MaD-WiSe: a Distributed Query Processor for Wireless Sensor Networks

[1] Giuseppe Amato, [1,2] Paolo Baronti, and [1,2] Stefano Chessa

[1] ISTI-CNR, via Moruzzi 1, 56124 Pisa, Italy, firstname.lastname@isti.cnr.it
[2] Department of Computer Science, University of Pisa, Largo Pontecorvo 2, 56127 Pisa, Italy.

Abstract. We propose a comprehensive approach to distributed query processing in wireless sensor networks. We consider various aspects related to database system design and we revise, reinterpret, and redefine them according to the wireless sensor networks context. We consider the aspects related to the definition of a query language, data model, query algebra, and query optimization strategies. All these aspects are consistently discussed and defined. We show that the proposed approach enables optimizations of the query plan which may reduce the costs, in terms of consumed energy, of orders of magnitude.

1 Introduction

Wireless Sensor Networks are networks specialized for environmental monitoring [6, 12]. They are composed of a set of (tiny) devices (hereafter called nodes or sensors), each of which is a microsystem comprising a processor, a memory, a set of transducers, and a low-range, low-bandwidth radio transceiver. Sensors are powered by on board batteries thus their lifetime is limited and their energy efficiency is critical in most applications. Typical applications of sensor networks include environment sampling, disaster areas monitoring, health monitoring, surveillance, security, inventory management, and they have also been envisioned as an architectural support for applications of pervasive computing.

The sensors can be easily deployed in the environment (sensing field) and they self-organize to form a (multihop) wireless network. They can be programmed to sample parameters of the surrounding environment, to process sampled data and to forward this information to a *sink node* which, in turn, provides connectivity between the network and the user. Trivial data gathering applications request sensors to forward periodic samples to the sink node which performs data processing, however in more sophisticate paradigms data processing is performed by the network itself. These approaches generally results in energy saves since a smaller amount of data needs to be transmitted and radio transmissions use an important fraction of the energy budget of the nodes.

A major issue, in these approach, is related to the injection of sensing tasks in the network. Recently proposed methods [23, 29, 25] suggest the use of database paradigms and query languages (generally SQL-like), to interact with sensor networks. A query submitted (or injected) to the sensor network specifies the sensing and data processing tasks to be performed by the network itself. With this paradigm, the database to be queried is the physical environment where the wireless sensor network is deployed.

This database is highly dynamic and it is not persistent. In fact, measured data change continuously and if a data is missed (for instance temperature is not acquired in a certain instant) it is missed forever. Measurable physical parameters of the real world form a continuous flow of data that can be filtered, processed, and cross-related by the wireless sensor network.

In this paper we propose a comprehensive approach to distributed query processing in wireless sensor networks. We consider the various aspects related to database system design and we revise, reinterpret, and redefine them according to the wireless sensor networks context. To our knowledge this is the first work were all these aspects are considered together and consistently to define a framework for distributed query processing in wireless sensor networks:

- We first define a *query language* with constructs specialized for sensor networks. The query language individually manipulates data sources consisting of specific transducers located on individual nodes. Queries can relate and compare data acquired by multiple (remote) nodes. Queries can also aggregate data in the spatial and temporal dimension. Virtual sources can be created and used in queries by combining basic sources. Section 4 discusses these aspects.
- In support to the query language we have defined a *data model* based on streams, where the origin of manipulated data is maintained explicit. The defined data model distinguishes between data acquired locally, data received from remote nodes, and data generated locally during query processing. Given that data acquisition and transmission is the primary concern for energy efficiency, the data model offers useful information to the query optimizer to generate query execution plans that minimize the execution cost. The data model is discussed in Section 5.
- We have defined a *query algebra* offering operators able to process queries specified in our language that exploit the features provided by the data model. A query is translated into a distributed query plan consisting of operators of the query algebra connected by streams. A query is executed by the wireless sensor network in a distributed fashion. Each node involved in a query execute the part of the query plan it was assigned to. The defined operators have a strictly pipelined behavior, and they control the phases of data acquisition and data transmission. The proposed query algebra is simple enough to be implemented by a distributed query processor running exclusively on nodes of the sensor network. The data model and the query algebra provide the query optimizer with features that allow the optimization of the query processing, by accurately orchestrating data transmission, data acquisition, and data processing. We define the operators of the query algebra in Section 6.
- We have defined a *cost model* to estimate the cost of a query plan execution, by taking into account data acquisition and transmission (implied by the various combinations of streams and operators) in conjunction with the expected selectivity of predicates used in the queries. Section 7 discusses these aspects.
- The cost model is an important element for the *query optimizer* that we have defined. We use an algebraic approach to query optimization that use transformation rules based on heuristics aimed at reducing the query execution cost. The transformations also exploit ordering strategies for the operators of the query plan. We

show that the query execution cost can be reduced by orders of magnitude using our approach. Query optimization issues are discussed in Section 8.

The approach proposed in this paper was implemented in the MaD-WiSe (Management of Data in Wireless Sensor networks) system. The distributed query processor of MaD-WiSe was implemented on the MICAz motes platform produced by CrossBow [1], relying on TinyOS [3]. The query parser and optimizer was implemented in Java and runs on a standard PC. Additional information and downloads can be found at the MaD-WiSe project web site [2]

2 Related work

The first attempt to define a data management paradigm in sensor networks is Directed Diffusion [19]. In Directed Diffusion the user queries the network by injecting *interests*. Each interest is associated with a sampling rate and it specifies a set of pairs (attribute,value) describing the kind of data the user is interested in. Interests are broadcasted to all nodes in the network and during the broadcast propagation each node sets up gradients directed to the sink. In practice the gradients set up a directed acyclic communication graph rooted at the sink. Data detected by the sensors which matches an interest is propagated to the sink along this graph. Directed diffusion also include some mechanisms aimed at the management of the interests in the network and it allows some simple form of data aggregation.

The natural evolution of Directed Diffusion had been towards paradigms integrating database management systems and sensor networks. These paradigms use query languages (such as SQL, for instance) to program the sensing task and translate the queries into query plans executable by the sensors. Among the various proposals pursuing this approach, pioneers and (to some extent) state of the art are considered Fijord [23], Cougar [29], and TinyDB [25].

Fijord is one of the first attempt in this direction. It is not bound to any specific query language, rather it offers an infrastructure allowing the user to issue queries to the network. In the Fijord architecture the sensors are clustered around high power machines called proxies with abundant memory and processing power, and a sensor can only interact with its proxy. The role of the proxies is two fold: they serve as intermediary between sensors and query plans and they manage the streams of data generated by the sensors. In this way the proxies can execute the query plans by issuing simple commands to the sensors and efficiently managing their energy. They can also merge streams and historical data local and enable sharing of work between queries. Proxies receive the query plan from a central machine via wired or wireless network and instantiate operators and queues. The former implement tuple processing (including selection and projection) over data streams while the latter interconnect operators, providing a dataflow-like computation architecture. The framework allows multiple queries by allowing operators to output their results to several output queues. Operators and queues implement both a *push* and *pull* operating mode wherein queries based on periodic sampling (acquisition-driven) and queries requesting data on demand can be implemented. Queues provide an uniform interface to connect either two local operators or a local and

a remote operator (i.e., one instantiated on a different proxy). Users can perform joins and aggregates over fixed-size windows which they define on streams. The main limitation of the Fjords approach is that it requires high-powered proxies to be located near the sensors and it only uses the latter for sampling (the rest of query processing happens on proxies). A proxy can only adjust the sampling rate of its sensors or switch them in power save mode (i.e. the sensors are switched off most of the time and they wake up periodically to check for new queries to be served). A limited amount of processing in the form of selections or aggregation can actually be performed on the sensors if that is compatible with all running queries.

Cougar is another sensor database that integrates stored data and sensor data. Stored data are maintained as relations in a traditional database and mainly contain information on sensors in the network, including the characteristics of every sensor in the network and their position. Sensor data refers to data generated by sensors as a consequence of measurements repetitively performed in the physical environment, and they are represented as time series. Sensor data is modelled by representing each type of sensor with an Abstract Data Type (ADT); the ADT interface includes signal processing functions to be used on the sensed data. For instance, the ADT of a temperature alarm sensor may include functions like `getTemp()` and `detectAlarmTemp(threshold)` which respectively return the measured temperature and the measured temperature when it is above the specified threshold. Note that signal processing functions are scalar and return a single value. Cougar obtains time series by repetitively executing these functions: once a value is obtained from the function, it is appended to the time series and the function is invoked again to obtain the next value. Relational operators are used to manipulate stored data, while sequence operators are used to deal with time series. Stored data and sensor data can be combined by using joins that take as input a relation and a sequence. This type of join is implemented by waiting on the sequence for a new record and checking if some record in the relation can be matched with the new record in the sequence. Query processing in Cougar is performed by adopting a three-layers architecture. The *first layer* is composed of sensor nodes. Sensor nodes have a lightweight query execution engine that is capable of executing signal processing functions (included in the corresponding ADT interface). Sensor nodes are grouped into clusters and a node is assigned to be the cluster leader. The *second layer* is composed of cluster leaders. They coordinate and execute aggregate operators on data produced by their cluster nodes and send the results to the database front-end. The database front-end is the *third layer*. It performs query processing on sensor data received by cluster leaders and stored data, and sends the results to the user interface.

Note that Cougar cannot process aggregates over time windows [13]. The reason is that sensor data are obtained by scalar signal processing function. Once a single value is obtained by a function, the query execution processes this data along with other data produced in the same time frame by the same function in other cluster nodes. Note also that Cougar can relate data acquired by different nodes (for instance to compare the temperature measured in two different rooms) by exploiting the query processor on the front-end of the network database. This is achieved with a centralized approach which allows for a lightweight query processor on the nodes, but, on the other hand, requires that data traverse the entire path from the sensor nodes to the front-end, an approach

which is not optimized for energy efficiency. Query optimization also encounters some limitations since the opportunity of in-network query processing is only exploited for execution of aggregates and signal processing functions.

In TinyDB a query is formulated in a computer (base station) connected to a sink node of the sensor network. The base station parses the query and generates an optimized query execution plan. The query plan is sent to all the nodes which should process it (potentially the entire network). The dissemination of the query plan sets up a routing tree (the *Semantic Routing Tree*) in the network which is used to collect sensor data at the sink. Each involved node autonomously processes the query and sends its results back to the sink node. TinyDB assumes the existence of a single (logical) table `Sensors`, containing data produced by the transducers of each node in the network, which is used to formulate queries. The table `Sensors` is filled assuming that time is divided into epochs: in each epoch every node produces a new single logical record for the table (hence the number of records produced in an epoch is equal to the number of nodes in the network). The information contained in a record includes the epoch, the identifier of the generating node, the values produced by the transducers in the node, and optionally additional node-dependent constants values (for instance the node's coordinates). The table `Sensors` is distributed across all nodes of the network: each node can access only its own records and has no access to records produced by other nodes. A query on the table `Sensors` is executed by each node involved in the query independently of each other (every node has a copy of the query). A query is processed repetitively every epoch, and it accesses only the data produced in the current epoch. Records that qualify the query are sent toward the sink node along the semantic routing tree. TinyDB can process aggregate queries on data produced in the same epoch by several sensors (spatial aggregates). Aggregate queries are executed hierarchically [24] along the routing tree. For instance in a query requesting the average temperature of a given area, a node contributes by aggregating the temperature it measured with the temperature received from its child nodes in the tree, thus minimizing the overall amount of data transmitted in the network.

