ETH zürich

ETH Library

# Optimization of concurrent queries in wireless sensor networks

**Report**

**Author(s):**
Müller, René; Alonso, Gustavo

**Publication date:**
2012

**Permanent link:**
https://doi.org/10.3929/ethz-a-006821058

**Rights / license:**
In Copyright - Non-Commercial Use Permitted

**Originally published in:**
Technical Report / ETH Zurich, Department of Computer Science 589

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

# Optimization of Concurrent Queries in Wireless Sensor Networks (Technical Report TR-589) *

René Müller
muellren@inf.ethz.ch

Gustavo Alonso
alonso@inf.ethz.ch

Systems Group, Department of Computer Science, ETH Zurich, Switzerland

## ABSTRACT

The lessons learned so far from a variety of deployments of wireless sensor networks is that they are still expensive and complex systems to deploy, run, and maintain. One way to make sensor networks more cost effective is to support concurrent queries rather than a single static query from a single application as it is today the case in most systems and deployments. To this end, in this paper we describe how to perform multi-query optimization in wireless sensor networks. The optimizer we propose takes a constant stream of queries as input and dynamically determines what is the best execution plan (or small set of plans) to run at the sensors while still being able to answer (a potentially much larger set of) user queries. As queries arrive and depart, our optimizer tries to merge and split the queries to optimize the work done at the sensor level. It then processes the resulting streams to answer all running user queries. The paper discusses the data model, query operators, cost models, query optimization strategies, and data stream operators we use.

## 1. INTRODUCTION

### 1.1 Background and Problem Statement

Existing work in query processing for wireless sensor networks (WSNs) does consider the possibility of concurrently running more than one query in the system [24, 14, 5]. However, the majority of the deployments and much of the research work focus on single application (single query) systems (e.g., [15, 10, 8, 21]). Interestingly, the current cost of a sensor network deployment is still very high and it is unlikely that it will significantly decrease in the near future. For instance, [10] report on a 100+ node deployment in an agricultural project that requires one full year of planning and a budget in excess of 400,000 US dollars. In such a deployment, economically, it would make sense to design the system so that it could efficiently support concurrent queries.

The difficulty with multi-query optimization is that while in many cases merging queries is a good strategy [16, 23], there are enough cases where alternatives like query splitting, and query rewriting are better options. In fact, an important contribution of this paper is showing that multi-query optimization in sensor networks needs to include more than just query merging. In this paper we explore the design space for a query optimizer whose input are SQL queries and its output a set of execution plans (e.g., programs) to be run in a sensor network. Queries can be submitted and withdrawn at any time. The optimization criterion we use in this paper is the overall energy consumption. In the paper we study mechanisms such as:

**Merging** Combining two or more queries into a single execution plan. Then extracting the results for each query from the combined data stream.

**Splitting** Dividing a query into sub-queries so that the sub-queries can either be answered from already existing result data streams or better merged with other queries. Then reconstructing the result data stream from all the different pieces.

**Parallelizing** If the overlap of queries is too small it can be better not to merge queries but to run them separately.

These mechanisms are applied to implement the following optimization strategies: *min-execution cost*, *delta-plan* and *single-plan*. We also study the min-execution cost policy in great detail as it is the one that works best under a wider range of query loads.

The contributions of the paper are (1) a precise and detailed discussion of the query merging problem in sensor networks; (2) determining at what multi-programming level merging is cost effective; (3) a novel and efficient algorithm for cost-based multi-query optimization that significantly improves the state of the art.

### 1.2 Architecture

The query optimizer described in the paper has been implemented and tested in SwissQM [17, 18]. The architecture comprises two levels: the *basestation* and the *execution*

*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.

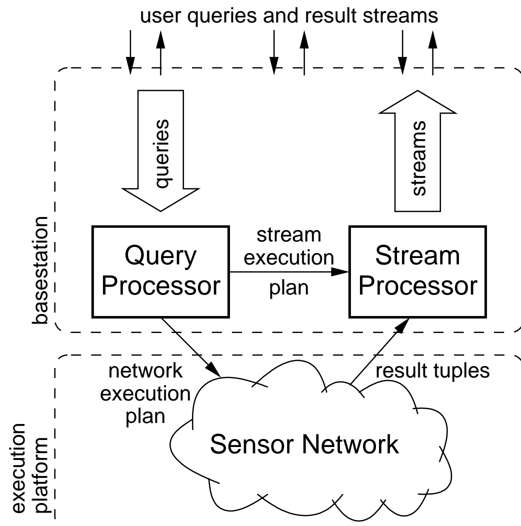



**Figure 1: Architecture overview of Query Processing system**

*platform*. The basestation takes user queries in SQL as input and returns data streams answering those queries. The basestation contains two modules, the *query processor* and the *stream processor* (Fig. 1). The query processor contains the query optimizer described above and all the machinery needed to produce *network execution plans* and *stream execution plans*. The network execution plans are the programs sent to the sensor nodes. The stream execution plans describe the operator pipeline to be executed by the stream processor to extract the answers to the user queries from the stream(s) of result tuples. The sensor network takes network execution plans as input and returns streams of result tuples. The work we describe can be implemented on top of any sensor network platform that allows clients to dynamically submit programs to be executed. Such systems are typically implemented as virtual machines running on the nodes of the network (e.g., [17, 11]). We do not make any further assumptions about the language of the execution plans or the underlying programming model. We just assume (as it is the case in all systems we are aware of) that the language provided is expressive enough to support the common operations performed in a sensor network (sampling sensors, sending and receiving data, basic arithmetic operations, in-network aggregation instructions, etc.).

## 1.3 Structure of the Paper

In the paper we describe the model used to express queries, the query algebra, and the set of query operators (Section 2). We also present a realistic cost model and discuss the optimization strategies that can be followed in a sensor network (Sections 3 and 4). We have also conducted extensive experiments that shed light on at what level of multi-programming each of the possible strategies is cost-effective (Section 5).

# 2. DATA MODEL AND OPERATORS

The basic data schema is a *virtual stream*, similar to that used in existing work [14, 24]. The stream contains tuples with timestamp indicating when the tuple was issued and a set of attributes. The virtual stream can be obtained from the sensor network with the query (the so called *universal query* [16]):

```
SELECT * EVERY Δt
```

$\Delta t$ is assumed to be the lowest possible sampling period. We also assume the sensor network implements a synchronization mechanism that allows to make correct correlations on the timestamps produced by every node.

## 2.1 Filtering Operators

In addition to the standard selection and projection operators, our approach also uses temporal operators that filter on the time axis.

**Sampling Operator:** The sampling operator $\tau$ is a temporal operator that materializes the virtual stream at the specified sampling intervals. For example, $\tau_{6\mathrm{s}}()$ generates a tuple stream that contains readings of all sensors of every node sampled once every six seconds.
**Rate Conversion Operator:** The rate conversion operator $\rho$ is a temporal operator that changes the sampling rate of a tuple stream. For lack of space we consider only down-sampling of the tuple stream (increasing the rate of the tuple stream is possible by interpolation [19] or model based sampling [3]). It has been shown that down-sampling in sensor networks is non-trivial due to inaccurate timers and jitter [16]. The same work proposes several ways to solve the problem. In this paper we assume that the necessary mechanisms for correcting these problems are in place. Hence, down-sampling is implemented as a simple rate-based m:1 sampling, e.g., $\rho_{m:1}$ implies that only every m-th incoming tuple is forwarded.

**Projection and Selection Operators:** As with traditional query algebra, we use projection $\pi$ and selection $\sigma$ operators to remove sensor attributes and to apply filter predicates. There is one significant difference, though. In sensor networks data is generated by sampling and not read as records from disk. Thus, for energy reasons, the sampling operator and the first projection operator are always fused in the execution plan. Consider, for example, the query

```
SELECT node, temp EVERY 6s
```

which can be represented as $\pi_{node,temp}\left(\tau_{6\mathrm{s}}()\right)$. The dynamics of selection and projection are different than in conventional query algebras. The reason is that selection and projection are not made over real attributes in a table but over sensors that need to be sampled. Hence, projection and sampling are fused such that only the required sensors need to be sampled.

## 2.2 Aggregation Operators

In sensor networks, aggregation can happen as temporal or spatial operators [7].

**Window Operator:** The window operator is a temporal operator. It can be implemented either as a *sliding window*

(SWINDOW or $\omega_{s,N}$) or a *tumbling window* (TWINDOW or $\omega_{t,N}$), with $N$ representing the size of the window. The supported operators over windows include MAX, MEAN, MIN and SUM.

**Fusion Operator:** The data fusion operator $\mu$ performs spatial aggregation, i.e., it aggregates data from several sensor nodes. However, it does not keep history state beyond the current sampling interval. The $\mu$ operator performs traditional in-network aggregation [13]. The operator can compute one or more aggregates. As in the *group by* clause of traditional SQL, the tuples can be grouped by one or more attributes. In this paper we consider the SQL aggregates AVG, COUNT, MAX, MIN, STDDEV, SUM, and VARIANCE.

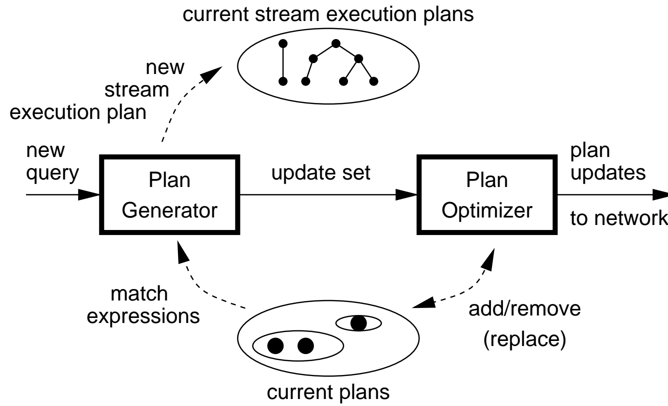



**Figure 2: Components of the Query Processor**

# 3. QUERY PROCESSING MODELS

## 3.1 Query Processor Overview

Submitted queries are passed to the *query processor* which then generates *network execution plans* and *stream execution plans*. Network execution plans are submitted to the sensor network and each one of them produces a data stream. These result data streams are processed by the stream execution plans that extract the final result streams for the user queries.

