVING a part of TinySQL has to be declared. In detail the necessary changes are:

- We need a structure to hold the necessary information about the HAVING clause. This is solved by declaring a class holding the information.
  ```java
class HavingStmt {
    AggOp aop;  // Aggregate option {SUM,MIN,MAX,AVG...}
    SelOp op;   // Having option {=,>,<,<> >= <=}
    short konst;  // Numeric constant to test against
    FieldInfo fi;  // Field to apply the HAVING clause to
  }
```

As we can see from the code all the necessary information about the HAVING clause is specified here.

- The HavingStmt object needs to be constructed and we need a variable declared to hold the information. A vector is declared for this purpose.
  ```java
  Vector havings = new Vector();
  ```

By declaring the variable as a vector, the code is prepared to handle more than one instance of the HAVING clause. This is useful if the TinySQL language is extended with sub selects. Then HAVING can be applied in sub selects as well, without changes to the structure that holds the data for the HAVING part of the query. For now the `havings` vector will only hold zero or one item.

- We need to deliver information to the `TinyDBQuery` about the query we are constructing. One of the things we must provide information about is the presence or absence of a HAVING clause. This is done by passing a boolean variable to `TinyDBQuery`. Therefore we need to declare a boolean variable `hasHaving` to indicate if this query has a HAVING expression.
  ```java
  Boolean hasHaving = false; //default set to false
  ```
• We need to make HAVING a legal option in the language. This is done by adding it to the list of terminals.

```
terminal ...
terminal SELECT, FROM, WHERE, GROUP_BY, HAVING

```

When this line is extended with HAVING as above, we have made it a legal option in the language.

• The non terminals have to be extended with an entry for description of the HAVING clause. This is done by adding `having_stat` to the list of non terminals.

```
non terminal ...
non terminal having_stat;
non terminal ...

```

• The next step is to declare where we actually are handling the HAVING option. This is done by appending the non terminal `query_core` expression with an expression including the `having_stat`. This will be a copy of the existing `query_core` expression with code for handling of the `having_stat` added to it. In detail it will look like this:

```
query_core::=select_stat from_stat where_stat
group_by_stat:gf epoch_stat
{: ........
:}
| select_stat from_stat where_stat group_by_stat:gf

```

The symbol `|` has the meaning of `or` in BNF notation. The code that is added is the following:

```java
// Add the having clause to the TinyDBQuery
e = havings.elements();
while (e.hasMoreElements()) {
    HavingStmt hs = (HavingStmt)e.nextElement();
    HavingExpr hEx = new
    HavingExpr(hs.aop,hs.fi.qf,hs.op,hs.konst);
```
tinyDBQuery.addHexprs(hEx);
}
tinyDBQuery.setHaving(hasHaving);

This code is adding the HAVING expression to the TinyDBQuery and the hasHaving variable in TinyDBQuery is set accordingly (i.e. true when HAVING is present otherwise false).

- The last step is to code the non terminal having_stat. A sensible place to locate this code is after the non terminal group_by_stat. The purpose with this code is to create a new HavingStmt object and populate it with the given parameters, and to make sure this is an aggregate field. In detail it will look like this:

```java
having_stat ::= HAVING agg:ao arith_expr:aObj RPAREN rel_op:ro CONSTANT:c
{:
    // System.out.println("in having_stat with agg = " + ao.toString());
    HavingStmt h = new HavingStmt();
    ArithExpressionClass a = (ArithExpressionClass) aObj;
    h.aop = ao;
    h.op = ro;
    h.konst = c.shortValue();
    h.fi = a.finf;
    havings.addElement(h);
    hasHaving = true;
    boolean funnet = false;
    Enumeration e;
    e = selList.elements();
    while (e.hasMoreElements()) {
        FieldInfo finf = (FieldInfo) e.nextElement();
        if (finf.isAgg) {
            if (h.aop.toString().equals(finf.af.fieldOp.toString()) &&
                h.fi.qf.getName().equals(finf.af.fieldData.finf.qf.getName()))
                funnet = true;
        }
    }
    if (!funnet) {
        SensorQueryer.errorMessage = "HAVING must be applied on an aggregated field in the list of selected fields."
        return null;
    }
};
```

This code evaluates the HAVING option and see to it that HAVING is only applied to aggregate expressions. Further on, it constructs a new HavingStmt and adds it to the havings vector.
These are the changes necessary in the *senseParser.cup* to make HAVING a part of the TinySQL syntax.

In *senseParser.lex* the semantics (i.e. meaning) of TinySQL is defined. In this file we have to define HAVING as part of TinySQL’s vocabulary. To be consequent with the rest of the vocabulary it is necessary to define that HAVING must be typed with lower or uppercase letters, but not with a mix of them. As we know, HAVING and GROUP BY are closely related to each other. Therefore, in the *senseParser.lex* file, we choose to let the definition of HAVING consequently follow the definition of GROUP BY. There are only two places to alter the file. The first place to add code is where all the key-words are defined. The new code will look like this:

```
GROUP_BY="GROUP BY"|"group by"
HAVING="HAVING"|"having"
```

The second place to alter the file is where the symbols are returned, and the new code will look like this:

```
<YYINITIAL> {GROUP_BY} {
    return new Symbol(sym.GROUP_BY, new TokenValue(yytext(), yyline, yychar, sourceFilename));
}
<YYINITIAL> {HAVING} {
    return new Symbol(sym.HAVING, new TokenValue(yytext(), yyline, yychar, sourceFilename));
}
```

The lines with group_by are not altered but are shown here to get a better understanding of where to alter the code. After the files have been changed, it is necessary to run the make command to generate a new parser for TinySQL. The makefile is delivered as part of the TinyDB implementation.

### 6.1.2 Changes in the Java API

There are several classes in the Java API that have to be changed to implement the handling of the HAVING clause. One new class, *HavingExpr*, has to be designed and
implemented. This is a class designed to hold the information about the HAVING clause in a query. With the disclaimers removed the **HavingExpr** class looks like this:

```java
package net.tinyos.tinydb;

/** HavingExpr represents a having expression; having is of the form:

HAVING AggOp(QueryField) SelOp Constant

Where AggOp(QueryField) is part of the select list in the Query. */
public class HavingExpr  {
    public HavingExpr(AggOp ao, QueryField field, SelOp op,
                        short value) {
        this.aop = ao;
        this.qf = field;
        this.op = op;
        this.konst = value;
    }

    public boolean isAgg() {
        return true;
    }

    public short getFieldId() {
        return qf.getIdx();
    }

    public AggOp getAggOp() {
        return aop;
    }

    public QueryField getQueryField() {
        return qf;
    }

    public SelOp getSelOp() {
        return op;
    }

    public short getConstValue() {
        return konst;
    }

    public String toString() {
        return ("Having " + aop.toString() + "(" + qf.getName() + ") " + op.toString() + "" + getConstValue() + 
\n";}

    public String getExpr() {
        return (aop.toString() + "(" + qf.getName() + ")";}

    private AggOp aop; // aggregate option min, max, avg, sum, count
    private QueryField qf; // selected field light, temp, noise
    private SelOp op; // the selected option =,<>,>,>=,<,<=
    private short konst; // a constant value eks 200
}
```
As we can see, in this class all the variables are made private and they are reached only by public methods. The variables are initialized during the construction of the class. We also see that the variables in this class correspond to what is in the HavingStmt class used in the senseParser.cup file. The general form of the HAVING clause has the form:

\[
\text{HAVING } \text{AggOp} \ (\text{QueryField}) \ \text{SelOp} \ \text{Constant}
\]

An example of use might be:

\[
\text{HAVING AVG(light)} > 200
\]

In addition to this new class, three existing classes in the Java API have to be changed.

The first class to alter is the TinyDBQuery. The changes to this class have to be done when we change the senseParser, but all changes to the TinyDBQuery class are explained here. The TinyDBQuery is the Java data structure that represents a query running (or to be run) on a set of motes. This is the place where the HAVING expression is kept for evaluation. The TinyDBQuery class has been extended with the necessary information for handling the HAVING clause. The data types used are standard Java data types and the changes are the following:

In the constructor of the TinyDBQuery the ArrayList is allocated.

\[
\text{hexprs} = \text{new ArrayList();}
\]

Three new variables are added:

\[
\text{private boolean hasHavng = false;}
\\text{private ArrayList hexprs;}
\text{private int hpos = 0;}
\]

The first variable indicates if a HAVING clause is contained in this query. It is default set to false. An ArrayList data type holds the list of HavingExpr. As explained before the list contains only one item, but it is prepared for handling queries with more than one HAVING clause in them. The third variable is an int data type, used as an index into the list of HAVING expressions pointing at the first free position in the list.
A public setHaving method for the private hasHaving variable is defined. This is used by the parser.

```java
public void setHaving(boolean having) {
    hasHaving = having;
}
```

A public method to check if the query contains a HAVING clause is provided.

```java
public boolean hasHaving() {
    return hasHaving;
}
```

To be able to actually add the HAVING clause to the query a public method addHexprs is added to TinyDBQuery.

```java
public void addHexprs(HavingExpr h) {
    hexprs.add(hpos++,h);
}
```

A public method getHavingExpr is provided to get HavingExpr #n from the query. An IndexOutOfBoundsException(n) is thrown if it occurs.

```java
public HavingExpr getHavingExpr(int i) throws ArrayIndexOutOfBoundsException {
    if (i >= numHavingExprs() || i < 0) throw new ArrayIndexOutOfBoundsException(i);
    return (HavingExpr) hexprs.get(i);
}
```

Finally the public toString() method needs to be extended with information about the HAVING clause.

```java
result += numHavingExprs() + " having exprs in query: \n";
for (i = 0; i < numHavingExprs(); i++) {
    try {
        hexp = getHavingExpr(i);
        result += hexp.toString();
        result += "  " + hexp.getFieldId();
    } catch (IndexOutOfBoundsException e) {} 
}
```
The approach used in the update of TinyDBQuery is the standard object oriented approach. The data of the object is made private with public methods to get and set the values in the variables.

When the updates to the TinyDBQuery are fulfilled we have extended the language with the HAVING clause, and we have updated the TinyDBQuery to keep the necessary information about it. The next step is to apply the HAVING clause to the query. In Section 6.1.3 we argue for the approach selected. The centralized approach is chosen, and this means that we can apply HAVING either on the root node on the motes, or in the Java API. It is considered easier to apply it in the Java API, therefore this approach is taken. The next step to figure out is where in the API we want to apply the HAVING clause.

In the Java API there are several different alternatives to where the HAVING clause might be applied; in TinyDBNetwork class, in QueryResult class or in the listeners. A complete result set for the query is constructed by the creation of a QueryResult object in TinyDBNetwork. It is possible to change the TinyDBNetwork to apply the HAVING expression to the resultset. The advantages of doing it here is that it is done as early in the process of handling the data as possible. The disadvantage is that TinyDBNetwork is made for handling network traffic and for converting queries between appropriate formats for sending and receiving messages over the network.