TinyDB has inspired several new works related to the database approach in wireless sensor network, and it is currently the most used tool for real applications. Its approach however, presents some limitations. TinyDB cannot execute queries that relate and compare data acquired by different nodes (for instance, check if temperature in room 1 is greater than that in room 2). This is due to the fact that the table `Sensor` is distributed across all nodes of the network. Each node exclusively owns a partition of the table (a single record per epoch) and processes the query just on that partition. A node (leaving out of consideration hierarchical aggregation) has no knowledge of data acquired by other nodes and cannot compare its data with an other node's data. TinyDB also presents some disadvantages concerning the optimization of queries. Given that the same query plan is processed autonomously in several nodes, the generation of an optimized query plan at the base station is done on global assumption valid for all nodes. Specifically, it is not possible to exploit statistics of individual sensors (or group of sensors). Consider for instance a query that requests to check if the temperature is lower than τ and the luminance greater than λ. Suppose that statistics say that in a certain node (or group of nodes) the temperature has high probability to be lower than τ. In this case it might be

convenient to first check the predicate on the temperature (because it is very selective) and then, just the few times that it is true, switch the light transducer on and check the other predicate. This predicate ordering would save energy in this case, but it would be inefficient in nodes where predicate on the luminance is the most selective. The TinyDB approach is very effective when several nodes (almost all nodes) have to perform the same task (process the same query), but it becomes less effective in applications where different portions of the network have to perform different tasks and when it is necessary to relate data produced in different zones of the network.

3 Overview of the proposed approach

The MaD-WiSe system [20] allows interaction with a wireless sensor network as a traditional database management system. In a traditional database system queries are used to search for data contained in a persistent storage repository. In a wireless sensor network, the data base consists of the environmental data that can be measured/acquired by the transducers available on the sensor nodes. Queries instruct nodes on the management, filtering and processing of the data acquired from the environment. The wireless sensor network and the software running on the nodes are the means that allow data to be acquired when needed from the environment, exactly in the way that a traditional database software allows data to be accessed on disks. In a wireless sensor network data is not stored anywhere: environmental data is acquired by transducers of the nodes when needed, in accordance with the query that the network is being processing. A new data is available every time a transducer is activated.

The environmental data base can be considered as a set of data streams, where new data can be potentially available at any time. Accordingly a wireless sensor network can be seen as a *distributed* data stream managements system. Every node of the network has access to different environmental data. These data can be accessed, processed, and cross-related using distributed query processing techniques [21, 10, 26].

The MaD-WiSe system consists of a set of modules that implement a distributed stream management system on a wireless sensor network.

A part of the MaD-WiSe modules run on the nodes of the wireless sensor network (network side) and the other part of the modules run on a client node (a laptop, a palmtop, etc.), connected to the wireless sensor network through a *sink* node. See Figure 1

The client side sub-system is composed of a query parser, an execution plan optimizer, and a query manager. The query parser takes an SQL-like query and translates it into an initial distributed query execution plan. Operators of the query execution plans are allocated on the nodes involved in the query execution. The query optimizer then generates a semantically equivalent query execution plan which organizes the nodes involved in the query execution, the operations to be executed, the transducer activations, and the radio communications in order reduce energy consumption and increase network lifetime. The query manager disseminates the optimized query execution plan in the network and receives the results obtained from in-network query execution.

The sensor side of MaD-WiSe is organized into three layers, as depicted in Figure 1. The layers interact through well defined interfaces and are autonomous with respect to

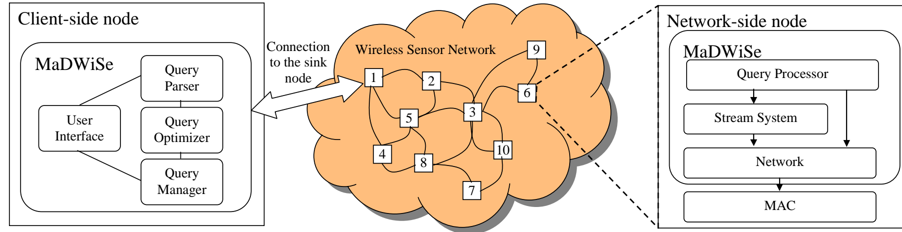

Fig. 1. The architecture of MaDWiSe.

each other. Each layer can be replaced with a new (different) implementation provided it complies with the existing interfaces.

The Network layer sits on top of the standard MAC layer of TinyOS [3]. It offers both connectionless and connection-oriented communication services. At network startup a distributed protocol assigns to each sensor a tuple of virtual coordinates which is used by a multi-hop geographic routing protocol [14]. The network layer also implements an application-driven energy efficiency protocol for the connection-oriented service [7].

The Stream System Layer offers abstraction mechanisms for data access by means of data streams. It can be thought of as the equivalent of a file system on a sensor network, the main difference being that, in the latter, data is continuously produced as a consequence of acquisition from transducers, communications between nodes, and data processing. As we will extensively discuss in Section 5, there are three types of streams: sensor, remote, and local streams. A sensor stream is connected to a transducer and it carries data originated from the transducer. For this reason the sensor streams are read-only. A remote stream is a data channel between two distinct sensors: writing to a remote stream happens on one sensor while reading from the stream happens on the other sensor. A local stream is local to a sensor in the sense that writing to and reading from the stream can only be requested by code running on the same sensor. The Stream System offers functionalities to create/remove streams as well as read and write records from/to existing streams. Data rates can be associated with sensor streams and remote streams. In the first case data rates determine at which speeds transducers associated with sensor streams should be activated to acquire data. In the second case, data rates are used by the network layer to optimize the radio scheduling: radio is switched on only when a data should be sent trough a remote stream [7]. Sensor stream can also be *on-demand*. In this case transducers are not activated at a fixed rate, rather, they are activated only in response to an explicit read request on the stream.

The Query Processor Layer implements the query processor of a distributed data stream management system over the Stream System layer. It can be programmed by the client-side subsystem in order to take part in a distributed query execution. Queries are defined in terms of operations connected by streams. Operations are basically primitives of the query algebra (see Section 6) which are applied to streamed relations implemented by the streams of the Stream System Layer. Note that in our model there

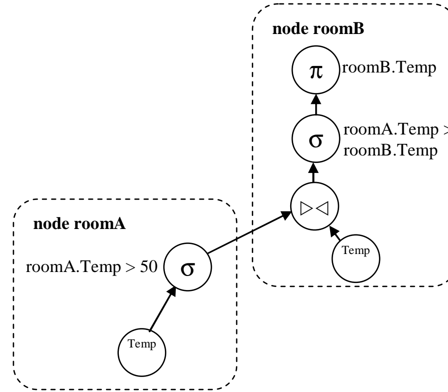

Fig. 2. Example of a distributed query plan.

are significant differences with respect to traditional relational algebra operations and relations. Relations (tables) are mainly static collections of records while streams are flowing records. Correspondingly, operators do not act on static relations but on continuously flowing data. In addition, given the limited resources available to sensor nodes (in terms of memory, processing power, and energy), data is processed on-the-fly when they arrive, using pipelined execution and avoiding as much as possible the storage of temporary data. This requires the use of non-blocking operations, and to exploit the inherent time ordering of data records. Although nodes could temporarily store data for later use, we avoid this to meet memory constraints.

To rigorously deal with these aspects we defin a data model (Section 5), based on streams, and a query algebra (Section 6), based on the relational algebra, both specifically addressed to distributed query processing in wireless sensor networks.

An example of query that can be executed by MaD-WiSe is the following:

SELECT roomB.Temperature
FROM roomA, roomB
WHERE roomA.Temperature > roomB.Temperature **and**
roomA.Temperature > 50
EVERY 10000

The query involves nodes in roomA and roomB. When the temperature in roomA is greater than 50 and also greater than the temperature in roomB, the temperature of roomB is given. This test is executed every 10 seconds. An optimized execution of this query, as that shown in Figure 2, requires actions in sensors of roomA and roomB to be correctly orchestrated. Every 10 seconds the node in roomA reads the temperature, it checks whether this value is greater than 50, and in this case it sends the temperature value to the node in roomB. Then the node in roomB reads the temperature, it checks if its temperature value is below the temperature of roomA and, in this case, it sends its temperature value to the base station. We will see that by exploiting the proposed data

model, query algebra, and architecture, we can express, optimize, and process a query like this.

4 Query Language

The query language used in MaDWiSe is named MW-SQL and shares its basic constructs with SQL. However sensor network peculiarities and the distributive nature of the database implementation introduce some differences.

MW-SQL allows users to express queries to manipulate, filter, and organize sequences of tuples generated by the sensors. MW-SQL relies on the concept of *source* to present the user with an abstraction of a sequence of tuples arriving from a precise origin.

As discussed below, sources have attributes that the user may refer in the query. MW-SQL queries are expressed through query statements having the form:

SELECT *select-list*
FROM *source*
[**WHERE** *condition*]
[**EPOCH** *samples* [**SAMPLES**]]
[**EVERY** *rate*]

A MW-SQL query selects the attributes (including temporal aggregates) specified by *select-list* from all tuples that satisfy a certain *condition* from the indicated *source*. Optionally, a query can request a sampling rate (*rate*) and an epoch duration (*samples*). A sampling rate specifies at which rate transducers should acquire data, in milliseconds. The epoch duration specifies how many samples are considered when processing temporal aggregation in queries. Keyword **SAMPLES** may be used but is redundant. A missing **WHERE** clause imposes no condition on tuple selection.

In the rest of this section we describe the main constructs of MW-SQL. Detailed specification of MW-SQL can be found in [20].

4.1 Sources

The **FROM** clause in the MW-SQL query statement defines a source of data tuples to be considered when generating query results. The **SELECT** clause expresses which of the attributes of the source are relevant for the query as a comma-separated list of attribute names. As a special case **SELECT** * means that all attributes are significant and none must be discarded.