The query processor consists of a *plan generator* and a *plan optimizer*. Figure 2 illustrates these two components and how they operate. New queries arrive at the plan generator. The plan generator uses the new query and the set of *current plans* (the plans currently running in the network) as input and outputs a stream execution plan and an *update set*. The update set specifies what additional data needs to be requested from the network to answer the new query (note that the update set can be empty). The update set is given as an input to the plan optimizer. The plan optimizer takes the update set and the set of current plans and produces a new set of current plans. It also forwards any changes to the current plans to the sensor network for execution.

Query optimization happens in two stages. The plan generator tries to answer the new query by using the result data streams already being produced. If it cannot, it generates the update set. The plan optimizer uses different cost model based strategies (Section 4) to minimize the cost of propagating changes and running the set of current plans.

Note that the plan optimizer does not need to generate network execution plans that acquire exactly the data asked by the queries. Depending on the optimization strategy, there might be *redundant data* (two separate plans return overlapping data sets) and *orphan data* (data obtained from the network for which there is currently no user query). The latter reduce the cost of removing plans if they are no longer needed. Leaving them around for a while may be better than removing them and having to re-insert them shortly afterwards.

## 3.2 Cost Model per Tuple

In this paper we aim at minimizing the average power consumption over all nodes. We start by calculating the cost of acquiring and transmitting a tuple to the basestation. The energy consumption $E_T$ of a sensor node for a single tuple is given by the following expression.

$$E_T = \underbrace{\sum_{s_k \in S} t_{s_k} P_{s_k}}_{\text{sampling}} + \underbrace{t_{rx} P_{rx}}_{\text{reception}} + \underbrace{t_{tx} P_{tx}}_{\text{transmission}} + \underbrace{t_{cpu} P_{cpu}}_{\text{CPU active}} + \underbrace{t_i P_i}_{\text{idle}} \quad (1)$$

The expression captures the cost of sampling, receiving and transmitting, running the CPU, and the power consumed while idle. The cost of sampling a sensor is the product of the time $t_{s_k}$ to take a sample and the power consumption $P_{s_k}$ of each sensor. The overall sampling cost is the sum over all sensors sampled. For message transmission and reception we use the time the transmitter and receiver circuits are active multiplied by the power consumption for sending and receiving. The same applies to the CPU. The idle power consumption $P_i$ is the power used by the node when the radio is powered-off and the CPU is in the lowest power mode.

Table 1 shows the values for the parameters we have obtained through shunt-measurements on the Tmote Sky sensor platform. For some of these parameters, what we have measured is the minimum value. The real values of these parameters are affected by the software running on the node. For instance, the receiving time $t_{rx}$ is application dependent (it depends on the duty cycle, message length, and size of the network). In our system we have measured the real value to be 50 ms.

Table 1 indicates as it also noted throughout the literature that energy consumption is dominated by the transmission costs. In fact, the major power drain occurs in the radio receiver[1]. For the model we assume a MAC layer that performs the necessary duty-cycling of the receiver. Thus, without any significant loss of accuracy, we can simplify the model by considering only the cost of transmitting tuples

[1]The sensors of the Tmote Sky node platform we are using do not have a significant energy consumption. For different sensors (e.g., gas sensors) sampling cost may no longer be negligible. In this case, the sampling cost has to be included in the parameter $C_a$.

**Table 1: Measured energy model parameter for the Tmote Sky platform**

| Parameter | | Value | Unit |
|---|---|---|---|
| Broad-band light sensor | $P_{tsr}$ | 4.94 | mW |
| | $t_{tsr}$ | 17.5 | ms |
| Narrow-band light sensor | $P_{par}$ | 5.01 | mW |
| | $t_{par}$ | 17.48 | ms |
| Temperature sensor | $P_t$ | 4.32 | mW |
| | $t_t$ | 220.4 | ms |
| Humidity sensor | $P_h$ | 4.28 | $\mu$W |
| | $t_h$ | 72.36 | ms |
| Radio transmitter | $P_{tx}$ | 61.1 | mW |
| Sending message (41 byte) | $\bar{t}_{tx}$ | 9.63 | ms |
| Radio receiver | $P_{rx}$ | 63.4 | mW |
| Receiving message (min) | $t_{rx} \geq$ | 2.02 | ms |
| CPU active | $P_{cpu}$ | 8.76 | mW |
| Idle power | $P_i$ | 2.64 | mW |

and disseminating network execution plans. Although our implementation uses messages of different sizes and supports tuple bundling in one message, in the cost model we assume each tuple costs at least one message. Some tuples are too long to fit into one message, hence, the cost $c(T)$ for sending a single tuple $T$ that contains $n$ attributes is given as follows:

$$C(T) = C_m \left\lceil \frac{n}{8} \right\rceil + C_a n \tag{2}$$

$C_m$ is the fixed cost for sending a message. In our system the fixed costs are the 25 header bytes of for each full 41 byte result message. $C_a$ is the cost of sending a single 16-bit attribute. The fraction $\frac{1}{8}$ comes from the maximum number of attributes we can fit into a message (in other systems and configurations this value may be different). $C_m$ and $C_a$ can be inferred from the bit-transmission cost. With our hardware we measured $C_m \approx 359\,\mu$J and $C_a \approx 28.7\,\mu$J. In the following, we take the simplifying assumption that a tuple is not split across multiple messages, i.e., $n \leq 8$. As an approximation we use $C_m \approx 12C_a$. All energy calculations are done in units of $C_a$.

## 3.3 Overall Cost Model

Sensor networks typically resort to multi-hop routing to transmit messages to the basestation. This is the basis for in-network data aggregation algorithms, e.g., [13].

There are several strategies to define the routing topology. For instance, in Directed Diffusion [6] the routing depends on the data and task at hand. To calculate the cost model we assume a tree routing topology that is independent of the queries and data collected. There are many possible tree topologies (Figure 3). The actual topology determines the number of messages required to get a tuple to the basestation. For the $N$-node topologies in Figure 3, the total number of messages is of the order of $O(N^2)$ for the chain, $O(N \log(N))$ for the binary tree, and $O(N)$ for the star. This overhead changes if in-network aggregation is used. If the aggregation state, i.e., the summary data that is used to compute the value of the aggregate, is of constant size, which is the case for non-holistic aggregates as described in [13], the number of messages that have to be sent is proportional

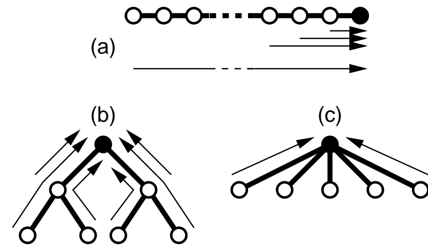



**Figure 3: Different collection tree topologies: (a) chain graph, (b) binary tree, (c) star graph**

to the number of edges in the routing graph, thus, $O(N)$.

To derive a cost model for the average power consumption over all nodes, we need to distinguish between spatial aggregation and non-aggregation queries. In the following, $t_p$ is used for the sampling interval, $C_T$ is a parameter describing the topology, and $N$ is the number of nodes in the network. With Equation (2) as a basis, Equation (3) describes the costs for the non-aggregation case, whereas Equation (4) is used for aggregation queries. The execution cost $C(p)$ for a given plan is the total electrical power used by the sensor nodes (physical units mW) to execute that plan:

$$C(p) = C_T \frac{1}{t_p} \left[ C_m + C_a n \right] \tag{3}$$

$$C(p) = (N-1) \frac{1}{t_p} \left[ C_m + C_a n \right] \tag{4}$$

In [23] similar equations are proposed. However, because they cannot determine the actual topology, the cost model they use is based only on Equation (4) (the lower bound). In our system we can estimate the value of the $C_T$ parameter as follows. A node $h$ hops away from the basestation leads to $h$ messages being sent towards the basestation. For the average energy consumption one can consider the average hop-distance $\bar{h}$ of a node. Together with the number of nodes, the topology parameter is then $C_T = N\bar{h}$, i.e., the average number of hops to the basestation multiplied by the number of nodes. In modern sensor networks the topology is typically dynamic. In our system we can obtain the current value $\bar{h}$ directly from the network by executing the following query in the background at a sufficiently large frequency to react properly to changes:

```
SELECT AVG(depth) EVERY 1min
```

This value is used by the query optimizer to constantly keep the cost model up to date. In Section 4.7 we extend this cost model to include query selectivity.

## 3.4 Plan Generator

A new query is sent to the plan generator after it has been parsed. The generator looks up all possible subexpressions of the query in the set of current plans. When no match is found, the corresponding expression is added to the update set which is then passed to the plan optimizer.

Ignoring selective queries (queries with filter predicates are described later in Section 4.7) a query can be regarded as a collection of expressions $\{E_1, E_2, \ldots\}$. Such an expression can either be a stream operator (as presented in Section 2) or an arithmetic expression. Every expression has an additional property in our stream processor, the *tuple rate* $r$. It determines how often the operator is expected to produce tuples. We explicitly note the rate $r$ together with the expression as an *annotated expression* $E@r$.

*Matching* expressions from a query with the ones from the execution plans involves comparing both expression subtrees and their rates. We define a binary matching relation $\prec$ for annotated expressions:

$$E_i@r_i \prec E_j@r_j \quad \iff \quad \big(E_i = E_j\big) \land \big(r_j \mod r_i \equiv 0\big)$$

According to this definition $E_j@r_j$ *matches* $E_i@r_i$ if, and only if, it contains the same subexpression and produces tuples at an integer multiple rate of $E_i@r_i$, i.e., a superset of the tuples of the left hand side, hence the $\prec$ symbol. Note that $\prec$ is not symmetric.

We can now provide a more formal definition of a query and a network execution plan. A query $q$ is a set of annotated expressions: $q = \{E_1, E_2, \ldots, E_k\}@r$. This notation emphasizes that all expressions of the query have the same rate $r$, determined by the *every* clause. The set is computed from the operator tree and it contains one element for each expression in the *select* clause.

