Validated Cost Models for Sensor Network Queries

Christian Y. A. Brenninkmijer, Ixent Galpin, Alvaro A. A. Fernandes, Norman W. Paton
School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, United Kingdom
{brenninc,ixent,alvaro,norm}@cs.man.ac.uk

ABSTRACT

Generating a good execution plan for a declarative query has long been a central problem in data management research. With the rise in interest in wireless sensor networks (WSNs) as query processing platforms, it was quickly noticed that the corresponding optimization problem is even more challenging than the classical one, since, in comparison to classical platforms, a WSN is a very constrained computational infrastructure (in terms of memory, processing, and communication capabilities, and, crucially, depletable energy stocks). Optimizing a declarative query for execution in WSNs is thereby made both more important and more challenging. One of the requirements for effective query optimization is the availability of effective models for estimating the cost of alternative execution plans. This paper describes how query cost models for space, time and energy were methodically derived and validated for an expressive algebra for continuous queries over sensor streams.

Categories and Subject Descriptors

H.2.3 [**Database Management**]: Languages—*query optimization, cost models, sensor networks*

1. INTRODUCTION

Starting with the seminal work in System R [12], there has always been widespread recognition of the important role that cost estimation models (CEMs) have in query optimization. Their availability means that a query execution plan (QEP) can be assessed, in isolation and in comparison to alternative QEPs, in terms of the extent to which it meets some non-functional property of QEPs (classically, the response time it delivers). Modern query optimization [3] uses estimation models to inform decisions as to which concrete algorithms and auxiliary data structures (such as indices) to use in evaluating algebraic operations (particularly, inherently expensive ones, such as join and group-bys).

As parallel and distributed query execution became more important in the more recent past, the importance of CEMs

only grew [5, 8, 11, 13, 16] as a consequence of the greater impact of decisions (compared to those in centralized execution) regarding resource allocation and scheduling as well as QEP decomposition and placement. Even more recently, the extension of query technology to data streams has once more highlighted the relevance of CEMs [4, 15]. In particular, the dynamic nature of the environment in which long-running, continuous queries evaluate often requires the evaluation process to adapt to runtime changes. No matter how effective the CEM used at compile time to decide on the best QEP, if, at runtime, the actually observed properties of either the data or the environment differ significantly from those assumed (from collected statistics) at compile time, the selected QEP may prove highly sub-optimal. The decision as to whether, and when, to adapt relies again on cost (as well as progress [10]) estimation models. The latest, and perhaps the most challenging, query optimization problem in which CEMs play a fundamental role is that of optimizing declarative queries for execution over WSNs [6, 7, 9].

In comparison to classical computational infrastructures (based on commodity PCs, server-grade hardware, or mainframes, and on the Internet), a WSN is a very constrained computational infrastructure: in a WSN, nodes have small amounts of primary and secondary memory, slow processors, and must rely on short-range wireless radio. Most importantly, rather than drawing their energy from continuous-supply utility grids as classical computational infrastructures do, WSNs are often battery-powered. These limitations mean that an optimizer for declarative queries that are targeted for execution in a WSN must choose a QEP whose estimated cost is low along several dimensions, e.g., memory consumed, time taken to deliver each tuple, and, crucially, energy used. Note, however, that the optimization problem is made even harder by the fact that such desiderata trade-off against one another. For example, minimizing delivery time is one likely consequence of using as few hops as possible in transporting data. This suggests one should transport data as early as possible over as few hops (and hence as long distances) as possible. This, in turn, suggests that minimizing delivery time is likely to lead to, on the one hand, minimization of memory consumption (because data is less likely to be held up in buffers) and, on the other, quicker energy depletion. This is because the energy required to transport data over radio grows disproportionately with the distance between source and destination. As a result, in WSNs, multihop links tend to be more energy-efficient than single-hop ones.

Optimizing declarative queries for execution in WSNs is

therefore both more important and more challenging due to the constrained nature of the platform. This, in turn, makes the availability of effective CEMs for estimating the costs of alternative execution plans an imperative. This paper describes how empirically-validated CEMs for the space, time and energy consumed by a QEP over a WSN were methodically derived and validated for an expressive algebra for continuous queries over acquisitional sensor-data streams.