The ultimate data sources for any query computation are transducers. We call such elementary sources *basic sources*. Conceptually, for each transducer TR available on sensor A there exists a basic source named A.TR with two attributes named Timestamp and A.TR, where A is a numeric sensor id and TR identifies a transducer. For example the transducer can be Light, Temperature, Audio, MagnetismX, MagnetismY, AccelerationX, or AccelerationY, subject to the actual availability of the transducer type on

the sensor. For instance, if a light transducer is available on sensor 1 the user may refer source 1.Light with attributes <Timestamp, 1.Light> [3]. Attribute Timestamp is a timestamp value for the reading contained in the other attribute. In the query

SELECT *
FROM 4.Light
EVERY 10000

the **FROM** clause refers basic source 4.Light while the **SELECT** clause indicates that both Timestamp and 4.Light are significant attributes and must be maintained in the query outcome.

A *complex source* is a source constructed by combining together several other sources (basic or complex) by means of join, spatial aggregation, and union operations.

Join: Joining sources means combining their tuples on the basis of a common timestamp value. The resulting source has all the attributes of the component sources with the exception that attribute Timestamp is replicated only once. A complex source obtained by joining several sources can be expressed as a comma-separated list of the source names. For instance in

SELECT 2.Temperature, 3.Temperature
FROM 2.Light, 2.Temperature, 3.Temperature
WHERE 2.Light > 20

we join basic sources 2.Light, 2.Temperature and 3.Temperature and request attributes 2.Temperature and 3.Temperature to appear in the query output. The complex source defined in the **FROM** clause has attributes Timestamp, 2.Light, 2.Temperature, and 3.Temperature and we only select two of them. Note that the meaning of the above query is substantially different from standard SQL. In SQL the above query whould have computed a simple cartesian product, rather than a join on the Timestamp attribute, given that an explicit join condition is not defined.

Spatial aggregation: Aggregation of data produced by a group of sensors is a very significant capability in wireless sensor networks. It allows the reduction of data that is sent to the sink. For instance one might want to compute the average, the maximum, or the minimum of the temperatures measured in different locations in a large room.

In MW-SQL spatial aggregates are expressed by using a functional notation in the **FROM** clause as the aggregation name (**max**, **min** or **avg**) followed by a parenthesized comma-separated list of basic sources or spatial aggregation sources. Consider

SELECT *
FROM avg(1.Temperature, 2.Temperature, 3.Temperature)
EVERY 1000

[3] Note that 1.Light is used both to denote the name of the basic source and the name of one of the attributes

We request an average spatial aggregation of the basic sources 1.Temperature, 2.Temperature and 3.Temperature and request sampling to occur once every second. The basic sources involved in the spatial aggregation must be of the same type i.e., they must all sample the same quantity (temperature in this case). It would not be legal (nor meaningful) to have a clause like

FROM max(1.Temperature, 2.Temperature, 3.Audio)

Aggregation is actually performed computing the desired function (max, min or average) on the sampling attributes from the basic sources, retrieved from tuples with the same timestamp value. The attributes of a spatial aggregation include the Timestamp and the name of the sampled attribute, deprived of any numeric prefix. For the previous example query the attributes of the aggregation source would be <Timestamp, Temperature> . Note also that spatial aggregation can be nested to obtain more complex queries.

Union: Sometimes it is useful to sequentially put in a single source the values read by different sensors. This can be obtained by using unions of sources. The query

SELECT *
FROM union(1.Temperature, 2.Temperature, 3.Temperature)

acquires temperature values from sensors 1, 2 and 3. The acquired values are sequentially returned by the query.

Note that the union of all nodes' readings is equivalent to the behavior of TinyDB [25], which employs a unique table (sensors) containing a record for every node per sampling period.

4.2 Topological selection of sources

A useful construct that can be used in the **FROM** clause is the functional **area**(). Each sensor is assigned coordinates that indicate its position in the two dimensional space. Functional **area**(x1, y1, x2, y2) takes 4 arguments indicating the top-left and bottom-right corners of a rectangle as illustrated in Figure 3. The functional must be qualified by a basic source type name and is used to denote all basic sources of the given type of all sensors located within the rectangle boundary. Assuming the area with (100,20) and (300,80) as corners contains sensors 4, 5 and 7

SELECT *
FROM area(100, 20, 300, 80).Light

is a shorthand for

SELECT *
FROM 4.Light, 5.Light, 7.Light

Observe that the functional **area**() can be combined with aggregates and joins possibly having different basic source types, as in:

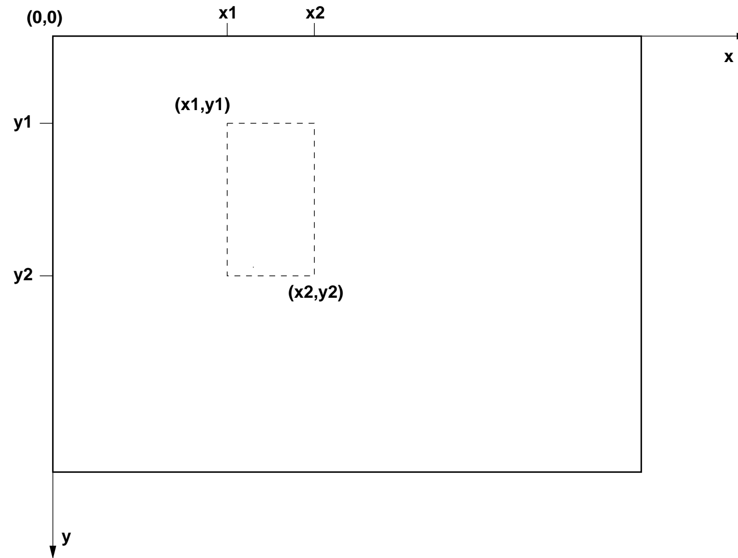

Fig. 3. Specifying a region in the **area**(x1,y1,x2,y2) functional.

SELECT *
FROM avg(**area**(100, 20, 300, 80).Light),
    area(100, 30, 200, 60).Temperature

Another shorthand consists in keyword **all** to represent all existing sensors. Hence for a sensor network with sensors 1, 2, 3 and 4 the clause **FROM all**.Light is equivalent to **FROM** 1.Light, 2.Light, 3.Light, 4.Light

4.3 Temporal Aggregates

The *select-list* in the **SELECT** clause normally consists of a comma-separated list of attribute names from the source. It is also possible to request that temporal aggregates be calculated on some of the attributes of the source. Four different temporal aggregates are supported: max, min, average and count, indicated by the functionals **max**(), **min**(), **avg**() and **count**() respectively. The argument to the functionals must be one of the attributes from the source defined by the **FROM** clause (see Section 4.1). When temporal aggregates are requested the **SELECT** clause must contain a comma-separated list of aggregate functionals and optionally the Timestamp attribute. Different temporal aggregates can be requested on the same attribute.

In the query

SELECT min(1.Light), **max**(1.Light), **avg**(2.Temperature)
FROM 1.Light, 2.Temperature
EPOCH 10 **SAMPLES**
EVERY 2000

we request the minimum and maximum temporal aggregates over the light values from sensor 1 and the average temporal aggregate over the temperature from sensor 2.

The **EVERY** and **EPOCH** clauses can be used to specify a sampling rate and the number of consecutive samples on which to calculate temporal aggregation. In the previous query we request that samples be taken every 2 seconds and that aggregation be performed every 10 samples (i.e., every 20 seconds).

4.4 Source Naming and Creation

MW-SQL has a virtual source creation statement that can be used in conjunction with the query statement:

CREATE SOURCE *source-name*
AS *source-definition*

In the simplest form such statement can be used to assign a name to anything that can appear in the **FROM** clause of a query statement. Note that virtual sources can also be nested. Definitions of virtual sources are maintained at the base station and they are translated in the appropriate combination of basic and complex sources during the query plan generation.

An example of virtual source definition is the following:

CREATE SOURCE RoomA
AS 1.Temperature, 1.Light

A query can simply use the defined source as in

SELECT RoomA.Light
FROM RoomA
WHERE RoomA.Temperature $>$ 40

to retrieve the light readings of sensor 1 (located in room A) whenever the corresponding temperature readings exceed 40.

5 The data model for query processing

A MW-SQL query is translated into a query execution plan expressed in terms of simple operators which manipulate the data acquired by sensors. In this section we define the model of data manipulated by these operators. Section 6 will then discuss the operators of the query algebra.

A natural way to imagine data processed by a wireless sensor network is that of a stream or sequence of tuples. Various models were recently proposed to model and process streams of data. A brief survey on data stream management is given in [18]. Sequences are modeled as finite ordered sequences of records in SEQ [28], while in [22] they are considered as possibly infinite ordered sequences records and the problem of non-blocking queries on sequences is also studied. In AURORA [5] a stream is modeled as an append only sequence of tuples. Data come from a variety of external data

sources and no direct control is possible on the ordering and regularity of data arrival. In Borealis [4] the model is extended by allowing tuples deletion and replacement in addition to insertions. In STREAM [9] a stream is considered as a sequence of data that arrive online. Once a data is accessed and processed it is discarded. No control on the sources of the streams is possible.

The above mentioned models address most of the issues for generic data stream management. However, given their generality, they do not consider some peculiar aspects that are critical for data management and processing in wireless sensor networks. Before giving our definition of streams for wireless sensor networks we will first discuss these aspects.

5.1 Peculiarities of streams in wireless sensor networks

Sensor networks perform the two main tasks of data acquisition from the environment and later data processing. However real environments produce continuous (and potentially infinite) flow of data, which do not need to be acquired and processed entirely. Rather, the sensor network decides if and when data should be acquired by the transducers on the basis of the received queries. The main reason for restricting data acquisition to a subset of samples is that activation of transducers and data processing have a cost in terms of energy consumption, and they should be done only when really needed. From this point of view, considering streams as append only sequences, or as sequences controlled by some external sources is unsatisfactory. In fact, controlling the way in which data are obtained and injected in a stream is a very important aspect here.

In a wireless sensor network data are initially acquired by the transducers of a subset of nodes. After a local (pre-)processing, data may be sent to other nodes to be further processed or combined with data coming from other nodes. All of these steps imply access to tuples of data travelling along the data streams. Note however that the the task of accessing a tuple is drastically different depending on where the tuple comes from. For instance, if the tuple is to be acquired from a transducer, the cost of accessing it includes the cost of activating the transducer itself. On the other hand, if a tuple is located on another node its access requires using the radio interfaces. This has a significant impact on the definition of the cost model used for optimizing queries, and should be made explicit in the data model. In our model the cost is estimated by considering the number of tuple accesses and the access modality, an approach which is rather different to that used in traditional databases which use the number of disk accesses and the size of temporary results to determine the cost of a query.

Another important issue is related to the use of storage resources which in the sensors are quite scarce. Since the storage of windows of tuples is memory demanding and may be impractical in many cases we opt for on-the-fly processing of stream tuples, that is, tuples are processed as soon as they arrive and they are discarded soon afterwards. This means that tuples must be processed using non-blocking operations that should not rely on the possibility of arbitrarily buffering data.