The second alternative is in QueryResult. The resultVector() method in this class returns a vector of strings corresponding to the result tuple for this query result. This method is used to get the result for the queries in the GUI. The advantage with applying the resultset here is that it is here the result sets are handled, and all the data needed to apply the HAVING expression are present. Another advantage is that it is relatively easy to apply the HAVING expression here. The disadvantage is that a null Vector might be returned and applying the HAVING expression might have been done in an earlier step in the process (in TinyDBNetwork or on the motes). This disadvantage is small compared to the advantage of handling the HAVING expression here.
The third alternative is to let the HAVING expression be applied by the listeners. The advantage is that we no longer have to handle null Vectors. The disadvantage is that the implementation has to be done in all listeners and it would be no consistency in the API. Handling of the HAVING expression is then pushed to the programmer. This solution seems to be the least attractive of them all.

The fourth alternative is to provide the QueryResult class with a public boolean method which checks to see if the query passes the HAVING test. This method can then be used both inside the QueryResult object and in objects where we have a QueryResult object available. The advantage with this approach is that it does the actual testing where we have a complete resultset, and it can be applied early, like in the TinyDBNetwork. This approach has another appealing quality, we do not have to handle null values. The latter is because we simply have the test before we send the query to the listeners.

Of the four methods discussed the fourth one is the most appealing, and this approach is taken when HAVING is applied to the query.

The QueryResult object receives a query result in the form of an array of bytes read of the network and a TinyDBQuery object. It parses the bytes and provides a number of utilities for user of the object to read the data that this object contains. In this object we add a new method named pasHaving, which is used to check if this object passes the HAVING test. The actual code is as follows:

```java
public boolean pasHaving() {
    // Does this result pass the Having test
    if (!myQuery.hasHaving()) return true; //Having test pass if no having

    Vector v = new Vector();
    int loc = 0;
    // First some house keeping to prepare for the test
    // include epoch
    v.addElement(new Integer(myEpoch).toString());

    if (myQuery.grouped()) {
        v.addElement(new Integer(myGroup).toString());
        loc++;
    }
    for (int i = 0; i < myQuery.numExprs(); i++) {
        if (myQuery.getExpr(i).isAgg()) loc++;
        if (valid[i]) {
            AggOp agg = ((AggExpr)myQuery.getExpr(i)).getAgg();
            String aggStr = "";
```
agg.finalizeValue();
aggStr = agg.getValue();
if (v.size() <= loc) v.setSize(loc + 1);
v.setElementAt(aggStr, loc);
}

// Default we pass the test
boolean passRes = true;
HavingExpr hexpr;
Vector hdrs = myQuery.getColumnHeadings();
hexpr = myQuery.getHavingExpr(0); // Pic first for now
String ex = hexpr.getExpr();      // ex. MAX(light)
int id = 0;
for (int j = 0; j < hdrs.size(); j++) {
    if (hdrs.elementAt(j).toString().equals(ex)) id = j;
}
short lim = hexpr.getConstValue(); // const to test against
String res;
res = (String) v.get(id);         // col value
short sRes;                       // Result as a short
Integer i = new Integer(0);
// Do the actual testing
try {
    i = new Integer(res.toString());
sRes = i.shortValue();
    switch (hexpr.getSelOp().toByte()){
        case SelOp.OP_EQ:  passRes = (sRes == lim); break;
        case SelOp.OP_NEQ: passRes = (sRes != lim); break;
        case SelOp.OP_GT:  passRes = (sRes >  lim); break;
        case SelOp.OP_GE:  passRes = (sRes >= lim); break;
        case SelOp.OP_LT:  passRes = (sRes <  lim); break;
        case SelOp.OP_LE:  passRes = (sRes <= lim); break;
        default: passRes = false;
    }
} catch (Exception e) {
    System.out.println("Error : Id = " + id + " Content to
test : " + res);
    passRes = false;
sRes = -1;
}
return passRes;

This method simply tests the content of the resultset against the HAVING clause and
sees if it passes. When no HAVING clause is present, it will always pass the test. We
see that this method is defined as public and it can be used both inside and outside the
QueryResult object. When this method is implemented we have the right tool to
apply the HAVING clause. This can be done in the QueryResult object for the GUI by
adding the following line of code to the ResultVector method in the QueryResult
class.

if (this.pasHaving()) return v;
else return null;

To prevent the different listeners to get result sets that do not pass the HAVING test the following lines of code is added in the addQueryResultToListeners method in TinyDBNetwork class:

/*
    check that this QueryResult will pass the Having test,
    if no HAVING is applied it will always pass.
*/
if (nqr.pasHaving()) return;

nqr is the new QueryResult object that is to be sent to the listeners.

When all this is done we have implemented HAVING with a centralized approach in TinySQL.

6.1.3 Applying the HAVING clause to the resultset

In Section 3.6.3 we presented two different approaches to how aggregates could be implemented in WSNs; the distributed and the centralized approach. The same approaches can be used for implementation of the HAVING clause. It can be applied distributed on each mote or it can be applied in a centralized manner. There are advantages and disadvantages with both approaches.

The main advantage with the centralized approach is that it is easy to implement. In a centralized approach you want to implement the HAVING clause at the root node in the tree and therefore it might as well be implemented in the Java API. This means that no changes have to be done on the software installed one the motes. The disadvantage with a centralized approach is that it is not power saving. In a centralized approach we risk sending data over the network which later will be discarded when the HAVING clause is applied to a query.

The main advantage with a distributed approach is that it is power saving. A distributed approach reduces the amount of data that has to be sent over the network, because when the HAVING clause is applied on the motes, data that does not comply
with the filter in the HAVING clause is rejected, and not sent on the network. There are two disadvantages with a distributed approach. First it is much harder to implement than the centralized approach, and second, not all types of aggregates can have the HAVING clause applied on the motes. As mentioned before the finalization of the Average aggregate can not be done before the root node, because it is not a monotonic function (Section 3.6.3). From this we see that a distributed approach can not replace a centralized approach for all the different aggregates, but the opposite is possible. Even when a distributed approach is implemented some of the aggregating functions have to be done on the root node or in the Java API. This is the handling of HAVING for non-monotonic aggregates like AVG. This means that some kind of centralized handling, for some of the aggregates, has to be implemented.

Based on this we choose the centralized approach for the implementation of the HAVING clause. The reason for this, is that we want to be able to not only describe how it might be implemented, but also actually perform the implementation. It could be hard to achieve this within the time available if the distributed approach is taken.

### 6.1.4 Changes to the GUI

When all the necessary changes are made to the Java API, changes have to be done in the GUI to enable the use of the HAVING clause in a Query. In Figure 6-1 a red ellipse is drawn around the visual changes in the GUI. The HAVING checkbox can not be selected unless there is an aggregate to apply the HAVING expression on.
Two classes have to be changed in order to apply the changes as illustrated above. The `MainFrame` class and the `GuiPanel` class have to be changed to allow for the use of the HAVING clause to be implemented. In `MainFrame`, only the height of the frame has changed from 535 to 575 pixels to give room for the new extended `GuiPanel`.

```
static final int FRAME_HEIGHT = 575;
```

The rest of the changes are in the `GuiPanel` class and are the following:

```
static final int HVN_LABEL_WID = 100; //width of "having" label
static final int HVN_AOPS_WID = 120; //width of having operations combo box
static final int HVN_OPS_WID = 100; //width of having operations combo box
static final int HVN_CONST_WID = 60; //width of having constant field box
```

The four variables are defined as static final int, and are used to hold information about the width of the different parts of the HAVING options in the GUI. Five variables are used for holding information for the layout of the HAVING options.

```
static final int HVN_LABEL_LEFT = CHECK_BOX_LEFT + CHECK_BOX_WID;
static final int HVN_AOPS_LEFT = HVN_LABEL_LEFT + HVN_LABEL_WID;
static final int HVN_OPS_LEFT = HVN_AOPS_LEFT + HVN_AOPS_WID;
```
As we can see, we calculate the leftmost position of the different parts of the HAVING options. This dynamic way of calculating where on the panel different objects are placed makes it easy to add new objects to the panel.

The position of the HAVING options is given with:

```java
static final int HVN_PANEL_TOP = GROUP_PANEL_TOP +
GROUP_PANEL_HEIGHT + BUT_SEP;
```

The panel following the HAVING panel is then calculated as:

```java
static final int SELECT_PANEL_TOP = HVN_PANEL_TOP +
HVN_PANEL_HEIGHT + BUT_SEP;
```

In the initComponents() method the following lines of code are added to initialize the panels and make them listen for actions which regards them. When the HAVING checkbox is selected and when changes are made to the selection, the updateSQL() method is called to ensure that the generated SQL is always in a correct state.

```java
selectHvnPanel = new JPanel();
selectHvnPanel.setFont(normalFont);
hvnPanel = new JPanel();
hvnCheckBox = new JCheckBox();
hvnLabel = new JLabel();
hvnAOps = new JComboBox();
hvnAOps.setLightWeightPopupEnabled(false);
hvnAOps.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        JComboBox cb = (JComboBox)evt.getSource();
        String op = (String)cb.getSelectedItem().toString();
        if (!op.equals("") ) {
            hvnAOps.setEnabled(true);
        } else {
            hvnAOps.setEnabled(false);
        }
        updateSQL();
    }
});

hvnOps = new JComboBox();
hvnOps.setLightWeightPopupEnabled(false);
addOpsToMenu(hvnOps);
hvnOps.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        JComboBox cb = (JComboBox)evt.getSource();
        String op = (String)cb.getSelectedItem().toString();
        if (!op.equals("")) {
            hvnOps.setEnabled(true);
        } else {
        }
    }
});
```
The following code is also in the initComponents() method. It disables the use of HAVING by default. Remember we must have an aggregation in the query before we can use the HAVING option. Finally it adds the different parts of the HAVING options to the hvnPanel variable.

```java
selectHvnPanel.setLayout(new java.awt.GridLayout(3, 0, 5, 5));
selectHvnPanel.setBorder(new BevelBorder(BevelBorder.LOWERED));

hvnPanel.setLayout(new AbsoluteLayout());
hvnCheckBox.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        hvnCheckBoxAction(evt);
    }
});

hvnPanel.add(hvnCheckBox, new AbsoluteConstraints(CHECK_BOX_LEFT, 0, CHECK_BOX_WID, BUT_HEIGHT));
hvnLabel.setText("HAVING  ");
hvnPanel.add(hvnLabel, new AbsoluteConstraints(HVN_LABEL_LEFT, 0, HVN_LABEL_WID, BUT_HEIGHT));
hvnPanel.add(hvnAOps, new AbsoluteConstraints(HVN_AOPS_LEFT, 0, HVN_AOPS_WID, BUT_HEIGHT));
hvnPanel.add(hvnOps, new AbsoluteConstraints(HVN_OPS_LEFT, 0, HVN_OPS_WID, BUT_HEIGHT));
```
When the button with the <<< is pushed, we are to remove the selected field from the query. When this happens we have to check if this influences on the use of HAVING. This is done by calling the removeHvnField method inside the removeSelButtonActionPerformed method.

```java
removeHvnField(value.toString());
```

The removeHvnField method is discussed later in this section. In equal manner when an aggregated field is added to the query, it has to be added into the list of available aggregates to test in the HAVING clause. This is done by calling the addHvnField method inside the addSelButtonActionPerformed method.

```java
if (op != null) {
    addHvnField(sf);
}
```

