Shared queries in sensor networks for multi-user support

Shared Queries in Sensor Networks for Multi-User Support *

René Müller and Gustavo Alonso
Department of Computer Science
Swiss Federal Institute of Technology Zurich (ETHZ)
Zurich, Switzerland
{muellren,alonso}@inf.ethz.ch

ABSTRACT

Wireless sensor networks are still quite limiting in the way they can be programmed and used. For instance, most existing platforms for sensors networks allow only a single application or user to run on them. This makes it difficult to, e.g., have mobile users connecting to a local sensor network as they arrive at a new location. In this paper we tackle the problem of supporting different applications over the same sensor network. The idea is to allow 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 single 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 user queries.

1. INTRODUCTION

Sensor networks are starting to be more widely used [1, 2, 3, 4]. In spite of these early successes, there is still a need for better tools to program sensor networks. An example is the lack of support for concurrent users. This makes it difficult to, e.g., implement a system where mobile users are granted access to local sensor networks so that each one can extract whatever data they need.

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 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 rates).

*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.

1.1 Sharing a sensor network

The 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 single 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.

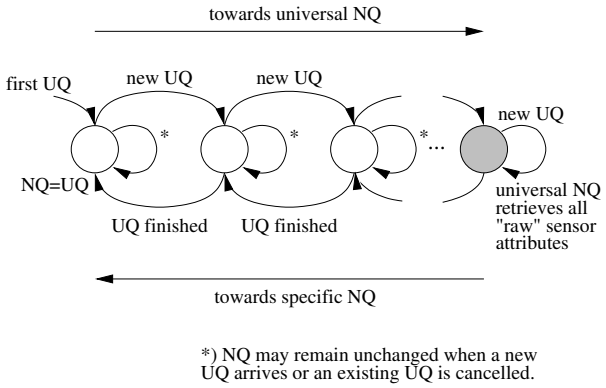


Figure 1: Moving between specific network queries and "universal" network queries as user queries are started and cancelled.

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 are highly unreliable – specially when under heavy load. It is thus unlikely that the universal network query will return all the data it requests [7, 8]. It is also not practical to run the sensor network at such high regime not 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 figure 1. Performing such a process dynamically 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 then 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.

1.2 Related Work

Our proposal complements the work of Jeffery et al. [7, 8], 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 [9, 10] 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 this paper, however, we do not focus on cleaning the data but on the more basic problem of extracting correct user data streams from the network data stream. Our approach is also related to the work in [11] where the need for a mapping layer between user and sensor network is also identified. However, the focus of their work is in-network aggregation and join operations using chains of flow blocks. This work also complements ours in that, once our system produces user data streams, additional operations could be applied to such streams to support more sophisticated user queries. 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. 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. Several papers [13, 14] describe the problem of time synchronisation in wireless sensor networks and propose different solutions. In this paper, we focus on post-processing data streams by explicitly taking inaccuracies of timing into account.

1.3 Contributions

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

- Algorithms are provided to merge multiple queries into a single network query. These algorithms support the dynamic addition and withdrawal of queries. We also discuss an approximated algorithm to match widely different sampling periods by guaranteeing that the resulting sampling period observed by the user differs from the requested sampling period by no more than parameter ε.
- Algorithms and 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 (and thus the underlying network query changes).
- The experimental results show that our system is capable of running several hundred different user queries over 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.

1.4 Organisation of the paper

The next section describes the concept of multi-query support. First, it gives a brief overview over the system and the working environment. Then two basic operations are described: query merging and data extraction. Section 3 provides a detailed description of how user queries are merged, i.e., how the network query is adapted when a new user query arrives or is withdrawn. Section 4 presents a number of algorithms that can be used to extract tuples from a network query in order to obtain results for user queries. Section 5 provides results from the experimental evaluation of our system. Section 6 concludes the paper.

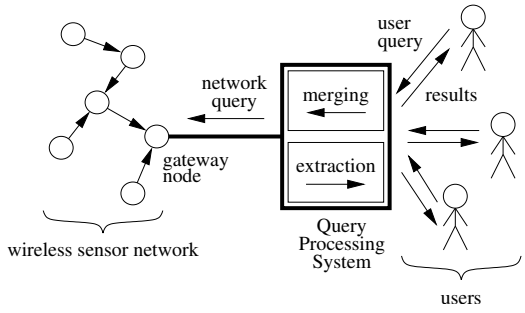


Figure 2: System Overview: wireless sensor network and query processing system

2. MULTI-QUERY SUPPORT

2.1 Working Environment

Our system is built atop TinyDB [5], 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 (as well as the gateway) 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 single *Network Query* (NQ). The network query is then sent to the gateway node which broadcast it into the wireless sensor network. User Queries as well as network queries have the following form:

```
SELECT select-list
  [FROM sensors]
  [WHERE where-clauses]
  SAMPLE PERIOD sample-period-in-msecs
```

This syntax is similar to SQL although with a restricted set of operators (e.g., joins are not supported). 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-clauses* are arbitrary boolean predicates (e.g., temperature ≤ 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 is is not filtered by a predicate in the "where" clause). 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.

2.2 Basic Operations

The query processing system *merges* multiple user queries into a single 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 single network query and how the result data is extracted, we use a simple example. Consider the following user queries:

```
UQ1 : SELECT nodeid, light FROM sensors
        SAMPLE PERIOD 5000
UQ2 : SELECT nodeid, light FROM sensors
        SAMPLE PERIOD 15000
UQ3 : SELECT light FROM sensors
        SAMPLE PERIOD 50000
UQ4 : SELECT nodeid, light, temp
        FROM sensors
        WHERE nodeid=1 AND temp>100
        SAMPLE PERIOD 20000
```

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:

```
NQ1 : SELECT nodeid, light, temp
        FROM sensors SAMPLE PERIOD 5000
```

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 $nodeid = 1 \wedge temp > 100$. 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.