5.2 Modeling streams in wireless sensor networks

The data model of MaDWise is consistent with the observations of the previous section. The data base (data acquired in the real world) is seen as a set of streams of tuples. In our model data generated externally are not passively injected in a stream, rather the stream itself maintains the information that specifies when data should enter the stream, for instance by having the associated transducer or the radio communication interfaces activated when needed.

We define three different types of streams to model the different tuple access modes: *sensor streams*, which represent streams of data acquired by the transducers, *remote streams*, which model streams of data sent by a source node to a destination node, and *local streams*, which represent streams of data generated by execution of local operations and sent as input of other local operations. These three types of streams model data acquisition (sensor streams), data transmission (remote streams) and pipelined processing on a single node (local streams). As discussed in the next sections, this allows the design of a cost model and of a query optimizer which distinguish the various types of streams and generate a query execution plan that minimizes the energy consumption of the wireless sensor network.

A generic stream is defined as follows:

$$S = (sd, t)$$

where sd is the stream descriptor, which specifies the nature (sensor, remote, or local) and update modality of the stream. The stream descriptor has a different structure for the various types of streams, as detailed in the next sections. The flow of tuples in a stream is modeled by a single *dynamically changing tuple* t. A stream maintains the last received tuple and every new tuple overwrites the previous one, thus forcing an on-the-fly processing of the tuples. Clearly, in a real implementation, data transmission trough streams and data processing might incur in some delays and a small buffer can be used to store the incoming tuples. However, this has mainly implementation advantages rather than modelling advantages. The extension to a fixed length buffer of tuples is straightforward and we do not discuss it in this paper.

The dynamically changing tuple t has a set of application specific data fields A_1, $\dots$, A_n, associated with a set of values $v_1, \dots, v_n$. Hereafter we will use notation $(A_1 = v_1, \dots, A_n = v_n)$ to denote a tuple and to emphasize its type and its value. A tuple that contains no data (because it is not yet available) is denoted with $(A_1 = void, \dots, A_n = void)$ or with the compact form *void*.

As an implementation note, we anticipate that we use an event-driven approach to process data. When a tuple is available on a stream (not matter the type of stream and the access modality) an event is fired, which wakes-up the operation waiting for it. This implements a continuous query processor, where long-life queries are executed to periodically monitor external environment: current tuple is processed, then the operation(s) implementing the query waits for the next tuple to be available.

In the following we will discuss the tree types of data streams in more detail.

5.3 Sensor streams

Sensor streams are used to sample environmental data. A sensor stream receives tuples of data acquired by the associated transducer

We have considered three different modes for updating the tuple of a stream:

1. In *periodic update* mode the transducer is activated with a fixed period to update the stream. We sometimes call *sampling period* or *sampling rate* the interval between two consecutive transducer activations.
2. In *on-demand update* mode the transducer is activated as a consequence of a read request and causes the stream update.
3. In *asynchronous activation* mode the tuple is updated when some external asynchronous event occurs.

For instance, periodic updates can be used to collect temperature readings every 10 seconds. On-demand updates can be used by the query processor to obtain light readings only when temperature readings of another stream are above a certain threshold. Asynchronous updates can be used to detect asynchronous environmental events.

The stream descriptor sd of a sensor stream is

$$sd = (n_id, TR, update_mode)$$

where n_id is the identifer of the node hosting the stream, TR is the name of the transducer associated with the stream, $update_mode$ indicates the update mode of the stream tuple.

The update mode of a sensor streams is decided at query plan generation and query optimization on the basis of the query and of the role of the stream in the query. In case of periodic activation, the $update_mode$ takes the form $\mathbf{p} = nn$, where nn indicates an activation interval in milliseconds. In case of on-demand activation, the $update_mode$ will be $\mathbf{od}$, and for asynchronous activation it we will be $\mathbf{as}$.

The dynamically changing tuple t of a sensor stream always has the form

$$t = (\mathbf{TS} = ts, TR = v)$$

and, of course, can also be $void$. The attribute TR is the name of the associated transducer, v is the last value acquired by the transducer, and ts is the timestamp indicating when the transducer was activated to acquire the value v.

For instance, the sensor stream

$$((20, \mathbf{Temp}, \mathbf{p} = 1000), (\mathbf{TS} = 123, \mathbf{Temp} = 27))$$

is located on Node 20, the temperature transducer is activated every second, and the current (last) reading is 27 degrees and it has been measured at timestamp 123.

The unique possible operation on sensor streams is $read$. Initially the tuple t is set to $void$.

The read operation returns the read tuple and sets t to $void$ if t is different than $void$, otherwise it waits for the tuple acquisition, returns the acquired tuple and sets t to $void$. In on-demand sensor streams the transducer activation (and then the tuple acquisition) is executed (on-demand) when the read operation is invoked.

5.4 Remote streams

A query is generally processed cooperatively by a group of nodes of the network. This implies that in some cases it is necessary that partial query results produced by a node be sent to another node in order to complete the query execution. For this purpose we make use of remote streams, where the source and destination ends reside on different nodes.

Communication between nodes is obtained through their radio interfaces. Radio activation is one of the primary causes of energy consumption, so it should be carefully managed as well. Suppose that a node acquires the temperature every ten seconds, and then sends it to another node. In this case the remote stream, used to transfer the temperature readings, will transport data at most every 10 seconds. Accordingly, activation of the radios along a communication path can be synchronized to transfer data from the source to the destination, every 10 seconds.

In our definition, remote streams hold the expected data rate of the stream. This information is extracted from the query and used in the query plan generation and optimization to synchronize the radio activity of the network.

An energy efficient radio communication protocol that uses this type of information to efficiently synchronize the radio activity along multi-hop communication channels is proposed in [7].

The stream descriptor sd of a remote stream is defined as

$$sd = (sn_id, dn_id, \mathbf{p} = nn)$$

,

where sn_id and dn_id are the source and destination nodes, respectively, and nn is the transmission period in milliseconds. The dynamically changing tuple t can be a generic tuple $t = (A_1 = v_1, \ldots, A_n = v_n)$, or $void$.

For instance, the remote stream

$$((10, 15, \mathbf{p} = 1000), (\mathbf{TS} = 30, \mathbf{Temp} = 27, \mathbf{Light} = 40))$$

has source Node 10, destination Node 15, data are sent every second, and the (last) tuple that traversed the stream is $(\mathbf{TS} = 30, \mathbf{Temp} = 27, \mathbf{Light} = 40)$.

Two operations are defined on a remote stream: the $read$ operation, which can be executed on the destination node, and the $write$ operation, which can be executed on the source node.

If the tuple is $void$ the read operation waits for a tuple to be received, then sets it back to $void$ and returns the tuple. If the tuple is not $void$ it returns the tuple and sets it to $void$. The $write$ operation sends a tuple toward the destination node. The sent tuple will overwrite the value of the tuple at the destination.

5.5 Local streams

Local streams model data transfers between operators located on the same node. They are used to connect operators residing on the same node in such a way that pipelined executions of operation can be achieved.

Note that, although remote streams could be used to this purpose, we prefer to distinguish local streams from remote streams, given their substantial difference in functionality. Furthermore remote and local streams have different costs in terms of energy consumption. Local streams mainly consume memory resources, while their energy consumption is negligible. On the other hand, the energy consumption of remote streams is predominant for the cost evaluation of a query execution plan. The distinction between local and remote streams makes it easier to cope with these issues.

The stream descriptor sd of a local stream is:

$$sd = (n_id)$$

where n_id is the node where the stream is located. The dynamically changing tuple t is the last tuple sent through the stream. Similarly to remote streams, the tuple t can be a generic tuple $t = (A_1 = v_1, \ldots, A_n = v_n)$, or $void$.

For instance, the local stream

$$((17), (\mathbf{TS} = 91, \mathbf{Magnetometer} = 39, \mathbf{Light} = 40))$$

is located on Node 17, the last tuple that traversed the stream is the tuple $(\mathbf{TS} = 91, \mathbf{Magnetometer} = 39, \mathbf{Light} = 40)$.

Both the $read$ and the $write$ operations can be executed on a local stream by the node where the stream is located.

Also for local streams, if the tuple is $void$ the read operation waits for a tuple to be seen on the stream, then sets it back to $void$ and returns the tuple. If the tuple is not $void$ it returns the tuple and sets it to $void$. The $write$ operation overwrites the current value of the tuple.

6 Operators of the query algebra

The operators of the query algebra supporting MW-SQL are inspired by the relational algebra [15]. However traditional relational operators are set-oriented and deal with set of homogeneous tuples, while our operators deal with streams of tuples according to the data model defined in Section 5. These operators have a strictly pipelined behavior that avoids the use of temporary buffers for producing results, and exploit the features of the specific types of streams used as operators input or output.

Operators take one or two streams as input and one stream as output, plus some operator specific parameters. With a few exceptions, operators can take any type of stream as input or output.

We define three basic operators, *selection*, *projection*, and *union*. In addition, we provide a special definition for the *join* operators (which relate tuples arriving on different streams) for the *spatial aggregates* (used to aggregate tuples produced by different streams), and for the *temporal aggregates* (used to aggregate data that arrive sequentially in a stream).

6.1 Selection

The semantics of the selection operator is equivalent to the relational selection. The selection operator puts in the output stream the tuples that satisfy the given predicate. More formally:

$S_O \leftarrow \sigma_p(S_I) =$
    while(**true**)$\{$
        $t = S_I.read();$
        if $p(t)$
            $S_O.write(t);$
    $\}$

where p is a predicate on the tuples t arriving on the input stream S_I. The type of the streams S_O and S_I must be the same.

Predicate p has the form $A_i\ op\ A_j$ or $A_i\ op\ const$, where A_i and A_j are attributes of the tuple t, op can be $=, <, \leq, >, \geq$, and $const$ is a constant.

The selection operator processes tuples coming from the input stream on-the-fly, as soon as they are available. It executes an infinite loop that waits for a new tuple t to be available in the input stream and writes it in the output stream if the tuple satisfies the given predicate.

6.2 Projection

The projection operator, similarly to the corresponding relational operator, produces output tuples which contain a subset of the elements contained in the input tuples.

$S_O \leftarrow \pi_{A_1,\ldots,A_n}(S_I) =$
    while(**true**)$\{$
        $t_I = S_I.read();$
        $t_O = (A_1 = t_I(A_1), \ldots, A_n = t_I(A_n));$
        $S_O.write(t_O);$
    $\}$

where the attributes $A_1, \ldots, A_n$ are a subset of the attributes of the tuples of the input stream S_I.

The projection operator processes tuples coming from the input stream as soon as they are available. It executes an infinite loop that waits for a new tuple t_I to be available in the input stream and writes the tuple t_O, which contains a subset of the element of t_I, in the output stream.