The variable op is an AggOp object, representing an aggregate option. We want to allow for use of the HAVING clause, only when it is appropriate. This means when the query contains a GROUP BY option. To achieve this we enable HAVING when the GroupCheckbox is selected and disable it when it is unselected. This is done by two calls inside the groupCheckBoxAction method.

```java
hvnCheckBox.setEnabled(true); // Permit having when group by is selected
hvnCheckBox.setEnabled(false); // Only permit having when group by is selected
```

To be consistent, every time the HAVING checkbox is selected or unselected we call a method to enable or disable the HAVING options.

```java
/** Called when the having check box is selected / deselected
   * Have to make sure that this is OK.
   */
private void hvnCheckBoxAction(ActionEvent evt) {
    if (hvnCheckBox.isSelected()) { //user selected the box
        if (checkSelectionConsistency(selectDestList,null)) {
            hvnLabel.setEnabled(true);
            hvnAOps.setEnabled(true);
        }
    }
}
```
The SQL string for the current query is generated in the updateSQL method. It has to be altered to include the HAVING option.

```java
if (hvnCheckBox.isSelected()) {
    String hvnAOp = (String) hvnAOps.getSelectedItem().toString();
    if (!hvnAOp.equals(""))
        sql += "HAVING " + hvnAOps.getSelectedItem().toString() + hvnOps.getSelectedItem().toString() + hvnConst.getText();
}
```

The two private methods mention earlier to add and remove entries in the list of available fields for the HAVING clause is defined as follows.

```java
/** Add an entry to the list of available fields for the having clause */
private void addHvnField(SelectionField sf) {
    int i = hvnAOps.getItemCount();
    hvnAOps.addItem(sf.toString());
    hvnCheckBox.setEnabled(true);
}

/** Remove an entry from the list of available fields for the having clause */
private void removeHvnField(String s) {
    for (int i = 0; i < hvnAOps.getItemCount(); i++){
        if (hvnAOps.getItemAt(i).toString().equals(s))
            hvnAOps.removeItemAt(i);
    }
    if (hvnAOps.getItemCount() == 0){
        hvnCheckBox.setSelected(false);
        hvnCheckBox.setEnabled(false);
    }
}
```
Finally we have the list of variables that have to be added.

```java
private JPanel selectHvnPanel; // The panel for HAVING
private JLabel hvnLabel; // The label for HAVING
private JCheckBox hvnCheckBox; // The checkbox for HAVING
private JComboBox hvnAOps; // The available aggregates
private JComboBox hvnOps; // The options
private JTextField hvnConst; // The constant to check against
```

This is in detail what has to be done in the `GuiPanel.java` file to extend it with the option of using the HAVING clause. When this is done we have extended TinySQL with the HAVING clause, and we have implemented the necessary changes in the Java API to handle it and we have updated the GUI accordingly.

### 6.2 The aggregates

We remember from Section 5.1 that the problem with incorrect results for aggregated queries arises because a new `QueryResult` object is generated from the `TinyDBQuery` and the `QueryResultMsg` before the old `QueryResult` object is released and sent to the listeners. Some of the fields in the `QueryResult` object are then overwritten with new values, and the error arises.

The idea used in the old implementation is good; For aggregate queries collect messages from the network until messages from the next Epoch arrive, and then send out the `QueryResult` from the previous Epoch. For non-aggregated queries just create the `QueryResult` and send it out to the listeners.

The same idea is used in the design of the new implementation. The main difference lies in that we wait with the creation of the new `QueryResult` until we have received all messages belonging to that `QueryResult`. The classes that need to be modified are the `TinyDBNetwork` class and the `QueryResult` class.

We start by looking at the new structure needed to keep information about all the queries and messages. Remember there can be many queries running simultaneously, but there will be only one instance of the `TinyDBNetwork` object.
Figure 6-2 is an illustration of the new structure needed to hold information about queries. The `TinyDBNetwork` object gets a message from the network and creates a `QueryResultMsg`. This `QueryResultMsg` is saved in a Vector named `msgs`. The `msgs` is saved in a HasTable named `savedMsgs`. The entry in the HasTable consists of the group and the `msgs`. These `savedMsgs` are saved in a Vector named `savedQueries`. The Java code is like this.

```java
Vector savedQueries = new Vector(); //obj = savedMsgs
Hashtable savedMsgs = null; //obj = group, obj = msgs
Vector msgs = new Vector(); //obj = QueryResultMsgs
```

If we relate this to aggregated queries and grouping, we have that the `savedMsgs` has a 1 to 1 relation to a query. The `msgs` has a 1 to 1 relation to a group for the current Epoch of that query. The `QueryResultMsgs` is a partial result for a group in the query.

The logic in the `messageReceived` method will now be like this.

1. Generate a `QueryResultMsg` for the message received.

2. Use the static public method `queryid` (which is explained later) to get the `queryid` of the query this message belongs to.
3. If this is not an aggregated query, just make a QueryResult object and send it to the listeners, like before.

4. If this is an aggregated query, loop through the savedMsgs, and add or merge this result to the query it belongs to.

5. If the Epoch of this message is larger than the one we have saved for this query, we generate a QueryResult object from the saved messages, and send it to the listeners.
   
   a. First we create a QueryResult using the TinyDBQuery and the first QueryResultMsg for this query as parameters.

   b. Then we loop through the other savedMsgs and merge them with the QueryResult, using the mergeAggs function described below.

   c. Then we send the QueryResult to the appropriate listeners.

   d. Finally we create a new Hashtable and start collecting data for this Epoch, starting with the message we just received.

Except from the variables shown above, all changes in TinyDBNetwork class are in the method messageReceived. The other place it is necessary to modify is in the QueryResult class.

In the QueryResult class we need to provide it with a static public method to have a convenient way to get the queryid of the query a message belong to (see #2 above). This simple little method is as follows.

```java
/** Given a query result message, return the query id it belongs to */
static public byte queryId(QueryResultMsg m) {
    return (byte)m.get_qid();
}
```

We also need a method to merge aggregates which belongs to the same query together. This method is called mergeAggs and it merges a given QueryResultMsg into the current query (see #5b above).
public void mergeAggs(QueryResultMsg m) {
    int exprIndex = (int)m.getElement_d_data(EXPR_IDX_B);

    //get group number
    int newGroup = ByteOps.makeInt(m.getElement_d_data(GROUP_B2),
        m.getElement_d_data(GROUP_B1));
    if (newGroup == myGroup && !valid[exprIndex]) {
        //merging av samme verdi skulle vært foretatt på motene
        /* This result corresponds to just one of
        (possibly several) aggregate expressions in the query. */
        AggOp agg = ((AggExpr)myQuery.getExpr(exprIndex)).getAgg();
        byte[] resultData = new byte[m.get_d_data().length - DATA_OFFSET];
        System.arraycopy(m.get_d_data(), DATAOFFSET, resultData, 0,
            resultData.length);
        agg.read(resultData);
        /* include in the result vector (see resultVector() */
        valid[exprIndex] = true;
    }
}

When this is implemented TinyDB will produce correct results also for the aggregated queries.

6.3 The logging

From Section 5.2 we know that the problem concerning the logging of aggregated data to PostgreSQL is fixed by changing the getColumnHeadings() method in the TinyDBQuery class. This method has to be altered to handle aggregate columns separately by building up their names without spaces.

By adding the following code, the logging of aggregates works fine.

if (isAgg()) {
    //it's an agg; the columns that are returned
    //are the group and the aggregate value
    for (int i = 0; i < numExprs(); i++) {
        QueryExpr e = getExpr(i);
        if (e.isAgg()) {
            AggExpr ae = (AggExpr)e;
            if (ae.getGroupField() != -1 && !addedGroupCol) {
                cols.addElement(groupColName());
                addedGroupCol = true;
            }
            String aggString = "";
            aggString += ae.getAgg().toString() + "(" +
                getField(ae.getField()).getName();
            if (ae.getFieldOp() != ArithOps.NO_OP) {
                aggString += ArithOps.getStringValue(ae.getFieldOp())
            }
        }
    }
}
When all this is implemented we have extended TinySQL with HAVING, aggregates are produced with correct values, and we are able to log to the PostgreSQL database both aggregated and non-aggregated queries. We are now in the position to test the new implementation of TinyDB.
7 Challenges regarding testing and debugging TinyDB implementations

WSNs have emerged as an exciting new area of research in computer science, and testing and debugging WSNs combines the challenges from different research domains. A WSN is a distributed system that shows a large degree of parallelism. Each mote is essentially an autonomous embedded system with stringent constrains on storage, energy and memory resources. In addition to this the wireless nature introduces an unreliable communication channel. In this chapter we look at the challenges regarding testing and debugging TinyDB implementations. The challenges mentioned here are not restricted to TinyDB implementations alone, but in general it is valid for all applications running in a WSN on top of TinyOS.

The application itself can be hard to understand and debug, in Section 7.1 we look at the background for this. In Section 7.2 we look at some of the other challenges related to development of a new implementation. Section 7.3 is about simulation of applications running in WSNs. No formal evaluation of simulator and emulators are performed, but an overview of what we have investigated is presented in the next two sections. Section 7.4 gives an overview of some simulators and finally in Section 7.5 we look at emulators with a special focus on the emulator used in this thesis.

7.1 Making TinyDB an executable application

As we know, TinyDB is an application forming a WSN using TinyOS. From the discussion of operating systems in Section 2.5 we know that TinyOS is not really an OS but more a library with a collection of components. Applications are made up of components wired together, this includes components from user applications and components from TinyOS.

If we once again look at Figure 2-6, which illustrates the compilation process, we can see that we end up with an executable, which contain application code wired together with TinyOS code. This is the code that runs on the motes.
7.2 Main challenges in testing and debugging

There are many challenges related to the development of applications for WSNs. Earlier we have mentioned some of the constraints like limitation in power, memory, and CPU speed. In addition to these constrains, which are not so important in a debugging and test phase, we have challenges directly related to testing and debugging TinyDB implementations.

- Documentation. There is not much code in the source file that explains what each module is there for, and what it is supposed to do. In general all files starts with the same disclaimer and often there is no more comments. We have not been able to find any system documentation that gives an explanation of the connection between the different parts of TinyDB. With one exception, under the tinyos/doc directory there is a file with TinyDB documentation. This is more a user documentation than system documentation.

- Developing applications for motes is a relative new area in computer science, and as far as we know there exists no good IDE available for doing this development. For the Java API however there are several IDEs available. In this thesis the NetBeans IDE is used.

- The TinyDB application consists of component based, asynchronous code. It is difficult to get a “birds-eye view” of the application, and to fully understand how the application works. It is written in an unfamiliar language, the nesC programming language, and it is challenging to understand how the asynchronous code is working in a concurrent fashion.

- WSN applications are, because of the nature of the motes, unreliable. This makes it challenging to test the application.

- The core of the TinyDB application is the TupleRouter component. This is a large, complicated component consisting of approximately 4000 lines of code. This component contains some documentation in its source code. This is
helpful when it comes to understanding it, but even so it is a challenging task to understand it.

- The use of debugging messages is a helpful tool when doing application development and testing. During the work with this thesis we early understood that we needed a good simulator or emulator to help us develop a new implementation of TinyDB. We were particularly interested in debug messages from the motes. Both the TOSSIM simulator and the Avrora emulator are supposed to have the ability to produce debug messages. However, we never managed to set up the environment in a way that worked for us as described in the documentation.

