Efficient Sharing of Sensor Networks

René Müller and Gustavo Alonso
Department of Computer Science
ETH Zurich, 8092 Zurich, Switzerland
Email: {muellren,alonso}@inf.ethz.ch

***Abstract*— In this paper we tackle the problem of allowing applications to request different data at different rates from different sensors of the same sensor network while still being able to run the sensor network in an efficient manner. Our approach is to merge an arbitrary number of user queries into a network query. By doing this, traffic is minimised and the sensors have better energy consumption behavior than if all user queries would have been directly sent to the network. In the paper we describe the algorithms for the transformation of queries and the resulting data streams. We also provide an extensive performance evaluation of the algorithms using sets of over hundred overlapping user queries executing on the same sensor network.**

I. Introduction

Sensor networks are starting to be more widely used [1]–[4]. In spite of these early successes, there is still a need for better tools to program sensor networks. In this paper we tackle the problem of supporting multiple applications over a single sensor network. We do this in the context of query based data acquisition systems [5] and use TinyDB [6] as the software running on the sensors. By doing this, we transform the problem of multi-user support into a multi-query optimisation problem and bring some of the tools of classical database query optimisation to bear on the problem. The approach is conceptually similar to that followed in projects like P2 [7] which map network protocols to a datalog based specification for both being able to express complex protocols in a compact manner and to exploit the advantages of a declarative specification. In the case of sensor networks, we show that such an approach not only allows declarative and compact specifications of user requests but that it also leads to a natural way of implementing efficient multi-user support.

A. Sharing a sensor network

The challenge behind multi-user support lies in resolving the trade-off between efficient operation of the network (reduced traffic, sparse duty cycles at the sensors, avoiding redundancy in measurements and messages) and the number of independent user requests for data that need to be supported (each one interested in a potentially different set of sensors and acquisition rate). Our system is based on two different query classes and a translation step between them: *User Queries* (UQ) and *Network Queries* (NQ). User queries are those submitted by users. These are then merged into a network query which is then sent to the sensor network for execution. The main idea behind our solution arises from the following observation. There is always a limit to the maximum amount of data that can be obtained from a sensor network: sample at the highest possible frequency and capture data from all the sensors of every node. We refer to a such request as the *universal network query*. This query would theoretically produce all the data ever needed to answer any user query.

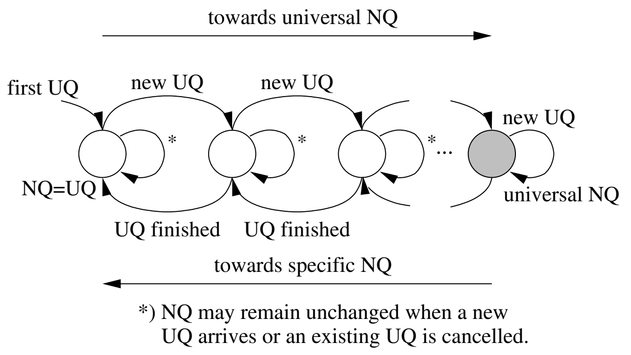


Fig. 1. Moving between specific network queries and "universal" network queries as user queries are started and cancelled.

The universal network query is as follows:

```
SELECT * FROM sensors
SAMPLE PERIOD min-sampling-period
```

This query has the lowest possible *selectivity* since it returns all available data. In practice, however, it has been shown that sensor networks exhibit low reliability—especially when under heavy load. It is thus unlikely that the universal network query will return all the data it requests [8], [9]. It is also not practical to run the sensor network at such a high rate without knowing whether the data will be used at all. Nevertheless, the intuitive notion of a universal query seems to indicate that it should be possible to merge a set of user queries into a single network query that will produce the data needed to answer all of the outstanding user queries.

In order to reduce the amount of data that flows through the network, the goal is to find the most selective network query that allows to answer all outstanding user queries. Thus, as new user queries arrive, the resulting network query is progressively expanded (it is made less selective) according to the new user queries. Ultimately the system could end up expanding the network query to be the universal network query. Conversely, when user queries are withdrawn, the system must in turn increase the selectivity of the network query so that it captures only the necessary data. This process of reducing selectivity (or moving toward the universal query as new user queries arrive) and increasing selectivity as user queries are removed is shown in Fig. 1. Performing such a process dynamically and in an efficient manner is the main challenge in the system we propose.

After the merging step, the resulting network query is sent to the sensor network which, in turn, starts to produce data. Extracting the user data streams from the network query data stream is the second part of the problem of sharing a sensor network. The challenge here arises from the need to down-sample the network data stream. If not done carefully, the user data stream might miss data and produce results at a different sampling period than actually requested. The way caching is implemented as well as how and when data is forwarded to the user play a big role in defining a correct solution.

B. Related Work

Our proposal complements the work of Jeffery et al. [8], [9], who have recently proposed a pipeline of processing stages for cleaning data from sensor networks. This pipeline could be used in our system to clean the data obtained from a network query to separate the data needed for each user query. Other methods [10], [11] that perform error correction and cleaning of data streams are based on a data model. These algorithms could also be implemented in the query mapping layer of our system. In [12], the problem of multiple query optimisation is also addressed but only for queries that use the same tuple rate. This is rather limiting as it does not allow applications to use any sampling period they may deem fit. It also simplifies the problem considerably as we will show in the paper. The processing is also done in batches (by groups of queries at a time) and not progressively as in the system we describe in this paper. This prevents new users from joining the system whenever they want and long-running queries have to be reissued multiple times.