6.3 Union

The union operator reads tuples from two input streams and writes them in a single output stream. It is defined as

$S_O \leftarrow \cup(S_{I_1}, S_{I_2}) =$
parallel{
while(**true**){
$t = S_{I_1}.read()$;
$S_O.write(t)$; };
while(**true**){
$t = S_{I_2}.read()$;
$S_O.write(t)$; };
}

The union operator executes two parallel infinite loops. Each loop reads an input stream. The tuple read from either stream is immediately written in the output stream.

Clearly, the name and the type of the attributes of the tuples of the two input streams S_{I_1} and S_{I_2} should be the same.

6.4 Join

Applying the traditional join operator on streams imply using potentially infinite buffers to maintain all tuples received from the streams. In fact, according to the definition of the relational join operator, a tuple received from one of the input streams might need to be joined with any tuple successively received from the other stream. However, in the wireless sensor network context, it is not particularly interesting relating any arbitrary pair of tuples received from two streams. Rather, it is very useful to relate tuples generated at the same instant (or approximatively the same instant) by different transducers and/or nodes. For example, one may be interested in comparing the temperature acquired in two adjacent rooms of a building. For this reason we define a join operator that relates tuples having the same timestamp **TS**.

Assuming that the sensor network is able to synchronize the clocks of the nodes with an acceptable precision (and consequently the timestamps), the join operator for a wireless sensor network checks if the last tuples read from the input streams have the same timestamps and, so it writes the tuple obtained by their combination in the output stream. In this respect, the join operator requires that the input streams have the timestamp **TS** attribute.

For every new tuple read on one of the input streams the join operator checks if the last tuple read from the other stream has the same timestamp. This operation is clearly non-blocking and its execution requires only single-position buffers. Given that in some cases tuples may incur in delays during query processing, the join operator could also maintain a very small buffer containing the last n tuples arrived on each stream, where n can be estimated from the average delay and the quality of service offered by the wireless sensor network.

The definition of the join operator is the following:

$S_O \leftarrow\bowtie (S_{I_1}, S_{I_2}) =$
parallel{
while(**true**){
$t_{I_1} = S_{I_1}.read()$;

```
        if t_{I_1}.TS = t_{I_2}.TS {
            t = t_{I_1}|(t_{I_2} \ TS)
            S_O.write(t); }
    };
    while(true){
        t_{I_2} = S_{I_2}.read();
        if t_{I_1}.TS = t_{I_2}.TS {
            t = t_{I_1}|(t_{I_2} \ TS)
            S_O.write(t); }
    };
}
```

The join operator executes two parallel infinite loops, each reading one input stream. When a new tuple arrives on either stream, the join checks whether the last tuple received on the other stream has the same timestamp. In this case it concatenates the two tuples onto a new one, taking care not to replicate the attribute $\mathbf{TS}$, which is contained in both tuples.

6.5 Sync-Join

In the previous definition of the join operator the two input streams are read independently of each other.

Hence, if several tuples arrive at the first stream before a tuple arrives on the second, all but the last readings in the first stream will be useless, but they will consume energy resources (due to transducers or radio activations). For instance consider the following query:

SELECT *
FROM 1.Light, 1.Temperature
WHERE 1.Temperature $>$ 200
EVERY 10 **SECONDS**

The above query retrieves the light and temperature readings only when the temperature is above the specified threshold. By the previous join definition, this query can be processed by using two periodic sensor streams associated with Light L and temperature T, respectively, and using the query execution plan given in Figure 4. In the query Temperature and light transducers are both activated every 10 seconds, and whenever the temperature is below the specified threshold no temperature tuple is sent to the join operator and the current light reading is lost, since it cannot be matched. Activating the light transducers when the temperature is below the threshold is useless and introduces and unneeded energy consumption. Suppose that the above query is used to detect an alarm situation. If the probability that the temperature is above the specified threshold is very low, the above query plan consumes energy for unnecessary light readings.

The query can be processed more efficiently by defining L as an on-demand sensor stream and by using a special implementation of the join operator that requests the activation of L only when a tuple arrives on the other stream. We call *sync-join* this implementation of the join operator:

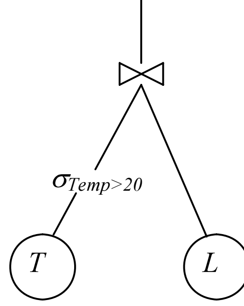

Fig. 4. Using the join operator, both sensor streams T and L should be periodic. Using the sync-join, the L sensor stream can be on-demand, i.e. it is activated only when needed

$S_O \leftarrow \bowtie_{sync} (S_{I_1}, S_{I_2}) =$
  while(**true**){
    $t_{I_1} = S_{I_1}.read();$
    $t_{I_2} = S_{I_2}.read();$
    if $t_{I_1}.\mathbf{TS} = t_{I_2}.\mathbf{TS}$ {
      $t = t_{I_1} | (t_{I_2} \setminus \mathbf{TS})$
      $S_O.write(t);$ }
  }

The sync-join operator executes a single infinite loop reading the input stream S_{I_1}. When a tuple from S_{I_1} is received, then the second stream is read. After a tuple is received on the second stream, the timestamps of the two tuples are compared and if they are the same then the output tuple is written on the output stream. The sync-join operator has a master-slave behavior: stream S_{I_1} is the master and slave stream S_{I_2} is read only when a tuple is received from the master.

6.6 Spatial aggregates

Spatial aggregates compute an aggregation of values acquired by a group of sensors. Examples of spatial aggregates are the maximum temperature and the average temperature measured in the rooms of the first floor of a building. We call them spatial aggregates since they can be used to compute a summary information of a phenomenon occurring in an area where the sensors are located.

One important feature of the aggregation is that a group of values is compressed into a single value. This is particularly important in wireless sensor networks applications where the amount of data exchanged between nodes should be minimized. For this reason the aggregation should be computed in the network by the nodes themselves, thus minimizing the consumed energy and by eliminating the hot spots (that is nodes that perform most of the computation). As suggested in other works [11, 25] an aggregate computation can be decomposed in a combination of simple operators organized in a

tree. Leaf nodes represent values that have to be aggregated and internal nodes are operators that compute partial states of the aggregation, by using values to be aggregated and/or partial states computed by children. The root node of the tree computes the final aggregation.

To simplify the strategy, we have defined simple binary operators to compute aggregations. The spatial aggregation of a group of sensors is obtained by building a binary tree. Each binary operator receives partially aggregated information, or yet to be aggregated information, and returns partially or completely aggregated information.

For some types of aggregations, like maximum or minimum, the aggregation can be obtained in a straightforward way without particular care of the management of the partial state. We just need a binary operator *max* (respectively, *min*) and produce a binary tree as shown in Figure 5a. Every node computes the max (or the min) between the value acquired locally and the maximum (or the minimum) computed by its child. The computed aggregate is passed to the parent node.

For other types of aggregation some care is needed in order to correctly report partial state upwards from the leaves to the root of the binary tree. For instance in spatial averages the partial state is given by the sum of the values to be aggregated and the number of values that have been summed. The average is computed in the root of the tree dividing the sum of the values by the number of values. This is achieved using two operators: one computes the sum and counts the values and it is used in the intermediate nodes of the binary tree, and the other operator is used at the root and divides the sum by the number of values. We have called the two operators partial average *pavg* and final average *favg*, respectively. Figure 5b shows an example where the average light measured by a group of sensors is computed. The *pavg* operator produces a tuple of type (**TS**, **Light**, #), where the **Light** attribute contains the sum of the light readings, and # contains the number of values that were summed. The *favg* operator produces a tuple of type (**TS**, **Light**), where the **Light** attribute contains the computed average. Other aggregates might need other types of partial state information to be transferred across the nodes of the tree.

Clearly, given a group of nodes, an aggregate can be computed by organizing and connecting the nodes in various ways. Even if the computed aggregate is the same, the cost for computing the aggregation may vary significantly depending on the order in which the aggregate is computed. In fact, the cost of communication between two nodes depends on their distance, which affects the number of intermediate nodes required for the multi-hop communication. In section 8 we will discuss how the query optimizer can produce a query plan consisting of a communication tree that minimize the energy consumption.

The definition of a generic binary operator for computing spatial aggregates is the following:

$S_O \leftarrow agg(S_{I_1}, S_{I_2}) =$
    parallel$\{$
        $t_{I_1} = void$
        $t_{I_2} = void$
        while(**true**)$\{$
            $t_{I_1} = S_{I_1}.read();$

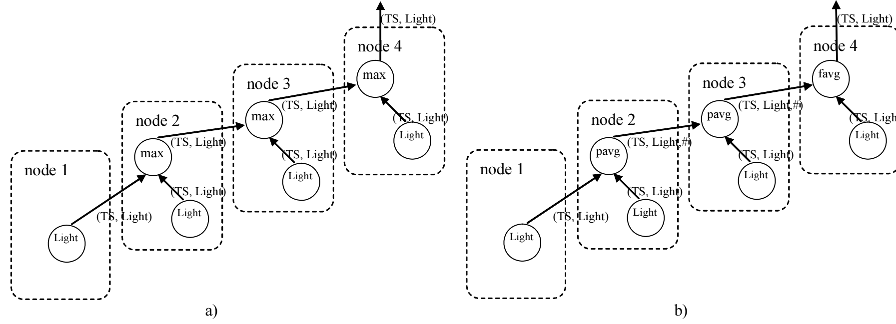

Fig. 5. Example of query execution plans for the spatial maximum a) and for the spatial average b). Spatial average uses two different operators: *pavg* (partial average) and *favg* (final average)

if $t_{I_2} != void$ {
$t = output_tuple(t_{I_1}, t_{I_2});$
$t_{I_1} = t_{I_2} = void;$
$S_O.write(t);$ }
}
while(**true**){
$t_{I_2} = S_{I_2}.read();$
if $t_{I_1} != void$ {
$t = output_tuple(t_{I_1}, t_{I_2});$
$t_{I_1} = t_{I_2} = void;$
$S_O.write(t);$ }
}
}

Also in this case there are two parallel infinite loops that read the two input streams. When a new tuple arrives on an input stream the operator looks for a tuple coming from the other input stream. If it arrives then the operator combines the two input tuples to generate the output tuple. The way in which the output tuple is generated varies for different aggregation operators.

6.7 Temporal aggregates

Another useful operation is the aggregation of values acquired over a time interval by the same transducer. For instance, one might be interested in computing the average or the maximum temperature measured during the day by a thermistor located in a room. We call these kind of operators *temporal aggregates*. As for spatial aggregates, also temporal aggregates are particularly important in wireless sensor network applications, given their property of reducing the amount of information that needs to be transferred from the source nodes. For instance computing the average daily temperature requires

only one single value per day to be transmitted, provided that the temporal aggregate is computed by the node performing temperature acquisition.