A network execution plan is also a set of annotated expressions: $p = \{E_1, E_2, \ldots, E_k\}@r$. The difference to a query is that each execution plan maintains a list of queries that currently use data generated by the plan. This list can be used as a reference counter of the query. When the list is empty a plan may be removed. We use $\mathcal{P} = \bigcup p$ to denote the set of all plans $p$.

Algorithm 1 computes the update set for a query $q$ considering the set $\mathcal{P}$ of currently executing plans. The function ComputeUpdateSet starts with an empty update set $U$ and then iterates over all expressions of the given query $q$ and calls MatchSubExpr to recursively match the subexpressions. MatchSubExpr returns a pair $(M, U)$ where $M$ contains the plans associated to those subexpressions that could be matched by any expression from the current set of plans. $U$ contains the unmatched subexpressions. The function first checks if the entire tree $E@r$ is matched by some plan (line 11). If not, the function is recursively applied to the subexpressions (line 16). The matching sets $M_i$ and update sets $U_i$ obtained from each subexpression tree are combined (line 17). If no match was found ($M = \emptyset$) the entire subexpression is added to the update set and returned (line 19). Otherwise, the union of the matching plans $M$ and the update set $U$ is returned (line 20). In ComputeUpdateSet the query $q$ is registered with all matching plans $p \in M$. The update set passed to the query optimizer is the union of the update sets $U_i$ obtained for each expression.

While matching the expressions of the update set with the current plans, the plan generator also creates a stream execution plan. For each match found, the attribute in the result stream of the corresponding network plan is extracted

**Algorithm 1** Computing the Update Set

1: **function** ComputeUpdateSet($q, \mathcal{P}$)
2: $\quad U := \emptyset$
3: $\quad$**for all** $E@r \in q$ **do**
4: $\quad\quad (M_i, U_i) :=$ MatchSubExpr($E@r, \mathcal{P}$)
5: $\quad\quad$ subscribe $q$ with all plans $p \in M_i$
6: $\quad\quad U := U \cup U_i$
7: $\quad$**end for**
8: $\quad$**return** $U$
9: **end function**
10: **function** MatchSubExpr($E@r, \mathcal{P}$)
11: $\quad$**if** $\exists p \in \mathcal{P} : \exists E_j@r_j \in p : E@r \prec E_j@r_j$ **then**
12: $\quad\quad$**return** $(\{p\}, \emptyset)$ $\quad$ ▷ match found in plan $p$
13: $\quad$**else**
14: $\quad\quad M := \emptyset, \quad U := \emptyset$
15: $\quad\quad$**for all** subexpressions $E_i@r_i$ of $E@r$ **do**
16: $\quad\quad\quad (M_i, U_i) :=$ MatchSubExpr($E_i@r_i, \mathcal{P}$)
17: $\quad\quad\quad M := M \cup M_i, \quad U := U \cup U_i$
18: $\quad\quad$**end for**
19: $\quad\quad$**if** $M = \emptyset$ **then return** $(\emptyset, \{E@r\})$
20: $\quad\quad$**else return** $(M, U)$
21: $\quad\quad$**end if**
22: $\quad$**end if**
23: **end function**

and, if necessary, some additional processing applied, e.g., down-sampling or applying a filtering operator. In order to join two result streams, e.g., computing $E_1 + E_2$ from two streams that provide $E_1$ and $E_2$, the tuples are combined using an equi-join. For spatial-aggregation queries the join is performed on the grouping attributes (specified in the *group by* clause). In all other cases, tuples that originate from the same node are joined, i.e., the join is performed on the *nodeid* attribute. We can save communication cost by extracting the *nodeid* attribute from the *origin* field of a result tuple message through the routing layer. Alternatively, the *nodeid* attribute could be added implicitly to every execution plan. However, this enlarges the size of a result tuple.

As an illustration of Algorithm 1 consider the example shown in Figure 4. Given a query consisting of a single expression $E_0@r = E_1@r + \omega_{t,10}(a_1@10r)$. Let $E_1@r$ be an unspecified expression tree, $\omega_{t,10}$ a tumbling window operator with a window size 10, and $a_1$ a sensor. There are currently two execution plans $p_1$, $p_2$ in the system. First, the possible subexpressions are matched with the plans. Since for $E_0@r$ no match was found, the subexpressions $E_1@r$ and $E_2@r = \omega_{t,10}(a_1@10r)$ are analyzed (line 16). A match for the $E_1@r$ in $p_1$ is found, hence, $p_1$ is added to the set $M$ and the recursion stops at this point. For $E_2@r$, no match can be found. Thus, recursion continues to $E_3@10r = a_1@10r$. Also in this case, no match is found. Algorithm 1 considers the entire branch $E_2@r$ to be unmatched and adds it to the update set (line 19). This update set is passed to the optimizer, which then modifies $\mathcal{P}$ such that results for $E_2@r$ are retrieved from the network. This is explained in the next section. The streams from $E_1@r$ and $E_2@r$ are joined in the stream execution plan by adding an addition-operator. Also, the query is subscribed with the existing plan $p_1$.

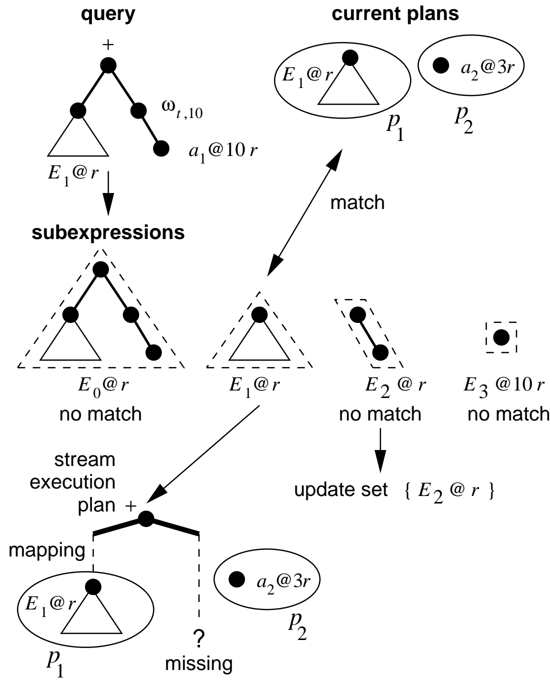



**Figure 4: Plan Generator Example**

# 4. QUERY PROCESSING OPTIMIZATION

Unmatched expressions that are passed to the optimizer by the plan generator need to be included into some execution plan. The optimizer has two choices to deal with the update set. Either it can add it as a new plan or merge it with an existing plan. All changes in the set of execution plans are associated with a cost. Hence, not only the execution of the plans but also the set up and removal costs of the plans have to be considered. There are different strategies the optimizer can apply, depending on the cost-aspect.

## 4.1 Cost of submitting Queries

Before we describe the possible optimization strategies we first state our assumptions on network execution plans:
(1) Execution plans are immutable, i.e., they cannot be modified after they have been generated. The reason is that the execution platform does not support modifications of plans. A modification (besides changing the sampling interval which is possible in, e.g., TinyDB) could in principle consist of large changes. The coordination of these distributed updates is difficult (global snapshot semantics on the state of a plan is required), therefore, we decided not to support updates on plans. The immutability implies that to update a plan, the plan has be replaced.
(2) The execution of a plan results in a power drain $C(p)$ in mW (Section 3.2). The actual energy consumption (Joules) depends on how long a plan is executed.
(3) Setting up a new execution plan gives rise to an energy cost $C_s$ (in Joules). This energy is spent in disseminating the plan through radio messages. These costs depend on the size of the network and the complexity of an execution plan. Our implementation uses a flooding mechanism, hence, a given plan message is retransmitted once by every node. This results in $m \cdot N$ transmissions for a plan that is disseminated with $m$ messages. We measured an energy consumption of $589\,\mu$J for sending a full radio message. Let $m(p)$ denote the number of messages required for representing plan $p$. Then the setup cost is

$$C_s(p) = 589\,\mu\text{J} \cdot Nm(p) \quad .$$

(4) For stopping and removing an execution plan an additional energy cost $C_r$ (in Joules) is defined. In our implementation a plan is removed when a *stop* message is received. $C_r$ is the cost for broadcasting one single single message and is independent of the plan. Thus, $C_r = 589\,\mu\text{J} \cdot N$.
(5) The number of execution plans that can be run concurrently in the sensor network is limited. This limitation is not only due to the limited bandwidth available for the transmission of result tuples but also to the limited CPU/memory on the sensor node. Our implementation on the Tmote Sky sensor nodes can execute up to 8 plans concurrently.
(6) Finally, as already mentioned in the previous section, all annotated expressions $E_i@r$ of a given plan must have the same rate $r$.

Obviously, there is a trade-off between cheap execution costs, i.e., "good" plans that best match the current set of queries, and a low update cost due to less frequent changes of the plan. The goal that is consistent with maximizing the battery life time is to minimize the energy used to process a given query load. The query load is characterized by the set of queries it consists of and the submission and withdrawal times of each query. In a realistic scenario we cannot assume that withdrawal times are known beforehand. However, the execution time of a query plays an important role, e.g., for a long running query it might be worthwhile to update the set of execution plans as the update costs are amortized by the long running queries.

## 4.2 Optimizer Rules

We study different strategies for the optimizer. The optimization strategy determines how the optimizer modifies annotated expressions from the set $\mathcal{P}$ of execution plans and the update set $U$. In general, the optimizer can modify $\mathcal{P}$ based on following rules:

1. If an expression $E_j@r_j$ is not matching expression $E_i@r_i$, i.e., $E_i@r_i \not\succ E_j@r_j$, because only their rates are incompatible while $E_i = E_j$, the rate of both expressions can be adapted to the *least common multiple* $\text{lcm}(r_i, r_j)$. The expressions can then be merged.

$$E@r_i, E@r_j \longrightarrow E@\text{lcm}(r_i, r_j)$$

   In order to adapt the rate of the result streams rate conversion operators $\rho$ have to be added to the stream execution plan. For the stream of $E@r_i$ the downsampling ratio is $\text{lcm}(r_i, r_j)/r_i : 1$.