C. Contributions

The paper describes a novel system for sharing a sensor network across multiple users. In doing so, several important problems are also addressed:

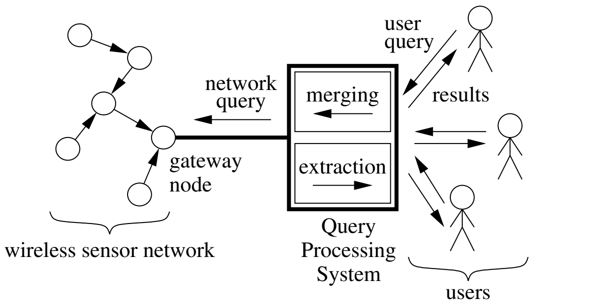


Fig. 2. System Overview: wireless sensor network and query processing system

- Algorithms are provided to merge multiple queries into a single network query. These algorithms support the dynamic addition and withdrawal of queries.
- Techniques are provided to extract user data streams from the network data stream with a correct sampling period. This is not trivial since the system must maintain a constant rate for the different users while the number of user queries and network conditions dynamically change.
- The experimental results show that our system is capable of running over hundred different user queries in a single sensor network. We are not aware of any platform for sensor networks capable of doing this.
- Although the paper works with queries both for reason of clarity and implementation, the ideas presented are generic and can also be applied to sensor networks that have a different interface for data acquisition.

II. Multi-query support

A. Working Environment

Our system is built atop TinyDB, a query-based wireless sensor network platform where the network is configured as a tree. The root node of the tree, called *gateway node*, receives all the data forwarded from the leaves and intermediate nodes. The gateway node is a dedicated node that is connected to the query processing system. The gateway node is used to send queries into the network and gather result data which is then forwarded to our system. The core of our solution is a *query processing system* that sits between the gateway node and the users and acts as intermediary between the two (Fig. 2). The query processing system is implemented in Java and can run on a portable device such as a PDA. Each sensor node (also called mote) in the network runs TinyDB. Users can request data from the sensors by posing *User Queries* (UQ) to the query processing system which then merges these queries into a *Network Query* (NQ). The network query is then sent to the gateway node which broadcasts it into the wireless sensor network. User queries as well as network queries have the following form:

```
SELECT select-list
[FROM sensors]
[WHERE where-expression]
SAMPLE PERIOD sampling-period
```

This syntax is similar to SQL although with a restricted set of expressions. The *select-list* indicates the values to be measured (e.g., light, temperature, etc.), the *sensors* are the nodes to be used (in this paper we just assume the query refers to the entire network as TinyDB currently does not support individual node addressing), the *where-expression* is a Boolean predicate (e.g., temperature $\leq$ 25), and the *sample period* indicates how often data has to be produced by the sensor network. For each sampling interval every node reads its sensors and emits a tuple (if it is not filtered by the "where" clause predicate). The time interval between the emission of two consecutive tuples is called an *epoch*. In order to correlate samples from different epochs they are given a monotonically increasing number which is inserted as an implicit attribute into the result tuples. In its current version, TinyDB can only run two simultaneous queries, only supports boolean predicates with conjunctions, and the minimum sample period is about 1000 ms.

B. Basic Operations

The query processing system *merges* multiple user queries into a network query and *extracts* tuples from the network query to produce result tuples for all associated user queries. To illustrate how user queries are merged into a network query and how the result data is extracted, we use a simple example. Consider the user queries:

UQ_1 :
```
SELECT nodeid, light
FROM sensors SAMPLE PERIOD 5s
```
UQ_2 :
```
SELECT nodeid, light
FROM sensors SAMPLE PERIOD 15s
```
UQ_3 :
```
SELECT light FROM sensors
SAMPLE PERIOD 50s
```
UQ_4 :
```
SELECT nodeid, light, temp
FROM sensors
WHERE nodeid=1 AND temp>100
SAMPLE PERIOD 20s
```

In spite of the fact that these queries are requesting different data with different sampling periods, they can be merged into the following network query:

NQ_1 :
```
SELECT nodeid, light, temp
FROM sensors SAMPLE PERIOD 5s
```

This network query, will deliver data on *nodeid*, *light* and *temp* every 5 seconds. The query processor takes this stream of data and extracts the answer for each one of the four user queries. For UQ_1 the rate mapping is 1:1 and only attributes *nodeid* and *light* must be extracted. For UQ_2 too the *temp* data is dropped but only one out of every three data points is used to enlarge the sampling period to the requested 15 seconds. For UQ_3, the light data must be extracted from one of each 10 data points to produce light measurements every 50 seconds. UQ_4 receives data from all sensors but only those tuples are selected where *nodeid* $= 1 \wedge$ *temp* > 50. Before the selection operation, the sampling period of NQ_1 is matched to fit the one requested by the user query by selecting every fourth tuple, such that a tuple is produced once every 20 seconds.

C. Query Merging

In this section we provide a more formal background for the mapping of user queries into a network query. We will refer to the set of user queries as $U = \{u_1, u_2, \ldots, u_m\}$. The network query will be denoted as n. For all queries (user or network) f denotes a set of attributes, s the sampling period of the query and p a list of predicates associated to the query.

A first requirement to meet is that the set of attributes $n.f$ of the network query must be a superset of the attribute set $u.f$ of any user query associated with network query n. This leads to the first condition that must hold when a set U of user queries is mapped to a network query n:

$$\forall u \in U \, : \, n.f \supseteq u.f \tag{1}$$

The next condition involves the sampling periods. The sampling interval $n.s$ must evenly divide the sampling interval $u.s$ of all its user queries. This guarantees that the data stream provided by n can be used to answer all queries u associated with n. Thus;

$$\forall u \in U : \exists k_u \in \mathbb{N} \, : \, u.s = k_u \cdot n.s \ . \tag{2}$$