In order to define temporal aggregates we introduce the concept of *epoch*. An epoch is a time window where several events (such as transducer activations) occur. Epochs are consecutive non overlapping fixed size time intervals associated with individual queries. Every epoch contains all tuples whose timestamp falls within the corresponding interval.

More formally, suppose that the evaluation of a query starts at time t_0, and that the epoch duration for that query is t_e. The i-th epoch of that query is the interval $E_i = (t^i_{start}, t^i_{end}]$, where $t^i_{start} = t_0 + i \cdot t_e$ and $t^i_{end} = t_0 + (i+1) \cdot t_e$.

Temporal aggregates are computed by grouping together tuples that have timestamps falling in the same epoch.

The definition of a the temporal aggregation operator is the following:

$S_O \leftarrow t_0, t_e \gamma_{agg_1(A_1),\ldots,agg_n(A_n)}(S_I) =$
    $i = 0$; //current epoch
    $ps_1 = \ldots = ps_n = init$; //initializing partial states
    while(**true**){
        $t_I = S_I.read()$;
        if $((t_I(TS) - t_0)/t_e) = i$
            $ps_1 = ps_agg_1(t_I(A_1), ps_1); \ldots, ps_n = ps_agg_n(t_I(A_n), ps_n)$;
        else{
            $t = (\mathbf{TS} = t_0 + i \cdot t_e, A_1 = agg_1(ps_1), \ldots, A_n = agg_n(ps_n))$;
            $S_O.write(t)$;
            $ps_1 = ps_agg_1(t_I(A_1), init); \ldots, ps_n = ps_agg_n(t_I(A_n), init)$;
        }
    }

The temporal aggregation operator groups together tuples belonging to the the same current i-th epoch, which is determined by using t_0, t_e, and the epoch counter i. If current epoch has not yet finished, partial states for the requested aggregates $agg_1, \ldots, agg_n$ are computed. When the epoch ends,the operator computes the final aggregate and deposits the result in the output stream. Then it computes the first partial state for the next epoch. The output tuple generated at the end of the epoch contains the computed aggregate and the timestamp set to the end of the epoch.

7 The cost model

Traditional databases evaluate the cost of a query plan by estimating the number of accessed records, the number of disc accesses, or the size of temporary results. In wireless sensor networks however these metrics are generally not significant and query optimization should rely on cost models based on other metrics. In particular, given that network lifetime plays the most important role, a query should be optimized with respect to the energy required.

In the following we define criteria to evaluate the cost of a query by considering the energy required to process it.

We measure the cost of a query execution plan by evaluating the power P needed to process it, that is the energy E consumed per unit of time ($P = E/t$). During the execution of a query, each operator receives a record from one or more streams and, depending on the operator semantics, it sends a record through another stream. Accordingly, given that the query execution activity is dominated by records traversing streams, we measure the cost by considering the energy required to send records across streams. The cost required by an operator to process a data received from a stream is in fact negligible with respect to the cost of sending data in a stream.

Let $E(S, s)$ be the energy required to send a single record of size s across the stream S. Let $f(S)$ be the frequency of records traversing the stream S. The cost of stream S, that is its power, is:

$$P(S) = f(S)E(S, s)$$

The cost of a query execution plan, say QEP, is the sum of the cost of its streams. Let $\mathbf{S}$ be the set of streams contained in the query execution plan QEP. The cost of QEP is

$$P(QEP) = \sum_{S \in \mathbf{S}} P(S)$$

In order to evaluate the previous expressions, we need to know 1) the energy required to send a record across each stream and 2) the frequency of records that traverse each stream. We will discuss these issues in the following.

7.1 Energy required for sending a record

The energy required to send data across a stream depends on the type of stream considered. The cost of sensor streams is dominated by the energy required to sample a value with the associated transducer. The cost of a local stream is that needed to store the records in the temporary buffer. The cost of a remote stream is dominated by the activity of the radio interfaces of the nodes in the multi-hop path from the source node to the destination node.

Since the records of sensor streams have fixed size and layout, the cost of a record that traverses the stream is independent of the size of the record and it solely depends on the transducer used. Given a sensor stream SS associated with transducer TR, we have:

$$E(SS, s) = energy_per_sample(TR).$$

Table 1 shows the energy consumption for a single sample from various transducers of the sensor board MTS, all produced by Crossobow [1].

For what concerns local streams the energy required to store a record in main memory is negligible with respect to the cost incurred by the other types of stream. Thus we can reliably consider a zero cost in this case. Given a local stream LS we have

Transducer	Energy per Sample (mJ)
Thermistor	0.0000891
Accelerometer	0.03222
Magnetometer	0.2685

Table 1. Energy required for a sample from various transducers of MTS310CA boards.

$$E(LS, s) = 0.$$

In case of remote streams the situation is a bit more complex. Transmission of a tuple along a remote stream requires that the nodes involved in the corresponding multi-hop path collaborate to forward the tuple toward the destination node. Determining the cost of using a remote stream depends on the specific radio communication strategy used in the underlying network.

For instance, [17] proposes a protocol based on preamble sampling, where nodes periodically switch their radio on to check if a message is coming. An incoming message is detected by listening for a specific preamble signal. When the preamble is detected the radio is kept switched-on for the entire duration of the message. The cost of periodic radio sampling is negligible with respect to the cost of maintaining the radio in receive mode. The B-MAC protocol [27] makes similar assumptions on the radio activity, and it is comparable from the energy consumption perspective. [7] proposes an energy-efficient multi-hop communication protocol that synchronizes the radios of the nodes in a multi-hop channel, so that their radios are switched on only when needed. This method can be combined with the preamble sampling technique to further reduce the cost.

According to these protocols, we disregard the cost of listening (that is, preamble sampling), thus we assume with a reliable approximation that only transmissions incur in costs. This is also compliant with the cost model proposed in [16], where it is assumed that the nodes are synchronized, so that the destination node starts listening just when the sending node begins transmitting.

The transmission cost includes the cost payed by the sender, the cost payed by the receiver, and the cost payed by the internal nodes to forward the message. Let $E_t(s)$ be the energy required for transmitting a record of size s over the radio interface, and $E_r(s)$ be the energy required for receiving it. Thus the energy required to send a record of size s over a remote stream RS along a n hops path can be approximated by

$$E(RS, s) = n(E_t(s) + E_r(s))$$

The above equation does not take into account interferences in the network and other issues that can affect the actual energy consumption. This energy estimation is just meant to judge the quality of a query execution plan, rather than precisely computing the actual consumed energy. For a more rigorous estimation of the energy consumption using specific communication protocols, we refer to the articles where the protocols were proposed and analyzed [17], [27], [7].

As an example we consider the energy consumption required to transmit and to receive a record of 50 bytes of the MICA2 and MICAz motes [1] shown in Table 2.

Mote	$\mathbf{E_t/E_r}$	Energy per record (mJ)
MICA2	$E_t(50)$	1.478169643
MICA2	$E_r(50)$	0.528729911
MICAz	$E_t(50)$	0.1494225
MICAz	$E_r(50)$	0.161445

Table 2. Energy required for sending and receiving data on MICA2 and MICAz motes.

Operator	$f(S_O)$
$S_O \leftarrow \pi_{_}(S_I)$	$f(S_I)$
$S_O \leftarrow \sigma_{pred}(S_I)$	$f(S_I) \cdot \mathtt{Pr}(pred = true)$
$S_O \leftarrow \bowtie (S_{I_1}, S_{I_2})$	$\min\{f(S_{I_1}), f(S_{I_2})\}$
$S_O \leftarrow \cup(S_{I_1}, S_{I_2})$	$f(S_{I_1}) + f(S_{I_2})$
$S_O \leftarrow agg(S_{I_1}, S_{I_2})$	$\min\{f(S_{I_1}), f(S_{I_2})\}$
$S_O \leftarrow t_0, t_e\gamma_{_}(S_I)$	$1/t_e$

Table 3. Frequency for local and remote streams connecting various operators

Statistics on the number of hops between two nodes can be obtained or estimated at the base station where the queries are submitted and optimized. In many cases this number is proportional to the distance between two nodes [8]. Therefore, we can express the energy needed to send a record of size s across a remote stream S, where source and destination nodes have distance d as:

$$E(S, s) = d \cdot c \cdot (E_t(s) + E_s(s)),$$

where c is a tuning parameter that depends on the density and transmission range of the nodes in the network.

7.2 Frequency of records in streams

The frequency of records across streams depends on the periodicity of the data acquisition of the sensor streams, and on the specific operators used to connect streams.

In case of sensor streams, we distinguish between periodic and on-demand streams. In a periodic stream S_p with period p data are acquired and sent with frequency $f(S_p) = 1/p$. An on-demand stream is intended to be used as input to a sync-join operator as in $S_O \leftarrow \bowtie_{sync} (S, S_{od})$, where S_{od}, is the on-demand sensor stream. We have that $f(S_{od}) = f(S)$ given that a record is requested from S_{od} every time a record arrives from S.

The frequency of local and remote streams depends on the operators that write in the streams and on the stream(s) where that operators read. The various possibilities are summarized in Table 3.

The output stream frequency of a projection operator is the same as the input stream of the operator. In case of the selection operator, the frequency of the output stream is that of the input stream multiplied by the probability that the selection predicate is true. This probability depends on the phenomenon measured. For instance, if the selection is used to filter an alarm situation, the probability that the predicate is true will be very

low. The frequency of the output of a join is the minimum of the frequencies of the input streams: the output record is produced only when a tuple is available from both input streams at the same time. The frequency of the output of an union is the sum of the frequencies of the input streams: as soon as a record is received on either stream it is written to the output stream. The frequency of the output of a spatial aggregate is the minimum of the frequencies of the input streams: as for the join the output aggregate is produced only when a record is available from both input stream at the same time. Finally, the frequency of the output of a temporal aggregate is equal to the inverse of the epoch interval (period) t_e.

8 Query optimization

Given a MW-SQL query there are several semantically equivalent query execution plans (defined in terms of combinations of operators of the query algebra and streams) that can be used to process it. Different query plans typically imply different query execution costs, thus the task of the query optimizer is to choose the query plan with the minimum cost.

In this paper we consider an algebraic optimization approach, that is based on transformation rules transforming a query plan into a semantically equivalent one with a lower cost. The final query plan is obtained by applying successive transformations to an initial query plan built from the MW-SQL query.

We will also discuss some issues related to the ordering of the operators, which can be affected by the transformation rules. Operators can be ordered according to topological considerations to favor communications along short paths, and according to the selectivity of selections and to the cost of data acquisition.

In the remainder of this section we use the following notation.

Each operator in a query plan is executed on a specific node. This is denoted by associating a superscript with each operator. When the superscript is not specified or when it is "_", the localization of the operator is not important. For instance, $\bowtie^j$ indicates that the join operator is executed on Node j, while $\bowtie^{_}$ indicates that the localization of the join operator is not relevant.

