Comprehensive Optimization of Declarative Sensor Network Queries

Ixent Galpin, Christian Y.A. Brenninkmeijer, Farhana Jabeen,
Alvaro A.A. Fernandes, Norman W. Paton
University of Manchester, United Kingdom
(ixent,brenninc,jabeen,alvaro,norm)@cs.man.ac.uk

ABSTRACT

We present a novel sensor network query processing architecture that (a) covers all the query optimization phases that are required to map a declarative query to executable code; and (b) does so for a more expressive query language than has heretofore been supported over sensor networks. The architecture is founded on the view that a sensor network truly is a distributed computing infrastructure, albeit a very constrained one. As such, we address the problem of how to develop a comprehensive optimizer for an expressive declarative continuous query language over acquisitional streams as one of finding extensions to classical distributed query processing techniques that contend with the peculiarities of sensor networks as an environment for distributed computing. We show that the approach is effective and offers measurable improvements in performance compared to related work.

1. INTRODUCTION

Problem Definition and Motivation This paper addresses the problem of optimizing the evaluation of declarative queries over sensor networks (SNs) [13]. Throughout, by *sensor networks* we specifically mean ad-hoc, wireless networks whose nodes are energy-constrained sensors endowed with general-purpose computing capability. Viewed as a particular type of distributed computing infrastructure, SNs are constrained to an unprecedented extent, and it is from such constraints that the challenges we have addressed arise. Addressing these challenges is important because of the prevailing expectations for the wide applicability of SNs [6].

In this paper, we explore the hypothesis that the classical *two-phase optimization* approach [14] from distributed query processing (DQP) can be adapted to be effective and efficient over SNs. Two-phase optimization is well established in the case of robust networks (e.g., the Internet, or the interconnect of a parallel machine), and involves the decomposition of query optimization into a *Single-Site* phase, and a subsequent *Multi-Site* phase, each of which is further decomposed into finer-grained decision-making steps. We aim to reuse well established optimization components where possible, and to identify steps that are necessarily different in SNs. We demonstrate that extending a classical DQP optimization architecture allows an expressive query language to be supported over resource-constrained networks, and that the resulting optimization steps yield energy efficiency benefits, as demonstrated through empirical evaluation.

Related Work There have been many proposals in the so-called SN-as-database approach (including [8, 10, 18, 28]). Surprisingly, none have followed an approach to query optimization founded on a classical DQP architecture. Indeed, there are few publications that provide systematic descriptions of complete query optimization architectures for SN query processors; the only such description found was for TinyDB [18], in which optimization is limited to operator reordering and the use of cost models to determine an appropriate acquisition rate given a user-specified lifetime. Arguably as a result of this, SN-as-database proposals have tended to limit the expressiveness of the query language. For example, TinyDB focuses on aggregation and provides limited support for joins. In many cases, assumptions are made that constrain the generality of the approach (e.g., Presto [8] focuses on storage-rich networks).

There has also been a tendency to address the optimization problem in a piecewise manner. To give some examples, the use of probabilistic techniques to address the trade-off between acquiring data often and the cost of doing so is proposed in BBQ [5]; the trade-off between energy consumption and time-to-delivery is studied in [27]; efficient and robust aggregation is the focus of [17, 19, 26]; a cost-based approach to adaptively placing a join which operates over distributed streams is proposed in [3]; and an algebraic approach is used to generate schedules for the transmission of data in order to maximize the number of concurrent communications in [29]. However, these individual results are rarely presented as part of a fully-characterized optimization and evaluation infrastructure, giving rise to a situation in which research at the architecture level seems less well developed than that of techniques that might ultimately be applied within such query processing architectures.

It can therefore be seen that the hypothesis of extending a classical DQP optimization architecture for the SN context has not been explored. In contrast, this paper aims to provide a comprehensive, top-to-bottom approach to the optimization problem for expressive declarative continuous queries over potentially heterogeneous SNs. In comparison with past proposals, ours is broader, in that there are fewer

compromises with respect to generality and expressiveness, and more holistic, in that it provides a top-to-bottom decomposition of the decision-making steps required to optimize a declarative query into a query execution plan (QEP).

Approach and Main Results A key aspect of our approach is that it supports the optimization and evaluation of SNEEql (for Sensor NEtwork Engine query language), a more expressive query language than others that have hitherto been proposed for SNs. SNEEql is inspired by classical stream languages such as CQL [1]. Thus, we do not start from the assumption that a query language for use with resource-constrained devices must, as a consequence, provide limited functionality. We then take a classical DQP optimizer architecture as a starting point, and adapt it to consider how the optimization and evaluation of queries for SNEEql differs if, rather than targeting classical distributed computational resources, we target SNs. We anchor our approach on the observation that a SN is a distributed computing platform, albeit a very constrained one. We identify what assumptions in classical DQP are violated in the SN case, and propagate the consequences of such violations into the design and engineering of a DQP architecture for SN data management. The differences we consider here that have led to adaptations and extensions, are: (i) *the acquisitional nature of the query processing task*: data is neither lying ready in stores nor is it pushed (as in classical streams), but requested; (ii) *the energy-constrained nature of the sensors*: preserving energy becomes a crucial requirement because it is a major determinant of network longevity, and requires the executing code to shun energy-hungry tasks; (iii) *the fundamentally different nature of the communication links*: wireless links are not robust, and often cannot span the desired distances, so the data flow topology (e.g., as embodied in a query operator tree) needs to be overlaid onto some query-specific network topology (e.g., a routing tree of radio-level links) for data to flow from sensors to clients, and the two trees are not isomorphic (e.g., some network nodes act as pure relay nodes); and (iv) *the need to run sensor nodes according to data-dependent duty cycles*: each element in the computational fabric must act in accordance with an agenda that co-ordinates its activity with that of other elements on which it is dependent or that depend on it, thereby enabling energy management (e.g., by sending devices to energy-saving states until the next activity). We note that these points also distinguish our approach from infrastructures for stream query processing (e.g., [1]), which do not operate in resource constrained environments.

Summary of Contributions The body of the paper describes the following contributions: (1) a user-level syntax (Section 2) and algebra (Section 4.1) for SNEEql, an expressive language for querying over acquisitional sensor streams; (2) an architecture for the optimization of SNEEql, building on well-established DQP components where possible, but making enhancements or refinements where necessary to accommodate the SN context (Section 4); (3) algorithms that instantiate the components, thereby supporting integrated query planning that includes routing, placement and timing (Sections 4.2.1–4.2.4); and (4) an evaluation of the resulting infrastructure, demonstrating the impact of design and optimization decisions on query performance, and the increased empowering to the user (Section 5) who is able to trade-off different QoS according to application needs.

```
Schema:
  outflow (id, time, temp, turbidity, pressure)
  outflow source sites: {0, 2, 4}
   inflow (id, time, temp, pressure, ph)
   inflow source sites: {4, 5, 7}

Query 1:
  SELECT RSTREAM *
  FROM   inflow[NOW]
  WHERE  pressure > 500;

Query 2:
  SELECT RSTREAM AVG(pressure)
  FROM   inflow[NOW]

Query 3:
  SELECT RSTREAM
         outflow.time, outflow.pressure, inflow.pressure
  FROM   outflow[NOW],
         inflow[FROM NOW - 1 TO NOW - 1 MINUTES]
  WHERE  outflow.pressure < inflow.pressure
  AND    inflow.pressure > 500

Quality of Service Expectations:
  ACQUISITION RATE = EVERY 3 SECONDS
  DELIVERY TIME = 5 SECONDS
```

Figure 1: Example queries in SNEEql

2. QUERY LANGUAGE

SNEEql is inspired by expressive classical stream query languages such as CQL [2]. A rich language is used even though our target delivery platform consists of limited devices because: (i) the results of queries written using inexpressive query languages may require offline post-processing over data that has to be delivered using costly wireless communication; and (ii) sensor applications require comprehensive facilities for correlating data sensed in different locations at different times (e.g., [20, 23]).