One way to enforce this condition is to make $n.s$ the greatest common divisor (GCD) of all user query sampling periods. Note that if the sampling intervals of two user queries are relative prime then $n.f = 1$ which is certainly undesirable. Also, in many cases, forcing an exact arithmetic match is too restrictive. Thus, instead of the above condition, we allow for a relative error in the sampling period up to some ε. Instead of using the GCD algorithm for determining the common sampling period we then use a *"Tolerant" Greatest Common Sampling period* (TGCS) such that for all user queries the effective sample period observed is within ε of what the user requested. The TGCS algorithm is described later in section III-C. Thus instead of using (2) we require:

$$\forall u \in U : \exists k_u \in \mathbb{N} \, : \, (1-\varepsilon)u.s \leq k_u \cdot n.s \leq u.s \tag{3}$$

Note that in some cases, the application submitting the user query may require the data to be delivered exactly at the specified time intervals. This can be achieved through operators that cache the result for a short period of time until it is time to send it to the user. Given the uncertainties in some of the measurements and the lack of precision of most sensor networks today, such time shifts in the measurements should be acceptable in most applications, specially if they can be constrained within a well specified error margin. Also, the formulation just provided allows to adjust below the requested period since it is easier to cope with more data than with missing data. The implementation of rate conversion of the tuple stream is described in section IV.

Selection queries—as implied by the predicate in the “where” clause—require special treatment. They are expressed in SQL with one additional restriction that the “where” clause expressions must be written in conjunctive normal form to emphasise the filtering property. Let $u.p$ be the set of predicate conjunction terms in the “where” clause of user query u, i.e., $u.p = \{nodeid = 1, temp > 100\}$. The selection can be expressed as $\sigma_{nodeid=1 \wedge temp>100}(\texttt{sensors})$ in relational algebra. In general a query can be represented as a selection over a conjunction of predicates p_i, p_j followed by a projection on a list of attributes a from the set of sensor attributes $u.f$, so for any two queries Q_i and Q_j in relational algebra:

$$\begin{aligned} Q_i: &\quad \pi_{a_i}\left(\sigma_{p_{i_1} \wedge p_{i_2} \wedge \ldots \wedge p_{i_n}}(\texttt{sensors})\right) \\ Q_j: &\quad \pi_{a_j}\left(\sigma_{p_{j_1} \wedge p_{j_2} \wedge \ldots \wedge p_{j_m}}(\texttt{sensors})\right) \end{aligned}$$

In order to allow sharing of common operations, the selection predicates of any two queries must be brought in relation. We define the relation “Q_i is at most as selective as Q_j” as $Q_i \leq Q_j$ where we use

$$Q_i \leq Q_j \;:=\; p_{j_1} \wedge p_{j_2} \wedge \ldots \wedge p_{j_m} \Rightarrow p_{i_1} \wedge p_{i_2} \wedge \ldots \wedge p_{i_n} \quad .$$

This leads to the last condition that must be met by the network query. The network query n must be “at most as selective” as any of its user queries u, i.e., $n \leq u$. Thus:

$$\forall u \in U \;:\; n \leq u \tag{4}$$

Summarising (1), (3) and (4) we obtain the following rules for mapping a set of user queries to a network query:

$$\begin{aligned} \forall u \in U \;:\; & n.f \supseteq u.f \;\wedge\; n \leq u \;\wedge \\ & \exists k_u \in \mathbb{N} \;:\; (1-\varepsilon)u.s \leq k_u \cdot n.s \leq u.s \end{aligned} \tag{5}$$

D. Data Extraction

The example given in II-B illustrates how user queries can be transformed into a network query. It also shows that once the result tuples for the network query become available, they need to be processed to produce the answers to the different user queries. This is done through a pipeline of data *operators* that is constructed when each individual user query is merged into the network query. For the purposes of this paper, the set of operators we are considering includes:

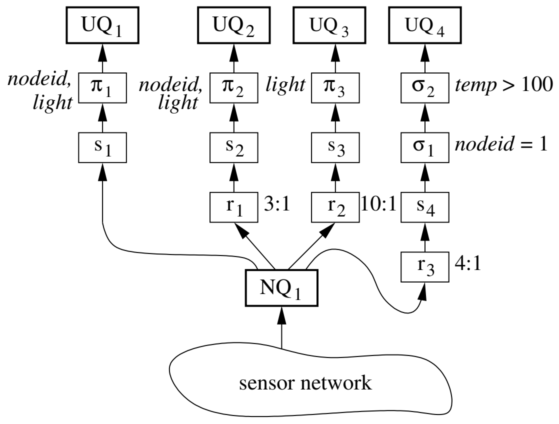


Fig. 3. Mapping of User Queries (UQ) to Network Queries (NQ)

Tuple rate conversion and down-sampling r_1, r_2, r_3: These operators take a stream of data points with a given period and produce a stream of data points with a larger sampling period (typically a multiple of the original sampling period). These operators could conceivably also be used to interpolate a series of data points to increase the sampling frequency. For this purpose, a model driven sampling strategy such as those discussed in [11] could be used.

Time and epoch shift $s_1, \ldots, s_4$: The data streams coming from the sensors have time (*epoch*) information attached to them that relate to the network query. The time and epoch shift operators set the counters to 0 when a user query starts and transforms the epoch information of the resulting data so that to the user it looks like the system just started producing data for that user query.

Attribute projection π_1, π_2, π_3: Similar to projection in relational algebra, a projection operator in our system removes unwanted attributes from the tuple stream.

Selection expressions σ_1, σ_2: Similar to the selection operator in relational algebra, selection operators apply predicates (those in the “where” clause of the user query’s SQL statement) so that the data delivered in response to a user query matches exactly what was requested. In some cases, the selection operator is not run on the query processing system but pushed down to the network query (which in turn results in less traffic since the use of predicates increases selectivity).

For the example given in II-B, the corresponding processing pipeline with all operators required to map tuples from NQ_1 to $UQ_1, \ldots, UQ_4$ is shown in Fig. 3. Arrows indicate the flow of tuples through the processing pipeline. In order to emphasise that a down-sampling operator r or a selection operator σ is associated to user query u we use the notation $u.op_r$ and $u.op_\sigma$ respectively.