In previous work [6, 7], we have described the SNEE framework for sensor network query processing, and, SNEEql, a continuous query language over acquisitional sensor streams. The logical algebra underlying SNEEql was given a formal semantics in [2]. In this paper, we firstly show how physical operators (i.e., concrete algorithms for WSN execution) can be assigned to the logical algebraic operators. Then, we show that carefully pitching the abstraction level at which the pseudocode for the physical operators is expressed, so as to bring out the features of operations that determine their space, time and energy expenditures, makes it easier to derive analytical CEMs for space, time and energy for the operators directly from their algorithmic structure. Finally, we show experimentally that the CEMs are valid (i.e., exhibit satisfactorily accurate results under emulations that are themselves accurate at the level of processor cycles).

The remainder of the paper is organized as follows. Section 2 provides the technical background to the contributions by briefly describing the SNEEql query language and its underlying logical algebra[1]. The contributions of this paper are reported in Sections 3-6. Section 3 describes how physical operators can be assigned to the logical algebraic ones. Section 4 describes the CEMs derived for the physical operators. Section 5 describes the strategy we have used to validate the CEMs empirically. Section 6 shows the results from the validation experiments. Finally, Section 7 draws some conclusions.

2. QUERY LANGUAGE AND ALGEBRA

SNEEql [2] is a declarative query language for WSNs inspired by expressive stream query languages, particularly CQL [1]. Fig. 1 (from [7]) shows an example SNEEql query over a schema comprising two logical extents, referred to as inflow and outflow, in a hypothetical water utility domain. The sensors measure conditions in distribution pipes. The example query emits tuples onto the output stream whenever the outflow pressure now is smaller than the inflow pressure a minute ago, provided that that inflow pressure is above 500. In a stream query language, where conceptually data is being consumed, and thus potentially produced, on an ongoing basis, the question exists as to when a query should be evaluated. In SNEEql, an *evaluation episode* is determined by the acquisition of data, i.e., setting an acquisition rate sets the rate at which evaluation episodes occur. Thus, whenever new data is acquired, it is possible that new results can be derived. In the example query, the desired acquisition rate is once every three seconds, and the expected time that it takes for a tuple to be delivered is five seconds[2].

---

[1]The material in this section is closely based on previous work [2, 6, 7] by the authors.

[2]It is beyond the scope of this paper to explain how such quality-of-service expectations are taken into account. The interested reader is referred to [7] for details. For the purposes of this paper, the reader need only note that CEMs play a crucial role making appropriate optimization decisions

```
schema: outflow (id,time,temp,pressure)    sources: {0,2,4}
        inflow  (id,time,temp,pressure,ph) sources: {4,5,7}

SELECT RSTREAM o.id, i.id, o.time, o.pressure, i.pressure
 FROM  outflow[NOW] o, inflow[FROM NOW-1 TO NOW-1 MINUTES] i
 WHERE o.pressure < i.pressure AND i.pressure > 500

QoS: {ACQUISITION RATE = 3s ; DELIVERY TIME = 5s}
```

Figure 1: Example SNEEql Query

In SNEEql, the only structured type is tuple. The primitive collection types in SNEEql are window, an instance of which is a relation whose content may implicitly differ at different evaluation episodes; and stream, an instance of which is a bag of tuples with indefinite cardinality whose content may implicitly vary throughout query evaluation. As in CQL, operations construct windows out of streams and vice-versa. Windows are used to convert streams into relations, relational operators act on those relations, and stream operators emit the resulting tuples into the output stream. Window definitions are of the form *LogicalExtent*[*Window Dimension*], where *LogicalExtent* is an application-specific stream defined in a schema (e.g., inflow in Fig. 1) and *Window Dimension* takes one of two forms. If *Window Dimension* is of the form NOW, then the corresponding window contains all the tuples in the corresponding *LogicalExtent* whose timestamp equals the current time (denoted by t later on). If *Window Dimension* is of the form FROM *Start* TO *End Units*, where *Units* is either ROWS or a time unit (viz., HOURS, MINUTES or SECONDS), then *Start* and *End* together denote the scope (in time or in rows) of the tuples that the window contains. *Start* and *End* take one of two forms. The first form is simply NOW. The second form is NOW-*IntegerLiteral*, and allows windows to be placed on the past. Output tuples of relational operators are emitted into the result stream by relation-to-stream operators [1], viz., ISTREAM, DSTREAM and RSTREAM, for *inserted-only*, *deleted-only* and *all-tuples*, resp..