2.3 Query Merging

In this section we provide a more formal background for the mapping of user queries and 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 sensors, 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 be met when a set of

m user queries U 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;

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

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 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 some ε of what the user requested. The TGCS algorithm is described later in section 3.3. Thus instead of using (2) we require:

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

Note that in some cases, the application submitting the user query may require the data to be delivered exactly in the time intervals specified. 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 4.

Selection queries – as implied by the predicate in the where clause – require special treatment. They are expressed in SQL with one additional restriction that predicates may only occur in conjunctions[1]. Let $u.p$ be the set of predicate conjunction terms in the where clause of user query u, i.e., $u.p = \{\texttt{nodeid} = 1, \texttt{temp} > 100\}$. Using relational algebra the query can be written as

$$\sigma_{\texttt{nodeid}=1 \,\wedge\, \texttt{temp}>100}\,(\texttt{sensors}) \ .$$

[1]Disjunctions are currently not supported by TinyDB however they can be easily implemented on our system in the mapping from user queries to network queries, further demonstrating the advantages of the architecture we propose.

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.

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

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 $Q_i \le Q_j$ where we use

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

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., $n \le u$. Thus:

$$\forall u \in U \,:\, n \le u \tag{4}$$

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

$$\forall u \in U : \quad n.f \supseteq u.f \,\wedge\, n \le u \,\wedge\, \exists x \in \mathbb{N} \,:\, (1-\varepsilon)u_i.s \le x \cdot n.s \le u_i.s \tag{5}$$

2.4 Data Extraction

The example given in 2.2 illustrates how user queries can be transformed into a single 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. These operators are in some cases independent of the adjustment and in all others can be directly derived from the mapping between the user query and the network query. For the purposes of this paper, the set of operators we are considering include:

- Tuple rate conversion and down-sampling $(r_1, \ldots, r_3)$. 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 [10] could be used.
- Time and epoch shift $(s_1, \ldots, s_4)$. Each user query is treated so that the user gets the impression that it is the only query running on the network. However,

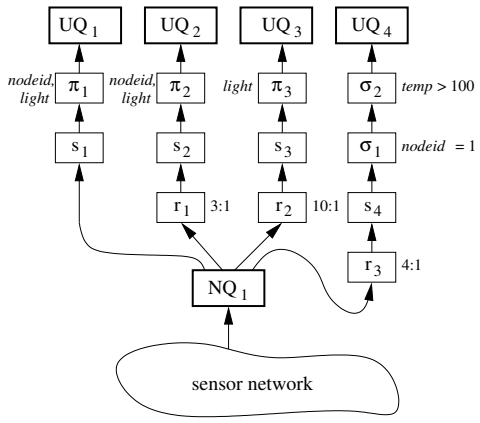


Figure 3: Mapping of User Queries (UQ) to Network Queries (NQ)

the data streams coming from the sensors have time (*epoch*) information attached to them that typically has nothing to do with any concrete user query. The time and epoch shift operators set the counters to 0 when a user query starts and transforms the epoch information on the resulting data set so that to the user it looks like the system just started producing data for that user query.

- Attribute projection $(\pi_1, \ldots, \pi_3)$. Similar to projection in relational algebra, a projection operator in our system removes unwanted attributes from the tuple stream to produce a new data stream with less attributes.
- 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 2.2, the corresponding processing pipeline with all operators required to map tuples NQ_1 to $UQ_1, \ldots, UQ_4$ is shown in figure 3. 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.

3. ALGORITHMS FOR QUERY MERGING

3.1 Adding a new User Query

The conditions above state 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 network query has to adapted when a new user query is added or and a user query is withdrawn. 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.

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_σ := ∅
7:         U := U ∪ {u}
8:         return
9:     end if
10:    if u.f ⊆ n.f ∧ n ≤ u then          ▷ use existing NQ
11:        u.op_σ = u.p\n.p
12:        U := U ∪ {u}
13:        if TGCS(u.s, n.s) = n.s then
14:            u.op_r := 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:            ∀u_j ∈ U : u_j.op_r := 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} ∪ U)
23:        n'.f := ⋃_{u_j∈U} u_j.f;   n'.p = ⋂_{u_j∈U} u_j.p
24:        inject new NQ n' into the network
25:        u.op_r := u.s/n'.s : 1;   u.op_σ = u.p\n'.p
26:        wait until n' has received τ tuples
27:        for all u_j ∈ U do   ▷ migrate UQs from n to n'
28:            u_j.op_r := u_j.s/n'.s : 1
29:            u_j.op_σ := u_j.p\n'.p
30:        end for
31:        U := U ∪ {u}
32:        remove NQ n
33:        n := n'
34:    end if
35: end procedure
```

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 3.3). 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 when the new user query includes attributes that are not retrieved by the current network query (condition (1) is not satisfied) or if the current network is more selective (condition 4 is not satisfied) than the user 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 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 setup in the network[2] the old network query n continues delivering tuples. As soon as n' has reached steady state (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. Thus in order to deliver tuples as specified by the user, further filtering has to be done outside of the network. As soon as no more user query exists for network query n, n is removed and n' becomes the new network query.

3.2 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[3].

Algorithm 2: Withdrawing a User Query