III. Algorithms For Query Merging

A. Adding a new User Query

Condition 5 states when a network query can be used to answer a given user query. The challenge however is how to dynamically manage a network query as user queries arrive (later on we will discuss what to do when user queries are withdrawn). The data acquisition system we are using to access the sensor network (TinyDB) has limited support for concurrent queries. In principle it does support the processing of multiple concurrent queries but due to severe memory restrictions, in general for average complex queries (i.e., queries with a 3–4 selected attributes and 1–2 predicates) no more than two network queries can be executed in parallel. Therefore we are forced to map all user queries to a single network query, while using the second network query only during the transition phase, i.e., when a new user query is added or withdrawn, and the network query has to be replaced. We are working on an alternative to TinyDB that will not have such a limitation. Once ready, it will be possible to consider optimisations involving the relative costs of merging queries or running them separately. In this paper, however, because of the current limitations of TinyDB, assume the goal is to merge user queries into a single network query. The detailed procedure for adding a new user query is shown in algorithm 1. On line 2, if no network query is running, the system makes the user query the network query. Otherwise the system checks if the network query that currently delivers data matches the criteria (1) and (4) described above. If both conditions are satisfied the existing network query n can be used and the manager inserts selection operators for all predicates that are not part of the network query n (line 11). Then it is verified whether the sampling period $n.s$ of the network query matches the one of u, i.e., criteria (3) is checked. If the condition is also satisfied the down-sampling operator $u.op_r$ is configured accordingly and integration of the new user query is completed. Note that in this case the sensor network is left untouched.

If the sampling interval of n cannot be matched to u (condition (3) is not satisfied but conditions (4) and (1) are) the system adjusts the sampling period of the already running network query in order to accommodate the new user query. Since changing the sampling period of a running network query is less expensive in terms of messages (and, thus, energy consumption) than starting a new query, we adapt the sampling period instead of setting up a new query. The new sampling period of n is computed using a variant of the *Euclidean algorithm* (described in section III-C). The sampling period $n.s$ is set to the largest period possible. Since the period of the network query has changed, all down-sampling operators of the existing user queries have to be reconfigured (line 18). The most difficult case occurs when the existing network query cannot be used. This happens if the new user query includes attributes that are not retrieved by the current network query (condition (1) is not satisfied) or if the user query is less selective (condition 4 is not satisfied) than the network query. The system now has to replace the current network query n by a new network query n' such that conditions (1), (3) and (4) are met. The sampling period $n'.s$ of new network query is determined by applying the "tolerant" greatest common sampling on all user queries (including the new query u). The attributes and the selection predicates of n' are also chosen based on the set of user queries (line 23). The attribute set is the union of the attribute sets of all user

Algorithm 1 Adding a new User Query

```
1:  procedure AddUserQuery($u$)
2:      if no NQ running then
3:          create new NQ $n$
4:          $n.f := u.f$;   $n.s := u.s$;   $n.p := u.p$
5:          inject new NQ $n$ into the network
6:          $u.op_r := 1 : 1$;   $u.op_\sigma := \emptyset$
7:          $U := U \cup \{u\}$
8:          return
9:      end if
10:     if $u.f \subseteq n.f \wedge n \leq u$ then    ▷ use existing NQ
11:         $u.op_\sigma = u.p \backslash n.p$
12:         $U := U \cup \{u\}$
13:         if TGCS$(u.s, n.s) = n.s$ then
14:             $u.op_r := \frac{u.s}{n.s} : 1$;
15:         else    ▷ adapt sample period of existing NQ
16:             $n.s :=$ TGCS $(U)$
17:             inject rate-change $(n.s)$ into the network
18:             $\forall u_j \in U : u_j.op_r := \frac{u_j.s}{n.s} : 1$
19:         end if
20:     else    ▷ setup new NQ and migrate UQs
21:         create new NQ $n'$
22:         $n'.s :=$ TGCS$(\{u\} \cup U)$
23:         $n'.f := \bigcup_{u_j \in U} u_j.f$;   $n'.p = \bigcap_{u_j \in U} u_j.p$
24:         inject new NQ $n'$ into the network
25:         $u.op_r := \frac{u.s}{n'.s} : 1$;   $u.op_\sigma = u.p \backslash n'.p$
26:         wait until $n'$ has received $\tau$ tuples
27:         for all $u_j \in U$ do    ▷ migrate UQs to $n'$
28:             $u_j.op_r := \frac{u_j.s}{n'.s} : 1$
29:             $u_j.op_\sigma := u_j.p \backslash n'.p$
30:         end for
31:         $U := U \cup \{u\}$
32:         remove NQ $n$
33:         $n := n'$
34:     end if
35: end procedure
```

queries and the selection predicate is the largest common set, i.e., the intersection set of all user query predicates. Next, the down-sampling and selection operators of the new user query u are configured and the new network query n' is injected into the network. It immediately produces data items for user query u. While n' is being disseminated in the network the old network query n continues delivering tuples. As soon as n' has been setup (detected by counting the received tuples, line 26) all user queries from n are migrated to n' and their down-sampling operators are reconfigured to the new common sampling frequency. The constant τ is a tuning parameter initially chosen to characterise the query setup behaviour of the network. For example for a deep network where the most distant nodes are several hops away from the root node, a larger value of τ has to be used. The attribute projections of the queries do not need to be reconfigured because the projections are performed implicitly in our implementation when tuples from network queries leave the operator chain. Additionally the insertion of selection operators between the user query and the network query might be necessary since the new network query can be less selective. As soon as no more user query exists for network query n, it is removed and n' becomes the new network query.