### 7.3 Simulation in WSNs

Simulation is an important step in the development cycle of embedded systems. Simulators allow for more detailed inspection of the dynamic execution of microcontroller programs and diagnosis of software problems before the software is deployed onto the motes. Simulation is the most common approach to developing and testing new software for a sensor network. The goal for any simulator is to accurately model and predict the behavior of a real world environment. In general, sensor networks face many types of problems that do not arise in traditional networks. Examples might be power constraints, limited memory, decreased reliability and typically, a higher density and number of nodes than found in conventional networks. Simulation is often used to test new software that is being developed because it is a lot easier than loading up all the motes with new software in a developing phase.

There is a difference between a simulator and an emulator. Emulators are different from simulators in that they run actual application code. Emulation is more useful for fine-tuning and looking at low level results. Emulators are effective for timing interactions between nodes and for fine tuning network level and sensor algorithms. Simulators are more useful when looking from a high level view. The effect of routing protocols, topology, and data aggregation can be best viewed at a top level and would be more appropriate for simulation.
It is out of the scope for this thesis to do an in dept survey of available simulators and emulators. Listed here are some of the simulators and emulators that would have been candidates for an in-depth analysis.

### 7.4 Simulators

There exist some commercial simulators which we have not looked at. The best known commercial simulators are OPNET and QualNet.

#### 7.4.1 Ns-2

Ns-2 is probably the most popular simulation tool for sensor networks. It began as Ns (Network simulator) in 1989 with the purpose of general network simulation. Ns-2 is an object oriented discrete event simulator. Its modular approach has made it extensible, and this approach is probably the reason why it is so popular for simulating sensor networks. Many other simulators are based on ns-2 or the ideas that ns-2 is built upon.

#### 7.4.2 SensorSim

SensorSim uses ns-2 as a base and extends it in three important ways. First it includes an advanced power model which takes into account each of the hardware components that needs power supply to operate. Secondly it includes a sensor channel, and the third extension is an interaction mechanism with external applications. The main purpose with this is to interact with actual sensor node networks. This would allow for real sensed events to trigger actions in the simulated environment. Unfortunately SensorSim is not being maintained and is not currently available to the public.

#### 7.4.3 J-Sim

J-sim is a general purpose Java-based simulator modeled using the ns-2 as its model. Unlike ns-2 J-Sim uses the concept of components, replacing the notion that each node should be represented as an object. J-Sim’s component based architecture scales better than the object oriented architecture in ns-2. J-Sim has an improved energy model and the ability to simulate the use of sensors. Like SensorSim applications may
be simulated, and there is support for the connection of real hardware sensors to the simulator. J-Sim also provides a script interface to allow integration with different script languages such as Perl and Python.

### 7.4.4 SENSE

The SENSE simulator is built up on principles from three other models. It attempts to implement the functionality of ns-2, the concept of components as in J-Sim, and it also has support for parallelism. The component based model in SENSE frees the simulation models from interdependences usually found in an object-oriented architecture. This makes the simulation model extendible. A new component can replace an old one if they have compatible interfaces. The removal of interdependence between models also promotes reusability. A component developed for one simulation can be used in another if it satisfies the latter’s requirements on the interface and semantic. Parallelization in SENSE is provided as an option. SENSE is developed to be extensible, reusable and scalable.

### 7.5 Emulators

When we started the work with this thesis the equipment used was three Mica2 motes and the option of launching queries to the motes or using the VMNet emulator. The motes used were the same all the time, but it was a desire to look at other emulators or simulators which were more flexible and reliable than VMNet.

#### 7.5.1 VMNet

VMNet is an easy to use emulator which aims at enabling a realistic performance evaluation for WSN applications. Every emulated mote has to run the same code in VMNet, not as in Atemu and Avrora where each emulated mote might have different versions of the code. VMNet does not have the accuracy of Atemu and Avrora and the CPU of an emulated mote is done at CPU clock cycle level of the PC. The sensing units and other hardware peripherals are emulated in a sufficient way. All data sensed like light and temp will share the same data file, for the same emulated node.
7.5.2 TOSSIM

TOSSIM is a discrete event emulator for TinyOS sensor networks and is designed specifically for applications to be run on the Mica motes. Instead of compiling a TinyOS application for a mote, users can compile it into the TOSSIM framework, which runs on a PC. This allows users to test, debug, and analyze algorithms in a controlled environment. As TOSSIM runs on a PC, users can examine their TinyOS code using debuggers and other development tools. TOSSIM comes with TinyOS and it simulates the network at the bit level but it does not provide detailed timing information. Despite the fact that the sensor’s data gathering hardware is simulated, the phenomena that trigger reactions are not emulated. There seems to be no updates of TOSSIM since 2003.

7.5.3 Atemu

Atemu is an Emulator for the Atemel processors used in Mica2. Atemu code is binary compatible with the Mica2 platform. The emulation is more fine-grained than in TOSSIM. Atemu uses a cycle-by-cycle strategy to run application code. This is done through emulation of the AVR microcontroller used in Mica2 motes. Atemu offers an accurate emulation model in which each emulated mote can run different application code. Atemu is more accurate than TOSSIM, but Atemu sacrifices speed and scalability to do this. Atemu is one of the most accurate sensor simulators available. Latest release of Atemu was Mars 31. 2004.

7.5.4 Avrora

Avrora is the emulator used in this master thesis. It is the youngest of the emulators investigated, and the last publicly released version, 1.6.0, was released in June 2005. A new release of Avrora will probably be released in the autumn 2007 according to Avrora’s main architect, Ben Titzer. Avrora has an active mailing list and a user group which contributes with changes to the system.

Avrora is written in Java and is a research project from the UCLA Compilers Group. It consists of a simulation and analysis framework for programs written for the AVR microcontroller and Mica2 sensor nodes. It is used extensively at the universities:
UCLA, UCSD, UCSB, and Utah for development and testing of sensor programs. With Avrora programs can be tested with cycle accurate execution times before they are deployed onto the hardware devices. Avrora offers a lot of different opportunities to set up the simulator to fit specific needs of the user. Avrora is flexible providing a Java API for developing analyses. Avrora offers the following opportunities:

- The centerpiece of Avrora is an emulator for the AVR microcontroller with cycle accurate simulation of the execution time where programs can be tested before they are deployed onto the motes. This allows programs to be run with precise timing, and Avrora also offers emulation of the behavior of the on-chip devices and provides a clean interface for interfacing off-chip devices.

- Avrora lets users add their own monitors by writing a new subclass of the MonitorFactory class. Avrora also comes with an extensible collection of predefined monitors, ready to use. Some of them are:
  
  - *calls* is a monitor that tracks the call/return behavior of the executing program. The stacking up of function calls and interrupt handlers are displayed.
  
  - *energy* is a monitor used to trace the energy consumption.
  
  - *interactive* is a monitor which allows the user to interact with the program using breakpoints and watch points.
  
  - *interrupts* is a monitor which tracks changes to the state of interrupts, this includes posting, enabling and invoking of interrupts.
  
  - *ioregs* is a monitor for tracking updates to the IO registers on the microcontroller.
  
  - *memory* is a monitor for collecting information about usage of memory, including the number of reads and writes to every byte in memory.
o **packet** is a monitor for that tracks packets send and received by the nodes in a sensor network.

o **serial** is the monitor that allows the serial port (UART) of a node in the simulation to be connected to a socket making it possible for data from the program running to be output and external data can be fed into the serial port of the simulated node.

o **trace** is a monitor that traces the execution of the entire program by printing every instruction it executes.

- Avrora allows programs that are running in the simulation to be instrumented so that information can be collected as they execute, and statistics reported after they complete. This is done without modifying the program source, binary or the simulator. The instrumenting mechanisms are probes, watches and events.

- Probe is a piece of instrumentation that can be added to any instruction in the program. If we are interested in recording all calls to a specific method, we add a probe to the first instruction in the method. The probe will then fire when this instruction is executed. It can produce output, record the time, or collect various statistics about the call that we might be interested in.

- Watches are much like probes, but this time on a memory location. This can be fired before or after any read or write to a memory location in the program.

- Events can be used to fire a specific event at a given point in time during the programs execution.

- Avrora comes with a tool which can generate a control flow graph of the executing program. This might be useful for understanding how a program is structured and what the compiler does with the code.
- Avrora has a stack checker tool that can be used to check the maximum size used for the stack of a program.

- The default action in Avrora is the simulate action. There are many other actions as well, among them “gui” which launches a GUI allowing the user to interactively create simulations.

It is several reasons for the selection of Avrora as the emulator used in this thesis. The most important one is that it is an emulator, and hence runs the same code that runs on the motes. The documentation of the different possibilities in Avrora is however sparse, and it is hard to find out all the options to use. Avrora only simulates the light sensor on the Mica2 board. Other sensors will always return zero as their sensed value. It can always be argued as to whether or not it is better for an emulator to support several phenomena well, or to support one phenomenon very well.
8 Research methods

This chapter discusses the methods used to answer the questions raised in this thesis. The theory is based on ACM’s taxonomy of computer science (Comer et al., 1989). Taxonomy is the practice and science of classification (Wikipedia, 2007), and ACM provide a framework for classification of scientific works. We start out with a broad scope and end up with a description of which subject matter this thesis falls in under. This is explained in Section 8.1, and in Section 8.2 the different research methods used in this thesis are explained.

8.1 Research theory

Figure 8-1 is a representation of ACM’s taxonomy of computer science. The left side in the figure shows the different paradigms or cultural styles that characterize the manner a subject area is approached. The right side of the figure shows the different subject areas in computer science.

![Figure 8-1 The ACM taxonomy of computer science (Akkok, 2004)]
According to the taxonomy, the subject matter of this thesis falls under subject areas #2 “Programming Languages”, #6 “Software Methodology and Engineering” and #7 “Databases and Information Retrieval”. These three subject areas cover the extending of the TinySQL language with the HAVING clause and correction of the errors in TinyDB. As mentioned earlier, TinyDB handles streams of data, but which subject area Data Streams fall into is not easy to decide. At first #7 “Databases and Information Retrieval” seems as an appropriate subject area. But this area, according to (Comer et al., 1989), deals with the organization of large sets of persistent, shared data for efficient query and update. The data in Data Streams are not persistent, but Data Streams are in many ways related to Databases and Information Retrieval. There is no doubt that handling of Data Streams are about Information Retrieval, so even if it is not about handling persistent data it has to fall under subject area #7.

In this thesis the “specify, design, implement and test against the given requirements” approach of the Design Paradigm of the taxonomy in Figure 8-1 is used for the implementation of the HAVING clause. The Abstraction Paradigm is used for the correction of the errors concerning aggregate queries.

According to (Akkøk, 2004) it is doubtful that a single paradigm is alone responsible for any research. There will always be elements of the other two paradigms to some degree supporting the primary paradigm. This seems also to be the case in this thesis.