2. If $E@r$ is a composite expression, i.e., an operator op $E@r = E_k@r_k \,\text{op}\, E_l@r_l$, it is replaced by the subexpressions, i.e., $E_k@r_k$ and $E_l@r_l$. The operator op is added to the stream execution plan such that the two streams are joined into a stream for $E@r$.

3. If $E@r$ is an aggregate expression, remove the aggregate. This is a special case of Rule 2. Example: $\texttt{MAX}(E_1 + E_2)@r$ is expanded into $E1@r + E2@r$. This

transformation has a profound impact on the execution cost, as now the aggregate MAX has to be computed at the basestation and therefore a tuple from every node needs to be sent to the basestation. However, if a stream for $(E_1 + E_2)@r$ is already available no additional costs occur. In fact, the costs are reduced as no processing is required in the network.

4. Combine common subexpressions in $\mathcal{P}$ and $U$. If, e.g., $E_1@r \,\mathrm{op}\, E_2@r$ is in a current execution plan $p$ and $E_1@r \in U$, then plan $p$ can be replaced by a plan $p'$ that contains $\{E_1@r, E_2@r\}$ instead.

## 4.3 Optimizer Strategies

How these rules are applied is determined by the optimization strategy. In this paper we study the following strategies:

**Min-Execution Cost Strategy** (Section 4.4) The optimizer aggressively reorganizes the current execution plan so that the execution cost $C(p)$ is minimized. Costs for the updates, i.e., setting up and removing plans are not considered. This strategy is motivated by the fact that update costs are negligible compared to the execution costs for long running queries, or for queries with low arrival rates.

**Delta-Plan Strategy** (Section 4.5) For each update set a new plan is created. When a new plan is added, the existing plans are not reorganized. This, of course only works up to a maximum number of plans that can be executed concurrently. A plan is removed if no query has subscribed to the plan. It obviously minimizes the update-costs, however, it can lead to orphan data being returned as a long running query might be subscribed with several query plans that, therefore, cannot be removed. These query plans can also return data not required by this query.

**Single-Plan Strategy** (Section 4.6) leads to a single execution plan, such that all queries are answered from a single stream. This approach was suggested in [16]. As new queries are added, the selectivity of the network execution plan decreases until it becomes SELECT * EVERY $\Delta t$ for the smallest possible $\Delta t$. From this point on, adding additional queries does not require any updates in the network. However, in order to include queries with different sampling intervals, the period of the resulting plan has to be set equal to the greatest common divisor of the queries' sampling interval. As the evaluation in Section 5 will show, the resulting sampling interval quickly approaches $\Delta t$, leading to redundant data.

**TTMQO Strategy** This strategy was described by Xiang et al. [23]. We have implemented TTMQO for comparison purposes. In the TTMQO strategy, instead of merging all queries to one single plan, a query $q$ is merged with the "most beneficial" existing plan. The benefit of a plan $p$ is defined as $C(p) + C(q) - C(\{p, q\})$, i.e., the savings when merging $p$ with the query $q$ compared to creating a new plan for $q$ and running it together with $p$. In [23] the authors only propose merging. Splitting a query and assigning it to different plans is not considered. Thus, their approach essentially bypasses the plan generator such that the update set contains all expressions of the submitted query. In the TTMQO strategy, a plan is not immediately removed as soon as the last query that uses data from it is withdrawn. Instead, a plan is kept, thus producing orphan data. The idea is to save update costs as a new query might be submitted that could make use of that plan. When such a plan is removed is determined by an aggressiveness parameter $0 \leq \alpha \leq 1$. In [23] the authors do not describe how to choose that parameter. Their plots seem to indicate that the parameter $\alpha$ does not affect the results significantly.

## 4.4 Min-Execution Cost Strategy

In this section we describe how to map queries such that the execution costs are minimized. The min-execution cost strategy tries to aggressively minimize the execution costs $\sum_{p \in \mathcal{P}} C(p)$ of the plans. Costs for starting $C_s$ and removing $C_r$ plans are not considered. As soon as query is added or withdrawn the set of current plans is recomputed and plans are replaced where necessary. This strategy works best if the query is in the system for a sufficiently long time as long execution times amortize update costs.

Before describing the strategy we motivate why the update costs can be ignored in a first approximation. When comparing the cost for updating a plan with the execution duration, it can be observed that in the worst case a plan is amortized soon after it has produced data as a simple calculation shows (ignoring selective plans and retransmissions). A plan update (broadcasting a *stop* command message and disseminating the new plan $p$ using $m(p)$ messages) leads to $N(m(p)+1)$ transmissions, assuming that each node forwards each message exactly once. A worst case plan in this case is a plan with the lowest possible execution cost, which is either a spatial aggregation plan, or a non-aggregation plan in a star topology (Fig. 3 (c)), leading both to $N$ transmissions per epoch. Hence a plan is amortized after $> m(p) + 1$ epochs. In SwissQM the plan size is only $1 \leq m(p) \leq 5$ ([18]) even for complex queries with simple user-defined functions. Thus, update costs can considered being amortized after having produced data for about 10 epochs. In the common case, the average depth of the tree is $\gg 1$. Then the update cost is amortized even earlier.

Consider a set of queries that leads to the following *annotated expression set*:

$$
\begin{aligned}
E_1 &: \quad (a+b)@2r \\
E_2 &: \quad (a+b)c@4r \\
E_3 &: \quad \omega_{10}(c)@r \\
E_4 &: \quad b@2r
\end{aligned}
$$

In this example $a, b$, and $c$ correspond to sensors whereas $@r$ specifies the requested tuple rate (frequency). The goal is to find the cheapest execution plans for the expression set. This also includes deciding how to split the expressions into execution plans $p_i$, that each produces a stream of tuples $T_i$ at one particular rate $r_i$.

The tree for the expressions $E_i$ can be presented as a connected graph (Figure 5 (a)). In the graph we add an edge (dashed arrows) from a *matching expression* to all expressions it matches. For example, since $(a+b)@2r \prec (a+b)@4r$, we add an edge from $(a+b)@4r$ to $(a+b)@2r$. The doubly encircled nodes are the root nodes of the expressions $E_i$. Values corresponding to these root nodes are requested by

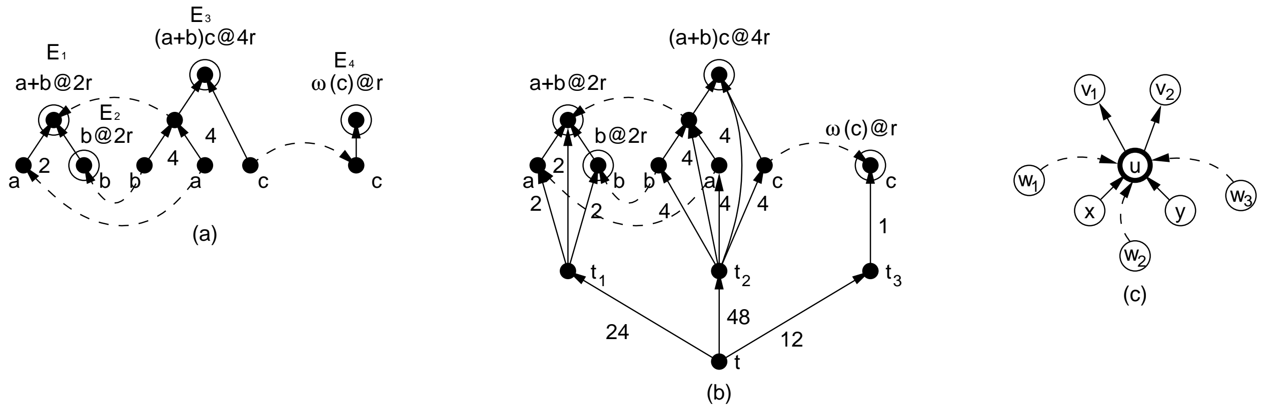



**Figure 5: (a) Graph for set of annotated expressions from the example. (b) Weighted graph model for ILP problem. (c) Construction of inequalities for the ILP problem.**

some user query.

## *Graph Modelling*

The graph from Figure 5 (a) can be extended as shown in Figure 5 (b). Let's call this extended digraph $G = (V, E)$ consisting of a set of vertices $V$ and edges $E$. An artificial root node $t$ is added to the graph. The expressions are grouped by their rate $r_i$. Also, for each group a new node $t_i$ is introduced. Since for each rate a fixed header cost has to be spent regardless of the number of selected attributes, an edge $(t, t_i)$ is added between the artificial root node $t$ and the common group nodes $t_i$. The edge is weighted by the cost $c(t, t_i) = r_i C_m$ and accounts for the header cost $C_m$ when using a plan at rate $r_i$. Additional edges $(t_i, u)$ are added between the group nodes $t_i$ and the remaining nodes $u$ of the expressions. These edges are weighted by the costs $w(t_i, u) = r_i C_a$. The weights reflect the cost of a field in the result message. Note, since $C_m \gg C_a$, as soon as a rate is selected, adding fields to a tuple is relatively cheap. All other edges, including the "matching expression" edges have zero weight. The idea is that computing an expression from an expression that is reached over a zero-weight edge is free, since the operand values can be retrieved from the tuple that is sent to the basestation. When following dashed lines (Figure 5 (b)), costs are actually saved as redundancy in the expression set is exploited.