Table 1: SNEEql Physical Algebra (partial)

SP_ACQUIRE[attrSList, predExpr, projList]	Acquire sensor readings for each source in attrSList and emit the tuple resulting from an application of SELECT[predExpr] and PROJECT[projList].
DELIVER[ ]	Emit onto the query requester component.
TIME_WINDOW[start, end, slide]	Emit a window with scope start time to end time, inclusive, computed over the input stream and recompute every slide time units.
RSTREAM[ ]	Emit all the tuples in the input window.
NL_JOIN[predExpr, projList]	Use a nested loop to join tuples on predExpr and emit after applying PROJECT[projList].
SELECT[predExpr]	Emit tuples for which predExpr is true.
PROJECT[projList]	Emit tuple with attributes in projList.
TRANSMIT[ ]	Pack input tuples into blocks up to the maximum packet size and send them over radio.
RECEIVE[ ]	Receive blocks of up to the maximum packet size, unpack the tuples and emit them.

The (physical) algebra underlying SNEEql is (partially) summarized in Table 1[3]. Fig. 2 shows an example WSN over which the example query in Fig. 1 could be executed; directed edges depict the routing tree superimposed over the wireless network, in which dashed edges depict unused alternative paths. Note that: N9 is the sink node; N4, N5 and N7 host the sensors that provide values for inflow;

---

[3]More details on these operators, as well as the ones not shown in Table 1 (viz., ROW_WINDOW, ISTREAM, RSTREAM, AGGR_INIT, AGGR_MERGE and AGGR_EVAL), can be found in [2].
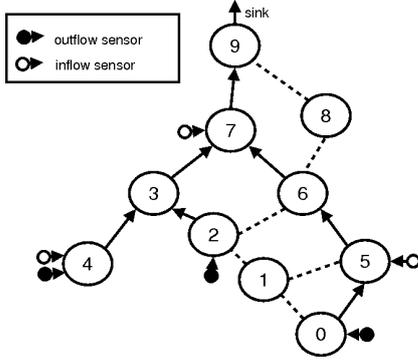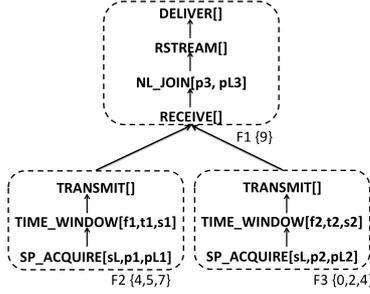
Figure 2: Example WSN with Routing Tree

Figure 3: Example Distributed QEP

N0, N2 and N4 provide values for outflow; and N3 and N6 are neither sources nor the sink of data. The techniques used to compile and optimize a query such as in Fig. 1 for execution in a WSN such as in Fig. 2 have been described in detail in [7]. The distributed QEP generated by SNEE for the query in Fig. 1 over the routing tree in Fig. 2 is shown in Fig. 3, where sL=[pressure], p1=pressure>500, p2=TRUE, p3 = o.pressure<i.pressure, pL1=[id,pressure], pL2 =[id,time,pressure], pL3=[i.id,o.id,o.time,o.pressure,i.pressure], f1=t1=t-6000, f2=t2=t-0, s1=s2=3000; the boxes denote distributed fragments; the solid arrows denote intranode dataflows; the dashed arrows internode dataflows using wireless links; and the call-outs denote the nodes where each fragment runs. Note that a fragment may be instantiated to run in several nodes [7]. Note also that some nodes act as relays only but, to avoid clutter, this is not shown in Fig. 3.

3. PHYSICAL OPERATORS

We now show how the operations in the **SNEEql** physical algebra can be expressed as concrete algorithms at a level in which it becomes possible to structurally derive CEMs for memory, time and energy for them. Due to lack of space, we illustrate our methodology with the **SP_ACQUIRE** and **TRANSMIT** physical operators because these illustrate representative data processing and movement capabilities. We have applied the same methodology to all other operators in the algebra, including the **RECIEVE** operator.