In SNEEql, the only structured type is tuple. The primitive collection types in SNEEql are: relation, an instance of which is a bag of tuples with definite cardinality; window, an instance of which is a relation whose content may implicitly vary between evaluation episodes; and stream, an instance of which is a bag of tuples with indefinite cardinality whose content may implicitly vary throughout query evaluation. As in CQL, operations construct windows out of streams and vice-versa.

Several example SNEEql queries are given in Fig. 1, motivated by the application scenario (but not actually occurring) in PipeNet, "a system based on wireless SNs [that] aims to detect, localize and quantify bursts and leaks and other anomalies in water transmission pipelines" [23]. The acquisitional streams (i.e., `inflow` and `outflow`) and the sensor nodes from which they stem are described in Fig. 1, while Fig. 3 depicts the topology of the network. The examples illustrate how SNEEql can express select-project (*Query 1*), aggregation (*Query 2*) and join (*Query 3*) queries.

In all the queries, windows are used to convert from streams to relations, relational operators act on those relations, and stream operators add the resulting tuples into the output stream. Window definitions are of the form *WindowDimension* [`SLIDE`] [*Units*], where the *WindowDimension* is of the form `NOW` or `FROM` *Start* `TO` *End*, where the former contains all the tuples with the current time stamp, and the latter contains all the tuples that fall within the given range. The *Start* and *End* of a range are of the form `NOW` or `NOW` –*Literal*, where the *Literal* represents some number of *Units*, which is

either ROWS or a time unit (HOURS, MINUTES or SECONDS). The SLIDE indicates the gap in *Units* between the *Start* of successive windows. The results of relational operators are added to the result stream using one of the relation-to-stream operators from CQL, namely ISTREAM, DSTREAM and RSTREAM, denoting *inserted-only*, *deleted-only* and *all-tuples*, respectively.

A typical requirement supported by the above window specifications is to correlate data from different locations at different times, which cannot be expressed with previous SN query languages. For example, *Query 3* aims to detect potential leaks by obtaining every three seconds, timestamped, correlated pressure readings in which the outflow pressure now is lower than the inflow pressure a minute ago (as long as the latter was above a certain threshold).

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 implementation, however, partial query results may be cached within the network with a view to minimizing network traffic, and different parts of a query may be evaluated at different times, reflecting QoS expectations, viz., the *acquisition rate* and the *delivery time*, as shown in Fig. 1.

Noteworthy features of SNEEql illustrated in Fig. 1 include: (i) extending CQL stream-to-window capabilities to allow for windows whose endpoint is earlier than now or than the last tuple seen, as happens in *Query 3* with the window on inflow, which emits tuples that were acquired one minute ago; (ii) allowing sensing capabilities to be logically abstracted in a schema (like Cougar [10], but unlike TinyDB, which assumes that all sensed data comes from a single extent denoted by the keyword SENSORS); (iii) allowing the logical streams to stem from more than one set of sensor nodes, and possibly intersecting ones (as is the case in Fig. 1); (iv) expressing joins, where the tuples come from different locations in the SN, as a single query and without using materialization points (as would be required in TinyDB); and (vi) allowing QoS expectations to be set for the compiler/optimizer, such as acquisition rate and delivery time.

3. TECHNICAL CONTEXT

We note that work on SN query processing necessarily makes certain assumptions about its technical context; for example, [18] assumes a homogeneous SN and targets the Mica mote platforms running TinyOS [12]. Furthermore, it is assumed that network management functionality (such as topology control, self-organization and synchronization) is the responsibility of the query processor. In this paper, our query optimizer currently targets Mica2 motes running TinyOS. We also make certain assumptions about the networking infrastructure. We observe that because sensor networks are ad-hoc wireless networks, they must, to some degree, self-organize. For example, each node must identify its neighborhood, and links must be established and maintained between neighbor nodes, in order for what is otherwise merely a collection of nodes to be constituted into, and operate as, an effective and efficient network. In this light, we assume that when sensor network data management systems do reach maturity they will likely conform to an architecture in which the set of functionalities relating to self-organization will be, from the point of view of query processors, encapsulated and logically abstracted.

To give an example of the implications of this assumption in the case of this paper, we acknowledge that query evaluation must be robust in the (likely) event of physical sensor node failure. The question arises as to whether the query compiler/optimizer should make decisions that aim to preempt the consequences of such failure or to delegate them to other architectural components. Delegating these decisions is tantamount to an independence assumption (not dissimilar, albeit at a different level, to the one that insulates query processing from the specifics of physical storage). In the example above, when a query fragment is assigned to a node, it would be assigned to a *logical* node. A logical node, in this sense, is an abstraction exposed by network management code that, as described, e.g., in [7], enables physical nodes to change roles in response to changes in the self-organizing physical fabric. Crucially, this layer of abstraction insulates the query processor from many changes at the level of the physical fabric. In particular, the query processor can be seen to exhibit a degree of resilience to physical node failures because the latter are the concern of the network management code layer below.

The various functionalities (e.g., neighborhood discovery, network formation, topology control, adaptive access to the communication medium, etc.) that, we postulate, can be abstracted in this way are the subject of great interest in network research (see [13] for an exhaustive and systematic survey). Our assumption is, therefore, that a collection of software abstractions will either emerge (similarly to those already identified in [16]) or be designed (as proposed in [7]) and, in time, will come to constitute an architectural layer which query processing technology can build on. While previous work on sensor network query processing [18, 27, 17, 26] has often felt it necessary to address self-organization concerns inside the query processor, we believe that architectural maturity will lead to separation of concerns, and that, from the viewpoint of database research, the more precise demarcation of responsibilities that we adopt in this paper, indicates the issues that query processing research uniquely can address.

4. QUERY COMPILER/OPTIMIZER

Recall that our goal is to explore the hypothesis that extensions to a classical DQP optimization architecture can provide effective and efficient query processing over SNs. The SNEEql compilation/optimization stack is illustrated in Fig. 2, and comprises three phases. The first two are similar to those familiar from the two phase-optimization approach, namely *Single-Site* (comprising Steps 1-3, in gray boxes) and *Multi-Site* (comprising steps 4-7, in white, solid boxes). The *Code Generation* phase grounds the execution on the concrete software and hardware platforms available in the network/computing fabric and is performed in a single step, Step 8 (in a white, dashed box). The query stack consists of around 15K effective lines of Java.

Classical kinds of metadata (e.g., schematic information, statistics, etc.) are used by the SNEEql optimizer, but we focus here on the requirement for a more detailed description of the network fabric. Thus, Fig. 3 depicts an example

Stream-to-Stream Operators	
ACQUIRE[AcquisitionInterval, PredExpr, AttrList](S) : S	Take sensor readings every AcquisitionInterval for sources in S and apply SELECT[PredExpr] and PROJECT[AttrList]. *LocSen*.
DELIVER[ ](S) : S	Deliver the query results. *LocSen*.
Stream-to-Window Operators	
TIME_WINDOW[startTime, endTime, Slide](S) : W	Define a time-based window on S from startTime to endTime inclusive and re-evaluate every slide time units.
ROW_WINDOW[startRow, endRow, Slide](S) : W	Define a tuple-based window on S from startRow to endRow inclusive and re-evaluate every slide rows. *AttrSen*.
Window-to-Stream Operators	
RSTREAM[ ](W) : S	Emit all the tuples in W since the previous window evaluation.
ISTREAM[ ](W) : S	Emit the newly-inserted tuples in W since the previous window evaluation.
DSTREAM[ ](W) : S	Emit the newly-deleted tuples in W since the previous window evaluation.
Window-to-Window or Relation-to-Relation Operators	
NL_JOIN[ProjectList, PredExpr]($R\|W$,$R\|W$) : $R\|W$	Join tuples on PredExpr condition using nested-loop algorithm. *AttrSen*.
AGGR_INIT[AggrFunction, AttrList]($R\|W$) : $R\|W$	Initialize incremental aggregation for attributes in AttrList for type of aggregation specified by AggrFunction. *AttrSen*.
AGGR_MERGE[AggrFunction, AttrList]($R\|W$) : $R\|W$	Merge partial-result tuples of incremental aggregation for attributes in AttrList for type of aggregation specified by AggrFunction. *AttrSen*.
AGGR_EVAL[AggrFunction, AttrList]($R\|W$) : $R\|W$	Evaluate final result of incremental aggregation for attributes in AttrList for type of aggregation specified by AggrFunction. *AttrSen*.
Any-to-Same-as-input-type Operators	
SELECT[PredExpr]($R\|S\|W$): $R\|S\|W$	Eliminate tuples which do not meet PredExpr predicate.
PROJECT[AttrList] ($R\|S\|W$): $R\|S\|W$	Generate tuple with AttrList attributes.