The goal is to select a subset $S \subseteq V$ of edges such that the corresponding induced graph is connected, and all expression roots $E_i$ can be reached from $t$. For optimality, the minimum cost subset $S$ is of interest. This leads to the following constraint optimization problem:

$$\min_{S} \sum_{(u,v)\in S} c(u, v) \tag{5}$$

## *0-1 Integer Linear Programming*

The optimization problem can be mapped to an *0-1 Integer Linear Programming Problem* (ILP). A 0/1 variable $x_e$ is introduced for each edge $e$. The edge $e \in S$ if, and only if, $x_e = 1$. The objective function is expression (5). Connectivity and reachability from $t$ is modelled using the constraint inequalities. The procedure is as follows:

Consider a node $u$ (shown in Figure 5 (c)). This node represents a binary operator, e.g., $x + y$. Hence, it is connected to the two operand nodes $x$ and $y$. In order to evaluate the expression the edges $(x, u)$ and $(y, u)$ have to be selected, i.e., the operands $x$ and $y$ have to be available. The dashed edges from $w_1$, $w_2$, and $w_3$ are from "matching expressions" that provide $x + y$, at a rate that is an integer multiple of the one requested by $u$. Node $u$ also has two outgoing edges to $v_1$, and $v_2$. The connectivity rules for the inner nodes are given by the following condition.

$$\begin{aligned}(u, v_i) \in S \quad &\Longleftrightarrow \\ &((x, u) \in S) \wedge ((y, u) \in S) \vee \bigvee_{w_j} ((w_j, u) \in S)\end{aligned} \tag{6}$$

In other words, an outgoing edge $(u, v_i)$ is in $S$ if and only if all operand edges $(x, u), (y, u)$ are in $S$ or $u$ is reachable by at least one "matching expression" edge $(w_j, u) \in S$. Note that the condition must hold for each outgoing edge $(u, v_1)$ and $(u, v_2)$. The constraints for expression roots (double circled nodes in Fig. 5) introduce a similar constraint: either all operand edges have to be in $S$ or at least one "matching expression" edge.

In a second step, the set of conditions (6) can be rewritten as linear inequalities by introducing a structural variable $x_{(u,v)}$ for each edge $(u, v)$. The equation corresponding to (6) is:

$$x_{(x,u)} + x_{(y,u)} \;+\; 2\sum_{w_j} x_{(w_j,u)} - 2x_{(u,v_i)} \geq 0 \tag{7}$$

It can easily be verified that inequality (7) corresponds to (6) by substituting the structural variables $x$ by values 0 and 1. The number of variables is equal to the number of edges $|E|$. For each out-bound edge there is a constraint inequality. An additional inequality is obtained for each root node of the expressions. The objective function of the ILP problem to be minimized is expression (5).

## *Example Continued*

The assignment of variables $x$ to edges is shown in Figure 6. For the example, 22 variables $x_1, \ldots, x_{22} \in \{0, 1\}$ are introduced. The weights $w_i$ are defined as indicated indicated in

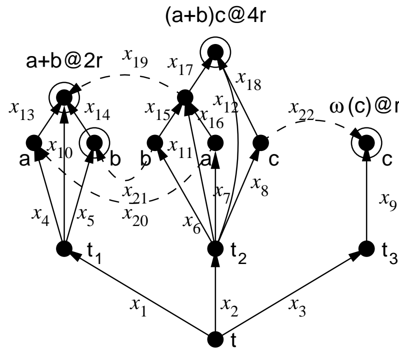



**Figure 6: Variables $x_e = 1$ correspond to selected edges $e \in S$ in ILP**

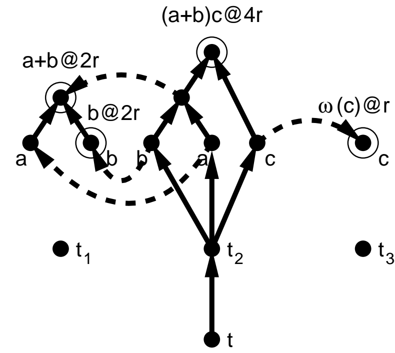



**Figure 7: Induced graph given by selected edges $S$, i.e., edges $e$ where $x_e = 1$**

Fig. 5 (b). The objective function to be *minimized* is

$$z(x_1, \ldots, x_{22}) = \sum_{i=1}^{22} x_i w_i \qquad .$$

The constraints for the expression roots $E_i$ are:

$$\begin{aligned} x_{13} + x_{14} + 2x_{10} + 2x_{19} &\geq 2 \\ x_5 + x_{21} &\geq 1 \\ x_{17} + x_{18} + 2x_{12} &\geq 2 \\ x_9 + x_{22} &\geq 1 \end{aligned}$$

The remaining constraints are:

$$\begin{aligned} x_1 - x_4 &\geq 0 \\ x_1 - x_5 &\geq 0 \\ x_1 - x_{10} &\geq 0 \\ x_2 - x_6 &\geq 0 \\ x_2 - x_7 &\geq 0 \\ x_2 - x_8 &\geq 0 \\ x_2 - x_{11} &\geq 0 \\ x_2 - x_{12} &\geq 0 \\ x_3 - x_9 &\geq 0 \\ x_4 + x_{20} - x_{13} &\geq 0 \\ x_5 + x_{21} - x_{14} &\geq 0 \\ x_6 - x_{15} &\geq 0 \\ x_6 - x_{21} &\geq 0 \\ x_7 - x_{16} &\geq 0 \\ x_7 - x_{20} &\geq 0 \\ x_8 - x_{18} &\geq 0 \\ x_8 - x_{22} &\geq 0 \\ x_{15} + x_{16} + 2x_{11} - 2x_{17} &\geq 0 \\ x_{15} + x_{16} + 2x_{11} - 2x_{19} &\geq 0 \end{aligned}$$

The problem is solved, e.g., using the *GNU Linear Programming Kit (GLPK) solver* and the following optimal integer solution can be found.

$$\begin{aligned} x_2 = x_6 = x_7 = x_8 = x_{13} = x_{14} = \ldots = x_{21} = x_{22} &= 1 \\ x_1 = x_3 = x_4 = x_5 = x_9 = x_{10} = x_{11} = x_{12} &= 0 \\ z(x_1, \ldots, x_{22}) &= 60 \end{aligned}$$

The selected edges $S$ and the induced graph are shown in Fig. 7. The solution contains the two "matching subexpression" edges $x_{20}$ and $x_{21}$ for the operands of $a + b@2r$ as well as the edge $x_{19}$ for the expression itself. This can be considered as sort of redundancy, but since the "matching subexpression" edges have zero weight they do not add an additional cost. However, the solution is plausibly minimal in terms of the cost $z$.

In a second stage the graph is traversed from $t$ in order to determine the expressions that are executed in the network, i.e., the expressions for the tuples fields. This is done by traversing the graph from $t$ and stopping at a node that either is the root of an expression $E_i$ or has an outgoing "matching expression" edge $\in S$. The expression rooted at this stop node is then used for that particular tuple field. The expressions are then assigned to plans by their common rate $r_i$. Concluding the example, the expressions $E_1, \ldots, E_4$ can be computed using a single plan $p = \{a, b, c\}@4r$.

### *Spatial Aggregation Queries*

For aggregation queries the mapping from the annotated expressions to the graph has to be extended. As shown by Equations (3) and (4) (in Section 3.3) spatial aggregate plans have smaller costs, as only a single message has to be sent over an edge in the data collection tree. Therefore, both the fixed costs as well as the attribute cost depend on whether the spatial aggregation is performed in-network or at the basestation.

The graph model has to reflect these two choices. The model is extended as follows: First, the cost in the graph, i.e., the edge weights, are specified as the total cost including the topology parameter $C_T$, rather than just $r_i C_a$ and $r_i C_m$ (topology independent). Second, for each rate group $r_i$ that contains at least one spatial aggregation expression the group node $t_i$ is split into two nodes $t_i$ and $t_{i,\text{agg}}$. If the latter is reachable over an edge $\in S$ the plan will be executed using a spatial aggregation plan. The weights for the edges $(t, t_i)$ and $(t, t_{t,\text{agg}})$ represent the header cost for running a non-aggregation plan at rate $r_i$ and, respectively, an aggregation plan at the same rate. Thus, using the cost metrics from Equations (3) and (4) the weights are:

$$\begin{aligned} w(t, t_i) &= r C_T C_m \\ w(t, t_{i,\text{agg}}) &= r(N-1) C_m \end{aligned}$$

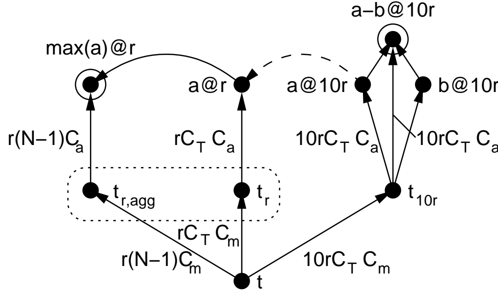



**Figure 8: Graph for spatial aggregation queries**

Next, the aggregation expressions are duplicated in the graph. Remember, that the second level edges $(t_i, a)$ to attributes model the costs of a field in a result tuple, which is also different for spatial aggregation expressions $f(a)$. Note, in an aggregation plan, leaf nodes are always aggregation expressions $f(a)$ and never the single attribute $a$, as attributes cannot be part of an aggregation plan by definition. The weight costs are as follows:

$$\begin{aligned} w(t_i, a) &= rC_T C_a \\ w(t, f(a)) &= r(N-1)s_f C_a \end{aligned}$$

Here $s_a$ is used to denote the size of the aggregation state of aggregate $f$. For example: 1 for `MIN`, `MAX`, `SUM`, 2 for `AVG`, 3 for `VARIANCE`, and `STDDEV`, etc. Finally, the graph is extended by the corresponding edges from the non-aggregation component to the aggregation component.

As an example consider the following two expressions:

$$\begin{aligned} E_1: &\quad \max(a)@r \\ E_2: &\quad (a-b)@10r \end{aligned}$$

In this example, $E_1$ is a spatial aggregation expression. The optimizer has to decide whether the expressions are to be computed using two separate plans $\max(a)@r, \{a, b\}@10r$, or whether they are merged into a single non-aggregation plan $\{a, b\}@10r$. The graph model is shown in Fig. 8. Note that for the aggregation expression $E_1$ the group nodes $t_r$ is split into $t_r$ and $t_{r,\mathrm{agg}}$ (dotted rectangle). Also observe the different weights for $(t, t_r)$ and $(t, t_{r,\mathrm{agg}})$. A "matching expression" edge from $a@10r$ to the $a@r$ node of the non-aggregation component of $E_1$ is added as well.

As it can be easily verified that as long as

$$C_T > \frac{1}{10}(N-1)\left(1 + \frac{C_m}{C_a}\right)$$

it is more efficient to run the two plans $\max(a)@r, (a-b)@10r$ than to run a single non-aggregation plan $\{a, b\}@10r$.