With the exception of sensor streams, we do not explicitly denote the streams types used to connect operators. Remote streams are implicitly used when two operators are located on different nodes, and local streams are implicitly used when two operators are located on the same node. For instance, in $\sigma_p^i(\bowtie^j (A, B)), i \neq j$, the join operator is connected to the selection with a remote stream, given that they are located on different nodes.

We use letters A, B, and C to denote generic expressions composed of combinations of various operators. We use ξ^* to denote an expression composed of a possibly empty sequence of unary operators π and σ.

When expressions are associated with a superscript, it means that the most external operator is executed on the indicated node. For instance, the most external operator in A^j is executed in node j. We denote the most external operator of an expression by $Root(A)$. We denote with $Attr(A)$ the attributes returned by the expression A, that is the attributes of the output stream of A.

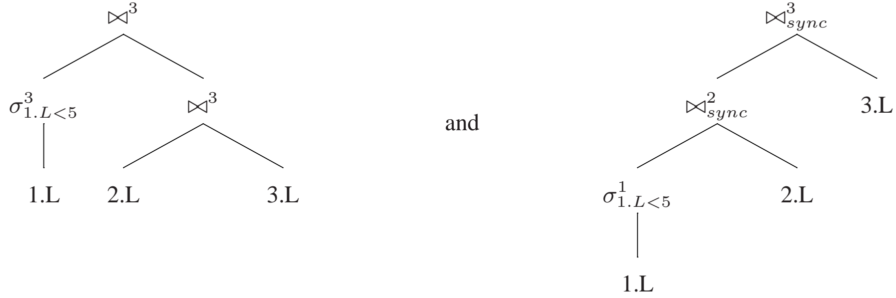

Fig. 6. An example of two equivalent query plans.

We also denote sensor streams using $\breve{S}$, and we (optionally) use a superscript to denote the node where it is located. In addition, when a sensor stream is used on the right side of a sync-join we always mean that it is an on-demand sensor stream.

8.1 Transformation rules

Several transformation rules proposed in the literature to optimize traditional database query execution can be applied in our context. For instances rules to push-down selections and projections, and selectivity-based ordering of selections are very useful since they contribute to reduce the amount of data to be transferred upward in a query plan. This implicitly reduce the amount of data traversing remote streams, and, in turn, it reduces the amount of radio activity and of energy consumed.

Here we discuss some transformation rules that are particularly useful in our context since they make optimal use of the data model and of the operators that we have defined.

An important feature of our approach is the possibility to use on-demand sensor streams, which acquire data only when requested. Their usage in conjunction with sync-joins and selections can reduce the energy required to acquire data. Consider the following query:

SELECT *
FROM 1.Light, 2.Light, 3.Light
WHERE 1.Light $<$ 5
EVERY 1000

Two possible equivalent query plans that process the previous query are shown in Figure 8.1. The query plan on the left acquires light readings from nodes 1, 2, and 3 every second. Node 1 sends all acquired data to Node 3 where data is filtered before being passed to the join. Node 2 also sends all acquired data to Node 3 to be joined with data acquired by Node 3. Obviously, acquisitions from nodes 2 and 3 are not necessary

when light on Node 1 is above 5, but they consume energy. In addition, data acquired by Node 1 and Node 2 are always sent to Node 3, even if they are not needed.

A more efficient query plan is shown on the right. This query plan acquires light every second on Node 1. Node 1 itself filters acquired data and it sends the data to Node 2 only when needed. Node 2 executes a sync-join where the left stream is a remote stream and the right stream is an on-demand sensor stream which acquires light in Node 2. Data is acquired by the on-demand stream only when a record is received from the remote stream. The result of the sync-join is sent to Node 3 through a remote stream and it is used as input to another sync-join. In turn Node 3 acquires the light reading only when a data arrives from Node 2. Summarizing Node 1 systematically acquires data every second. Suppose now that the probability that light on Node 1 is below 5 is very low. Under this assumption, transmissions of data from Node 1 to Node 2, from Node 2 to Node 3, and acquisitions of light readings in Node 2 and 3 are very rare and a lot of energy is saved.

The observations on the previous example help us to define some useful transformation rules.

1. Sync-join and on-demand streams should be used whenever possible.
2. Given that a sync-join requires a sensor stream on the right side, trees representing query plans should be unbalanced to the left (Left Deep Join Trees). In this way, the chance that a sensor stream (a leaf node) is found as the right argument of a join is increased.
3. Unary operators such as selections, projections, and temporal aggregates (which reduce the amount of data being forwarded) should be moved as close as possible to the node where data is acquired.

Transforming a join into a sync-join: According to our previous observations we define rules that transform a join into a sync-join. The idea is that if a periodic sensor stream is on the right side of a join, the join can be transformed into a sync-join and the sensor stream into an on-demand sensor stream. If there are some unary operators between the sensor stream and the join, the unary operators can be moved after the join (to process the output of the join) and the transformation can still take place. Note that even if the unary operators are moved up, this is not a problem, given that the sensor stream is activated only if needed.

Formally the transformation rule is the following:

$$\frac{\bowtie^k (B, \xi^*(\breve{S}^h)) \qquad \text{where } \xi^* \neq \emptyset \vee k \neq h}{\widetilde{\xi^h}(\bowtie^h_{sync} (B, \breve{S}^h))} \tag{1}$$

where $\widetilde{\xi^h}$ is obtained from the sequence ξ^*, where each π_X is transformed to $\pi_{X \cup Attr(B)}$, and all elements are localized on Node h.

If the sensor stream is on the left side of the join we can use the following rule, which exploits the commutative property of joins:

$$\frac{\bowtie ((\xi^*(\breve{S}^h), B)) \qquad \text{where } B \text{ is not a sensor stream}}{\widetilde{\xi^h}(\bowtie^h_{sync} (B, \breve{S}^h))} \tag{2}$$

Obtain Left Deep Join Trees: The following rule rearranges the joins in a query plan to obtain a left deep join tree, and facilitate transformation of joins into sync-joins by means of the previous rules.

$$\frac{\bowtie^{-} (A^h, \xi^*(\bowtie^{-} (B^k, C^j)))}{\widetilde{\xi^{j*}}(\bowtie^j (\bowtie^k (A^h, B^k), C^j))} \tag{3}$$

where $\widetilde{\xi^{j*}}$ is obtained from ξ^* by transforming all π_X into $\pi_{X \cup Attr(A)}$, and all elements are located on Node j.

This rule can be profitably used with standard rules to push down selections and to order joins and selections so that the most selective selections are performed first, reducing the amount of data traversing the tree.

Moving unary operators close to the source of data: Unary operators are moved close to the source of the data, as suggested by the third observation, by using the following rules in addition to traditional push-down transformation rules.

$$\frac{\pi^h_X(A^k) \quad \text{where} \quad h \neq k}{\pi^k_X(A^k)} \tag{4}$$

$$\frac{\sigma^h_c(A^k) \quad \text{where} \quad h \neq k}{\sigma^k_c(A^k)} \tag{5}$$

$$\frac{t_0, t_e\gamma^h_{\{agg_1(a_1),..,agg_n(a_n)\}}(A^k) \quad \text{where } h \neq k}{t_0, t_e\gamma^k_{\{agg_1(a_1),..,agg_n(a_n)\}}(A^k)} \tag{6}$$

8.2 Query optimization example

Let us suppose that we submit the following MW-SQL query:

SELECT *
FROM 1.Magnetism, 2.Acceleration, 3.Temperature
WHERE p_1(1.Magnetism)
and p_2(2.Acceleration)
and p_3(3.Temperature)
EVERY 1000

where p_1, p_2, and p_3 are some predicates on magnetism, acceleration and temperature readings, respectively, with probability $\texttt{Pr}(p_1) = 0.01, \texttt{Pr}(p_2) = 0.05, \texttt{Pr}(p_3) = 0.1$, respectively.

Figure 7 shows three possible equivalent query plans that can be used to process the above query.

QP1, on the left, is obtained by applying the left deep join trees rule. It first acquires all specified data and then joins them before applying the three selections on the last node. This requires that all magnetism readings be sent to Node 2 and joined with the

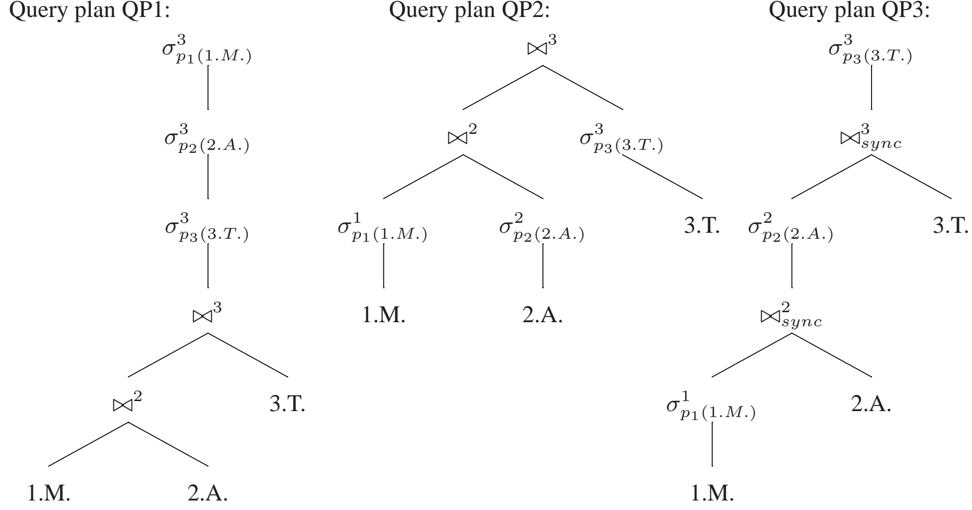

Fig. 7. Three possible execution plans for the same query: QP1, on the left, is obtained applying ”left deep join tree” rule on an initial query plan; QP2, in the middle, is obtained from QP1 by applying push-down of selections, and allocation of selection on the node where data are generated; QP3, on the right, is obtained from QP2 by transforming joins into sync joins.

acceleration readings. The result of the join is sent to Node 3 where it is joined with the temperature reading and then the three selections are applied.

QP2, in the middle, is obtained from QP1 by using the selections push-down rule and their allocation on the node where data are generated. In this query plan all data must be acquired. However, magnetism is sent to Node 2 only if it satisfies p_1. The join on Node 2 is thus executed only if both p_1 and p_2 are satisfied and in this case the result is sent to Node 3. The join in Node 3 is executed only if all three predicates are true, and in this case the result is sent to the sink.

QP3, on the right, is obtained from QP2 by using rules for transforming joins into sync-joins. In this case, magnetism is always acquired. It is sent to Node 2 if it satisfies predicate p_1 and in this case the acceleration is also acquired and joined with the magnetism. If p_2 is satisfied, the result is sent to Node 3 and temperature is acquired. If p_3 is satisfied the result is eventually sent to the sink.