Table 1: SNEEql Physical Algebra

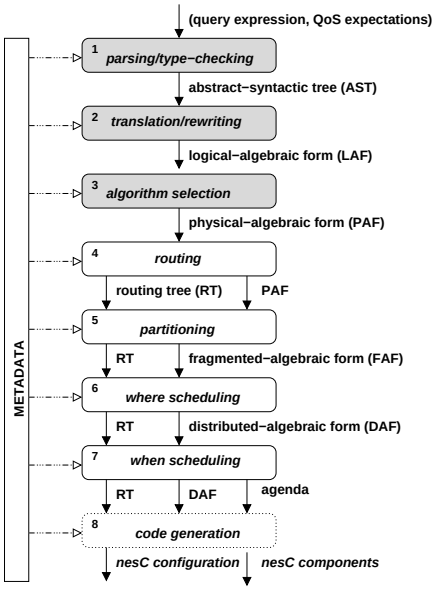

Figure 2: SNEEql Compiler/Optimizer Stack

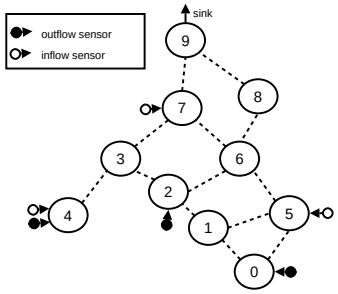

Figure 3: Connectivities and Modalities

network for the case study in Fig. 1 in terms of a weighted connectivity graph and the sensing modalities available in each site. Dotted edges denote that single-hop communication is possible. Weights normally model energy, but more complex weights could be used. Here, we assume links to have unit costs, but this need not be so. It is assumed that all sites may contribute computing (e.g., processing intermediate results) or communication (e.g., relaying data) capabilities.

4.1 Phase 1: Single-Site Optimization

Single-site optimization is decomposed into components that are familiar from classical, centralized query optimizers. We make no specific claims regarding the novelty of these steps, since the techniques used to implement them are well-established. In essence: *Step 1* checks the validity of the query with respect to syntax and the use of types, and builds an abstract syntax tree to represent the query; *Step 2* translates the abstract syntax tree into a logical algebra, the operators of which are reordered to minimize the size of intermediate results; and *Step 3* translates the logical algebra into a physical algebra, which, e.g., makes explicit the algorithms used to implement the operators.

Fig. 4 depicts the outcome of Steps 1 to 3 for *Query 3* from Fig. 1, expressed using the *physical-algebraic form* (PAF) that is the principal input to multi-site optimization. Table 1 describes the physical-algebraic operators, grouped by their respective input-to-output collection types. A signature is given for each operator of the form OPERATOR_NAME[Parameters](InputArgumentTypes):OutputArgumentTypes where the argument types are denoted R, S and W, for relation, stream and window respectively, and a vertical bar indicates a choice of one of the types given. Properties of operators, used by the optimizer, are also given: *LocSen* specifies that an operator is location sensitive, i.e., there is no leeway as to which node(s) in the SN it may execute on; *AttrSen* specifies that an operator is attribute-sensitive, i.e., in order to carry out partitioned-parallelism the optimizer needs to consider how the input(s) are partitioned in order

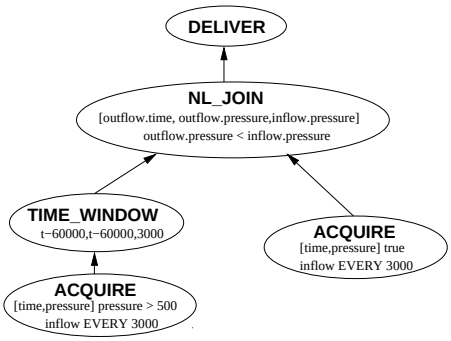

Figure 4: Physical-Algebraic Form of *Query 3*

to preserve operator semantics.

ACQUIRE and DELIVER can be seen as location-sensitive operators denoting data sources and sinks. TIME_WINDOW and ROW_WINDOW specify the window dimensions and the gaps between the starts of successive windows. For example, in *Query 3*, from Fig. 1, windows are time-based, and refer to points or intervals in time rather than number of tuples. The window on `outflow` is represented in the algebra as TIME_WINDOW[t,t,1], and the window on `inflow` is represented as TIME_WINDOW[t-60000,t-60000,30000]. In the algebra, times are in milliseconds, and are relative to `t`, which is bound in turn to the time in which each evaluation episode of the query starts. Aggregates (such as average) are computed in three phases (*initialize*, *iterate* and *evaluate*), each implemented as a separate physical operator, as is done in TinyDB. This helps reduce radio traffic and allows the computation of different operands in the algebraic expression to be scheduled separately. The SELECT and PROJECT operators (unlike in CQL) operate on both windows *and* streams (as well as relations, of course).

4.2 Phase 2: Multi-Site Optimization

For distributed execution, the physical-algebraic form is broken up into QEP fragments for evaluation on specific nodes in the network. In a SN, consideration must also be given to routing (the means by which data travels between nodes within the network) and duty cycling (when nodes transition from being switched on and engaged in specific tasks, and being asleep, or in power-saving modes). Therefore, for Steps 4-7, we consider the case of robust networks and the contrasting case of SNs.

For execution over multiple nodes in robust networks, the second phase is comparatively simple: one step partitions the physical-algebraic form into fragments and another step allocates them to suitably resourced sites, as in, e.g., [24]. One approach to achieving this is to map the physical-algebraic form of a query to a distributed one in which EXCHANGE operators [11] (that encapsulate all of control flow, data distribution and inter-process communication) define boundaries between fragments.

However, for the same general approach to be effective and efficient in a SN, a response is needed to the fact that assumptions that are natural in the robust-network setting cease to hold in the new setting and give rise to a different set of challenges, the most important among which are the following: **C1**: *location and time are both concrete*: acquisitional query processing is grounded on the physical world, so sources are located and timed in concrete space and time and the optimizer may need to respond to the underlying geometry and to synchronization issues; **C2**: *resources are severely bounded*: sensor nodes can be depleted of energy, which may, in turn, render the network useless; **C3**: *communication events are overly expensive*: they have energy unit costs that are typically an order of magnitude larger than the comparable cost for computing and sensing events; and **C4**: *there is a high cost in keeping nodes active for long periods*: because of the need to conserve energy, sensor node components must run tight duty cycles (e.g., going to sleep as soon they become idle).

Our response to this different set of circumstances is reflected in Steps 4-7 in Fig. 2, where rather than a simple partition-then-allocate approach (in which a QEP is first partitioned into fragments, and these fragments are then allocated to specific nodes on the network), we: (a) introduce Step 4, in which the optimizer determines a routing tree for communication links that the data flows in the operator tree can then rely on, with the aim to address the issue that paths used by data flows in a query plan can greatly impact energy consumption (a consequence of **C3**); (b) preserve the query plan partitioning step, albeit with different decision criteria, which reflect issues raised by **C1**; (c) preserve the scheduling step (which we rename to *where-scheduling*, to distinguish it from Step 7), in which the decision is taken as to where to place fragment instances in concretely-located sites (e.g., some costs may depend on the geometry of the SN, a consequence of **C1**); and (d) we introduce *when-scheduling*, the decision as to when, in concrete time, a fragment instance placed in a given site is to be evaluated (and queries being continuous, there are typically many such episodes) to address **C1** and **C4**. **C2** is taken into account in changes throughout the multi-site phase.