8.2 Research methods used in this thesis

In this section we look at another theoretical framework for describing different approaches to research in information systems. It is from (Akkøk, 2004) and gives a description and characterization of 16 different approaches for research in information systems. The different research approaches used in this thesis are then described.
According to the alternatives in Figure 8-2 this thesis falls in under the following categories:

- **Laboratory Experiments.** The experiments are performed on both live sensors and with the use of an emulator. We choose to load the data from the motes
into a PostgreSQL database. By logging the data to the PostgreSQL database we get the data structured in a way that makes it easier to analyze them. The research consists of generating queries that aggregate and measure data in a way not possible on data streams. SQL is used for analysis of the results. As explained in Section 3.9 TinyDB is prepared for connection to PostgreSQL and the logging is done automatically by simply selecting the Log to Database checkbox in the Query Constructor panel, and in addition the database server (postmaster) must be running.

- **Surveys.** According to the classification in Figure 8-2 surveys include literature survey. There is not much literature available describing TinyDB. We had no previous knowledge of the area, and the ACM article (Madden Samuel et al., 2005) and the Ph.D thesis (Madden Samuel, 2003 A) gives an thorough description of the different aspects of TinyDB. These two documents are used as the source for gaining the knowledge of TinyDB that is needed for the work with this thesis.

- **Empirical** research is defined as formal observation and verification/falsification based on collected data. Empirical research methods can be divided into two categories (Moody, 2002):
  
  - **Qualitative research methods:** Such methods collect qualitative data like text, images and sounds. These data are drawn from observations, interviews and documentary evidence. The data are then analyzed using qualitative data analysis methods. This method is not used in this thesis.

  - **Quantitative research methods:** Such methods collect numerical data and analyze it using statistical methods. It is often conducted through setting up controlled experiments or collecting data through case studies. The Quantitative research method is used in this thesis. As mentioned in this section data are logged to a PostgreSQL database for analysis of the results. This technique is used to ensure that the
implementation of the HAVING clause is working properly. As well, we aim to analyze the reliability of TinyDB.

When a query is launched in TinyDB, it is a request for service to TinyDB. This request can have three possible outcomes: TinyDB may perform the service correctly, incorrectly or refuse to perform the service. The three metrics associated with the three outcomes, namely successful service, error, and unavailability, are also called speed, reliability, and availability metrics (Jain, 1991). When a system performs a service incorrectly, an error is said to have occurred. It is then interesting to look at two different aspects, the probability of an error to occur and the time between errors. We have looked at the probability of an error to occur. In addition to just state that an error has occurred we are interested in what influences the error rate.

i. Does the order of aggregates in a query influence on the number of null values produced?

ii. Does the number of concurrent queries influence the correctness of the data produced?

iii. Does the number of aggregates on the same sensing unit influence the correctness of the data produced, and does it influence the number of null values produced for a query result?

iv. As mentioned in Section 2.1.3 the Mica2 motes have a packet loss of about 20% – 30%. How does this relate to the experiments conducted?

v. When the same query is launched a number of times in the WSN, does the generated result differ from time to time with respect to null values and undelivered packets?
vi. Does the sampling rate of a query influence on the number of null-values produced?

vii. As mentioned before, some of the data produced for aggregated queries had incorrect values prior to the implementation of the changes in TinyDB. After the changes to TinyDB are implemented, does the system produce correct data?

• Engineering. Engineering is a large part of this thesis and is defined as development of technology, which involves construction and re-construction of systems. This method is used in three different areas:

  o Engineering is used when the handling of the HAVING clause is implemented in the Java API.

  o It is also used for correction of the errors in the aggregating queries.

  o Finally, it is used to correct the problem regarding logging of aggregated data to the PostgreSQL database.

• Simulation. When working with this thesis it was vital to have an environment that was controllable and to have the ability to force the motes to produce known values. One way to achieve this is to use emulators instead of real motes. As mentioned in Chapter 7, different simulators and emulators were looked at, but no formal evaluation of them was conducted.

To select an appropriate research method might be critical to the success of any research project. It must be driven by the research question and the state of knowledge in the area being studied. In general, and as in this thesis, a combination of research methods may be the most effective in achieving a particular research objective.
8.3 Selection of techniques and metrics

The selection of an evaluation technique and a metric is an important step in all performance evaluation projects (Jain, 1991). According to Jain there are three techniques available, Analytical Modeling, Simulation and Measurement. Often it is useful to use two or even all three techniques. In this thesis Measurements is the main technique used, and some Simulation is used as well. A number of considerations help us to decide which evaluation technique to use. The key consideration is the life-cycle stage in which the system is. Measurements, as used in this thesis, are possible because there is a system to perform measurement on, and we are designing and implementing an improved version of the product. Jain lists 6 more considerations\(^{18}\), but the life-cycle is the most important one.

In general the metrics used when evaluating computer systems are related to speed, accuracy, and availability of service. In this thesis the metrics used are reliability (accuracy) and availability. Speed is not used as a metric in this thesis.

When a query is sent to TinyDB, the three possible outcomes are the following: It delivers correct results for the query, it delivers incorrect result for the query, or it does not deliver result for the query at all. The first two cases tell us something about the reliability of TinyDB. The ratio between the two cases when TinyDB deliver a result set, and when TinyDB does not deliver a result set, says something about the availability of service.

---

\(^{18}\) Time required, Tools, Accuracy, Trade-off evaluation, Cost and Saleability (listed from most to least important).
9 Implementation and results

The actual implementation of the code for TinyDB is conducted according to the description in Chapter 6. This chapter discusses the environment in which the development, test and experiments are conducted. It gives an overview of the configuration of the different system elements, and discusses the tests conducted. The configuration parameters for TinyDB, TinyOS, and Avrora are described in Section 9.1. The SerialForwarder program has to be used when we use the Avrora emulator. The SerialForwarder program is described in Section 9.2. In Section 9.3 we describe the experiment setup of hardware and software used for the experiments in this thesis. In Section 9.4 we discuss the tests conducted. The overall reliability of the motes is discussed in Section 9.5. The implementation of the enhancements and error corrections are discussed in Section 9.6. Finally in Section 9.7 the results from the experiments are summarized.

9.1 Configuration parameters

TinyDB configuration parameters are located in a file named tinydb.conf. The file is located in the directory tinyos-1.x/tools/java/net/tinyos/tinydb. A parameter is on the form:

```
Parameter:parameter-value
```

Lines that start with a % are considered as comments, and “$” is used in place of “:” in the parameter-value. Only the most important parameters are shown here, a complete list, with comments, is in the Appendix A.

First some parameters used to enabling the connection of TinyDB to PostgreSQL.

```
postgres-user:tele
postgres-passwd:tiny
postgres-db:task
postgres-host:localhost
```
The parameters are used to set userid, password, name of the database and location. The comm-string parameter tells what source TinyDB is going to talk to. The most common is when talking to the real motes with the parameter:

```
Comm-string:serial@COM1$57600
```

Another common option is when you use an emulator and the SerialForwarder:

```
comm-string:sf=localhost$9001
```

The last parameter in the `tinydb.config` file is the source for the catalog-file. As described earlier the catalog file defines all the attributes that are available for selecting in TinyDB.

```
catalog-file:net/tinyos/tinydb/catalog.xml
```

TinyOS configuration parameters are specified in the `CompileDefines.h` file which resides in the catalog tinyos-1.x/tos/lib/TinyDB. The compile-time options are set up with `#define` or `#undefine` statements. The options are explained in Section 4.1. In general all options in the list are left unchanged with their default values in this thesis. In Section 2.5 it is explained how the TinyOS kernel and libraries are linked together with the application specific nesC code.

Avrora offers a lot of different opportunities to set up the emulator to fit your specific needs. Avrora is started from the Avrora/bin directory by executing the command

```
java avrora/Main
```

The options used with Avrora are listed in the Appendix C.

### 9.2 The SerialForwarder program

The SerialForwarder program is used to read data from a serial port on the PC and forward it to programs that are written to communicate with the sensor network over the serial port on the PC. TinyDB is set up to read and write from the serial port on the PC. When running a TinyDB simulation in Avrora, the SerialForwarder is set up to handle the communication between the root node and the PC. Both parts think they
are sending and receiving from the serial port. As explained earlier, to run TinyDB using the SerialForwarder, the `tinydb.conf` parameter `comm-string` needs to be changed to

```
Comm-string: sf@localhost$9001
```

All the messages will now be forwarded between the motes root node and the PC by the SerialForwarder. The SerialForwarder is launched from the directory tinyos-1.x/tools/java with:

```
java net.tinyos.sf.SerialForwarder -comm
network@127.0.0.1:2390
```

This will open up the GUI window in Figure 9-1.

![SerialForwarder](image)

**Figure 9-1 The SerialForwarder execution panel**

The `–comm` argument tells SerialForwarder which address and port to communicate over. In this example SerialForwarder communicates with the emulator on port 2390. SerialForwarder does not display the packet data itself. It updates the packet counters in the lower-right hand corner of the window. Once running, the SerialForwarder listens for client connections on a given TCP port. Port 9001 is the default port. SerialForwarder simply forwards TinyOS messages between the emulator and the
network client connection. Multiple applications might connect to the SerialForwarder at once, and all of them will receive a copy of the messages from the sensor network.

9.3 Experiment setup

The experiments are performed with the following equipment:

- Mica2 motes with the equipment described in Section 2.3.2. Three motes are used, a root mote connected to a PC and two motes with sensing capabilities. As mentioned before the motes used in this thesis are capable of sensing light, temperature and noise.

- The PC which the motes are connected to has a 3GHz dual Pentium 4 processor and 1 GB memory. The OS is Fedora core 5.

- The topology of the network is restricted by the fact that we have only two motes with sensing capabilities. The Root does not have any sensing capabilities. The three different topologies used are shown in Figure 9-2. Topology 1 is used to test the kQUERY_SHARING option see Section 0. In Topology 1 we have a parent child relationship between Mote 1 and Mote 2, with Mote 1 as the parent. The messages from Mote 2 are routed through Mote 1 to the Root. Topology 3 is used when we investigate the percent of undelivered messages when only one mote is present, see Section 9.4.4. Topology 2 is used for the other experiments conducted. In this Topology messages are delivered from both motes directly to the Root.
The operating system on the motes is the TinyOS version 1.1.7. No changes to TinyOS are performed during this master thesis.

The version of TinyDB used is the latest publicly released version which is the 0.2 version, released as part of the TinyOS distribution 1.1.7 from September 2003. As described earlier in this thesis, all changes to TinyDB are in the Java API and in the parsers for this version. No changes are done to the software running on the motes.

The database used for logging of the query results is a PostgreSQL database version 7.3.2. As explained in Section 3.9 TinyDB is prepared for connection to PostgreSQL and the logging is done automatically by simply selecting the Log to Database checkbox in the Query Constructor panel (see Figure 3-9), and in addition the database server (postmaster) must be running.

For the experiments, motes as well as the Avrora version 1.6.0 emulator is used.

9.4 The experiments

We are especially interested in the reliability of TinyDB, and seven different questions were raised in Section 8.2. To answer these questions several experiments are conducted. In general each query is launched 5 times, and the data is logged to the
PostgreSQL database for later investigation. When the experiments were conducted we tried to have each run of the queries to run for approximately 100 Epochs. If TinyDB has a complete delivery for all Epochs there would have been logged 200 rows to the database for each run of the query, resulting from one delivery from each mote for each Epoch. The reason for the variation in number of rows logged to the database is that for some Epochs we do not get any results. When a query is launched we start to receive results into the GUI result frame (see Figure 3-11) and we can just look at the Epoch and estimate how many Epochs we actually have received results for, before we stop the query.