Broadly, the methodology is as follows: (1) We define, in pseudocode, the processing logic of an operator that gets executed at an evaluation episode (i.e., the equivalent to a getNext() in a classical physical operator that is designed for pipelined execution. (2) We declare, in the pseudocode, the state kept by the operator to support the processing logic in (1). (3) We fairly directly derive from (2) a CEM for mem-

SP_ACQUIRE[attrSList, predExpr, projExpList](tick)
1   **dependencies** ▷ *CPU, Sensor Board: on; Radio: off*
2   **State**
3   sensedValues: **array of** float **size** length(attrSList)
4   result: **array of** float **size** length(projExpList)+1
5   **begin**
6   ▷ ACQUIRE
7   **for** i=1 **to** length(attrSList):
8   sensedValues[i] ← SENSE(typeof(attrSList[i]))
9   ▷ SELECT
10 **if** apply(predExpr, sensedValues):
11 **then** ▷ PROJECT
12   result[0] ← tick
13   **for** j=1 **to** length(projExpList):
14   result[j] ← apply(projExpList[j], sensedValues)
15   **return** bagof(result)
16 **else return** bagof([ ])
17 **end**

Figure 4: Pseudocode for SP_ACQUIRE

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

Figure 5: Pseudocode for TRANSMIT

ory. (4) We derive from the algorithmic structure of the operator (as revealed in the pseudocode) a CEM for duration in the classical way, i.e., we take into account the most expensive steps, using multipliers when the step is iterated. (5) We derive from the CEM for duration obtained in (4) a CEM for energy by multiplying each addend in the former by the corresponding unit cost in energy. We define the **SP_ACQUIRE** and **TRANSMIT** operators in the remainder of section, and the CEMs derived for them in Section 4.

The **SP_ACQUIRE** physical operator (in Fig. 4) performs three operations from the corresponding logical algebra: it

acquires a tuple of sensed data as defined by attrSList, then it performs a select operation using predExpr, and finally, on those tuples that satisfied the selection condition, it performs a project operation using projList. The TRANSMIT physical operator (in Fig. 5) obtains (a pointer to) the results from its child operator and then packs tuples into a block containing as many tuples as will fit given the system-wide MAX_PACKET_SIZE, making sure that the last block is padded with NULL bytes if it is not full.

4. COST MODELS

In this section, we show how the style of pseudocode used in Section 3 can be built upon with a view to deriving memory, duration and energy CEMs for the corresponding operator. Some less conventional notations we use in our CEM formulation are as follows. We use lower-case sans-serif font for primitive functions and operations. We use UPPER-CASE SANS-SERIF font for physical operators and for CEM parameters (see Section 5 for parameter values corresponding to the sensors we have used in validating the CEMs). When expressing an aggregation, e.g., sum over a set of values $V = \{v_1, \ldots, v_n\}$, rather than write $\sum_{v \in V}(v)$, we write $\mathsf{sum}\{v \mid v \in V\}$. Finally, we note that the energy CEMs contain cross-references to the corresponding duration CEMs for the purposes of abbreviation only. Such references point to an addend in one equation in the duration CEM and should be interpreted in terms of textual substitution, i.e., textually replacing the reference with the expression it is a reference to yields the non-abbreviated form of the CEM. The CEMs for SP_ACQUIRE are collected in Fig. 6, those for TRANSMIT in Fig. 7.

Since the pseudocode declares the state kept in support of its processing logic, deriving a CEM for memory is tantamount to writing a summation in which each addend is the result of applying a primitive like sizeof to each scalar variable and summing the scalar elements in collection variables. This process is clearly illustrated in Eq. (1) in Fig. 6, bearing Fig. 4 in mind.

The derivation of a CEM for duration is similarly straightforward. The only additional concerns are: (a) to focus on steps which use processing more intensively (disregarding, e.g., atomic steps that do not involve calls to potentially expensive functions), and (b) to formulate the expressions that act as multiplicands on the processing blocks that are iterated over and that quantify the number of passes in the iteration. This process is illustrated in Eq. (2) in Fig. 6, bearing Fig. 4 in mind. Some parameters of interest in Eq. (2) are, $D_{\mathsf{SENSE(typeof}(s))}$, the time it takes to sense a value of a given type, A_P and A_E, the time it takes to evaluate an atomic Boolean and an atomic arithmetic expression, respectively. Finally, note that, for complex predicate and arithmetic expressions, we abstract the workload involved in terms of the number of atomic expressions the expression tree contains.