For each of the following subsections that describe Steps 4 to 7, we indicate how the proposed technique relates to DQP and to TinyDB, the former because we have used established DQP architectures as our starting point, and the latter because it is the most fully characterized proposal for a SN query processing system. The following notation is used throughout the remainder of this section. Given a query Q, let P_Q denote the graph-representation of the query in physical-algebraic form. Throughout, we assume that: (1) operators (and fragments) are described by properties whose values can be obtained by traditional accessor functions written in dot notation (e.g., P_Q.Sources returns the set of sources in P_Q); and (2) the data structures we use (e.g., sets, graphs, tuples) have functions with intuitive semantics defined on them, written in applicative notation (e.g., for a set S, ChooseOne(S) returns any $s \in S$; for a graph G, EdgesIn(G) returns the edges in G); Insert((v_1,v_2),G) inserts the edge (v_1,v_2) in G.

4.2.1 Routing

Step 4 in Fig. 2 decides which sites to use for routing the tuples involved in evaluating P_Q. The objective is to compute a routing tree for P_Q that reduces the total energy cost with respect to the edges over which tuples are transmitted. Let $G = (V, E, d)$ be the weighted connectivity graph for the target SN (e.g., the one in Fig. 3). Let P_Q.Sources $\subseteq G.V$ and P_Q.Destination $\in G.V$ denote, respectively, the set of sites that are data sources, and the destination site, in P_Q. The aim is, for each source site, to minimize the

```
ROUTING(P_Q, G)
1  rtV ← {P_Q.Destination}
2  rtE ← ∅
3  remainingV ← P_Q.Sources
4  while remainingV ≠ ∅
5        do from ← ChooseOne(remainingV)
6           to ← ChooseOne(rtV)
7           path ← Shortest-Path(from, to, G)
8           rtE ← rtE ∪ EdgesIn(path)
9           rtV ← rtV ∪ VerticesIn(rtE)
10          remainingV ← remainingV \ rtV
11 return (rtV, rtE)
```

Figure 5: Computing a SNEEql Routing Tree

total cost to the destination. We observe that this is an instance of the *Steiner tree* problem, in which, given a graph, a tree of minimal cost is derived which connects a required set of nodes (the *Steiner nodes*) using any additional nodes which are necessary [13]. Thus, the SNEEql-optimal routing tree R_Q for Q is the Steiner tree for G with Steiner nodes P_Q.Sources $\cup$ $\{P_Q$.Destination$\}$. The problem of computing a Steiner tree is NP-complete, so the heuristic algorithm given below (and reputed to perform well in practice [13]) is used to compute an approximation. First, the algorithm (see Fig. 5) makes the destination site a vertex in the Steiner tree. Then, it removes the remaining Steiner points one by one after finding the shortest path between the removed point and some point already in the tree, adding to the tree all the sites in the computed path and stopping once all Steiner points appear in the tree. For the physical-algebraic form in Fig. 4, given the network topology in Fig. 3, the routing algorithm computes the overlay routing tree depicted in Fig. 6 by arrows between the nodes. Note that nodes 1 and 8 are not in the routing tree (i.e., have no incoming or outgoing data flows), and therefore, do not participate in any way in the query. This allows conservation of their resources.

Relationship to DQP This step has been introduced in the SN context due to the implications of the high cost of wireless communications, viz., that the paths used to route data between fragments in a query plan have a significant bearing on its cost. Traditionally, in DQP, the paths for communication are solely the concern of the network layer. In a sense, for SNEEql, this is also a preparatory step to assist the subsequent *where-scheduling* step, in that the routing tree imposes certain constraints on the data flows, and thus on where operations can be placed.

TinyDB In TinyDB, routing tree formation is undertaken by a distributed, parent-selection protocol at runtime. Our approach aims, given the sites where location-sensitive operators need to be placed, to minimize the distance traveled by tuples. TinyDB does not directly consider the locations of data sources while forming its routing tree, whereas the approach taken here can potentially make finer-grained decisions about which depletable resources (e.g., energy) to make use of in a query. This may prove useful, e.g., if energy stocks are consumed at different rates at different nodes.

4.2.2 Partitioning

Step 5 in Fig. 2 defines the fragmented form F_Q of P_Q by breaking up selected edges $(child,\ op) \in P_Q$ into a path

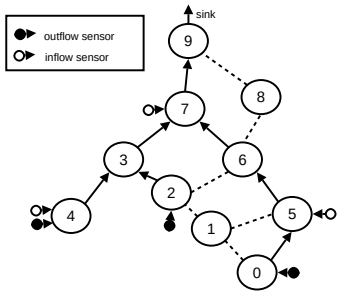

Figure 6: Routing Tree for the PAF form in Fig. 4

```
FRAGMENT-DEFINITION(P_Q, Size)
1  F_Q ← P_Q
2  while   ▷ post-order traversing F_Q,
           ▷ let op denote the current operator
3        do for each child ∈ op.Children
4              do if Size(op) > Size(op.Children)
5                 or op.LocationSensitive = yes
6                 or op.AttributeSensitive = yes
7                    then Delete((child, op), P_Q)
8                         Insert((child, e_p), P_Q)
9                         Insert((e_p, e_c), P_Q)
10                        Insert((e_c, op), P_Q)
11 return F_Q
```

Figure 7: The partitioning algorithm.

$[(child,\ e_p),\ (e_c,\ op)]$ where e_p and e_c denote, respectively, the producer and consumer parts of an EXCHANGE operator. The edge selection criteria are semantic, in the case of location- or attribute-sensitive operators in which correctness criteria constrain placement, and pragmatic in the case of an operator whose output size is larger than that of its child(ren) in which case placement seeks to reduce overall network traffic. Let Size estimate the size of the output of an operator or fragment, or the total output size of a collection of operator or fragment siblings. The algorithm which computes F_Q is shown in Fig. 7. Fig. 9 depicts the distributed-algebraic form (the output of where-scheduling, described in Section 4.2.3) given the routing tree in Fig. 6 for the physical-algebraic form in Fig. 4. The EXCHANGE operators that define the four fragments shown in Fig. 9 are placed by this step. The fragment identifier `Fn` denotes the fragment number. The assigned set of sites for each fragment (below the fragment identifier) are determined subsequently in where-scheduling. EXCHANGE has been inserted between each ACQUIRE and the JOIN, because the predicate of the latter involves tuples from different sites, and therefore data redistribution is required. Note also that an EXCHANGE has been inserted below the DELIVER, because the latter is (as is ACQUIRE) *location sensitive*, i.e., there is no leeway as to where it may be placed.

Relationship to DQP This step differs slightly from its counterpart in DQP. EXCHANGE operators are inserted more liberally at edges where a reduction in data flow will occur to favor radio transmissions to take place along such edges whenever possible (Note that we use cost models to determine the worst-case upper bounds of operator output sizes).

TinyDB Unlike SNEEql and DQP, TinyDB does not par-

tition its query plans into fragments. Rather, the entire query plan is shipped to sites which are required to participate in it, even if they are just relaying data. As discussed in Section 4.3, this is potentially wasteful of scarce memory resources.

4.2.3 Where-Scheduling

Step 6 in Fig. 2 decides which QEP fragments are to run on which routing tree nodes. This results in the distributed-algebraic form of the query. Creation and placement of fragment instances is mostly determined by semantic constraints that arise from location sensitivity (in the case of ACQUIRE and DELIVER operators) and attribute sensitivity (in the case JOIN and aggregation operators, where tuples in the same logical extent may be traveling through different sites in the routing tree). Provided that location and attribute sensitivity are respected, the approach aims to assign fragment instances to sites, where a reduction in result size is predicted (so as to be economical with respect to the radio traffic generated).