B. Withdrawing a User Query

What we have discussed so far allows new queries to be submitted to the system. It remains to be seen how to deal with user queries that are withdrawn. The routine `StrengthenNetworkQuery` (described in algorithm 2) performs this step. This routine is called periodically. It is possible to call the routine as soon as user query is stopped, however changing the network query too often would create too much traffic on the network. By calling the routine periodically with a sufficiently large interval the overhead is minimised. Recall that the query executed by the sensor network can be changed in two different ways. First, the network query can be replaced by a different query. Second, the sampling period of the running network query can changed directly. Since any modification is associated with a cost, algorithm 2 first analyses whether it is cost effective to change the network query. The algorithm is based on two *penalty functions* that indicate how well the current network query matches the user queries. The first penalty function $f_r(n, U)$ compares the sampling periods of all user queries from the set U with the one of network query n. The shorter the sampling period $n.s$ of the network query compared to the "tolerant" common sampling period of the user queries, the larger is the penalty value. We define this penalty function as follows

$$f_r(n, U) = \frac{\text{TGCS}(U)}{n.s} - 1 \ , \tag{6}$$

such that $f_r(n, U) = 0$ for the optimal sampling period of the network query. The second penalty function $f(n, U)$ determines the overall matching by also including the set of selected attributes and selection predicates. It counts the number of attributes selected by the network query that are no longer needed by any user query. This is done by computing the difference set of the network query's attribute set and that of every user query. Additionally, it counts the selection predicates all user queries have in common that do not appear in the network query (e.g., these predicates might have been missing in a previous user query, thus preventing their inclusion in the network query). The value of $f(n, U)$ is a weighted sum of these three terms:

$$f(n, U) = f_r(n, U) + \alpha \cdot \left| n.f \setminus \bigcup_{u \in U} u.f \right| + \beta \cdot \left| \bigcap_{u \in U} u.p \setminus n.p \right| \tag{7}$$

The parameters α and β are tuning parameters that are initially chosen to reflect the characteristics and the desired behaviour of the sensor network. Changing the sampling period of a network query is associated with a cost threshold ϕ_r. The expense for replacing a network query and migrating the user queries to the new network query is associated with a migration cost threshold ϕ_m. In general, setting up a new network query is more expensive than changing the sampling period of an existing network query, i.e., $\phi_r < \phi_m$. If the value of the penalty function is larger than the corresponding cost the change is performed, i.e., if $f(n, U) > \phi_m$, the network query is replaced. Since replacing a network query may also include an update in the sampling period, an explicit rate-change is only done if $f(n, U) \leq \phi_m$ and $f_r(n, U) > \phi_r$. The cost thresholds are set at configuration time and are chosen to reflect the characteristics of the network and the desired behaviour of the system. For example, if the sensor network is not very reliable, i.e., many messages are lost, and the energy consumption for performing a rate-change or setting up a new query is large, larger values for the cost thresholds ϕ_r and ϕ_m are chosen, such that changes are performed less frequently.

On line 2 in algorithm 2 the penalty function is evaluated to determine whether it is worthwhile to replace the network query. If so the system sets up a new network query n' with the lowest possible sampling period and only those attributes that are required by any user query (line 4). By choosing all selection predicates that are common to all user queries the selectivity of the network query can be maximised (line 5). Next the query is injected into the network. As in algorithm 1 the system waits until the new query is set up before migrating the user queries. Then the old network query is removed. If the penalty $f(n, U) \leq \phi_m$ and $f_r(n, U) > \phi_r$ (line 14)

Algorithm 2 Withdrawing a User Query

```
1: procedure StrengthenNetworkQuery
2:     if $f(n, U) > \phi_m$ then
3:         create new NQ $n'$
4:         $n'.s :=$ TGCS$(U)$;   $n'.f := \bigcup_{u \in U} u.f$
5:         $n'.p := \bigcap_{u \in U} u.p$
6:         inject new NQ $n'$ into the network
7:         wait until $n'$ has received $\tau$ tuples
8:         for all $u \in U$ do        ▷ migrate UQs to $n'$
9:             $u.op_r := \frac{u.s}{n'.s} : 1$
10:            $u.op_\sigma := u.p \backslash n'.p$
11:        end for
12:        remove NQ $n$
13:        $n := n'$
14:    else if $f_r(n, U) > \phi_r$ then
15:        $n.s :=$ TGCS$(U)$   ▷ change interval of NQ
16:        inject rate-change $(n.s)$ into the network
17:            ▷ adjust down-sampling operators $\forall u \in U$
18:        for all $u \in U$ do
19:            $u.op_r := \frac{u.s}{n.s} : 1$
20:        end for
21:    end if
22: end procedure
```

the sampling rate of the current network query is adapted to the "tolerant" common greatest sampling period (line 15). Next the down-sampling operators of all user queries are reconfigured to the new sampling period of the network query (line 19).

C. A "tolerant" algorithm for greatest common sampling period determination

This section describes the "tolerant" greatest common sampling period determination algorithm (TGCS). It is related to the Euclidean greatest common divisor algorithm (GCD). The problem is to find the greatest common sampling period for a set of user queries $U = \{u_1, u_2, \dots u_m\}$ having a sampling period of $u_1.s, u_2.s, \dots, u_m.s$ milliseconds each. Since the query processing on the sensor motes is performed in discrete time steps, such called *heart beats* of length R ($R = 256$ ms in TinyDB), the specified period $u_i.s$ is quantised into $\lfloor \frac{u_i.s}{R} \rfloor$ units of R giving an effective sampling interval duration of $\lfloor \frac{u_i.s}{R} \rfloor R$ milliseconds.