In Section 9.4.1 we want to investigate how the ordering of aggregates in a query influences on the number of null values produced. In Section 9.4.2 we investigate how concurrently running queries influence on the data produced. It is interesting to find out how many aggregates on the same sensing unit influence the data produced. This is investigated in Section 9.4.3. It is well known that messages are lost in WSNs. In Section 9.4.4 we investigate the loss of messages. In Section 9.4.5 we investigate what effect launching of the same query has on the results produced from time to time with respect to null values and undelivered messages. Finally, in Section 9.4.6 we investigate what effect the sampling rate of a query has on the data produced.

### 9.4.1 Aggregates order

We want to investigate if the ordering of aggregates in a query has any influence on the number of null values TinyDB produces. Two queries are constructed, and the only difference between them is the order of the aggregates:

**Query 1:**

```sql
SELECT nodeid, MAX(light), MAX(noise) FROM sensors
GROUP BY nodeid SAMPLE PERIOD 2048
```

**Query 2:**

```sql
SELECT nodeid, MAX(noise), MAX(light) FROM sensors
GROUP BY nodeid SAMPLE PERIOD 2048
```
We can see from Figure 9-3 there is a significant difference in the number of null values in the two aggregates. The aggregate placed first in the query had between 6% and 9% null values delivered. The aggregate placed second had null values delivered in 34% of the result sets.

The reason for this result is not easy to explain. We know the following. In a query each of the aggregates are produced separately. When we query the same sensing unit like MAX(light), MIN(light) in the same query, the light sensor is sampled twice. Each of these readings results in a separate message that is sent to the Java API where they are collected and assembled to a QueryResult. Queries with aggregates on different sensors follow the same pattern. Null values arise because messages are missing when the QueryResult is produced. The messages are corrupted, and hence not delivered to the Java API, when two or more messages are sent on the network at the same time.

In this experiment Topology 2 from Figure 9-2 is used, and hence there are no values produced from child motes for any of the motes. From Section 2.3.2 we know that a CSMA-like media access protocol is used, and that we might have collisions. Three messages might cause collisions: First, the heartbeat message from the root (see Section 3.3). Second, the message from the first aggregate on the other mote. Third, the message from the second aggregate from the other mote. We know that a mote listen first to assure that no one else is sending, before it starts sending. Given a situation where one of the motes, Mote 1, manages to start sending out the first message before the other one, Mote 2. Mote 2 will now wait a random time before it
starts the sending procedure again. From this we see that the message from the first aggregate might collide with the first aggregate from the other mote and the heartbeat message. The message from the second aggregate might collide with the heartbeat message, the first message from the other mote and the second message from the other mote. In the example above, if Mote 2 first wait because Mote 1 sends the message from the first aggregate, and then waits because Mote 1 is sending the second aggregate, it might not be time enough within the current Epoch to produce a reading for the second aggregate.

9.4.2 Concurrent queries

We wanted to find out if the number of concurrent running queries had any influence on the correctness of the data produced. We never managed to get more than two queries to run on the motes simultaneously. With only two queries running at the same time there was not much difference on the results produced when it comes to null values etc. There is an undocumented option in the TinyOS parameter file named MAX_NUM_SERVICES. It is default set to 2. We tried changing this to several different other values, but after compiling and installing the new implementation on the motes, the queries did not produce results any more. When changed back to the original configuration it worked ok for two queries again.

9.4.3 More than one aggregate on the same sensing unit

In Section 9.4.1 we investigated if the ordering of aggregates had any influence on the number of null values produced. It is also interesting to see what effect many aggregates on the same sensing unit have. Three queries were constructed where we started with one aggregate and ended with three aggregates on the same sensing unit.

Query 1: One aggregate
SELECT nodeid, MAX(noise) FROM sensors GROUP BY nodeid SAMPLE PERIOD 2048

Query 2: Two aggregates
SELECT nodeid, MAX(noise), MIN(noise) FROM sensors GROUP BY nodeid SAMPLE PERIOD 2048

Query 3: Three aggregates
SELECT nodeid, MAX(noise), MIN(noise), AVG(noise)
FROM sensors GROUP BY nodeid SAMPLE PERIOD 2048

Figure 9-4 Percentage of time when delivery is from one node only

We can see in Figure 9-4 that in 20% of the Epochs for the query with one aggregate, Mote 1 did not deliver a result. With two aggregates this drops to 13% and for three aggregates this is down to 12%. Also for Mote 2 we see the same pattern, number of Epochs with delivery increases with the number of aggregates in the query. With delivery in this context we mean delivery with result for at least one of the aggregates. The result is not surprising. Remember that each aggregate in the query is sent as a separate message, and because when the number of messages from one node increases, the larger probability there is for at least one of them is successfully transmitted to the root node.

The next natural step is to look at how the number of aggregates influence the number of null values produced.
In Figure 9-5 we can see that the number of rows with null-values increases as the number of aggregates on the same sensing unit increases. This comes as no surprise since when there are more messages to be sent, more of them will collide with other messages.

### 9.4.4 Investigation of message loss

According to (Madden Samuel, 2003 A) Mica2 motes drop packets (messages) about 20% to 30% of the time. We assume that every null value and every Epoch that lacks information from one of the nodes represent lost messages. The query

```sql
SELECT nodeid, light, noise FROM sensors SAMPLE PERIOD 2048
```

is used to investigate the message loss for none aggregated queries. The test reveals that there are no null values in the resulting data sets. Either the result is produced with data in all columns, or the result is not produced at all. This is not surprising, since non-aggregated results are delivered as a single message from the root to the Java API, whereas aggregated messages are delivered as one message for each aggregated column.
The average percentage of lost messages is about 18%. The percentage of lost messages is even less than the 20% to 30% predicted by Madden.

If we take the same query and make it an aggregated query by using the MAX aggregate we get the following query:

```
SELECT nodeid, MAX(light), MAX(noise) FROM sensors
GROUP BY nodeid SAMPLE PERIOD 2048
```

The result from this query gives the following distribution of undelivered messages.
In Figure 9-7 a QueryResult is considered undelivered if it either has a null value in one or more of the fields selected, or if there is no delivery at all for this Epoch. As expected, the situation when it comes to aggregate queries is quite different from the one with non-aggregated queries. The results from the different motes are now sent in more than one message (since each aggregate is sent as a separate message), and hence we have more collisions. Even though we expect more lost messages for aggregates, the result is surprising. For run 2 and 3 in the test, no complete result is delivered from Mote 1, and this is considerably different from the prediction of 20% to 30% message loss.

If we count QueryResult which contains both data and null values as delivered results we get a different result.

![Bar chart showing percentage of undelivered results for aggregate queries when nulls are acceptable.](image)

**Figure 9-8 Percentage of undelivered results for aggregate queries when nulls are acceptable**

If we have to accept a loss of messages for about 20% to 30% of the time, we see from Figure 9-8 that Mote 2 has acceptable reliability for all the queries, but Mote 1 has this for only 2 of the five runs. The message loss from Mote 1 is considerable for run number 2 and 3 where the message loss is 85% and 94%. Another interesting fact with these two runs is that they delivered correct results from either Mote 1 or from Mote 2, but never from both motes at the same Epoch. We have a collision 100% of the time resulting in lost messages from one or the other mote.
Why it is like this is hard to figure out. If we had missed all values from one of the motes, it might not have received the query. This is not the case here, since it actually produces some results.

Using Topology 3 we wanted to investigate the message loss when we have only one mote with sensing capabilities in the network. This removes the collision of messages sent from different motes. The parameter that is changing for the different queries is the sampling rate. For the last two queries the heartbeat rate was changed from the default once every 256th millisecond to the maximum value once every 1700th millisecond. The query used in this test is as follows:

```
SELECT nodeid, MAX(light) FROM sensors GROUP BY nodeid SAMPLE PERIOD n
```

The sample rate for run 1 and 2 is 2048, and for run number 3 it is 1024. The result for the runs of the query is given in the following figure.

![Figure 9-9 Percentage of undelivered messages when only one mote is used](image)

With only one mote in the network, the message loss for aggregated queries is considerably lower than when two motes are present in the network. We see that when the sampling rate increases from once every 2048th millisecond, in run 1, to once every 1024th millisecond in run 2, we get less message loss. This is natural since we now have twice as much sample messages and the same number of heartbeat messages for the same period.
For run 3 in this test the heartbeat rate was changed from the default once every 256 milliseconds to once every 1700 milliseconds, and we see that longer time between each heartbeat message leads to even fewer undelivered messages from the mote. It all makes sense since we only have collisions between the heartbeat messages and the messages with results from the query.

An external factor that might influence on the message loss is interference from other devices using the same frequency. We know that Mica2 motes use a 917 MHz RFM radio. According to National Table of Frequency Allocations for Norway the frequency band from 915.5 – 917.5 MHz is allocated to MOBILE service but marked as free (Post- og teletilsynet, 2007). This indicates that there is little interference from external devices, and that the message loss occurs as a result of the sending of messages from the different nodes in the network. This includes the sending of heartbeat messages down the tree.

9.4.5 Same query started many times

We want to find out if launching of the same query a number of times in the WSN, generates different results from time to time with respect to null values and undelivered messages. To investigate this we look at the results both for aggregated and non-aggregated queries.

First, we look at the non-aggregate queries. If we again look at Figure 9-6 we can see that for Mote 1, 4 out of 5 runs of the query have a variation of only 2%. But the 5th run has almost twice as many undelivered messages. For Mote 2 we have a variation of only 3% for 4 out of 5 runs. With the 5th run having 8% higher rates of undelivered messages than the lowest one. The number of null values delivered is zero for all runs.

For non-aggregated queries we have that in 80% of the queries the variation is little and messages are delivered in 80% or more of the time. This is much as expected from what is mentioned about message loss in the previous section.

Second, when we look at aggregate queries we use the data illustrated in Figure 9-8. For Mote 1 the variation between the three best and two worst cases is considerable. As we can see the range of undelivered messages are from 20% to more than 90%.
For Mote 2 however the situation is more like the one for non-aggregate queries in that the variation between the best case and worst case is from 5% until 20% message loss. The number of null values in the result sets for Mote 1 varies from 5% to 60%. For Mote 2 the variation is less. The number of null values in the result sets for Mote 2 varies from 16% to 37%.

From this we can see that the variation of messages delivered is much higher for aggregated queries than for non-aggregated queries, and also the number of undelivered messages is higher for aggregated than for non-aggregated queries.

9.4.6 The sampling rate of a query

It is interesting to look at how the sampling rate of a query influences the number of null values in aggregated queries. To find out this we launched the following query six times with the SAMPLE PERIOD varying from 128 milliseconds to 131 072 milliseconds per Epoch.

```sql
SELECT nodeid, MAX(noise), MIN(noise), AVG(noise)
FROM sensors GROUP BY nodeid SAMPLE PERIOD n
```

The result from the runs of the query gives the following distribution of QueryResults without null values.

![Figure 9-10 Percentage of QueryResults delivered without null values.](image)

We see from Figure 9-10 that with a sampling rate of once every 128th millisecond no QueryResult is delivered without null values. It does not help to reduce the sampling rate from the default value of one instance every 2048th millisecond to one every...
131 072th millisecond. The number of `QueryResult` delivered without null values does not increase. In the query with the sampling rate once every 131 072th millisecond the reliability is even worse. In the figure we see that the query with sample period once every 512th millisecond has the highest average of `QueryResult` delivered without null values, but this query has nearly all packets delivered from Mote 2 only.