### *Limit on the number of concurrent plans*

The number of concurrent plans that are generated is equal to the number of selected edges $(t, t_i) \in S$. If the sensor network imposes a limit of at most $N_{\mathrm{Pmax}}$, this limit can be enforced by adding another inequality

$$\sum_i x_{(t,t_i)} \leq N_{\mathrm{Pmax}} \quad .$$

If this constraint is added, it is in principle possible that the linear program has no feasible solution, i.e., the constraints lead to an empty feasible region. In this case, the optimizer learns that a cost-optimal solution is not possible. Instead, a sub-optimal solution is sought that uses fewer plans but inevitably leads to data not requested by any query. The optimizer iteratively reduces the number of rate groups and, hence, possible plans, by combining groups $t_i$ and $t_j$ until a feasible solution is found. Assume that in group $i$ the set of selected sensors attributes is $A_i$ and $A_j$ for group $j$ then the amount of superfluous data obtained when merging the groups is estimated as

$$\frac{\mathrm{lcm}(r_i, r_j)}{r_i}|A_i \backslash A_j| + \frac{\mathrm{lcm}(r_i, r_j)}{r_j}|A_j \backslash A_i| \quad .$$

The optimizer now greedily combines groups that introduce the least amount of superfluous data until a feasible solution is found.

This solution for solving the min-execution cost optimization problem is NP-hard [9]. We believe that there is no efficient solution for this problem. Nevertheless, for the number of queries that is reasonable to run in a wireless sensor network, this approach is feasible in practice.

## 4.5 Delta-Plan Strategy

For each update set $U$ that is generated when a query is added a new plan will be created. However, plans are not reorganized once they are started. The idea for this strategy is to include the update set with as few updates as possible, which is clearly the case when the entire set is added as a new plan. A plan is always removed as soon as no more query is subscribed to an expression of this plan.

This strategy has two problems. First, a plan might produce large amounts of orphan data. Consider, for example, a plan producing tuples with ten fields. All queries are removed until the last query, which only extracts one single field of the tuples, resulting in nine orphan fields. However, if this query is long running the plan is not removed and since it is not reorganized, the orphan data can significantly increase the cost. In TTMQO [23] the authors consider this as an advantage, as new queries might request one of the orphan fields. In that case, the queries can be answered directly without any update costs. In fact, in [23], unlike in the delta-plan approach, even plans that contain only orphaned fields are not always removed. They are intentionally left in the sensor network for this reason. TTMQO provides a tuning parameter that allows adjusting how aggressively orphaned plans are removed.

Second, as updates lead to new plans, the number of concurrent execution plans rapidly increase. Thus, this approach works only up to the maximum number of concurrent plans. If the limit is reached, the plans have to be reorganized, e.g., using the min-execution cost strategy.

## 4.6 Single-Plan Strategy

This strategy was described in [16] and is motivated by the notion of the *universal query* that returns all attributes of the sensor nodes in the entire network at the highest possible rate. As new queries arrive, the selectivity of the single execution plan is gradually increased towards the universal query. If a query is withdrawn, a plan is gradually made less selective, i.e., more specific to the queries.

This approach works well for a workload with many concurrent queries, in particular if there is large overlap in the requested attributes. The disadvantage is that result rate is the least common multiple of all user queries. For example, if the sampling intervals specified in the queries are relative prime, the greatest common divisor is one, leading to a plan with the shortest possible sampling interval $\Delta t$. [16] suggests using introducing a slack in the resulting sampling interval (a tolerant sampling interval). The optimizer then can freely pick any sampling intervals within the tolerance window specified such that the sampling rate of the resulting execution plan is minimized.

## 4.7 Queries with Predicates

In this section we extend the expression annotation introduced in Section 3.4 to include filter predicates, i.e., queries with $\sigma$ operators. Just as with the sampling rate $r$, we also annotate the selection predicate $p(E_j)$ with the expression. Multiple predicates are represented in conjunctive normal form. The resulting annotated expression is:

$$E@r|p(E_j) \wedge \ldots \wedge p(E_k)$$

For example, for the single query the annotated expressions in the execution plan are:

```
SELECT nodeid, (light+lightpar)/2
WHERE humid<50 EVERY 1s
```

$$p = \{\texttt{nodeid}@1\text{s}|(\texttt{humid} < 50), (\texttt{light} + \texttt{lightpar})/2@1\text{s}|(\texttt{humid} < 50)\}$$

When the matching relation is redefined appropriately, the plan generator does not have to be changed. The matching operator $\prec$ for predicates is redefined as:

$$E_i@r_i|p(E_k) \prec E_j@r_j|p(E_l) \iff E_i = E_j \wedge r_j \mod r_i \equiv 0 \wedge p(E_k) \Rightarrow p(E_l)$$

The superset condition still holds. If $p(E_k) \Rightarrow p(E_l)$ the stream $E_j@r_j|p(E_l)$ is a superset for $E_i@r_i|p(E_k)$. For the plan optimizer we add the following rewrite rule.

5. If an expression $E_i@r_i|p(E_j)$ is bound to a predicate $p(E_j)$ the predicate is removed. The predicate $p(\cdot)$ is then computed at the basestation, i.e., it is added as a $\sigma$ operator to the stream execution plan. However, in order to evaluate the predicate at the basestation, a stream for the predicate expression $E_j$ has to be available, thus, it also has to be added to the expression set together with $E_i$:

$$E_i@r_i|p(E_j) \longrightarrow E_i@r_i,\ E_j@r_i$$

For the query above the `humid` attribute is also added to the rewritten annotated expressions.

$$\texttt{nodeid}@1\text{s},\ (\texttt{light} + \texttt{lightpar})/2@1\text{s},\ \texttt{humid}@1\text{s}$$

The plan generator needs to first decide on the evaluation order of a list of predicates $E@r|p(E_j) \wedge \ldots \wedge p(E_k)$, and whether a predicate is applied in the network execution plan at all, i.e., whether rule (5) should be applied.

Again, the cost model is used. However, since execution plans now can contain selective expressions, the tuple cost depends on the average selectivity $\bar{s}$ of a predicate. Hence, the cost Equations (3) and (4) are redefined as

$$C(p) = C_T \frac{\bar{s}}{t_p}\left[C_m + C_a n\right] \quad (8)$$

$$C(p) = (N-1)\frac{\bar{s}}{t_p}\left[C_m + C_a n\right] \quad . \quad (9)$$

In order to apply this model, the optimizer needs to know the selectivity of the predicates in the execution plans. As stated earlier, we use the *origin* field in the result message to associate a tuple with a sensor node, hence, the selectivity $s$ of a predicate can be estimated at the basestation for every sensor node by counting the number of received tuples. This estimate is inherently affected by message loss, as missing tuples (due to lost messages) cannot be distinguished from filtered tuples. The optimizer uses the average $\bar{s}$ over all nodes, thus reducing the influence of individual losses. Additionally, the decision on whether it makes sense to continuously use a selective plan, and thereby reducing reuse for different queries, is made if the selectivity is very low, e.,g., $< 10\%$. This makes mispredictions due to message loss negligible as the delivery probability of a message is sufficiently high $(0.7 - 0.9)$. In order to determine $\bar{s}$ for selective in-network aggregation plans, the number of fused readings needs to be explicitly counted (e.g., the number of values that contributed to a `MAX` aggregate). This statistical data has to be sent along with the aggregate state. Only for `AVG` and `COUNT` this information can be directly deduced from the aggregate state. Our system currently does not estimate the selectivity of aggregation plans. The optimizer assumes $\bar{s} = 1$ in those cases.

# 5. EXPERIMENTAL RESULTS

## 5.1 Query Workload

In order to assess the performance of the different optimization strategies, a set of queries has to be used. Lacking a benchmark for query processing in sensor networks, we use a set of randomly generated queries. For each query in the workload, a submission time and execution duration is given.

The queries are selected from a considerably large query space consisting of $\approx 2 \cdot 10^{13}$ possible queries. The expressions of the queries are composed using elements that are randomly picked from a predefined set. In this paper we are not considering selective queries as results depend on the actual sensor values. Furthermore, choosing useful predicates that are repeatable despite the varying nature of the sensor readings is difficult. The workload set consists of three different types of queries: (1) spatial aggregation queries,

Table 2: Query Workloads

| Query Workload | WL1 | WL2 |
|---|---|---|
| non-aggregation queries | 50% | 20% |
| spatial aggregation queries | 30% | 60% |
| temporal aggregation queries | 20% | 20% |

(2) temporal aggregation queries and (3) non-aggregation queries. Not all queries from the available space are equally likely to occur in practice. A reasonable assumption is that exotic queries such as

$q_1$ : `SELECT (temp+light/3)*nodeid EVERY 17s`

occur less frequently and that sampling intervals will most likely take values from an "even" range, such as 1 s, 5 s, 15 s, 30 s, 1 min, 5 min, 30 min, 1 h, etc. The query components, number of selection expressions, sensor attributes, and subexpressions are thus selected using a non-uniform distribution. The sampling intervals take one of 17 possible values. We are aware of the fact that sampling intervals that are integer multiples of each other greatly increases reuse of execution plans for multiple queries. Note that constraining the possible sampling intervals actually corresponds to using a slack value as proposed in [16].

As shown in Section 3.3 the cost for running spatial aggregation queries is very different from the cost for non-aggregation queries. We use two different query workloads with different query mixtures (Table 2) to investigate these effects. In workload WL1 non-aggregation queries dominate, whereas WL2 is dominated by spatial aggregation queries. An excerpt of queries from the two workloads is listed in the appendix.

The submission and withdrawal times of each query are determined using two random models. First, we rely on the common assumption that the query arrivals are Poisson distributed. Second, the time a query is run until it is withdrawn by the user is exponentially distributed. The expected number of queries present in the system in steady state can be estimated by Little's Law. Assuming a query arrival rate $r$ and an average execution duration $T$ there are $rT$ queries expected in the systems. This does not consider the ramp-up phase at the beginning and the ramp-down phase at the end after the last query has been submitted. A simple calculation shows that the expected number of queries in the ramp up phase is $rT(1-e^{-\frac{t}{T}})$. For $t \to \infty$ this leads to the asymptotic case of Little's Law. For the ramp-down phase started at $t_0$ the number of queries decrease as $rT(1-e^{-\frac{t_0}{T}})e^{-\frac{t-t_0}{T}}$. The system is ergodic during steady state phase, hence, it is possible to estimate statistic characteristics (averages) using a time average, i.e., by choosing a sufficiently large execution window. In the following, it is thus sufficient to consider a single (large) query set for a given set of parameters.