```
 1: procedure StrengthenNetworkQuery
 2:     if f(n,U) > φ_m then   ▷ penalty > migration cost
 3:         create new NQ n'
 4:         n'.s := TGCS(U);   n'.f := ⋃_{u∈U} u.f
 5:         n'.p := ⋂_{u∈U} u.p
 6:         inject new NQ n' into the network
 7:         wait until n' has received τ tuples
 8:         for all u ∈ U do   ▷ migrate UQs from n to n'
 9:             u.op_r := u.s / n'.s : 1
10:             u.op_σ := u.p \ n'.p
11:         end for
12:         remove NQ n
13:         n := n'
14:     else if f_r(n,U) > φ_r then ▷ penalty > change cost
15:         n.s := TGCS(U)          ▷ change interval of NQ
16:         inject rate-change (n.s) into the network
17:           ▷ adjust down-sampling operators of all u ∈ U
18:         for all u ∈ U do
19:             u.op_r := u.s / n.s : 1
20:         end for
21:     end if
22: end procedure
```

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

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

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

[2]We observed that using TinyDB it takes about 1 min (depending on the network load and the chosen sampling period) for the new network query to reach "steady state", i.e., during the transient phase tuples are lost more frequently.

[3]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. Therefore the routine is called periodically with a sufficiently large interval to minimise overhead.

their inclusion in the network query). The value of $f(n, U)$ is a weighted sum of these three terms:

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

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 its 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) 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).

3.3 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 $U = \{u_1, u_2, \ldots u_m\}$ having a sampling period of $u_1.s, u_2.s, \ldots, u_m.s$ milliseconds each. Since the query processing on the sensor motes is performed in time steps (heart beats[4]) of length R 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.

[4]In TinyDB the heart beat length is $R = 256$ ms.

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[5]. 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. Also there are at most $\lfloor \frac{\min(u_i.s)}{R} \rfloor$ "candidate" lengths of $n.s$. 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 $n.s_{\min}$ 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: Fast "Tolerant" Greatest Common Sampling period algorithm (TGCS)