The derivation of a CEM for energy, given the corresponding duration CEM, is also straightforward. If we bear in mind that sensor nodes consume different amounts of unit energy for sensing, processing, receiving and transmitting, then the derivation of an energy CEM from a duration CEM essentially amounts to (1) classifying each addend in the duration CEM by the types of energy being spent in its duration, and (2) multiplying the durations thus obtained by the corresponding unit energy cost. This process is illustrated in

$$M_{\mathsf{SP\_ACQUIRE}(\mathrm{attrSList,predExpr,prjExpList})} = \qquad (1)$$
$$M_{\mathsf{SP\_A\_OVERHEAD}} + \mathsf{sizeof}(tick) +$$
$$\mathsf{sum}\{\mathsf{sizeof}(s) \mid s \in \mathrm{attrSList}\} +$$
$$\mathsf{sum}\{\mathsf{sizeof}(a) \mid a \in \mathrm{projExpList}\}$$

$$D_{\mathsf{SP\_ACQUIRE}(\mathrm{attrSList,predExpr,prjExpList,sensedValues})} = \quad (2)$$
$$D_{\mathsf{SP\_A\_OVERHEAD}} + \qquad\qquad\qquad\qquad (3)$$
$$(\mathsf{sum}\{D_{\mathsf{SENSE(typeof}(s))} \mid s \in \mathrm{attrSList}\}) + \quad (4)$$
$$(D_{\mathsf{APPLY}}(\mathsf{A\_P}, \mathrm{sensedValues}) * \qquad\qquad (5)$$
$$\mathsf{count}\{p \mid p \in \mathrm{predExpr} \wedge \mathsf{atomic}(p)\}) +$$
$$(\mathsf{sum}\{D_{\mathsf{APPLY}}(\mathsf{A\_E}, \mathrm{sensedValues}) * \qquad (6)$$
$$\mathsf{count}\{e \mid a \in \mathrm{projExprList} \wedge e \in a \wedge \mathsf{atomic}(e)\})$$

$$E_{\mathsf{SP\_ACQUIRE}(\mathrm{attrSList,predExpr,prjExpList})} = \qquad (7)$$
$$(E_{\mathsf{SENSE}} * Addend[4].Eq(2)) +$$
$$E_{\mathsf{PROCESS}} *$$
$$(Addend[3].Eq(2) +$$
$$Addend[5].Eq(2) +$$
$$Addend.[6].Eq(2) * \mathsf{selectivity}(\mathrm{predExpr}))$$

Figure 6: CEMs for SP_ACQUIRE

$$M_{\mathsf{TRANSMIT}(tick,child)} = \qquad\qquad\qquad (8)$$
$$M_{\mathsf{TRANS\_OVERHEAD}} + \mathsf{sizeof}(\mathsf{pointer}) +$$
$$(\mathsf{sizeof}(tuple) * \lfloor(\mathsf{MAX\_PACKET\_SIZE}/\mathsf{sizeof}(tuple))\rfloor) +$$
$$\mathsf{MAX\_PACKET\_SIZE} + \mathsf{sizeof}(i)$$

$$D_{\mathsf{TRANSMIT}(tick,child)} = \qquad\qquad\qquad (9)$$
$$D_{\mathsf{TRANS\_OVERHEAD}} + \qquad\qquad\qquad (10)$$
$$(D_{\mathrm{child.getNext}(tick)}) + (\qquad\qquad (11)$$
$$(D_{\mathsf{RX\_OVERHEAD}} + \qquad\qquad (12)$$
$$D_{\mathsf{TX\_OVERHEAD}} + \qquad\qquad (13)$$
$$(D_{\mathsf{TX\_BYTE}} * \mathsf{sizeof}(\mathrm{block}))) * \qquad (14)$$
$$\lceil\mathsf{count}\{tuple \in \mathrm{resultsFromChild}\}/\mathsf{length}(\mathrm{block})\rceil)$$