The idea of a "tolerant" version is to allow an error in the effective sampling period in anticipation of a higher common sampling period (even if some $\lfloor \frac{u_i.s}{R} \rfloor$ are relative prime). The largest relative error that can be tolerated is specified by a constant ε. In the current implementation this parameter is used for all queries. But it could just as well be specified for every query individually. We enforce that the effective common sampling period $n.s$ of the network query remains within the error bounds such that condition (3) holds. Note that the common sampling interval may be less but never larger than any $u_i.s$ because the user application is more likely to be able to handle a surplus of data than missing tuples. The condition (3) states that there must be a sampling period within the interval $[(1-\varepsilon)u_i.s, u_i.s]$ that is evenly divisible by $n.s$. In order to prevent errors from an additional quantisation on TinyDB when processing the network query, we force $n.s$ to be an integer multiple of the heart beat length R. Since the "tolerant" greatest common sampling period $n.s$ is no larger than the shortest user interval, $n.s \leq \min(u_i.s)$ must hold, resulting in at most $\lfloor \frac{\min(u_i.s)}{R} \rfloor$ "candidate" lengths for $n.s$. The TGCS algorithm then iterates over all possible candidate lengths and checks if condition (3) is satisfied, as sketched in algorithm 3. The largest candidate length is always returned because the algorithm starts the largest "candidate" period and then stepwise reduces the period until the shortest common sampling interval $n.s_{\text{min}}$ is reached, which can be specified as additional system parameter. It prevents the system from entering an inefficient mode of operation due to congestion of the network when the sampling interval of a network query is chosen too small.

Algorithm 3 "Tolerant" Greatest Common Sampling period (TGCS)

```
1: function TGCS$(u_1, u_2, \dots, u_m, R, n.s_{\text{min}}, \varepsilon)$
2:     $p := \lfloor \frac{\min(u_i.s)}{R} \rfloor$
3:     while $pR > n.s_{min}$   $\wedge$
4: $\neg\big(\forall u_i.s : \exists k_i \in \mathbb{N} : (1-\varepsilon)u_i.s \leq pRk_i \leq u_i.s\big)$ do
5:         $p := p$ - 1
6:     end while
7:     return $pR$
8: end function
```

Algorithm 3 requires at most $\lfloor \frac{\min(u_i.s)}{R} \rfloor - \frac{n.s_{\text{min}}}{R}$ iterations. During each iteration, condition (3) has to be checked for all m user queries. The time complexity (for the range checks) therefore is $\mathcal{O}\left(m \cdot \min\{u_i.s, \forall i\}\right)$. Although the algorithm is in principle expensive, our experimental evaluation has shown that the overhead is acceptable and completely hidden behind other costs (e.g., sending a network query to the sensor network).

IV. Data Extraction

The "tolerant" common sampling algorithm computes the sampling period for the network query such that the delivered tuples can be used as results for the user queries. However the period determined by the TGCS algorithm is (in general) smaller than those of the user queries. Thus the tuple stream received from the network must be down-sampled before being delivered to the users. This is done by an operator that is placed between the network query and a user query (Fig. 3). The result stream is fed into a down-sampling operator which then emits tuples at a rate specified by the user. In the following, two different intervals are distinguished: the *incoming sampling period*, i.e., the sampling period of the tuples originating from the network query and the *outgoing sampling period*, i.e., the sampling period at which the user requests tuples. A valid implementation of a down-sampling operator must solve two problems: (1) how forwarded tuples are selected from the incoming stream and (2) how the epoch value of the forwarded tuples is (re-)computed. The algorithm presented in this section uses time stamp information obtained at the arrival of previous tuples to determine which element is to be forwarded.

Tuples sent by the sensor nodes do not contain fine-grained time stamps. The coarse-grained epoch only allows to associate a tuple to a specific sampling epoch. As the sampling period chosen by the user can be arbitrary large, the time information in the epoch number too coarse and insufficient for down-sampling. Hence, a time stamp is added when a tuple arrives at the gateway node. The down-sampling operator remembers the time stamp t_k of the last forwarded tuple and then forwards the first tuple which arrives at a time t_l such that $t_l - t_k \geq u_i.s$ where $u_i.s$ is the sampling period specified in user query u_i. The operator also remembers the epoch value e of the last tuple forwarded. It then increments e and assigns this value to the forwarded tuple, yielding a correct epoch value for the user query stream. However, as simple as this idea appears at first, it has two problems:
1) If there is more than one sensor mote that generates tuples, the down-sampling does not work as expected. Assume there are m sensor motes. If no tuples are lost or filtered by a predicate in the "where" clause, m result tuples are received per epoch. Thus the arrival time between tuples from the same epoch is less than period of the network query. Therefore the down-sampling operator as sketched above would only return at most one tuple out of an epoch consisting of m tuples.
2) Slightly too short sampling periods caused by clock drift lead to a large error in the effective sample period seen by the user. The problem is that tuples that arrive too early are simply dropped. The next tuple forwarded will then be late, introducing a large error in the effective sampling period.

There are remedies for these problems. The difficulty for the first is that the time stamp and epoch count of the last forwarded tuples has to be stored for *every* node in the network. Only then are the tuples received in the same epoch forwarded This modification introduces two new difficulties. First, it requires additional storage for each node in the network. Two integer numbers need to be kept for each node, the epoch value and the time stamp of the last forwarded tuple. Second, this approach requires that the *nodeid* attribute is always present in the result field. If the network query does not select the *nodeid* field, then there is no way for the operator to associate a result with a node. A solution is to include the *nodeid* by default on every network query.