Let G, P_Q and F_Q be as above. Let $R_Q = \textsc{Routing}(P_Q, G)$ be the routing tree computed for Q. The where-scheduling algorithm computes D_Q, i.e., the graph-representation of the query in distributed-algebraic form, by deciding on the creation and assignment of fragment instances in F_Q to sites in the routing tree R_Q. If the size of the output of a fragment is smaller than that of its child(ren) then it is assigned to the deepest possible site(s) (i.e., the one with the longest path to the root) in R_Q, otherwise it is assigned to the shallowest site for which there is available memory, ideally the root. The aim is to reduce radio traffic (by postponing the need to transmit the result with increased size). Semantic criteria dictate that if a fragment contains a location-sensitive operator, then instances of it are created and assigned to each corresponding site (i.e., one that acts as source or destination in F_Q). Semantic criteria also dictate that if a fragment contains an attribute-sensitive operator, then an instance of it is created and assigned to what we refer to as a confluence site for the operator.

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

```
FRAGMENT-INSTANCE-ASSIGNMENT(F_Q, R_Q, Size)
 1  D_Q ← F_Q
 2  while   ▷ post-order traversing D_Q
            ▷ let f denote the current fragment
 3        do if op ∈ f and op.LocationSensitive = yes
 4              then for each s ∈ op.Sites
 5                        do Assign(f.New, s, D_Q)
 6           elseif op ∈ f and op.AttributeSensitive = yes
 7              and Size(f) < Size(f.Children)
 8              then while     ▷ post-order traversing R_Q,
                               ▷ let s denote the current site
 9                        do if s Δ op
10                              then Assign(f.New, s, D_Q)
11           elseif Size(f) < Size(f.Children)
12              then for each c ∈ f.Children
13                        do for each s ∈ c.Sites
14                              do Assign(f.New, s, D_Q)
15              else  Assign(f.New, R_Q.Root, D_Q)
16  return D_Q
```

Figure 8: The where-scheduling algorithm.

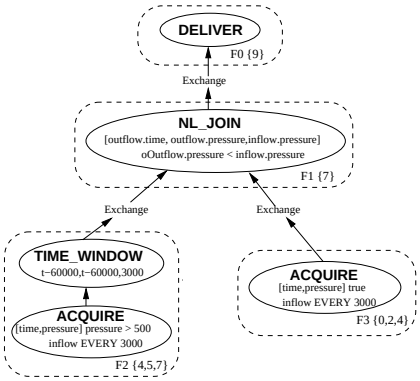

Figure 9: Distributed-Algebraic Form of Fig. 4

Let P_Q and R_Q be as above. Let s Δ op be true iff s is the deepest confluence site for op. The algorithm that computes D_Q is shown in Fig. 8.

As shown in Fig. 9, fragments that are not location-sensitive (only F1 in this case) are placed according to their expected output sizes, to reduce communication. Note also the absence of site 3 in Fig. 9 wrt. Fig. 6. This is because site 3 is only a relay node in the routing tree.

Relationship to DQP Compared to DQP, here the allocation of fragments is constrained by the routing tree, and operator confluence constraints, which enables the optimizer to make well-informed decisions (based on network topology) about where to carry out work. In classical DQP, the optimizer does not have to consider the network topology, as this is abstracted away by the network protocols. As such, the corresponding focus of where-scheduling in DQP tends to be on finding sites with adequate resources (e.g., memory and bandwidth) available to provide the best response time (e.g., [24]).

Related Work Our approach differs from that of TinyDB,

since its QEP is never fragmented. In TinyDB, a node in the routing tree either (i) evaluates the QEP, if the site has data sources applicable to the query, or (ii) restricts itself to relaying results to its parent from any child nodes that are evaluating the QEP. Our approach allows different, more specific workloads to be placed in different nodes. For example, unlike TinyDB, it is possible to compare results from different sites in a single query, as in Fig. 9. Furthermore, it is also possible to schedule different parts of the QEP to different sites on the basis of the resources (memory, energy or processing time) available at each site. The SNEEql optimizer, therefore, responds to resource heterogeneity in the fabric. TinyDB responds to excessive workload by shedding tuples, replicating the strategy of stream processors (e.g., [4]). However, in SNs, since the query processor has control over data acquisition, and furthermore, there is a high cost associated with acquiring a sensor reading, load shedding is an undesirable option. It seems more appropriate to tailor the optimization process so as to select plans that do not generate excess tuples in the first place.

4.2.4 When-Scheduling

Step 7 in Fig. 2 stipulates execution times for each fragment. Doing so efficiently is seldom a specific optimization goal in classical DQP. However, in SNs, the need to coordinate transmission and reception and to abide by severe energy constraints make it important to favor duty cycles in which the hardware spends most of its time in energy-saving states. The approach adopted by the SNEEql compiler/optimizer to decide on the timed execution of each fragment instance at each site is to build an agenda that, insofar as permitted by the memory available at the site, and given the acquisition rate α and the delivery time δ set for the query, buffers as many results as possible before transmitting. The aim is to be economical with respect to both the time in which a site needs to be active and the amount of radio traffic that is generated.

The agenda is built by an iterative process of adjustment. Given the memory available at, and the memory requirements of the fragment instances assigned to, each site, a candidate buffering factor β is computed for each site. This candidate β is used, along with the acquisition rate α, to compute a candidate agenda. If the makespan of the candidate agenda exceeds the smallest of the delivery time δ and the product of α and β, the buffering factor is adjusted downwards and a new candidate agenda is computed. The process stops when the makespan meets the above criteria. Let Memory, and Time, be, respectively, a model to estimate the memory required by, and the execution time of, an operator or fragment. The algorithm that computes the agenda is shown in Fig. 10.

The agenda can be conceptualized as a matrix, in which the rows, identified by a relative time point, denote concurrent tasks in the sites which identify the columns. For Fig. 9, the computed agenda is shown in Fig. 11, where α = 3000ms, $\beta = 2$ and δ = 5000ms. Thus, a non-empty cell (t, s) with value a, denotes that task a starts at time t in site s. In an agenda, there is a column for each site and a row for each time when some task is started. Thus, if cell $(t, s) = a$, then at time t in site s, task a is started. A task is either the evaluation of a fragment (which subsumes sensing), denoted by `Fn` in Fig. 11, where `n` is the fragment number, or a communication event, denoted by *tx n* or *rx n*, i.e., respectively, tuple transmission to, or tuple reception from, site *n*. Note that leaf fragments `F2` and `F3` are annotated with a subscript, as they are evaluated β times in each agenda evaluation. Blank cells denote the lack of a task to be performed at that time for the site, in which case, an OS-level power management component is delegated the task of deciding whether to put the node into an energy-saving state.

In SNEEql (unlike TinyDB), tuples from more than one evaluation time can be transmitted in a single communication burst, thus enabling the radio to be switched on for less time, and also saving the energy required to power it up and down. This requires tuples between between evaluations to be buffered, and results in an increase in the time-to-delivery. Therefore, the buffering factor is constrained by both the available memory and by user expectations as to the delivery time. Note that, query evaluation being continuous, the agenda repeats. The period with which it does so is $p = \alpha\beta$, i.e., $p = 3000 * 2 = 6000$ for the example query. Thus, the acquisition rate α dictates when an ACQUIRE exe-

```
WHEN-SCHEDULING(D_Q, R_Q, α, δ, Memory, Time)
 1  while   ▷ pre-order traversing R_Q,
         ▷ let s denote the current site
 2      do reqMem_e ← reqMem_f ← 0
 3         for each f ∈ s.AssignedFragments
 4             do x ← Memory(f.EXCHANGE)
 5                reqMem_f ← + Memory(f) - x
 6                reqMem_e ← + x
 7         β*[s] ← ⌊ (s.AvailableMemory − reqMem_f) / reqMem_e ⌋
 8  β ← min(β*)
 9  incr(β)      ▷ to set up adjustment through repeat ... until
10  repeat
11         decr(β)
12         agenda ← BUILD-AGENDA(D_Q, R_Q, α, β, Time)
13    until agenda.Makespan ≤ min(α * β, δ)
14  return agenda

BUILD-AGENDA(D_Q, R_Q, α, β, Time)
    ▷ schedule leaf fragments first
 1  for i ← 1 to β
 2      do for each s ∈ R_Q.Sites
 3             do nextSlot[s] ← α * (i − 1)
 4         while   ▷ post-order traversing D_Q
                   ▷ let f denote the current fragment
 5             do if f.IsLeaf = yes
 6                   then s.f.ActAt ← [ ]
 7                        for each s ∈ f.Sites
 8                            do s.f.ActAt.Append nextSlot[s]
 9                               nextSlot[s] ← + Time(s.f)
    ▷ schedule non-leaf fragments next
10  while   ▷ pre-order traversing R_Q,
         ▷ let s denote the current site
11      do while   ▷ post-order traversing D_Q
                   ▷ let f denote the current fragment
12             do if f ∈ s.AssignedFragments
13                   then f.ActAt ← nextSlot[s]
14                        nextSlot[s] ← + Time(f)*β
         ▷ schedule comms between fragments
15         s.TX.ActAt ← max(nextSlot[s],nextSlot[s.Parent])
16         s.Parent.RX(s).ActAt ← s.TX.ActAt
17         nextSlot[s] ← + Time(s.TX)
18         nextSlot[s.Parent]) ← + s.Parent.RX
19  return agenda
```

Figure 10: The when-scheduling algorithm.