$$E_{\mathsf{TRANSMIT}(tick,child)} = \qquad\qquad\qquad (15)$$
$$(E_{\mathsf{PROCESS}} * Addend[10].Eq(9)) +$$
$$(E_{\mathsf{PROCESS}} * Addend[11].Eq(9)) + (($$
$$((E_{\mathsf{PROCESS}} + E_{\mathsf{RX}}) * Addend[12].Eq(9)) +$$
$$((E_{\mathsf{IDLE}} + E_{\mathsf{TX}}) * Addend[13].Eq(9)) +$$
$$((E_{\mathsf{IDLE}} + E_{\mathsf{TX}}) * Addend[14].Eq(9))) *$$
$$\lceil\mathsf{count}\{tuple \in \mathrm{resultsFromChild}\}/\mathsf{length}(\mathrm{block})\rceil)$$

Figure 7: CEMs for TRANSMIT

Eq. (7) in Fig. 6, bearing Fig. 4 in mind. In SP_ACQUIRE, there is no use of radio, so the unit energy costs involved are E_{SENSE}, the unit energy cost for sensing, and E_{PROCESS}, the unit energy cost for processing. Eq. (7) uses these two platform-specific parameters as multiplicands on the durations specified in the corresponding CEM, i.e., Eq. (2) in Fig. 6. Thus, energy is spent on sensing for the duration computed by Addend[4] in Eq. (2), i.e., the SP_ACQUIRE

section (ll. 7-8 in Fig. 4). Additionally, energy is spent on processing for the duration computed by Addend[5] and Addend[6] in Eq. (2), i.e., the **SELECT** and **PROJECT** sections (l. 10 and ll. 12-15, resp., in Fig. 4).

The CEMs for **TRANSMIT** are collected in Fig. 7. Due to lack of space, we do not provide a detailed analysis but we stress that the methodology used to derive them is the same as the one described above for **SP_ACQUIRE**. We have used the same methodology to derive memory, duration and energy CEMs for all the operators in Table 1.

5. VALIDATION STRATEGY

The accuracy of CEMs is a determinant factor in the quality of the decisions taken by an optimizer. The **SNEE** query optimizer [7] is particularly sensitive in this respect due to the objective of optimizing in a way that meets quality of service targets. We have therefore carried out an empirical validation of the CEMs using the Avrora emulator [14]. The results are described in Section 6. Avrora emulates in software the Mica2 mote hardware with cycle-level energy and duration accuracy. It also provides tools for taking measurements that would be very hard to reproduce in real motes.

Table 2 gives values to the parameters used in the CEMs, and Table 3 the unit energy costs[4], for Mica2/Avrora[5].

Table 2: Parameters for Mica2/Avrora

Parameter	Memory (bytes)	Duration (cycles)	Energy (μJ)
SP_A_OVERHEAD	14	124	(0.376)
SENSE	3	2542	(2.39)
APPLY(A_P,*)	(0)	8	(0.024)
APPLY(A_E,*)	(0)	8	(0.024)
TRANS_OVERHEAD	59	1215	3.680
RX_OVERHEAD	n/a	14353	(99.53)
TX_OVERHEAD	n/a	62446	(381.887)
TX_BYTE	n/a	3072	(18.787)

x

Table 3: Unit Energy Costs for Mica2/Avrora

Parameter	Energy per cycle
E_{SENSE}	0.0031826(0.0009402) μJ
E_{PROCESS}	0.0030286 μJ
E_{IDLE}	0.0013100 μJ
E_{RX}	0.0039061 μJ
E_{TX}	0.0048054 μJ

Recall Figs. 1 and 2. Section 6 shows the results obtained with five queries, Q1-Q5, of the form **SELECT**⟨ *projList* ⟩ **FROM** `inflow`, where *projList* grows from one attribute to five in steps of one, the attributes being `id, pressure, temperature, ph, time`. Of these, the id and time are not sensed. Id and the sensed attributes are two bytes long, and the time is four, bytes long. **TRANSMIT** was measured with Q1 sending three tuples per packet; Q2 and Q3, two; and Q4 and Q5, one.