### 9.5 The overall availability of the motes

In the experiments described in Section 9.4 the results are based on queries that actually started and delivered data for logging to the PostgreSQL database. Motes are unreliable, and many times queries have to be restarted a number of times to actually start delivering data.

During the experiments 86 queries have been started with the option to log the results to the database. 20 (23%) of these queries produced no results at all. The queries that did not produce any result at all were all queries with aggregates. This indicates that queries with aggregates more often will produce no results than queries without aggregates.

### 9.6 Evaluation of the new implementation of TinyDB

We know that for aggregated queries the original implementation of TinyDB deliver incorrect values. We want to investigate if the changes we have made to TinyDB have corrected this. When we look at the result for the query described in Figure 4-1 we can see that for each aggregate within the same Epoch we always have the same value, or a null value, for both the Motes. In Section 9.6.1 we look at how the new implementation has eliminated the errors mentioned. We also want to test that the enhancements to TinyDB, the addition of the HAVING clause to the language, is implemented correctly. This is described in Section 9.6.2. The correction of the error regarding the logging of aggregated queries to PostgreSQL is not discussed further. The fact that we now are able to log data to PostgreSQL, and the fact that we were unable to do this before the correction, is proof enough for the error regarding logging of aggregated data is corrected.
9.6.1 Evaluation of the error corrections

In the current implementation of TinyDB we always have the same value from aggregates for both the motes within the same Epoch. We want to test that the error regarding aggregates is correct in the new implementation. The query used is the following:

\[
\text{SELECT nodeid, MAX(light), MIN(light), AVG(light) FROM sensors GROUP BY nodeid SAMPLE PERIOD 2048}
\]

We first perform the experiment using Avrora with the motes set up to produce known values. The emulator was set up with Mote 1 producing the value 277, and Mote 2 producing the value 435 using Topology 2. The motes never deliver values produced by the other mote, and they always have different values for the same Epoch. This indicates that the new implementation produces correct values when the emulator is used.

We also want to perform this test on live sensors, and place one of the motes in a place with significantly less light than the other one. This ensures that the motes do not produce values in the same range. When we look at Figure 9-11 we see that there is no overlap between the range of values produced from Mote 1 and Mote 2, hence the error does not occur a single time in the experiments conducted with the use of live motes. This strongly indicates that the error is corrected.

<table>
<thead>
<tr>
<th></th>
<th>MIN(light)</th>
<th>MAX(light)</th>
<th>AVG(light)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mote 1</td>
<td>862 - 890</td>
<td>862 - 890</td>
<td>862 - 890</td>
</tr>
<tr>
<td>Mote 2</td>
<td>591 - 632</td>
<td>364 - 632</td>
<td>590 - 632</td>
</tr>
</tbody>
</table>

Figure 9-11 Light readings from the motes
9.6.2 Evaluation of the implemented HAVING clause

We want to assure that the implemented HAVING clause works correctly. To test this against the motes we use a query that senses light. One of the motes is placed inside a paper box. This way the light sensors for the two motes produce non overlapping values. The query used is the following:

Query 1:
SELECT nodeid, MAX(light) FROM sensors HAVING MAX(light) > 500 GROUP BY nodeid

Query 2:
SELECT nodeid, MAX(light) FROM sensors HAVING MAX(light) < 500 GROUP BY nodeid

Query 3:
SELECT nodeid, MAX(light) FROM sensors GROUP BY nodeid

In order to try to get as equal conditions as possible during the experiment, the queries are launched without any justification to the light. From the experiment we find that the HAVING clause filters out correctly and delivers only correct results. The third query is used to ensure that we actually have values both above and below 500.

We also test with both nodes reading approximately same light values, and with the HAVING filter set so that it should filter out some of the values from both nodes. The queries used are the following:

Query 1:
SELECT nodeid, MAX(light) FROM sensors HAVING MAX(light) >= 870 GROUP BY nodeid

Query 2:
SELECT nodeid, MAX(light) FROM sensors HAVING MAX(light) < 870 GROUP BY nodeid

Once again both queries produce results as expected. Query 1 has results from both nodes in the range from 870 and above. Query 2 has results from both nodes in the range from 869 and below. Based on this results, we have confidence in that the HAVING clause is correctly implemented in the enhanced version of TinyDB.
9.7 The result from the experiments

When we look at the tests performed and the results conducted we see that there is a variation in the availability. We get results in about 80% of the time, for the last 20% of the time the results from the query simply do not appear in the Java API. This means that the availability of service in TinyDB for the experiments conducted in this thesis is about 80%.

When we look at the reliability for TinyDB, we see that there is a variation between aggregated and non-aggregated queries. We know that motes can be unreliable and that they lose packets (messages) in 20% to 30% of the time (Madden Samuel, 2003 A). The tests conducted here indicate that we are within this range for non-aggregated queries. For aggregated queries, there is large variation in the reliability. Worst case, as we have seen is 0% delivery of results without null values. Best cases have been well inside the 20% to 30% loss of packets, and 80% of the aggregated queries have reliability within the expected range.

A description of PostgreSQL usage and some of the queries used in the reliability study of TinyDB is found in Appendix D.
10 Conclusion and future work

In this chapter we discuss the results achieved and the major conclusions. The conclusions drawn from the work with this thesis is found in Section 10.1, and in Section 10.2 we describe possible further work and loose threads.

10.1 Conclusion

In this section we summarize the contributions of this thesis in Section 10.1.1, and conclude on the results from the reliability study of TinyDB in Section 10.1.2.

10.1.1 Contributions

Handling of aggregates is an important feature in WSNs, and when there is an aggregate in a query there is also a (possibly implicit) GROUP BY clause associated with it. Grouping is important since rather than computing a single aggregate value over the entire set of data values, a grouping clause partitions the values into groups based on some attribute. The HAVING clause can then be used to filter aggregated values.

There are two different approaches to where the HAVING clause could be applied. A centralized approach applies the HAVING clause on the root node in the WSN, or in the Java API. A distributed approach applies the HAVING clause on the motes. A centralized approach is easier to implement than a distributed approach, and it can handle all type of aggregates. A distributed approach can be applied only for aggregates that have the monotonic property. This means that aggregates like MAX, MIN, COUNT and SUM can have HAVING applied on the motes, whereas AVG can not.

A Non-monotonic aggregate like AVG has to be implemented centralized. When this implementation is performed, it is relatively easy to extend it with the monotonic aggregates as well. A distributed implementation for the monotonic aggregates is much harder to develop. The main advantage with a distributed approach is that it is power saving. The main advantage with a centralized approach is that it is easier to
implement than a distributed approach. We want not only to design a solution to the implementation of HAVING, but also actually perform it. It could be hard to achieve this within the time available if the distributed approach is taken. Based on this, TinyDB is extended with HAVING, using the centralized approach, and the implementation is done in the Java API which is much easier to perform than in the root node in the WSN.

We have experienced that the original implementation of TinyDB handle queries with aggregates incorrectly. The results from a query containing aggregates always have the same aggregated value for the same Epoch for all the motes. The reason for this is a logical error regarding the construction of results for aggregated queries. The error arises when a new result for an aggregated query is constructed. The result that is to be sent to the listeners has some of its values overwritten by values belonging to the next query result.

The solution to this problem is to extend TinyDB with a new data structure for holding information about queries with aggregates. This new data structure is proposed and implemented in the Java API.

Logging of results from queries to the PostgreSQL database is important, since this enable easier data analysis. In the original implementation of TinyDB, only non-aggregated queries could be logged. Performing measurements and analyses on the stream of data received from the motes is more or less impossible to do without the ability to log these data to a database. The reason that the logging of aggregated queries in the original implementation of TinyDB is not working is due to that headers for aggregated queries are generated with spaces. These headers are used as column names in PostgreSQL where column names including blanks are prohibited. A new function for generating headers for aggregates is proposed and implemented.

From this we see that the enhancements to TinyDB presented in the problem statement of this thesis are implemented.

The enhanced and improved implementation of TinyDB is currently used in courses in informatics at the University of Oslo.
10.1.2 Results from the reliability study

In WSNs it is a known problem that we have packet loss. We know that the motes are unreliable and we know that the Mica2 motes will drop 20% to 30% of the messages received at a range of about 16 - 32 feet. Using the enhanced and improved implementation of TinyDB, we want to investigate the reliability of TinyDB with aggregation particularly in mind.

The tests conducted show that there is a significant variation between aggregated and non-aggregated queries. For non-aggregated queries we have a packet loss of 20% to 30% or less. For the aggregated queries there is much more variation in the results. Worst case was a reliability of delivery equals to 0%. The best cases are well inside the 20% to 30% loss of packets. About 80% of the aggregated queries have reliability within the expected range.

When a service is requested from TinyDB we have three possible outcomes. We may have a successful delivery, a delivery with errors or the service might be unavailable and deliver no result at all. The latter is the availability metric. Sometimes when we launch a query we get no response back from TinyDB. When we did the reliability test the availability of TinyDB was about 80%.

The study shows that for non-aggregated queries the reliability is as expected. For aggregated queries however, we have a message loss that varies a lot, and worst case is way above the 20% to 30% predicted by (Madden Samuel, 2003 A).

10.2 Future work

In this thesis aggregates and HAVING are implemented using the centralized approach. We know that HAVING can (and should) be applied on the motes for aggregates that hold the monotonic property. Power saving is of outmost importance in WSNs. A shift on the motes, from sending to computation will save power. A centralized approach to implementation of HAVING is not power saving. The next natural step would be to implement HAVING on the motes for the aggregates where
this is feasible. It is obvious that a distributed implementation is an improvement over a centralized implementation.

The implementation of HAVING demonstrates how new features are implemented in the query language. This gives us a platform to further enhance the query language with features like the OR and NOT operator in the WHERE and HAVING clauses.

In TinyDB, when you have more than one aggregate for the same sensor the sensor value is fetched once for every aggregate. A different approach is to fetch the value from the sensor only once for each Epoch, and then use the fetched value in the different aggregates.

The Avrora emulator offers you to write your own monitors for the system. Monitors can collect statistics and generate a report once simulation is complete. Monitors can track program variables, function executions, interrupts and monitor packets transmitted over the radio. Unfortunately it has not been time enough to investigate these features of Avrora. This is an area where it is interesting to do more research.

Debugging is a complicated task when developing software for the motes. It would have been helpful to have an easy way of getting out debug messages from the motes. The debug messages could have been filtered out by the SerialForwarder and displayed in a separate window.