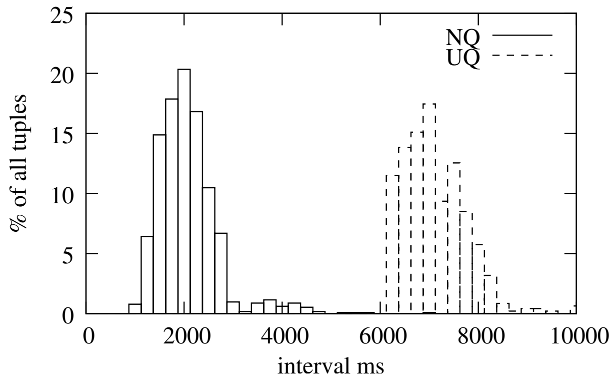
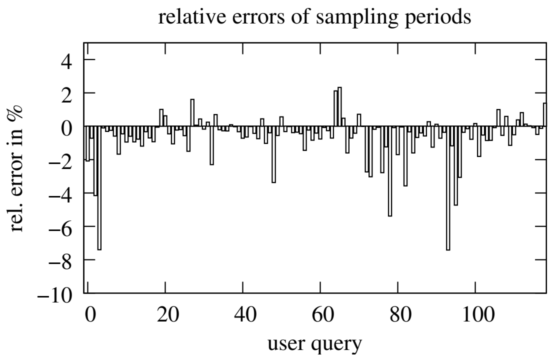


Fig. 4. Histogram of measured intervals between user query tuples (UQ) with a sampling period of 6144 ms down-sampled from a network query (NQ) with a sampling period of 2048 ms.

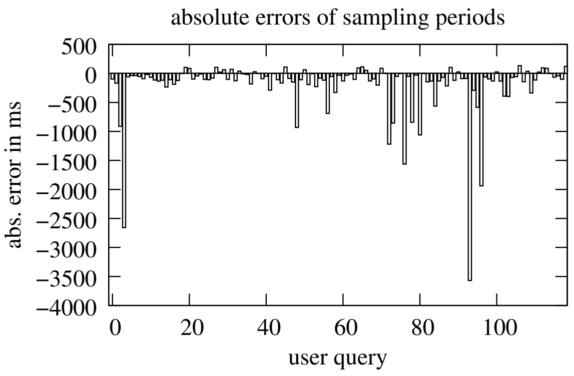
The second problem can be alleviated by caching the latest tuple that arrives before its expected time instead of dropping it. The tuple found in the cache, i.e., the latest tuple, is then forwarded when the tuple is due, as specified in the user query. After being forwarded, the tuple is removed from the cache. Caching reduces the number of tuples that are intentionally dropped to the minimum. Also note that the time stamp of a tuple is not modified when it is emitted by the operator. Changing the time stamp obviously would introduce a zero error (if there are always tuples available). The meaning of a time stamp changes, though, since the time stamp should specify *when* the tuple arrived and thus relates to the measurement. Fig. 4 shows the histogram of the measured tuple intervals for both a user query and network query tuples. The tuple stream from a networking query with a sample period of 2048 ms is transformed by a caching down-sampling operator to a user query stream with 6144 ms sampling period, i.e., a down-sampling ratio of 1:3 is applied. It can be seen that the specified user query sample period (6144 ms) represents a lower bound of the measured tuple interval. Even with caching this implementation of the operator may introduce errors. This is due to the fact that tuples are dropped if they arrive within the same user query period. However, a dropped tuple typically results in a longer period for the next epoch.

V. System Evaluation

We implemented our system in Java on top of the TinyDB network interface component. The gateway node of the sensor network was connected via serial interface (RS232) to the Java system. The deployment consisted of three Mica2 sensor motes. The sensor modes were



(a) Relative Error



(b) Absolute Error

Fig. 5. Error in sampling period (measured from 120 random queries)

equipped with a light, temperature and sound sensor as well as an AD converter for monitoring the battery voltage. The shortest sampling period the network reliably delivers data was 1024 ms. Thus, this period was configured as limit $s_{\min}$ for the TGCS algorithm. Its relative error tolerance ε was chosen as 10%. In order to model user behaviour, i.e., submitting and cancelling queries, we implemented a random query generator. For the queries, the size of the attribute set was uniformly distributed and well as the selection of the attribute fields. The user queries were created from a Poisson process at a rate of 1 query/min with an exponentially distributed sampling interval (average 30 s) and an exponentially distributed execution duration (average 10 min). Data was gathered for 120 queries, then the average sampling interval between consecutive tuples was measured. After 120 queries had been executed, the experiment was stopped and the recorded results were analysed offline. The average relative error in the sampling interval, measured as the interval between two tuples with a consecutive epoch number is shown in Fig. 5(a). Fig. 5(b) shows the absolute sampling errors in milliseconds. The largest error for a measured query interval was -7.42%. This demonstrates that the sampling period error laid indeed within the 10% error margin specified and that the system is able to process the given query load. The results also show that the network query had only to be replaced once. Starting with the fourth user query, the network query contained all five sensor attributes. However, the less expensive rate-change operation occurred 21 times. The changes of the sampling period of the network query are depicted in Fig. 6. In the figure the highest sampling rate with 1024 ms period was only used during 26.4% of the experiment time (when the TGCS algorithm is used to determine the common sampling frequency). The largest sampling period reached during the test is 4608 ms. This corresponds to an improvement by a factor 4.5 compared to using the sampling period of the universal network query. For evaluating the TGCS

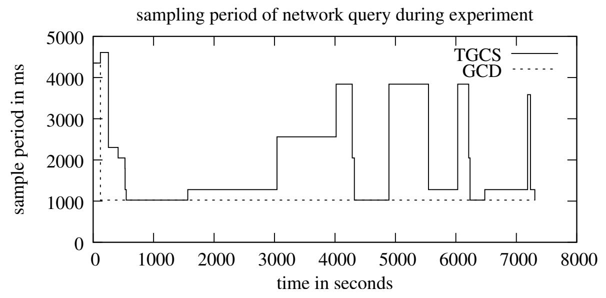


Fig. 6. Sample period of the network query during experiment (120 random queries) with TGCS and GCD algorithm

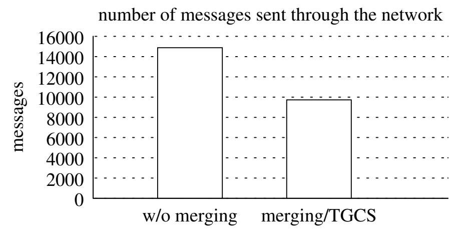


Fig. 7. Number of messages sent during the experiment, without merging of user queries and merging using TGCS algorithm.

algorithm against the basic GCD algorithm, the sampling periods of the network query is shown in Fig. 6. Starting from the second user query, GCD always returned the sampling period of the universal network query. This has a significant impact on the number of messages that are sent. Fig. 7 represents the number of messages that were sent through the network. The bar on the left in Fig. 7 indicates the number of messages that would have been sent if the user queries had been directly broadcast into the network without merging. Since TinyDB is not capable of executing a large number of queries concurrently, the message count for the non-merging case in the figure was computed based on the queries' sample period and execution duration. The right bar in the figure shows the message count for the merged queries that were sent into the network by our system. It can be seen that by applying merging and the TGCS algorithm the number of result messages is reduced by 35%.