```
1: function TGCS(u_1, u_2, ..., u_m, R, n.s_min, ε)
2:     p := ⌊min(u_i.s)/R⌋
3:     while pR > n.s_min   ∧
4:       ¬(∀u_i.s : ∃x ∈ ℕ : (1 − ε)u_i.s ≤ pRx ≤ u_i.s) do
5:         p := p - 1
6:     end while
7:     return pR
8: end function
```

Algorithm 3 requires at most $\lfloor \frac{\min(u_i.s)}{R} \rfloor - \frac{n.s_{\min}}{R}$ iterations. During each iteration, condition (3) has to be checked for all m user queries. The time complexity (for the range checks) therefore is $\mathcal{O}(m \cdot \min\{u_i.s, \forall i\})$. 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).

4. ALGORITHMS FOR 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 user queries. This

[5]Note: While this parameter is the same for all queries it could just as well be specified for every query individually.

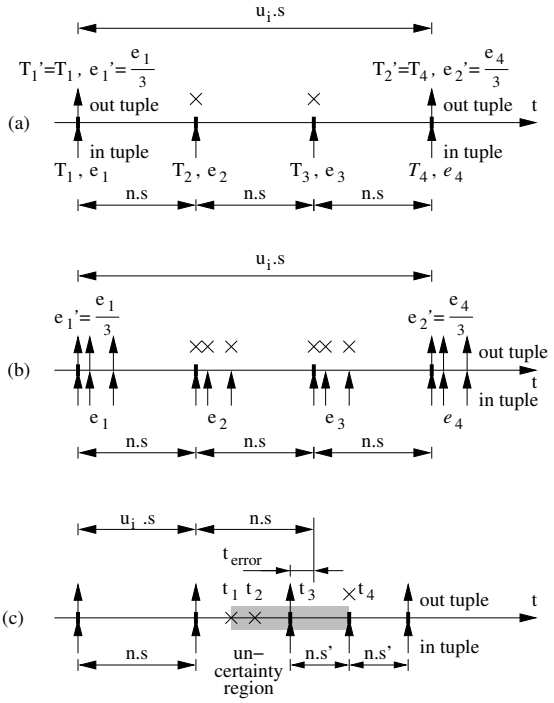


Figure 4: Ratio-based Down-sampling: (a) Down-sampling ratio 3:1 for network consisting of one sensor node, (b) Down-sampling with ratio 3:1 for network consisting of tree sensor nodes, (c) Timing uncertainties during rate-change.

is done by an operator that is placed between the network query and a user query. 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. In what follows, we describe two different approaches. The first method forwards every xth tuple based on the ratio of the outgoing and incoming sampling period. Unfortunately, this approach accumulates inaccuracies of the sampling period, due to *jitter*, present in the incoming tuple stream. This effect does not occur in second approach where time stamp information from previous tuples is used to determine which element is to be forwarded.

4.1 Ratio-based Down-sampling

The *effective sampling period* $u_i.\tilde{s} = n.s\lfloor\frac{u_i.s}{n.s}\rfloor$ of the user query u_i is not less than $(1-\varepsilon)u_i.s$ and an integer multiple of the network query. This is guaranteed by the determination of $n.s$ using the TGCS algorithm. The ratio-based operator takes advantage of this by forwarding only every $\frac{u_i.\tilde{s}}{n.s}$th tuple from a node. The *epoch* field of the tuple received is then divided by $\frac{u_i.\tilde{s}}{n.s}$.

This is illustrated in Fig. 4 (a). Tuples $T_1, \ldots, T_4$ have epoch values $e_1, \ldots, e_4$. In the figure $\frac{u_i.s}{n.s} = 3$, thus every third tuple is forwarded, i.e., tuple T_1 and T_4. The epoch values of the forwarded tuples are divided by three, e.g., $e_1' = \frac{e_1}{3}$ for the forwarded tuple T_1'. Note that, in general, a tuple is received every epoch from every sensor in the network, i.e., more than one tuple is received per epoch. Furthermore some tuples may arrive late or may be missing at all. The forwarding decision thus has to be made based on the epoch value (see Fig. 4 (b)).

Assume that a tuple T, having epoch number e is sent to the operator which then decides whether the tuple is to be forwarded or dropped. It will be forwarded if the effective outgoing interval $u_i.\tilde{s}$ *evenly divides* the time value computed by the tuple's epoch number and the incoming interval $n.s$, i.e., $u_i.\tilde{s}|(e \cdot n.s)$. The epoch value of the emitted tuple will be computed as

$$e' = \frac{e \cdot n.s}{u_i.\tilde{s}}$$

However, since the sampling period of the network query may change, the ratio-based down-sampling operator occasionally must be reconfigured. Unfortunately just changing the incoming interval from $n.s$ to $n.s'$ alone does not suffice. There are several difficulties involved (illustrated in Fig. 4 (c) where the sampling rate is doubled at time t_1):

1. The rate-change could be initiated at any time (t_1 in Fig. 4 (c)) during the interval between two user query tuples. The down-sampling operator has no control *when* the change is initiated.
2. Once the change is initiated, i.e., the *rate-change* message has been sent, the change does not immediately take effect. In Fig. 4 (c) the rate-change message reaches the sensor nodes at time t_2 where the change takes effect. However, t_2 is unknown to the down-sampling operator.
3. The down-sampling operator cannot determine if a tuple received from the network was created before or after the rate change took effect by looking at the epoch number or time stamp of this tuple.

The only way for the system to recognise the rate-change is to observe the time interval between epochs in the incoming stream. Assume that at t_3 the first tuple after the change arrives at the operator. Until the down-sampling operator has seen the next tuple at t_4 and is able to compute $t_4 - t_3$ it cannot decide whether the change has already occurred. Thus the tuple arriving at t_3 is forwarded yielding a sampling period error of t_{error}.

4.1.1 Advantage

The ratio-based down-sampling operator has the advantage that it does not require additional storage to maintain state for the tuples received from every node. Also it allows "late" tuples from distant nodes to be forwarded as it does not directly count tuples. Instead, it uses the epoch number.

Therefore there are no restrictions in the arrival times of the tuples.

4.1.2 Disadvantage

The disadvantage of all ratio-base operators is the dependency on the precision sampling period of the incoming tuple stream. In fact inaccuracies in the sampling periods are accumulated by this operator as the following analysis shows:

Assume that the ratio-based operator down-samples a data stream with a ratio $x : 1$, i.e., forwards every xth tuple. Assume that the sampling period is not precisely constant as it is the case for TinyDB. The tuple stream from the network has a strong variance in the tuple interval (jitter). Assume just for the sake of argument that I is the interval length between tuples of consecutive epochs in the result stream arriving from a given network node. In order to model the random behaviour of the jitter we consider I as a random variable with mean $E[I] = n.s$ and a variance $VAR[I]$. Let I' be the time interval between tuples in the stream that leaves the operator. Since the operator effectively counts the tuples and forwards every xth tuple, the length of I' is the sum of the last x intervals I. Thus, as expected, it follows that $E[I'] = E[xI] = x \cdot n.s$, but, unfortunately also $VAR[I'] = VAR[xI] = x \cdot VAR[I]$. This means that the variance in the outgoing stream is x times larger than the variance in the incoming stream. In other words the *jitter of x epochs is accumulated*, i.e., the overall jitter is amplified by a factor x. This severely limits the practical use of a ratio-based down-sampling operator.

4.2 Time-based Down-sampling

The time-based down-sampling algorithm does not count tuples and therefore does not suffer from the cumulative jitter effect. It decides which tuples are to be forwarded by looking at their timestamps. It 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, m result tuples are received per epoch, i.e., results on groups of m tuples with the same epoch number will arrive at the root node every $n.s$ milliseconds where $n.s$ is the sample period of the network query. However the time between the arrival of two tuples from the same epoch is less than $n.s$ (see tuple arrival in Fig. 4 (b)). Therefore the down-sampling operator as sketched above will only return one tuple out of an epoch consisting of m tuples.

2. Too short sampling periods caused by jitter in the network tuple stream can lead to a large error in the effective sample period seen by the user[6]. This is illustrated in Fig. 5 (a). Assume a network query with sampling period $n.s$ but due to inaccuracies in timing, tuples arrive with a shorter period $(1-\varepsilon)n.s$. Further assume that the user has specified a sampling period $u_i.s = 2n.s$ and that the last forwarded tuple had a time stamp $t_0 = 0$. The next two tuples will then be received at $t_1 = (1-\varepsilon)n.s$ and $t_2 = 2(1-\varepsilon)n.s$ ms. However since $t_2 - t_0 < 2n.s$ the tuple with time stamp t_2 will not be forwarded but dropped. Only the next tuple arriving at $t_3 = 3n.s(1-\varepsilon)$ will satisfy the inequality and thus be forwarded. By comparing the timestamps between the forwarded tuples $t_3 - t_0$ it can be seen that the resulting interval for the user is $3(1-\varepsilon)n.s$ instead of $2n.s$. This corresponds to a relative error of $\frac{1}{2}(1-3\varepsilon)$, i.e., the upper bound is 50%. Note that this error does not originate from the "tolerant" greatest common sampling algorithm. The error is introduced solely by the down-sampling operator and inaccuracies in the timing of the network.

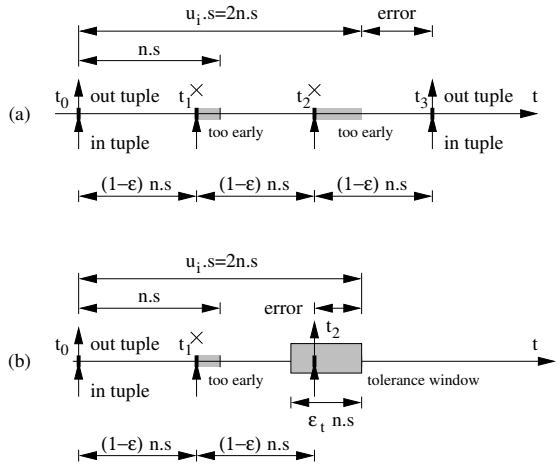


Figure 5: Time-based Down-sampling: (a) without tolerance window, (b) with tolerance window

There are remedies for these problems. The problem concerning multiple sensor motes is addressed first. The difficulty 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 (provided that spacing of the tuples inside the group and between the group is accurate enough). 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.

[6]If the operator is used for a 1:1 mapping between the network query and a user query the resulting error can be up to 100%.

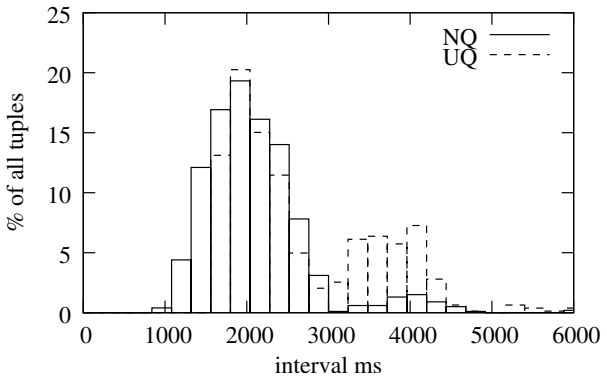


Figure 6: Histogram of measured intervals of user query tuples from a network query with a sampling period of 2048 ms for a user query with the same sampling period, using the time-based down-sampling operator ($\varepsilon_t = 0.2$)

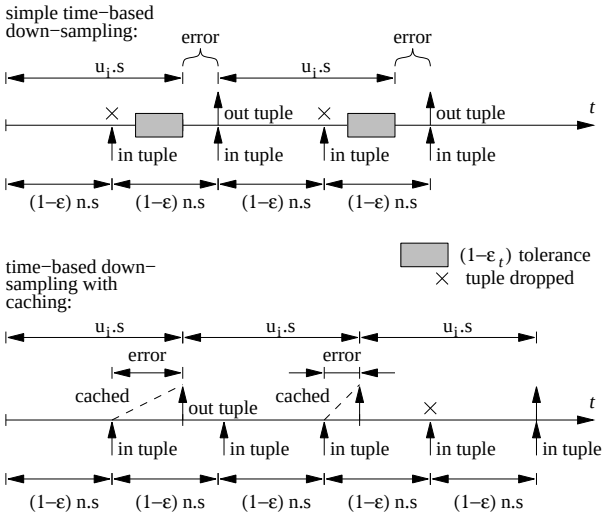


Figure 7: Effect of caching on tuples that arrive too early.

The second problem can be alleviated by weakening the condition $t_l - t_k \geq u_i.s$ that must hold between two consecutive forwarded tuples with time stamp t_l and t_k. A *tolerance window* is added, i.e., the sample period in the stream delivered to the user may have a relative error in the range $[-\varepsilon_t, 0]$ (shorter, but never longer). Now tuples with a too-short sampling period, expressed by a relative error $-\varepsilon$, are still forwarded if $\epsilon \leq \varepsilon_t$. Fig. 5 (b) shows the effect of using a tolerance window on tuples arriving with the same error as in figure 5 (a). If the tolerance window parameter is chosen such that $\varepsilon_t \geq \varepsilon$, the tuple arriving at $t_2 = 2n.s(1-\varepsilon)$ will be forwarded. This reduces the relative error from $\frac{1}{2}(1-3\varepsilon)$ to ε. Then the down-sampling operator does no longer introduces errors. The problem is, of course, only postponed. If the tuple time stamp happens to be just shortly before the tolerance window begins, the tuples will still be dropped.

Figure 6 shows the measured intervals between consecutive tuples for a single user query with 2048 ms sample period. Since there is only one user query, the network query does have the same sampling period, i.e., the mapping from the network query to the user query is 1:1. In principle no down-sampling is needed, but the ratio 1:1 can be used to illustrate the effect this operator has when applied on a tuple stream that is subjected to jitter. The solid lines represent the histogram for network query tuples and the dashed lines the one for the user query tuples. First, observe that the jitter is already present in the tuple stream from the network query. In addition to the peak around 2000 ms, there is another peak around 4000 ms. The second peak around 4000 ms is a result of single tuples that are lost, effectively doubling the sampling interval. More important, however, is the observation that the amount of tuples that arrive too late is larger for the user query than for the underlying network query that provided the tuples. Thus, the algorithm introduces further delays. The reason for this behaviour are tuples that arrive even earlier than $(1-\varepsilon_t)n.s$. The tuples arriving too early will be dropped, thus the next tuple which then will be forwarded introduces a large error in the sampling period.