TinyDB runs on version 1.x of TinyOS. In 2006 TinyOS 2.0 was released. TinyOS 2.0 is a redesign of TinyOS 1.x, but it is not backwards compatible with the previous version. TinyOS 2.0 is an improvement in many areas such as improved scheduler, a much richer set of timers, improved communication, support for more sensor platforms, improved network protocol where delivery rate have been up to 98%. TinyOS 2.0 represent the next step of development, and the architecture of TinyOS is pushed forward in several directions. This hopefully leads to simpler and easier application development. Since TinyOS 2.0 is much better than TinyOS 1.x in many ways, and TinyDB currently are running only on TinyOS 1.x it is interesting to port TinyDB to TinyOS 2.x. This is extremely important for the future applicability and interest of TinyDB. TinyDB has many nice features that make it easier to develop
applications for WSNs. The improved network protocol and the communication in TinyOS 2.0 will probably extend the reliability of TinyDB and thereby make the system even more useful. If TinyDB is not ported to TinyOS 2.0 it will probably be less and less used and eventually it will simply not be used any more and disappear from the WSN arena.
References


MADDEN SAMUEL (2003 B) TinyDB A Declarative Database for Sensor Networks.


MADDEN SAMUEL, HELLERSTEIN JOE & HONG WEI (2003 D) TinyDB: In-Network Query Processing in TinyOS.


PISTER, K. (2001) SMART DUST.


Appendix A

TinyDB configuration parameters

This is a complete list of all parameters in the file tinydb.conf. The parameters explained in Section 9.1 are also repeated here for completeness. The file is located in the directory tinyos-1.0/tools/java/net/tinyos/tinydb. The parameter is on the form:

Parameter:parameter-value

Lines that start with a % are considered as comments, and “$” is used in place of “:” in the parameter-value. The following parameter values are used for the setup of the TinyDB implementation used in this thesis. First some parameters used to enabling the connection of TinyDB to PostgreSQL.

postgres-user:tele
postgres-passwd:tiny
postgres-db:task
postgres-host:localhost

The parameters are used to set userid, password, name of the database and location. Next follow two parameters to tell TinyDB which windows that should be shown when the application starts up.

show-command-window:true
show-status-window:true

Here we show both the command window and the status window when the application starts up. These two windows are the two small windows on the right side of the query construction panel and they are explained in Section 3.9.

The next parameter is used to enable logging of data to the PostgreSQL database.

enable-logging:true

The root-id parameter can be used to force a value to the root-id.

root-id:0
The am-group-id parameter tells TinyDB which am-group-id these motes belong to. Default is -1 telling that this network does not contain any subgroups.

\[ \text{am-group-id: -1} \]

The next parameter is default set to false, it is not documented but it has to do with generic communication.

\[ \text{gb-root: false} \]

The comm-string parameter tells what source TinyDB is communicating with. When communicating with the real motes the parameter is set to:

\[ \text{Comm-string: serial@COM1$57600} \]

Another option is when you use an emulator and the Serial Forwarder:

\[ \text{comm-string: sf@localhost$9001} \]

The default query can be set with the default-query parameter. The string provided will be a query string that can be selected from the Query constructor’s text panel.

\[ \text{default-query: create buffer sam size 5 as (select nodeid, light)} \]

The last parameter in the tinydb.config file is the source for the catalog-file. The catalog file defines all the attributes that are available for selecting in TinyDB.

\[ \text{catalog-file: net/tinyos/tinydb/catalog.xml} \]
Appendix B

Construction of a query with aggregates.

This is a step by step explanation of how a query with aggregates is constructed, distributed to the motes, and returned to the Java API. The procedure is as follows:

First the query is constructed in the Java API. The information about the query is contained in the TinyDBQuery object. This is done in the GUI panels, or in a suitable application. The TinyDBQuery is handed over to the TinyDBNetwork object, which divides up the TinyDBQuery in Messages. One Message for each field, aggregate and selection predicate in the query.

Each Message is send out on the network where it is received by the TupleRouter. The TupleRouter is the core of the TinyDB system. It is the entry point on the motes, and has three main execution paths.

1. Handling of new queries.

2. Handling results from neighbouring nodes.
3. Generating local results and deliver data to parent nodes.

The TupleRouter receives new messages from the network and creates a local state for them by converting them to ParsedQueries. Each query is identified by a globally unique id. Messages contain part of a query, and all the messages describing a single query must arrive before the router will begin routing tuples for that query. The different parts of a query are:

- A single field or attribute to retrieve.
- A single selection predicate to apply.
- A single aggregation predicate to apply.

When all data for a query have arrived the ParsedQuery is created. It is a compact representation of a query where field names are replaced with field IDs that are used as offsets into the sensors local catalog. Given a ParsedQuery the TupleRouter allocates space at the end of it to hold the collected data. The last step when a query arrives is for the TupleRouter to call the SetSampleRate to start the mote’s 32Khz clock to fire at the appropriate data-delivery rate for all of the queries currently in the system.

Whenever a clock event occurs the TupleRouter must perform the following actions: Deliver tuples which were completed on the previous clock event. If the query contains an aggregate, deliver the aggregate data from the aggregate operator, if not, deliver the tuple filled out during the last iteration. Reset the counters that indicate when these queries should be fired again. Decrement the counters for all queries. Any query whose counter reaches zero need to have data delivered. Reset expression specific state for these queries. This is specific to the expressions in the query. MIN aggregates, for instances, will have their current minimum value set to a large positive number.
Fetch the data fields for each query firing this epoch, and loop through all fields for all queries. Fetch them and fill in the appropriate values in the tuples on the appropriate queries.

Route the filled in tuples to query operators. First route to selections and then, if it exists, the aggregates. If any selection rejects a tuple, stop routing it.

When a result for an aggregated query arrives from a neighbour it needs to be integrated into the aggregate values being computed locally. It is then forwarded to the AGG_OPERATOR component for aggregation. If the result is not an aggregated query, it is simply forwarded up the routing tree towards the root.

The results generated by the TupleRouter ends up as messages which are received by the TinyDBNetwork. Based on the contents of the TinyDBQuery and the messages for the query the TinyDBNetwork generates QueryResults (a complete resultset), which are delivered to the applications listening for that query.
Appendix C

Avrora configuration parameters

This is the Avrora 1.6.0 configuration parameters used in this thesis. Avrora offers a lot of different opportunities to set up the simulator to fit your specific needs. Avrora is started up by, in the Avrora/bin directory, execute the command:

```
java avrora/Main
```

The options used with Avrora are as follows:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-topology=</td>
<td>Name of a file containing a description of the topology of the nodes we want to test.</td>
</tr>
<tr>
<td>topo.top</td>
<td></td>
</tr>
<tr>
<td>-platform=mica2</td>
<td>Which platform to test. Mica2 nodes are used for the experiments in this thesis.</td>
</tr>
<tr>
<td>-nodecount=3</td>
<td>Number of nodes in the simulation, included node number 0 which has no data but is the root node, hence the connection point to the PC.</td>
</tr>
<tr>
<td>-simulation=</td>
<td>Type of simulation to run.</td>
</tr>
<tr>
<td>sensor-network</td>
<td></td>
</tr>
<tr>
<td>-sensor-data=</td>
<td>Name of the data files containing data for the different nodes for the simulation. Makes it possible to run test with known data, which makes the prediction of the results easier.</td>
</tr>
<tr>
<td>&quot;light&quot;:1:m1.dat,</td>
<td></td>
</tr>
<tr>
<td>&quot;light&quot;:2:m2.dat</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>-monitors=serial</code></td>
<td>Type of monitor to simulate. Serial is one of the predefined monitors in Avrora it allows the serial port (UART) of a node in the simulation to be connected to a socket so that data from the program running in the simulation can be outputted to the serial port. This is only one of many predefined monitors in Avrora. When started with this parameter Avrora waits for a Serial Forwarder to connect when it starts up.</td>
</tr>
<tr>
<td><code>-seconds=3000</code></td>
<td>Number of seconds the simulation should run.</td>
</tr>
<tr>
<td><code>-update-node-id=true</code></td>
<td>Makes sure the nodeid in the simulation is updated by Avrora.</td>
</tr>
<tr>
<td><code>-stagger-start=100</code></td>
<td>Number of clock cycles between start of each node. First one is started at 0 next at 100, third at 200 and so on.</td>
</tr>
<tr>
<td><code>-verbose=verbose</code></td>
<td>Gets information from Avrora when it runs.</td>
</tr>
<tr>
<td>TinyDBApp.od</td>
<td>Name of the program file we want to run. The file must be in the od format, and located in the directory where Avrora is started up. To get the program to the od format lunch the command: [avr-objdump -zhD main.exe &gt; TinyDBApp.od']</td>
</tr>
</tbody>
</table>
Appendix D

Queries used in the reliability study of TinyDB

When the result of a query is logged to the database it is logged into two tables, the first named *queries* is holding information about the query that is performed. It consists of the following columns: *qname*, which is the name of the query, starting with *q1* then *q2* and so on. The next column is the *query_time*, which is a timestamp giving the time when the query is saved to the database. The last column is the *query_string* which contains the text string of the query performed.

The next table is a table for holding the result for each query. These tables are named dynamically given the name you find in the *qname* column in the *queries* table. The result for the first query is stored in table *q1*, the next in *q2* and so on. These tables have columns reflecting the columns in the result set. First we have a column named *Result_time*, second comes *Epoch*, and finally one column for each heading in the result set.

As an example we can look at query #72. The query inserted into the network is:

```
SELECT nodeid, MAX(light) FROM sensors GROUP BY nodeid SAMPLE PERIOD 1024
```

In the queries table we find the following entry:

<table>
<thead>
<tr>
<th>qname</th>
<th>query_time</th>
<th>query_string</th>
</tr>
</thead>
<tbody>
<tr>
<td>q72</td>
<td>2007-09-12 11:40:19.92</td>
<td>SELECT nodeid, MAX(light) FROM sensors GROUP BY nodeid SAMPLE PERIOD 1024</td>
</tr>
</tbody>
</table>

The q72 table looks the following:

<table>
<thead>
<tr>
<th>Result_time</th>
<th>Epoch</th>
<th>nodeid</th>
<th>MAX_light</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007-09-12 13:44:57.409601</td>
<td>2</td>
<td>2</td>
<td>842</td>
</tr>
<tr>
<td>2007-09-12 13:44:58.35734</td>
<td>3</td>
<td>2</td>
<td>837</td>
</tr>
</tbody>
</table>

We are aware of the fact that the timestamp in the queries table is two hours wrong, but there have not been time to figure out the reason for the discrepancy between the
two tables. We also notice that there is a difference in the precision for the timestamps.

When the data is stored in the result tables we can construct queries to analyse the reliability of TinyDB. Given below are some of the queries used. The data for the graph in Figure 9-3 are produced with the query below. The following query is run 1 time for each result table (q6-q15).

\[
\text{select (select count(*) from q6 where max\_light is null) * 100 } \\
/ \text{ count(*) as agg1, (select count(*) from q6 where max\_noise is null) * 100 / count(*) as agg2 from q6;}
\]

The data for the graph in Figure 9-4 are produced with the following query:

\[
\text{Select count(*) * 100 / (select max(epoch) - min(epoch) from q16) from q16 a where nodeid = 1 and not exists (select * from q16 b where nodeid = 2 and b.epoch = a.epoch);}
\]

This gives us the percent of queries which have result from mote 1, and not from mote 2. By switching the nodeid = 1 and nodeid = 2 with each other we get the result for the opposite. This must be run for each of the result tables that are the basis for the graph (q16-q25).
Appendix E

CD

Enclosed to the thesis is a CD that contains the SenseParser files, the files that are changed in the Java API, the PostgreSQL database tables and the EndNote library with all the references with URL’s where appropriate. Finally the thesis is enclosed.