Time (ms)	Sites							
	0	5	6	2	4	3	7	9
0	$F3_1$	$F2_1$		$F3_1$	$F2_1$		$F2_1$	
63					$F3_1$			
3000	$F3_2$	$F2_2$		$F3_2$	$F2_2$		$F2_2$	
3032				*tx3*		*rx2*		
3063	*tx5*	*rx0*			$F3_2$			
3125					*tx3*	*rx4*		
3157		*tx6*	*rx5*					
3313			*tx7*				*rx6*	
3469						*tx7*	*rx3*	
3688							F1	
3719							*tx9*	*rx7*
4313								F0

Figure 11: Agenda for the Query Plan in Fig. 9

cutes; α and the buffering factor β dictate when a DELIVER executes.

Relationship to DQP The time-sensitive nature of data acquisition in SNs, the delivery time requirements which may be expressed by the user, the need for wireless communications to be co-ordinated and for sensor nodes to duty-cycle, all make the timing of tasks an important concern in the case of SNs. Clearly, in DQP this is not an issue, as these decisions can be delegated to the OS and network layers.

Related Work In TinyDB, cost models are used to determine an acquisition rate to meet a user-specified lifetime. The schedule of work for each site is then determined by its level in the routing tree and the acquisition rate, and tuples are transmitted downstream following every acquisition without any buffering. In contrast, our approach allows the optimizer to determine an appropriate level of buffering, given the delivery time constraints specified by the user, which results in significant energy savings as described in Section 5. Note that this differs from the orthogonal approach proposed in [22], which achieves energy savings by not sending a tuple if an attribute is within a given threshold with respect to the previous tuple. It would not be difficult to incorporate such a technique into the SNEEql optimizer for greater energy savings. In [29] a subset of the when-scheduling problem is addressed; an algebraic approach to generating schedules with a focus on maximizing the number of non-interfering, concurrent communications is proposed. It is functionally similar to the proposed BUILD-AGENDA algorithm, although it only considers the scheduling of communications, and not computations as we do.

4.3 Phase 3: Code Generation

Step 8 in Fig. 2 generates executable code for each site based on the distributed QEP, the routing tree and the agenda. The current implementation of SNEEql generates nesC [9] code for execution in TinyOS [12], a component-based, event-driven runtime environment designed for wireless SNs. nesC is a C-based language for writing programs over a library of TinyOS components.

Physical operators, such as those described in this and the previous section, are implemented as nesC template components. The code generator uses these component templates to translate the task-performing obligations in a site into nesC code that embodies the computing and communication activity depicted in abstract form by diagrams like the one in Fig. 12. The figure describes the activity in site 7, where the join (as well as sensing) is performed. In the figure, arrows denote component interaction, the black-circle end denoting the initiator of the interaction. The following kinds of components are represented in the figure: (i) square-cornered boxes denote software abstractions of hardware components, such as the sensor board and the radio; (ii) dashed, round-cornered boxes denote components that carry out agenda tasks in response to a clock event, such as a communication event or the evaluation of a QEP fragment; (iii) ovals denote operators which comprise fragments; note the the correspondence with Fig. 9 (recall that an EXCHANGE operator is typically implemented in two parts, referred to as producer and consumer, with the former communicating with the upstream fragment, and the latter, the downstream one); and (iv) shaded, round-cornered boxes denote (passive) buffers onto which tuples are written/pushed and from which tuples are read/pulled by other components.

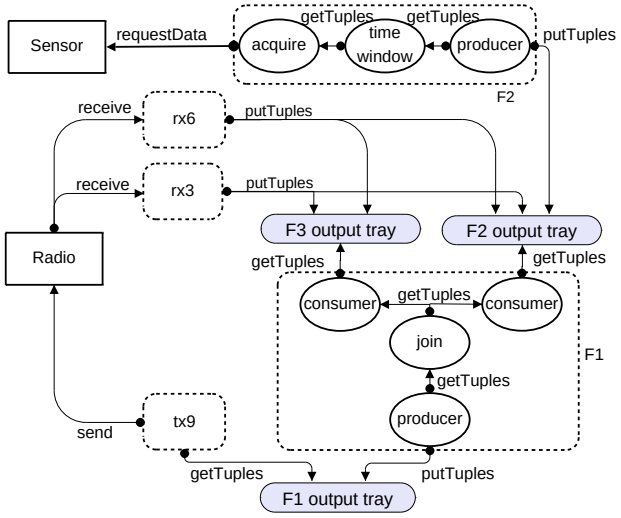

Figure 12: Generated Code for Site 7 in Fig. 11

Fig. 12 corresponds to the site 7 column in the agenda in Fig. 11 as follows. Firstly, the acquisitional fragment F2 executes twice and places the sensed tuples in the F2 output tray. Subsequently, tuples are received from sites 6 and 3, and are placed in the F2 output tray and F3 output tray accordingly. Inside fragment F1, an exchange consumer gets tuples from F2 and another one gets tuples from F3 for the NL_JOIN. The results are fetched by a producer that writes them to the F1 output tray. Finally, tx9 transmits the tuples to site 9.

Relationship to DQP and Related Work In classical DQP, typically each node has an interpreter which evaluates the query plan fragment(s) assigned to it. Similarly, the approach taken by TinyDB is for each node to interpret the parsed query data structure, and to conditionally execute parts of the query which are relevant to it. On a MICA mote (see below), TinyDB requires 65K program memory (PM) for the query evaluator, including code for all the operators and the supporting TinyOS libraries, and 2.8K RAM. SwissQM [21], a SN virtual machine specialized for data processing (which is query language independent, so this step could alternatively target it) occupies 33K PM for the interpreter and instruction-set code and 3K RAM. In contrast, our approach generates nesC code (and is therefore at a lower level of abstraction) with only the specific

	β	Sites							
		0	5	6	2	4	3	7	9
PM	1	13.8	15.3	14.2	13.8	15.7	14.3	17.0	12.0
RAM	1	0.5	0.6	0.5	0.5	0.6	0.5	0.8	0.5
PM	10	15.1	16.7	15.3	15.1	17.4	15.4	18.5	13.0
RAM	10	0.8	1.0	0.9	0.8	1.0	0.9	2.2	1.5

Figure 13: Program Memory and RAM, in K, used by sites in the example query plan in Fig. 9 for varying β

functionality required at each site, allowing it to be more economical. Fig. 13 shows how, as the buffering factor β increases, both PM and RAM consumption of *Query 3* rise. The larger PM is due to the increased number of rows in the agenda as more acquisition tasks are added, resulting in a larger nesC implementation of the query plan. RAM consumption increases as more space is allocated for the trays which accommodate the buffering of tuples between query evaluations. Note that site 7, which has the largest footprint (as it executes the join), only requires 18.5K PM and 2.2K of RAM (with $\beta = 10$), which is considerably less than TinyDB and SwissQM. This allows significant scope for other code to be executed at the same time by the SN using the remaining PM and RAM.