4.2.1 Advantage

In contrast to the ratio-based operator the simple time-based down-sampling operator does not accumulate the jitter. In contrast to the ratio-based operator the time-based operator does not need to know the sampling period of the incoming tuple stream. Therefore, also no update mechanism is necessary when the sampling interval of a network query is changed.

4.2.2 Disadvantage

One disadvantage is that the epoch number and the time stamp of the last tuple forwarded must be stored for every node in the network. The *nodeid* attribute must be present in the network query. Additionally the down-sampling operator introduces further interval errors due to inaccuracies in the incoming tuple stream. As figure 6 shows, the operator in fact amplifies the sampling period errors in the incoming stream.

4.3 Time-based Down-sampling with Caching

Caching can solve one problem of the down-sampling operator: the large intervals in the user data stream that result if the incoming interval is too short in the network data stream. Instead of dropping a tuple that arrives before its expected time, the tuple will be cached and then forwarded when the tuple is due. This is illustrated in figure 7. The upper time plot depicts the time-based down-sampling operator described in the previous section and the lower plot shows the effect of caching early tuples. In both cases the tuples are delivered at a period of $n.s(1-\varepsilon)$ instead of $n.s$. In the figure the user query sample period u_i is considered to be equal to $n.s$. It can be seen that without caching the first and the third tuple are dropped. With caching the tuples will only be forwarded when they are due. A tuple is only dropped if there was another, more recent tuple, received before the deadline of the next user query tuple (the fourth tuple).

Note that when no caching is used, a tuple will be for-

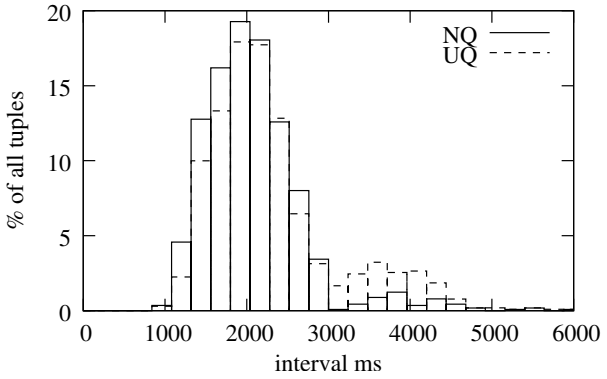


Figure 8: Histogram of measured intervals of user query tuples from a network query with a sampling period of 2048 ms for a user query with the same sampling period, using the time-based down-sampling operator with caching

warded at the moment a new tuple is received. However, with caching a timer must be used that triggers the release of the last received tuple. After being sent, the tuple is removed from the cache. As the operator described in the previous section, this operator must also store the epoch number and the time stamp of the last tuple forwarded for every node. However, it also requires additional storage per *nodeid* as the tuples themselves also need to be cached. Thus memory restrictions have a stronger impact on the caching operator. When an epoch of a user query is due, the operator will forward the cached tuple for every *nodeid*. 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 measurement was taken.

Figure 8 shows the histogram of the measured tuple intervals for a single user query with a sample period of 2048 ms using the cached down-sampling operator. The network query that provides tuples for this single user query uses the same sampling period. When figure 8 is compared with figure 6 it can be seen that the percentage of tuples that have an error of 100% or more was reduced. Unfortunately the deviation from the requested delivery period for the user is still high. About 29% of all user query tuples have an interval that is more than 20% larger than the specified user query sample period. However, only 19% of the network query tuples have an interval that is more than 20% larger than specified. Even with caching this implementation of the down-sampling operator introduces further 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.

4.3.1 Advantage

In contrast to the non-cached version, tuples that arrive too early will not be dropped. They will be forwarded when the next user query epoch ends. Thus less tuples are dropped by the down-sampling operator. Only multiple tuples that were received during a user query epoch are dropped. Measurements show that using caching a more precise result stream can be generated by the operator (Fig. 8).

4.3.2 Disadvantage

The caching operator also relies on the availability of the *nodeid* attribute. Here even more memory is required as the last tuple received needs to be stored for each node. Measurements show that the operator amplifies the jitter in the tuple stream.

4.4 Time-based Down-sampling with Caching and Retransmission

All down-sampling operators presented so far amplify the sampling period error of the network data stream. Caching prevented dropping tuples that arrived to early. Instead, the tuples are stored and forwarded when a new user query tuple is due. However since a tuple is only emitted at most once, the effect of missing tuples or too long intervals in the incoming stream also propagates through the operator. If there is no new tuple the operator cannot forward any tuple when the next user query tuple is required to be emitted. The operator described below uses a different approach by slightly changing the semantics of the tuples in the result stream: it *allows* retransmission of a cached tuple. If no new tuple is received, the old tuple is forwarded again.

If a tuple is resent by the operator, its time stamp and epoch number must be adapted. The client application expects tuples with a strictly monotonically increasing time stamp and a strictly monotonically increasing epoch number for every nodeid. This is achieved by setting the time stamp of the emitted tuple equal to the time when the timer goes off for sending a user query tuple. Since the epoch counter already stored is part of the node state it can simply be incremented and the new value used for the user query tuple. So, in principle the user application is not able to recognize duplicate tuples from their originals, i.e., whether a tuple is sent for the first time.