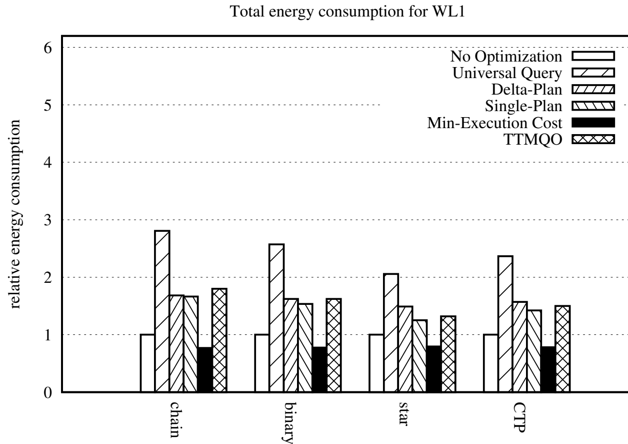
## 5.2 Few Concurrent Queries

We consider a workload of 10 concurrent queries in average. We select the average arrival rate $r = 1/100$ queries/s and the average execution time $T = 1000$ s. This leads to 10 concurrent queries on average. The number of queries in a workload set is chosen sufficiently large such that the duration of the steady state phase is large compared to ramp-up and ramp-down times. When choosing workload sizes of 200 queries the expected duration of the three phases is distributed as follows: 19% for ramp-up, 62% for steady state, and the remaining 19% for ramp-down phase. We consider steady state after $> 9.9$ (expectation) queries are in the system.

We determine the execution costs for the workloads for four different topologies in a sensor network consisting of 32 battery powered sensor nodes and one basestation. The first three are the topologies shown in Fig. 3. The average hop-distance in the chain is $\bar{h} = 16.5$ graph, in the binary tree ($\bar{h} = 3.375$), and in the star graph $\bar{h} = 1$. The fourth topology is obtained from the dynamic tree routing mechanism CTP (Collection Tree Protocol [4]) of the TinyOS 2.0 programming platform using a real deployment. The structure established by the protocol with $\bar{h} = \frac{59}{32} \approx 1.8$ is close to the star topology.

We implemented the min-execution cost, delta-plan, and the single-plan strategy, as well as the TTMQO from [23]. For the latter we set the aggressiveness parameter to $\alpha = 0.5$. We also determined the execution cost when no optimization would be used by plugging-in the duration and topology into the cost model. We use these values as a baseline for comparison. Additionally, we computed the cost of running the universal query `SELECT * EVERY 1s` for the processing duration of entire workload set. For this evaluation we are considering all costs, i.e., costs for execution, setup and removal. Fig. 9 shows the total energy relative to the non-optimization scenario for both workloads. A first observation is that all strategies except min-execution cost lead to a higher energy consumption than running each query with a separate plan. As further analysis showed, for the delta-plan and TTMQO strategies, it turns out there are few active plans and many orphan expressions remaining in the network. For the single-plan strategy, the resulting network plan does have orphan expressions but it runs at a very short interval (1–5s), already close or equal to shortest possible interval. From Fig. 9 one can see that switching to the universal query is not beneficial. Compared with the single-plan strategy, the universal query not only runs at a too high sampling rate, it also returns sensor attributes not asked by any query. The figure also shows the influence of the mixture of the query load as well as the topology.

As expected, in WL2, dominated by spatial-aggregation queries, the improvement of the min-execution cost algorithm is smaller than for WL1. For the chain topology the min-execution cost strategy actually requires slightly (5%) more energy than the no-optimization strategy. The reason is not completely clear to us. Obviously, the update costs are higher for min-execution cost strategy. In the no-optimization strategy each query is mapped into a separate plan which is never replaced. For the chain topology, if a spatial aggregation query is executed by a non-aggregation plan, $\bar{h} = 16.5$ more messages are generated than by an aggregation plan. Thus, deciding to do aggregation outside of the network is most heavily penalized by the chain topology. We also note that the TTMQO has a very high energy consumption, in particular for WL2.



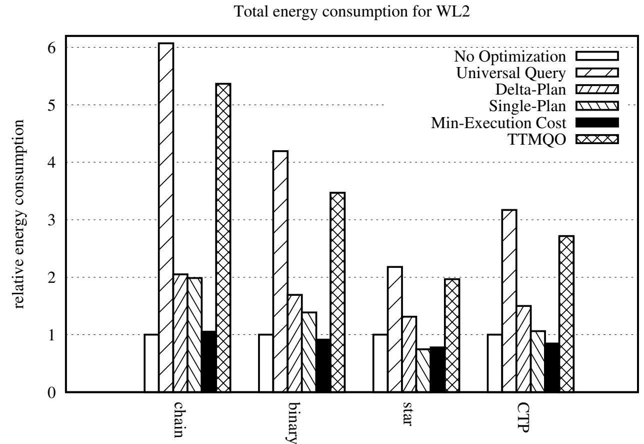




**Figure 9: Total execution costs for workloads WL1 and WL2 using different strategies and topologies. The costs is shown relative to the no optimization case**

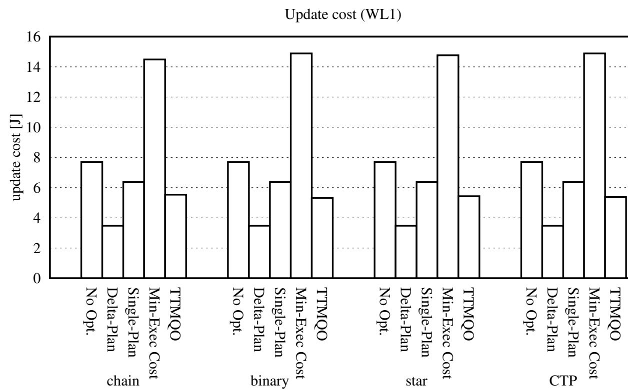



**Figure 10: Absolute update costs for WL1**

Fig. 10 shows the absolute update costs for plan submission and removal when executing WL1[2]. As expected, the aggressive updating of plans by the min-execution cost strategy leads to largest fraction of update costs. The other strategies have lower update costs than the non-optimization case. This is due to the orphan expressions and plans which suppress creation of new plans when new queries arrive.

Fig. 11 shows the maximum number of concurrent plans produced by the individual strategies. As described earlier, the delta-plan strategy may lead to a large number of concurrent plans as they are not reorganized. In TTMQO there are relatively few plans in the network but for a long time, which also implies orphan data and, hence, high energy cost (as seen in Fig. 9).

## 5.3 Many Concurrent Queries

[2]The plot for WL2 is omitted as it has similar update cost, although, as Fig. 9 indicates, different total costs.

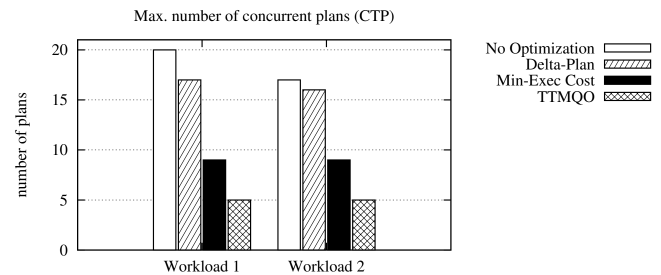



**Figure 11: Maximum number of concurrent plans**

The results for WL1 and WL2 with 10 concurrent queries on average indicate that the min-execution cost strategy performs well. In order to figure out the behavior for smaller and larger sets the number of concurrent queries is varied within the interval 1–50. In this run, the 200 queries from WL1 and WL2 with the same execution duration $T$ are used except that the arrival rate $r$ is chosen such that on average $rT$ queries are in the system.

Fig. 12 shows the total execution costs for each of the optimization strategies for query loads WL1 and WL2. For just a few queries (around less than 5), running the optimizer does not pay off. The queries should be run independently. The delta-plan strategy obviously has a bad performance over the entire interval, although it works better for WL2. Thus, a strategy that solely tries to minimize update costs is not sufficient. The single-plan strategy becomes very efficient as the number of queries increase. The TTMQO strategy performs exponentially bad for less than 20–25 concurrent queries. With $n \geq 23$ queries (WL1) and $n \geq 33$ (WL2) TTMQO outperforms the min-execution cost strategy. This an artefact of the experiments that actually favors TTMQO as the workloads used in the benchmark generate a constant query load. In this case, the orphan expression can quickly be reused by new queries. TTMQO will perform

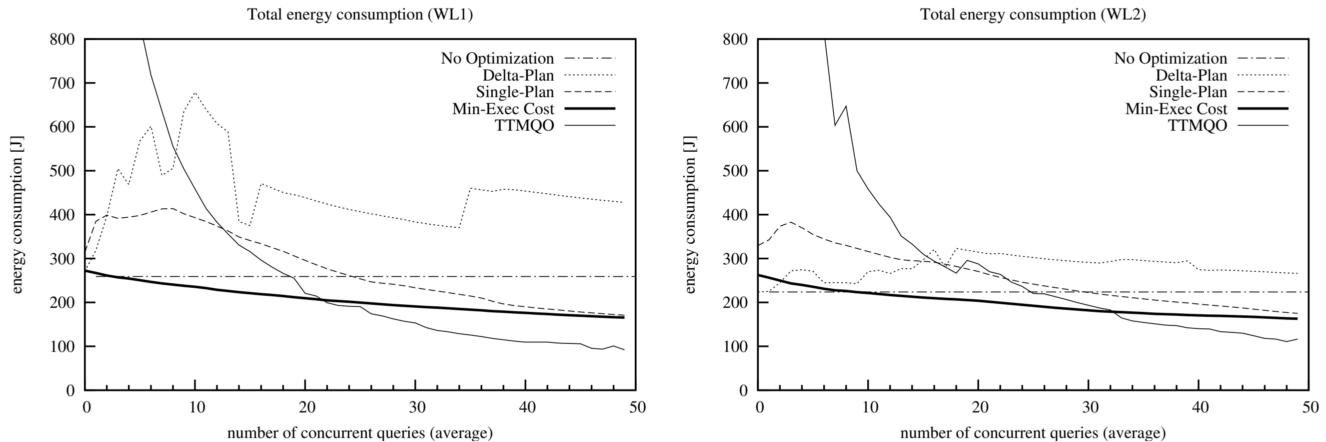



**Figure 12: Total energies for different numbers of concurrent queries (arrival rates) using CTP topology**

worse on bursty loads or when the query arrival rate is not high enough to quickly reuse orphans. When comparing the graphs from Fig. 12 only the min-execution cost plan and the single-plan strategies show the same characteristics for both loads.