5. EXPERIMENTAL EVALUATION

The goal of this section is to present experimental evidence we have collected in support of our overall research hypothesis, viz., that the extensions (described in Section 4) to DQP techniques that are proven in the case of robust networks lead to effective and efficient DQP over SNs. The approach that emerges from such extensions offers many more opportunities for co-ordinated optimization strategies than previous work on SN query processing, which often has focused on trying to obtain a single goal by addressing a specific type of optimization decision.

Experimental Design Our experimental design centered around the aim of collecting evidence about the performance of our approach for a range of query constructs and across a range of quality of service (QoS) expectations. The QoS expectations we used were acquisition interval and delivery time. The evidence takes the form of measurements of the following performance indicators: network longevity, average delivery time, and total energy consumption. Four experiments were analytical, i.e., aimed at collecting evidence as to the performance of the code produced by the SNEEql compiler/optimizer. One experiment was comparative, i.e., aimed at collecting evidence as to the performance of the code produced by the SNEEql compiler/optimizer relative to that of TinyDB. The SNEEql queries used are shown in Fig. 1 (denoted Q1, Q2 and Q3 in the graphs).

Experimental Set-Up Experiments 1–4 were run using the Avrora simulator [25], which simulates the behavior of SN programs at the machine code level with cycle-accuracy, and is able to provide detailed energy consumption information for each hardware component. Experiment 5 was run using Tossim [15], an interrupt-level discrete event simulator for TinyOS networks. Note that, for this experiment, radio traffic was counted, as we were unable to get reliable energy consumption measurements using Avrora. Both simulators ran executables compiled from TinyOS/nesC 1.1. Being closed-world simulators, the environment has no impact on the results obtained. Our experiments ran for 600 simulated seconds. The SN simulated in the experiments was the one described in Fig. 3. The sensor nodes we have simulated were [Type = Mica2, CPU = 8-bit 7.3728MHz AVR, RAM = 4K, PM = 128K, Radio = CC1000, Energy Stock = 31320 J (2 Lithium AA batteries)].

Experiment 1: *Impact of acquisition interval on total energy consumption.* SNs often monitor remote or inaccessible areas (e.g., [20]), and a significant part of the total cost of ownership of a SN deployment is influenced by the energy stock required to keep it running. Results are reported in Fig. 14. The following can be observed: (i) As the acquisition rate α is increased, the total energy consumption decreases, as the query plan is acquiring, processing and transmitting less data overall, and sleeping for a longer proportion of the query evaluation process. (ii) A point is reached when increasing the acquisition rate leads to a marginally lower energy saving, due to the constant overhead incurred by having most of the components in a low-power state. The overhead is higher for the default Mica2 sensor board which is simulated, as it does not have a low power state, and is therefore always on. (iii) The radio energy consumption shrinks disproportionately compared to the CPU, because in relative terms, the energy saving when in a low power state is much greater for the radio than it is for the CPU. (iv) *Query 2* has the lowest energy consumption because in the aggregation, tuples from 3 source sites are reduced to a single tuple; in contrast, *Query 3* joins tuples from the `inflow` and `outflow` extents which comprise 3 tuples in each acquisition, and therefore consumes the greatest amount of energy.

Experiment 2: *Impact of acquisition interval on network longevity.* The lifetime of a SN deployment is a vital metric as it indicates how often the SN energy stock will need to be replenished. Note that, here, network longevity is assumed to be the time it takes for the first node in the routing tree to fail, so this is, in a sense, a shortest-time-to-failure metric. Fig. 15 reports the results obtained. It can be observed that as acquisition interval increases, energy savings accrue, and hence network longevity increases, albeit with diminished returns, for the same reasons as in *Experiment 1.*

Experiment 3: *Impact of buffering factor on delivery time.* This experiment studies the increase in delivery time as more buffering is performed by the EXCHANGE operators in a query plan. Results are reported in Fig. 16. It is observed that: (i) At first, delivery time increases in a linear fashion with respect to the buffering factor β; and (ii) Delivery time plateaus when the available memory precludes any further increase in β. Note that this happens earlier for queries in which a greater amount of data is generated (e.g., *Query 3* compared to *Query 1*). In this and the remaining experiments, the acquisition interval was set to 3 seconds.

Experiment 4: *Impact of delivery time on network longevity.* For some applications, network longevity is of paramount importance and a delay in receiving the data may be tolerated (e.g., [20]), whereas in other applications it may be more important to receive the data quickly with less regard to preserving energy (e.g., [23]). Results which report the relationship between delivery time and longevity are reported in Fig. 17. It can be observed that: (i) The optimizer reacts to an increase in tolerable delivery time δ by increasing β, which in turn leads to an increase in network longevity. In other words, the optimizer empowers users, by enabling

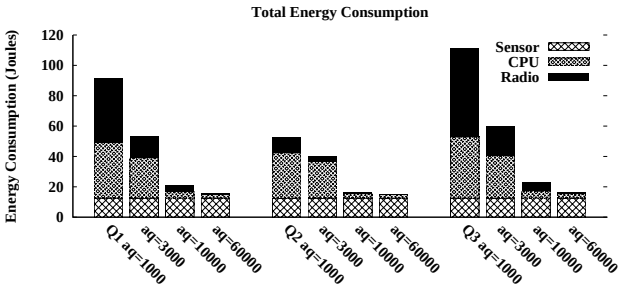

Figure 14: Energy consumption vs. α

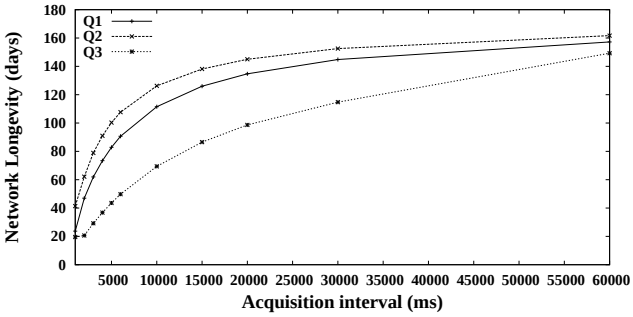

Figure 15: Network Longevity vs. α

them to trade-off delivery time for greater network longevity; and (ii) Inflection points occur when increasing the buffering factor does not reduce energy consumption. This is because the when-scheduling algorithm assumes that increasing the buffering factor is always beneficial; this is however not always the case. A higher buffering factor may lead to a number of tuples being transmitted at a time that cannot be packed efficiently into a message, leading to padding in the message payload. As an example, consider the case where a single source node is being queried, $\beta = 4$, the number of tuples/message = 3. Packets will be formed of 3 tuples and then 1 tuple, which is inefficient. As a result, more messages need to be sent overall, leading to a higher energy consumption, and hence, a decreased network lifetime. This demonstrates that in order to ascertain an optimal buffering factor, minimizing the padding of messages is also an important consideration. However, we note that maximizing the buffering does lead to an overall improvement in network longevity.

Experiment 5: *Contrasting radio traffic with TinyDB.* This experiment compares the number of query result messages sent by TinyDB to SNEEql with different buffering factors. Results are reported in Fig. 18. Note that *Query 3* cannot be expressed in TinyDB without using materialization. It can be observed that: (i) For $\beta = 1$, the number of messages sent in SNEEql and TinyDB are similar. It is slightly less for SNEEql, which is probably due to the formation of the routing tree being informed by the location of data sources, resulting in a shorter distance being traveled by tuples. (ii) At $\beta = 2$, SNEEql sends significantly less messages. As demonstrated by Experiment 4, radio communication is energy intensive and therefore a key indicator of SN lifetime. As as result, as β increases, the network lifetime in SNEEql will become increasingly greater than for TinyDB. The increase in β leads to greater delivery times, but such trade-offs are not supported by TinyDB.

To summarize the findings of the experimental evaluation, we can now ascertain that (i) the SNEEql optimizer exhibits desirable behaviors for a broad range of different scenarios; (ii) SNEEql performs favorably against TinyDB, as workloads can be scheduled to specific sites in the SN, and furthermore, buffering allows more efficient use of energy resources; (iii) SNEEql caters for more expressive queries and allows different qualities of service (e.g., delivery time and lifetime) to be traded-off. This demonstrates that SNEEql delivers quantifiable benefits vis-à-vis the seminal contribution in the SN query processing area.