Resending a tuple essentially "creates" a new measurement with new time stamp. However the measurement is not associated with any physical process that occurred at that time. This "virtual" measurement in this sense is a zero-order extrapolation of the previous value. Note that the semantics of the user query tuple stream is thus changed: not every tuple originates from a real measurement. However it is reasonable to assume that the application is more likely to be able to handle duplicate tuples than missing tuples.

However, there are cases when the user application must be able to recognize a tuple that has already been sent. For example if an application itself computes duplicate-sensitive aggregate values (e.g., sum or average) on some attributes of the tuples. To facilitate the detection, another implicit attribute field *tupleid* is added to the tuple in the user data stream. When a tuple from a network query is received, it will be assigned a unique identification number *tupleid*. This number uniquely identifies those attributes of the tuple that were explicitly selected by the query, e.g., *nodeid*, *light* etc. The *tupleid* will not be modified by any operator. The user then can detect a duplicate tuple by looking at its *tupleid*

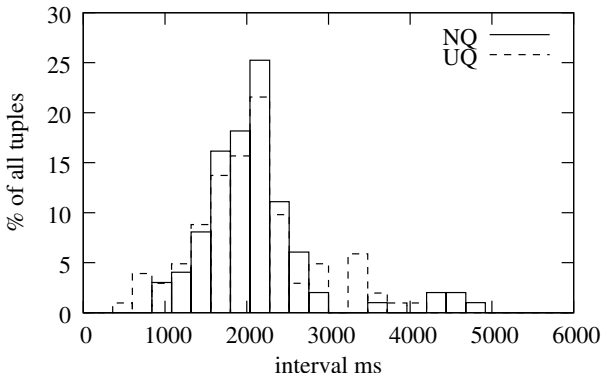


Figure 9: Histogram of measured intervals of user query tuples from a network query with a sampling period of 2048 ms for a user query with the same sampling period, using the time-based down-sampling operator with caching and retransmission. Note that interval is computed from the tuple timestamps.

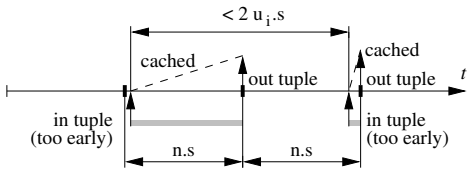


Figure 10: Tuples from a time-based down-sampling operator with caching can still have an interval error up to 100% even if there are no tuples lost (we assume $u_i.s = n.s$).

number.

Figure 9 shows the measured interval times between two tuples for a user query and its underlying network query, both with a specified sampling period of 2048 ms. Note that the tuple interval of the user query is computed from the tuple time stamp, *not* from the time when the tuples were delivered to the user application. With retransmission, the query processing system is always able to deliver tuples every 2048 ms, yielding a precise sampling period for the user. There is an upper bound of the interval error at 4000 ms. This corresponds to a 100% error which is caused by the arrival of the tuples as illustrated in figure 10: if one tuple arrives almost an entire sampling period early and the next arrives shortly before expected. However, in any case the error cannot be larger than 100% because of the resending mechanism.

4.4.1 Advantage

In addition to the advantages of the caching operator without resending, this operator has the additional advantage that the positive sample period error of the tuple stream is bounded.

4.4.2 Disadvantage

In order to allow retransmission of the old tuple, the semantics of the user query tuple stream must be slightly changed. A tuple can now be a result of a "virtual" measurement caused by the resending process. In order to distinguish between "real" measurement tuples and "virtual" measurement tuples an additional implicit *tupleid* field has to be added to every tuple delivered to a user. The *tupleid* uniquely identifies an element from the network query stream.

5. SYSTEM EVALUATION

5.1 Implementation

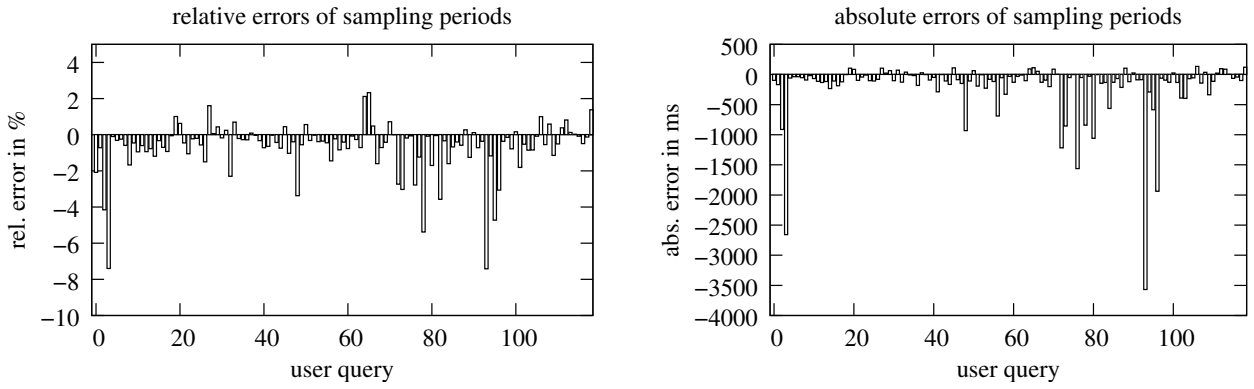
The query processing system was implemented in Java and is run on a Standard Edition JVM. The size of the entire application is 3 MB. This also includes the components from TinyDB that required to access the sensor network. The size of the query processing system alone is 528 kB. The gateway node of the network is connected to the serial interface (RS232) of the portable device that runs the query processing system.

5.2 Evaluation

We tested our implementation on top of TinyDB running on a deployment of three Mica2 sensor motes [16, 15]. The sensor modes are 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 delivers reliable is 1024 ms. Thus, this period was configured as limit s_min for the the TGCS algorithm. For down-sampling, the *time-based down-sampling operator with caching and retransmission* (described in section 4.4) was used and the TGCS algorithm was applied with a relative error tolerance ε of 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 is uniformly distributed and well as the selection of the attribute fields. The user queries emanate 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 have been executed, the experiment was stopped and the recorded results 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. 11(a). The average relative sampling interval error was found to be -0.71%, i.e., the sampling rate is on average faster than expected. This is due to the design of the TGCS algorithm as it always determines a sampling period for the network query that is at least as fast as the fastest user query period. The largest error for a query interval is 7.42%. This shows that the sampling period error lies indeed within the 10% error margin specified and also demonstrates that the system performs well with a sensor network consisting of a very limited sensor nodes.

Fig. 11(b) shows the absolute sampling errors in milliseconds. The largest (absolute) error value is -3570 ms for



(a) Relative Error

(b) Absolute Error

Figure 11: Error in sampling period (measured from 120 random queries)

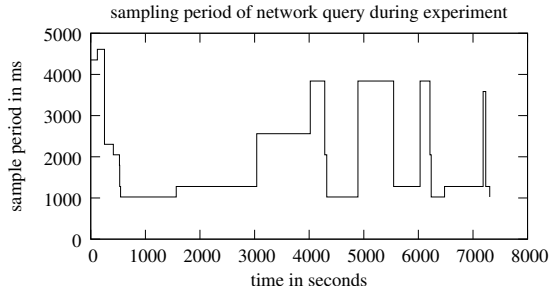