---

[4]In Table 3, l.1, the first figure is for a Mica2 mote, but because in the Avrora emulator, the sensor board cannot be switched off, we have used the figure in brackets in our validation, as it compensates for that limitation.

[5]Note that, in Table 2, the values in parentheses are not needed in the CEMs but are given here for completeness.

In order to isolate and measure the cost of a specific operator, two versions of the nesC executable corresponding to the QEP were generated, run and compared, as follows. The control version only includes operators up to the operator of interest (plus all overheads, including timers [7]). The complete version includes all the code from the control version plus the code for the operator of interest. Neither includes code for operators downstream from the operator of interest, in order to avoid their affecting the cost measurements.

SNEE uses code generation, i.e., it emits nesC source [7]. In nesC, all memory is allocated at compile time. The measured memory cost for an operator is the reported RAM for the complete version minus the reported RAM for the control version. Thus, the reported RAM values for the Q5 **TRANSMIT** are 640 for the control and 699 for the complete version, indicating a measured memory cost for the Q5 **TRANSMIT** of 59 bytes. The duration of each operator was measured by turning an LED on and off at the beginning and at the end of the operator of interest. Such LED events are monitorable in Avrora. The difference between the two LED operations, adjusted for the LED overhead, provides a cycle accurate duration measurement. Thus, the second Q5 **TRANSMIT** occurred between cycles 37537720 and 38276315. Discounting the LED overhead (12 cycles), gives us 737,582 cycles or 100.18 ms. Energy measurements were derived by dividing the difference between the reported energies by the number of evaluations, which equals the simulation duration divided by the evaluation rate. Thus, the Q5 **SP_ACQUIRE** cost for a simulation lasting 100 seconds with evaluation episodes every 5 seconds was ($140155.05\mu J$ - 42476.31 μJ) / ($100/5$) = $4883.94\mu J$/episode.

As Avrora is only able to emulate a luminosity sensor, we have simulated the sensing of other physical quantities by repeatedly calling the luminosity sensor. While this did provide us with increasing costs as the number of sensed attributes increases, validating the cost models, the results are somewhat lower than the costs reported by hardware makers. Similarly, Avrora does not emulate turning off the sensor board so the reported sensor energy costs do not include the costs incurred by the sensor hardware. The CEMs do take into account sensing durations that vary depending on the type of sensor, as happens in practice.

6. EXPERIMENTAL RESULTS

Table 4 shows that the validation results for the memory, duration and energy CEMs for **SP_ACQUIRE** vary as the number of sensors and the tuple size vary. We note that despite the artificially low SENSE cost required by Avrora, the number of sensors plays a much larger role than the size of the output tuple, especially for duration and energy. This is a parameter, and hence does not compromise the quality of the CEM, let alone of the methodology we have described. Table 5 shows that, as expected, the duration and energy cost of sending six tuples varies greatly as tuple size increases. Small increases occur between Q2 and Q3 and again between Q4 and Q5 as an extra two bytes are sent per packet. However the most significant difference occurs when the number of tuples that fit in a packet decreases from three to two at Q2 and from two to one at Q4. Table 5 does not include memory, as we have found that the nesC compiler inlines the variable-size *block* into the fixed-size *packet*, thereby removing the expected variability in the memory CEM for **TRANSMIT**.

Table 4: Validation of the SP_ACQUIRE CEMs

Q	Sensors	Size	Model	Actual	Diff	Error
			RAM usage in bytes			%
Q1	1	6	27	27	0	0.00 %
Q2	2	8	32	32	0	0.00%
Q3	1	10	31	31	0	0.00%
Q4	2	12	36	36	0	0.00%
Q5	3	14	41	41	0	0.00%
			Duration in cycles			%
Q1	1	6	2698	2696	2.0	0.07 %
Q2	2	8	5248	5241	7.0	0.13 %
Q3	1	10	2706	2704	2.0	0.07 %
Q4	2	12	5256	5249	7.0	0.13 %
Q5	3	14	7806	7804	2.0	0.03 %
			Energy in μJ			%
Q1	1	6	3.25	3.26	0.01	0.20 %
Q2	2	8	5.67	5.65	0.02	0.27 %
Q3	1	10	3.27	3.28	0.01	0.20 %
Q4	2	12	5.69	5.67	0.02	0.27 %
Q5	3	14	8.11	8.10	0.01	0.08 %