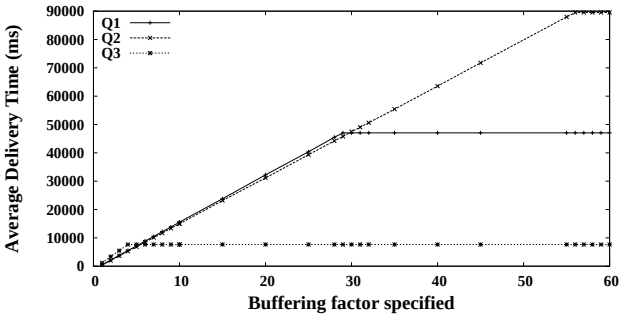

Figure 16: Avg. Deliv. Time vs. β

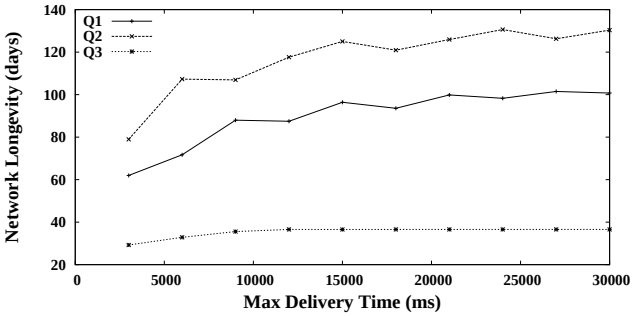

Figure 17: Network Longevity vs. δ

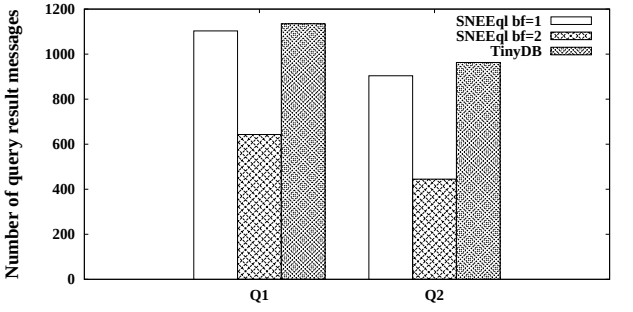

Figure 18: Radio Traffic: SNEEql vs. TinyDB

6. CONCLUSIONS

In this paper we have described SNEEql, a SN query optimizer based on the two-phase optimization architecture prevalent in DQP. In light of the differences between SNs

and robust networks, we have highlighted the additional decision-making steps which are required, and the different criteria that need to be applied to inform decisions. We have demonstrated that, unlike TinyDB which performs very limited optimization, the staged decision-making approach in SNEEql offers several benefits, including: (1) The ability to pose queries using a rich, expressive language based on classical stream query languages. This is beneficial as environments in which there are less expressive languages are likely to incur greater in-network processing costs, conveying data to offline nodes for subsequent analysis. (2) The ability to schedule different workloads to different sites in the network, enabling more economical use of resources such as memory, and potentially support for heterogeneity in the SN. (3) The ability to empower the user to trade-off conflicting qualities of service such as network longevity and delivery time.

The effectiveness of the SNEEql approach of extending a DQP optimizer has been demonstrated through an empirical evaluation, in which the performance of query execution is observed to be well behaved under a range of circumstances, and compares well with TinyDB, the seminal first-generation SN database. It can therefore be concluded that much can be learned from DQP optimizer architectures in the design of SN optimizer architectures.

7. ACKNOWLEDGEMENTS

This work is part of the DIAS-MC project funded by the UK EPSRC WINES programme under Grant EP/C014774/1. We are grateful for this support and for the insight gained from discussions with our collaborators in the project. C.Y.A. Brenninkmeijer thanks the School of Computer Science, and F. Jabeen, the government of Pakistan, for their support.

8. REFERENCES

[1] A. Arasu, B. Babcock, et al. STREAM: The Stanford Stream Data Manager. *IEEE Data Eng. Bull.*, 26(1):19–26, 2003.

[2] A. Arasu, S. Babu, et al. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[3] B. J. Bonfils and P. Bonnet. Adaptive and Decentralized Operator Placement for In-Network Query Processing. In *IPSN 2003*, pages 47–62.

[4] D. Carney, U. Çetintemel, et al. Monitoring Streams - A New Class of Data Management Applications. In *VLDB 2002*, pages 215–226.

[5] A. Deshpande, C. Guestrin, S. Madden, et al. Model-based approximate querying in sensor networks. *VLDB J.*, 14(4):417–443, 2005.

[6] D. Estrin, D. Culler, et al. Connecting the Physical World with Pervasive Networks. *IEEE Pervasive Computing*, pages 59–69, January-March 2002.

[7] C. Frank and K. Römer. Algorithms for Generic Role Assignment in Wireless Sensor Networks. In *SenSys 2005*, pages 230–242.

[8] D. Ganesan, G. Mathur, et al. Rethinking Data Management for Storage-centric Sensor Networks. In *CIDR 2007*, pages 22–31.

[9] D. Gay, P. Levis, et al. The nesC language: A holistic approach to networked embedded systems. In *PLDI 2003*, pages 1–11.

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

[11] G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD 1990*, pages 102–111.

[12] J. Hill, R. Szewczyk, et al. System Architecture Directions for Networked Sensors. In *ASPLOS 2000*, pages 93–104.

[13] H. Karl and A. Willig. *Protocols and Architectures for Wireless Sensor Networks*. John Wiley and Sons, June 2005.

[14] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[15] P. Levis, N. Lee, et al. TOSSIM: accurate and scalable simulation of entire tinyOS applications. In *SenSys 2003*, pages 126–137.

[16] P. Levis, S. Madden, et al. The Emergence of Networking Abstractions and Techniques in TinyOS. In *NSDI 2004*, pages 1–14.

[17] S. Madden, M. J. Franklin, et al. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI 2002*.

[18] S. Madden, M. J. Franklin, et al. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[19] A. Manjhi, S. Nath, et al. Tributaries and Deltas: Efficient and Robust Aggregation in Sensor Network Streams. In *SIGMOD 2005*, pages 287–298.

[20] I. W. Marshall, M. C. Price, et al. Multi-sensor Cross Correlation for Alarm Generation in a Deployed Sensor Network. In *EuroSSC 2007*, pages 286–299.

[21] R. Müller, G. Alonso, et al. SwissQM: Next Generation Data Processing in Sensor Networks. In *CIDR 2007*, pages 1–9.

[22] M. A. Sharaf, J. Beaver, et al. TiNA: A Scheme for Temporal Coherency-Aware In-Network Aggregation. In *MobiDE 2003*, pages 69–76.

[23] I. Stoianov, L. Nachman, et al. PIPENET: a wireless sensor network for pipeline monitoring. In *IPSN 2007*, pages 264–273.

[24] M. Stonebraker, P. M. Aoki, et al. Mariposa: A Wide-Area Distributed Database System. *VLDB J.*, 5(1):48–63, 1996.

[25] B. Titzer, D. K. Lee, et al. Avrora: scalable sensor network simulation with precise timing. In *IPSN 2005*, pages 477–482.

[26] N. Trigoni, Y. Yao, et al. Multi-query Optimization for Sensor Networks. In *DCOSS 2005*, pages 307–321.

[27] N. Trigoni, Y. Yao, et al. WaveScheduling: Energy-Efficient Data Dissemination for Sensor Networks. In *DMSN 2004*, pages 48–57. ACM Press.

[28] Y. Yao and J. Gehrke. Query Processing in Sensor Networks. In *CIDR 2003*.

[29] V. I. Zadorozhny, P. K. Chrysanthis, et al. A Framework for Extending the Synergy between Query Optimization and MAC Layer in Sensor Networks. In *DMSN 2004*, pages 68–77. ACM Press.