Figure 12: Sample period of the network query during experiment (120 random queries)

query 94. This query has a sampling period of 48100 ms and is only running for 46 seconds. The interval of the underlying network query is 4096 ms at that time, which yields a theoretical effective interval of $\lfloor 48100/4096 \rfloor \cdot 4096$ ms = 45056 ms. This query represents an extreme case, thus the tolerant sampling interval is close to the lower bound yet still inside the tolerance region.

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 21 times. The changes of the sampling period of the network query are shown in figure 12. In figure 12 the highest sampling rate with 1024 ms period is only used during 26.4% of the experiment time. The largest sampling interval 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.

In order to examine the TGCS algorithm, the sampling period of the network query was also computed using the greatest common divisor (GCD) approach. Starting from the second user query, GCD always returned the sampling period of the universal network query.

6. CONCLUSIONS

We have presented a middleware platform for query-based sensor networks that provides simultaneous access for multiple users to a sensor network that is limited in the number of concurrent queries it can process. The platform is able to merge user queries into a single 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 (GCD) 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 tradeoff between the length of the computed sampling period and the resulting error.

The tolerant common sampling period is always as fast as the shortest user query sampling period. Therefore 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. Down-sampling of a tuple stream is not trivial. Four different approaches have been discussed and tested. A ratio-based downs-sampling where only every xth tuple is forwarded has the advantage that it does not not require additional storage for maintaining the state of the sensor nodes, as it is the case for the other methods. However it does accumulate jitter present in the data stream from the network query (proportionally to the sampling-period ratio between network and user query). This jitter accumulation can be avoided when a time-based down-sampling approach is used. We discussed time-sampling with and without caching of tuples and with and without retransmission of tuples. The analysis showed that caching helps reducing missing data when the network is delivering data in a too short interval. Retransmission provides an

upper bound of the sampling period error even in case of missing tuples. In order to detect duplicate tuples (generated by retransmission) a unique identification number is attached to each tuple delivered by the network query.

We believe that the platform described in this paper alleviates many of the current shortcomings of sensor networks as experienced by the user. It provides concurrent access to the sensor network for multiple users and applications. This allows, e.g., mobile users to easily access a sensor network, without disrupting the long-term data acquisition. Without concurrent queries, the sensor network remains monopolised by one single application, thus preventing, e.g., other users from taking advantage of already deployed infrastructure.

The ultimate goal of this project is to provide a declarative interface that completely abstracts from the underlying sensor network as it is the case for current traditional databases. In order to reach this goal the middleware platform has to be extended. Further research will focus on aggregate queries and queries with `GROUP BY` and `HAVING` clauses. Additionally we will explore techniques for post-processing sensor data, such as outlier-detection and fusing data from several sensors.

7. 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. Proceedings of the Second IEEE Workshop on Embedded Networked Sensors (EmNetS-II), May 2005.

[3] Szewczyk R., Polastre J., Mainwaring A., Culler D.: Lessons From a Sensor Network Expedition. Proceedings of the First European Workshop on Sensor Networks (EWSN), January 2004.

[4] Szewczyk R., Mainwaring A., Polastre J., Anderson J., Culler D.: An Analysis of a Large Scale Habitat Monitoring Application. Proceedings of the 2nd international conference on Embedded Networked Sensor Systems, 2004.

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

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

[7] Jeffery S. R., Alonso G., Franklin M. J., Hong W., Widom J.: Virtual Devices: An Extensible Architecture for Bridging the Physical-Digital Divide. Technical Report UCB/CSD-5-1375, University of Berkeley, March 2005.

[8] Jeffery S. R., Alonso G., Franklin M. J., Hong W., Widom J.: A Pipelined Framework for Online Cleaning of Sensor Data Streams. ICDE 2006.

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

[10] Deshpande A., Guestrin C., Madden S. R., Hellerstein J. M., Hong W.: Model-Driven Data Acquisition in Sensor Networks. Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004.

[11] Yao Y., Gehrke J.: Query Processing for Sensor Networks. Proceedings of the 2003 CIDR Conference.

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

[13] Ganeriwal S., Kumar R., Srivastava M. B.: Timing-sync Protocol for Sensor Networks. Proceedings of the 1st internaltion conference of Embedded Sensor Systems, 2003.

[14] Aakvaag N., Mathiesen M., Thonet G.: Timing and power issues in wireless sensor networks — an industrial test case. Parallel Processing, ICPP Workshop, June 2005.

[15] Hill J., Culler D.: Mica: a wireless platform for deeply embedded networks. IEEE Micro Volume 22, Issue 6, pp. 12–24, November-December 2002.

[16] MICA2 Wireless Measurement System: Data Sheet. Crossbow Technology Inc. `http://www.xbow.com`