A. Further Optimisations

The moderate reduction of number of messages by 35% shown above is due to the fact in this evaluation all user queries are merged into a single network query. This is not always the best option. Consider, for example, two user queries with sampling periods $u_1.s = 7$ s and $u_2.s = 5$ s and the heart beat length of the system $R = 1$ s. The common sampling period for the network query determined by the TGCS algorithm using $\varepsilon = 0.1$ is then $n.s = 1$ s as any other choice would violate the specified error boundaries. Hence, one message/s is sent by every node in the network. However, when executing the two queries separately only $\frac{12}{35} \approx 0.34$ messages/s are sent. Thus, merging queries does not always reduce the message count. As a consequence we are investigating methods to merge a set of user queries into more than one network query. We are working on a cost model that extends the presented system such that it is able to build optimal groups of user queries. The cost model includes the sampling period, i.e., the message number, as well as the energy required to access the sensors. User queries within a group are then merged using the algorithms described in this paper and sent into the network. However, having an $m : n$ merging for user to network queries instead of $m : 1$ requires that the network is able to process n queries in parallel which is, as already mentioned, not feasible with the current version of TinyDB.

VI. Conclusions

We have presented a solution to the problem of multi-user support in sensor networks. For this purpose we take advantage of using declarative requests in the form of queries to turn the problem of multi-user support into a problem of query rewriting and merging. The platform is able to merge user queries into a network query which is then sent into the network and executed by the sensor nodes. The stream of tuples from the network query is then processed by a chain of operators that adapt the tuple rate and filter out tuples or attributes not requested by the user queries. The proposed "tolerant" version of the common sampling period determination algorithm yields a larger sampling period for the network query than the greatest common divisor of the sampling periods of all user queries. Thus the network can be operated more efficiently while still providing all data requested by the user queries. The idea is to allow an error in the effective user query interval in anticipation of a larger common sampling period. The algorithm uses a tolerance parameter, that allows adjusting the trade-off between the length of the computed sampling period and the resulting error. In general the sampling period of the result stream from the network query must be enlarged by down-sampling to match the one specified in the user query. A down-sampling operator that uses tuple time stamps has been discussed and evaluated. Caching the last tuple received enlarges the epoch yield when the network is delivering data in too a short interval.

The evaluation, albeit on a small-scale deployment but with a large number of concurrent queries, shows that system is not only able to merge the user queries, but also to deliver result tuples to the user at the specified rate. Scalability in terms of network size is mainly restricted to the underlying sensor network, TinyDB in our case. As mentioned, we are working on an alternative to TinyDB that will eliminate many of such restrictions. The message count can further be reduced if the user queries are merged into more than one network query. A more optimal grouping of user queries can also be obtained using a cost model that is currently under development.

Acknowledgment

The work presented in this paper was supported (in part) by the National Competence Center in Research on Mobile Information and Communication Systems NCCR-MICS, a center supported by the Swiss National Science Foundation under grant number 5005-67322.

References

[1] Szewczyk R., Osterweil E., Polastre J., Hamilton M., Mainwaring A., Estrin D.: Habitat Monitoring with Sensor Networks. Communications of the ACM **47**:6, pp. 34–40, June 2004.

[2] Jeongyeup P., Chintalapudi K., Govindan R., Caffrey J., Masri S.: A Wireless Sensor Network for Structural Health Monitoring: Performance and Experience. Workshop on Embedded Networked Sensors (EmNetS-II), 2005.

[3] Szewczyk R., Polastre J., Mainwaring A., Culler D.: Lessons From a Sensor Network Expedition. European Workshop on Sensor Networks, EWSN'04.

[4] Szewczyk R., Mainwaring A., Polastre J., Anderson J., Culler D.: An Analysis of a Large Scale Habitat Monitoring Application. Embedded Networked Sensor Systems, 2004.

[5] Madden S., Franklin M., Hellerstein J., Hong W.: TinyDB: an acquisitional query processing system for sensor networks. ACM TODS **30**:1, pp. 122–173, March 2005.

[6] TinyDB web page. `http://telegraph.cs.berkeley.edu/tinydb/`.

[7] Loo B., Condie T., Hellerstein J., Maniatis P., Roscoe T., Stoica I.: Implementing Declarative Overlays. SOSP'05.

[8] Jeffery S. R., Alonso G., Franklin M. J., Hong W., Widom J.: Declarative Support for Sensor Data Cleaning. PERVASIVE'06.

[9] Jeffery S. R., Alonso G., Franklin M. J., Hong W., Widom J.: A Pipelined Framework for Online Cleaning of Sensor Data Streams. ICDE'06.

[10] Mukhopadhyay S., Panigrahi D., Dey S.: Data aware, Low cost Error correction for Wireless Sensor Networks. WCNC 2004.

[11] Deshpande A., Guestrin C., Madden S., Hellerstein J., Hong W.: Model-Driven Data Acquisition in Sensor Networks. VLDB'04.

[12] Trigoni N., Yao Y., Demers A., Gehrke J.: Multi-query Optimization for Sensor Networks. Technical Report TR2005-1989, Cornell University, 2005.