As a consequence from these measurements, the optimizer should choose between (1) no optimization, (2) min-execution cost, and (3) eventually the single-plan strategy. The decision can be made based on the number of queries that are present in the system. For a few concurrent queries it should generate a separate plan for each query. For $5 < n \leq 30$ the min-execution cost strategy should be applied. For $n > 30$, the best strategy is probably the single-plan as it does not involve any optimization overhead. TTMQO can be used for a large number of queries if the load is of the right type, but its effectiveness is highly dependent on the query arrival rate, the aggressiveness factor and the speed at which orphan expressions can be reused. Thus, it is not a general solution. Switching from min-execution to a different strategy for larger $n$ has the advantage that it prevents the optimization problem from being intractable. For large $n$ the ILP problem that results from the min-execution cost strategy is hard to compute. Thus, difficulties of the NP-hardness of ILP are not observed in practice.

# 6. RELATED WORK

Processing of continuous queries is a well known problem in data streaming systems. In [12] an adaptive multi-query processing strategy based on Eddies [1] is presented. Our approach is different, as in contrast to traditional data streaming systems, data streams in sensor networks are pulled into the stream processing engine rather than pushed. In our case, execution plan generation also involves creating the data stream at the source nodes in the sensor network.

The need for sharing wireless sensor networks was also recognized by [25] and [17]. Both approaches provide a multi-programming environment on the sensor nodes. Both platforms can be used to execute the merged queries generated by our approach. The execution plans need to be compiled into a program and disseminated in the sensor network.

Crespo et al. describe query merging [2] in a publish-subscribe system in a multi-cast environment. They provide a solution for a "battlefield awareness and data dissemination" scenario where several client submit subscriptions for events with geographically overlapping ranges. Although completely different in nature, there is a similar trade-off between processing inside the network and outside (at the clients) for query areas.

Multi-query optimization for sensor networks for spatial aggregation queries is shown in [22]. Query execution is performed in rounds, where the plans are recomputed and disseminated every round. Queries and sensors are represented in a vector space. This representation leads to a reduction of the amount of data transmitted if the concurrent queries are linearly dependent. The number of queries that are actually processed is equal to the number of dimensions of the subspace spanned by the query vectors. The linearity property leads to a reduction of transmission costs for aggregation queries. The evaluation in the paper is limited to queries having the same aggregate and the same sampling frequency. Additionally, the representation chosen makes it difficult to use predicates other than on node IDs.

A recent paper proposes a two-way multi-query optimization system (TTMQO [23]). The first stage is performed at the basestation by query rewrite in order to reduce redundancy in the executed queries. The second stage is performed in the sensor network using query-aware routing and the properties of the broadcast medium. The work does not consider the network topology. They state that the topology is hard to predict, and instead the authors opt for a lower bound for transmitted messages, essentially assuming in-network aggregation or a star topology. Additionally, they do not consider splitting queries. We implemented the TTMQO strategy. The results using our implementation indicate that this strategy is only applicable when the query load contains many concurrent queries and there is a constant stream of new queries. Otherwise, the strategy does not pay off and leads to worse performance than if no optimization is used.

In [20] the efficient evaluation of multiple spatial aggregates

on a subset of source nodes is discussed. In contrast to our setup the aggregate values are not sent to the basestation. Instead, they are sent to destination nodes inside the network, e.g., to control actuators. The proposed solution minimizes communication costs while the sharing of partially aggregated values is maximized. In their work, they assume a multi-cast tree rooted at every destination node that connects the corresponding source nodes. This is a much stronger requirement for the routing layer than the single collection tree.

# 7. CONCLUSIONS

In this paper we present a model for aggregation (spatial and temporal) and non-aggregation streaming queries for sensor networks. We tackle the problem of efficiently executing multiple concurrent queries from a dynamic workload by applying different multi-query optimization strategies. We showed that different strategies have different effects on the total execution cost. Furthermore, there does not seem to exist a strategy that performs best for any number of concurrent queries. Instead, the query optimizer should choose the best strategy based on the multi-programming level.

In the paper we also present the min-execution cost strategy which aggressively reorganized the plans using a cost model and 0-1 integer linear programming. This strategy performs well across a wide range of multi-programming levels and takes into account important factors like network topology.

# 8. REFERENCES


[1] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.

[2] A. Crespo, O. Buyukkokten, and H. Garcia-Molina. Query merging: Improving query subscription processing in a multicast environment. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):174–191, 2003.

[3] A. Deshpande and S. Madden. MauveDB: supporting model-based user views in database systems. In *SIGMOD*, pages 73–84, 2006.

[4] R. Fonseca, O. Gnawali, K. Jamieson, S. Kim, P. Levis, and A. Woo. TEP 123: Collection Tree Protocol (CTP). http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html.

[5] J. Gehrke and S. Madden. Query processing in sensor networks. *IEEE Pervasive Computing*, 03(1):46–55, 2004.

[6] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.

[7] S. R. Jeffery, G. Alonso, M. J. Franklin, W. Hong, and J. Widom. Declarative support for sensor data cleaning. In *PERVASIVE*, pages 83–100, 2006.

[8] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. *SIGOPS Oper. Syst. Rev.*, 36(5):96–107, 2002.

[9] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[10] K. G. Langendoen, A. Baggio, and O. W. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *14th International Workshop on Parallel and Distributed Real-Time Systems*, page 8, 2006.

[11] P. Levis and D. E. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS 2002*, pages 85–95, 2002.

[12] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–60, 2002.

[13] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[14] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[15] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA*, pages 88–97, 2002.

[16] R. Mueller and G. Alonso. Shared queries in sensor networks for multi-user support. In *MASS*, 2006.

[17] R. Mueller, G. Alonso, and D. Kossmann. SwissQM: Next generation data processing in sensor networks. In *CIDR*, 2007.

[18] R. Mueller, G. Alonso, and D. Kossmann. A virtual machine for sensor networks. In *EuroSys*, 2007.

[19] A. Silberstein, A. Gelfand, K. Munagala, G. Puggioni, and J. Yang. Making sense of suppressions and failures in sensor data: A Bayesian approach. In *VLDB*, pages 842–853, 2007.

[20] A. Silberstein and J. Yang. Many-to-many aggregation for sensor networks. In *ICDE*, pages 986–995, 2007.

[21] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the Redwoods. In *SenSys*, pages 51–63, 2005.

[22] N. Trigoni, Y. Yao, A. Demers, J. Gehrke, and R. Rajaraman. Multi-query optimization for sensor networks. In *DCOSS*, 2005.

[23] S. Xiang, H. B. Lim, K.-L. Tan, and Y. Zhou. Two-tier multiple query optimization for sensor networks. In *ICDCS*, 2007.

[24] Y. Yao and J. Gehrke. Query processing in sensor networks. In *CIDR*, 2003.

[25] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *SenSys*, pages 139–152, 2006.


# APPENDIX

## Queries Loads

For illustration purposes the first 16 queries of the two randomly generated workloads are shown in Tables 3 (WL1) and 4 (WL2).

**Table 3: First 16 queries of Workload WL1**

| ID | Query |
|---|---|
| 0 | SELECT humid/temp, temp, light*temp, light EVERY 2min |
| 1 | SELECT TWINDOW(temp, 3, MEAN) EVERY 60s |
| 2 | SELECT SWINDOW(voltage, 20, MIN) EVERY 30s |
| 3 | SELECT light/temp EVERY 5min |
| 4 | SELECT temp+light, light-temp, light EVERY 2min |
| 5 | SELECT lightpar*parent EVERY 60s |
| 6 | SELECT depth/light, temp, parent, humid/lightpar EVERY 2min |
| 7 | SELECT SWINDOW(parent, 3, SUM) EVERY 5min |
| 8 | SELECT depth EVERY 2s |
| 9 | SELECT parent*parent, humid EVERY 30s |
| 10 | SELECT COUNT(*) EVERY 30s |
| 11 | SELECT AVG(light) EVERY 5min |
| 12 | SELECT SUM(humid) EVERY 2min |
| 13 | SELECT MIN(humid) EVERY 2min |
| 14 | SELECT MAX(depth) EVERY 2min |
| 15 | SELECT temp/humid, lightpar EVERY 5s |

**Table 4: First 16 queries of Workload WL2**

| ID | Query |
|---|---|
| 0 | SELECT TWINDOW(humid, 5, SUM) EVERY 15s |
| 1 | SELECT light EVERY 15s |
| 2 | SELECT MIN(parent) EVERY 10s |
| 3 | SELECT COUNT(*) EVERY 30s |
| 4 | SELECT COUNT(*) EVERY 15s |
| 5 | SELECT COUNT(*) EVERY 30s |
| 6 | SELECT TWINDOW(depth, 5, SUM) EVERY 5s |
| 7 | SELECT AVG(light) EVERY 60s |
| 8 | SELECT nodeid, depth-depth EVERY 30s |
| 9 | SELECT temp, temp EVERY 30s |
| 10 | SELECT MAX(nodeid) EVERY 4s |
| 11 | SELECT parent, depth, depth EVERY 2s |
| 12 | SELECT parent*parent, humid EVERY 30s |
| 13 | SELECT COUNT(*) EVERY 30s |
| 14 | SELECT AVG(light) EVERY 5min |
| 15 | SELECT SUM(humid) EVERY 2min |