Table 5: Validation of the TRANSMIT CEMs

Query	Size	Actual	Model	Diff	Error
		Duration in cycles			%
Q1	6 bytes	302497	299287	3210	1.07 %
Q2	8 bytes	416160	412958	3202	0.78 %
Q3	10 bytes	453024	449815	3209	0.71 %
Q4	12 bytes	720285	704791	15494	2.20 %
Q5	14 bytes	757149	738583	18566	2.51 %
		Energy in μJ			%
Q1	6 bytes	1869.98	1950.18	81.20	4.34 %
Q2	8 bytes	2575.84	2671.82	95.98	3.73 %
Q3	10 bytes	2801.28	2937.73	136.42	4.87 %
Q4	12 bytes	4470.99	4641.69	170.70	3.82 %
Q5	14 bytes	4696.43	4883.94	187.51	3.99 %

7. CONCLUSIONS

This paper has described a methodological approach to the derivation of memory, time and energy CEMs for QEPs intended for execution over WSNs. As technical background to the contributions of this paper, we have described (albeit in a necessarily brief manner), the SNEE framework of distributed query processing over sensed streams, its associated query language, called SNEEql, and the corresponding physical algebra [2, 6, 7]. The contributions described in this paper are the specification of concrete algorithms for physical algebraic operators in such a way, i.e., in a format and pitched at such an abstraction level, that the task of deriving memory, time and energy CEMs for them becomes straightforward by reduction to a structural traversal of the pseudocode. Moreover, we have shown how the energy CEM is fairly directly derivable from the duration CEM alone. Finally, we have shown that the CEMs derived by our methodology are valid (i.e., exhibit satisfactorily accurate results under emulations that are themselves accurate at the level of processor cycles). Given the challenging nature of the problem of optimizing a declarative query for execution in WSNs, and given the dependency of any solution on empirically validated CEMs, we believe that the contributions reported in this paper, particularly in their methodological import, constitute a significant step towards better optimizers for declarative sensor query processing frameworks.

8. REFERENCES

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, and J. Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[2] C. Y. A. Brenninkmeijer, I. Galpin, A. A. A. Fernandes, and N. W. Paton. A Semantics for a Query Language over Sensors, Streams and Relations. In *BNCOD*, pages 87–99, 2008.

[3] S. Chaudhuri. An overview of query optimization in relational systems. In *PODS*, pages 34–43, 1998.

[4] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE*, pages 345–356, 2002.

[5] S. de F. Mendes Sampaio, N. W. Paton, J. Smith, and P. Watson. Measuring and modelling the performance of a parallel odmg compliant object database server. *Concurrency and Computation: Practice and Experience*, 18(1):63–109, 2006.

[6] I. Galpin, C. Y. A. Brenninkmeijer, F. Jabeen, A. A. A. Fernandes, and N. W. Paton. An Architecture for Query Optimization in Sensor Networks. In *ICDE*, pages 1439–1441, 2008.

[7] I. Galpin, C. Y. A. Brenninkmeijer, F. Jabeen, A. A. A. Fernandes, and N. W. Paton. Comprehensive Optimization of Declarative Sensor Network Queries. In *SSDBM*, pages 339–360, 2009.

[8] M. N. Garofalakis and Y. E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *VLDB*, pages 296–305, 1997.

[9] J. Gehrke and S. Madden. Query Processing in Sensor Networks. In *IEEE Pervasive Computing*, volume 3. IEEE Computer Society, 2004.

[10] G. Luo, J. F. Naughton, C. J. Ellmann, and M. Watzke. Toward a progress indicator for database queries. In *SIGMOD*, pages 791–802, 2004.

[11] M. T. Roth, F. Ozcan, and L. M. Haas. Cost models DO matter: Providing cost information for diverse data sources in a federated system. In *VLDB*, pages 599–610, 1999.

[12] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[13] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.

[14] B. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN*, pages 477–482, 2005.

[15] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, pages 37–48, 2002.

[16] Q. Zhu, Y. Sun, and S. Motheramgari. Developing cost models with qualitative variables for dynamic multidatabase environments. In *ICDE*, pages 413–424, 2000.