Costs for these query plans are analyzed in Table 4 where all actions relevant for cost estimations, according to Section 7, are listed together with their required energy. The frequency of these actions and the corresponding power are listed separately for the various query plans. The total cost for every query plan is reported as well. For simplicity, in this example, we suppose that all remote streams consist of single hop paths. In next section we will discuss the case of multihop paths.

As reported in the table, the cost of QP2 is approximatively $1/3$ of QP1. Cost of QP3 is a slightly better than QP2. This means that the expected lifetime of a network running QP2 or QP3 is about 3 times longer than the lifetime expected when running QP1. The lower cost of QP2 with respect to QP1 is due to the reduced number of

		QP1:		QP2:		QP3:	
Action	**Energy(mJ)**	**Freq.**	**Power**	**Freq.**	**Power**	**Freq.**	**Power**
Acquire M.	0.2685	1	0.2685	1	0.2685	1	0.2685
Send M.	0.31087	1	0.31087	0.01	0.00310	0.01	0.0031
Acquire A.	0.03222	1	0.03222	1	0.03222	0.01	0.00032
Send M.A.	0.31087	1	0.31087	0.0005	0.00016	0.0005	0.00016
Acquire T.	0.00009	1	0.00009	1	0.00009	0.0005	4.46E-08
Send M.A.T.	0.31087	5.0E-5	1.55E-05	5.0E-5	1.55E-05	5.0E-5	1.55E-05
Total Cost:			0.92256		0.30408		0.2721

Table 4. Costs of the three executions plans in Figure 7. We suppose that selectivity of selections is as follow: $\texttt{Pr}(p_1) = 0.01, \texttt{Pr}(p_2) = 0.05, \texttt{Pr}(p_3) = 0.1$. We also suppose that all remote streams consist of single hop paths.

communications that it requires. The lower cost of QP3 with respect to QP2 is due to the combined reduction of communications and acquisitions.

The performance improvement of QP3 with respect to QP2 is very limited. However, we will show in the next section that the use of sync-joins, as produced for QP3, with appropriate ordering of operators can provide significant performance improvements.

8.3 Ordering of operators

In the previous section we have seen that QP3 is the best query plan among the considered ones. However, several other equivalent query plans maintaining the same structure of QP3 can be obtained by changing the order of the operators in the tree. Figure 8 shows three different query plans with the same structure but with a different order of the operators.

Operators can be ordered according to different criteria, leading to different performance depending on the query and the statistics about data and node distribution.

Here we discuss three different ordering criteria. Operators can be ordered so that i) more selective selections are pushed down in the tree; ii) less selective operators are pushed down in the tree; iii) cost of remote connections is minimized.

The first criterion give precedence to very selective predicates to filter immediately useless data, thus reducing communications and data acquisitions by means of sync-joins.

The second criterion gives precedence to low cost acquisitions. High cost acquisitions are thus executed with low probability since they are high in the tree, and the data collected at the lower levels of the tree must pass the selections first.

The third criteria reduces the communication costs by choosing an ordering of the operators and their allocation to the nodes such that the multihop communication paths are shortened.

Examples of the application of these criteria are given in Figure 8. Differently from the previous section multihop paths are taken into account here.

In QP3 operators are ordered according to criterion i), criterion ii) is used in QP4, and criterion iii) is used in QP5.

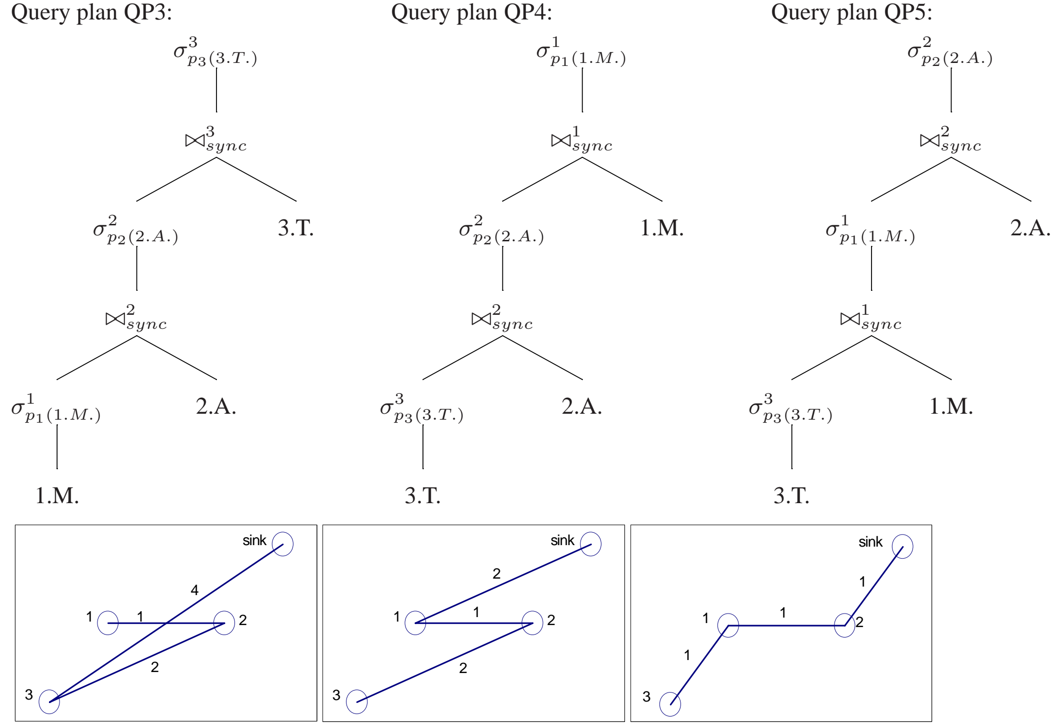

Fig. 8. Three possible execution plans for the same query using joins. Below each execution plan is shown the geographical placement of nodes and the remote streams connecting the nodes (remote streams are labeled with their number of hops). In QP3 predicates and joins are ordered according to selectivity; in QP4 they are ordered according to the cost of acquisition, and in QP5 they are ordered according to topology.

The costs of QP3, QP4, and QP5 are given in Table 5. It is seen that the cost of QP4 (0.068 mJ) is one order of magnitude smaller than the cost of QP3 (0.27 mJ). The cost of QP5 (0.058 mJ) is slightly smaller than the cost of QP4. This means that the expected lifetime of a network running QP4 or QP5 is about 5 times longer than the lifetime expected when running QP3.

However, this is not a proof that ordering according to the topology of the network is always the best solution. The results can vary depending on the selections selectivity and on the acquisitions costs. In general, there is not a best ordering strategy. The optimizer must generate different orderings according to the various criteria and choose the one providing the best performance. As shown in our example, this may lead to performance improvements of orders of magnitude.

However we observe that that the ordering according to the topology of the network is always useful in those cases where the number of acquisitions and communications cannot be reduced. This is the case of queries without selection predicates or queries with spatial aggregates.

QP3:			
Action	**Energy(mJ)**	**Freq.**	**Power**
Acquire M.	0.2685	1	0.2685
Send M.	0.31087	0.01	0.00311
Acquire A.	0.03222	0.01	0.00032
Send M., A.	0.62174	0.0005	0.00031
Acquire T.	0.00009	0.0005	4.46E-08
Send M., A., T.	1.24347	0.00005	6.21E-05
Total Cost:			0.2723

QP4:			
Action	**Energy(mJ)**	**Freq.**	**Power**
Acquire T.	0.00009	1	0.00009
Send T.	0.62174	0.1	0.06217
Acquire A.	0.03222	0.1	0.00322
Send T., A.	0.31087	0.005	0.00155
Acquire M.	0.2685	0.005	0.00134
Send T., A., M.	0.62174	0.00005	3.11E-05
Total Cost:			0.06841

QP5:			
Action	**Energy(mJ)**	**Freq.**	**Power**
Acquire T.	0.00009	1	0.00009
Send T.	0.31087	0.1	0.03109
Acquire M.	0.2685	0.1	0.02685
Send T., M.	0.31087	0.001	0.00031
Acquire A.	0.03222	0.001	0.00003
Send T., M., A.	0.31087	0.00005	1.55E-05
Total Cost:			0.05838

Table 5. Cost of the query plans QP3, QP4, and QP5. Differently from Table 4 here the number of hops for remote streams is not supposed to be always 1. In this case ordering according to selectivity is clearly better

Consider for instance the query plans QP7 and QP8 (shown in Figure 9) of the following query:

SELECT *
FROM avg(1.Temperature,2.temperature, 3.Temperature)
EVERY 1000

In QP7 operators are ordered so that shorter communications paths are used. As reported in Table 6 the cost of QP7 is about 4 times smaller than that of QP6. If the number of nodes involved in the query is high further improvements can be expected.

9 Conclusions

In this paper we have presented a comprehensive and consistent approach to query processing in wireless sensor networks.

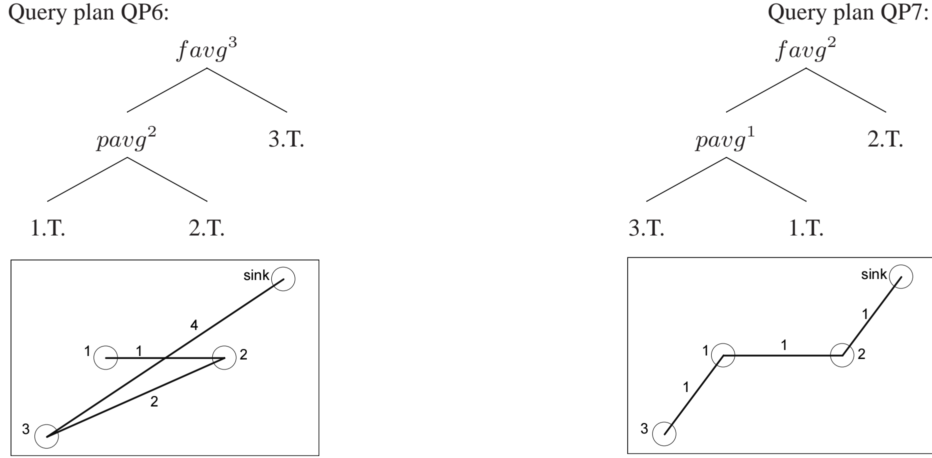

Fig. 9. Two possible execution plans for the same query using spatial aggregates. They differ on the ordering of operators. Below each execution plan is shown the geographical placement of nodes and the remote streams connecting the nodes. Remote streams are labeled with their number of hops. The operators order in QP7 provides the shortest paths.

QP6:		**QP7:**	
Action	**Power**	**Action**	**Power**
Acquire 1.T	0.00009	Acquire 3.T	0.00009
Send 1.T	0.31087	Send 3.T	0.31087
Acquire 2.T	0.00009	Acquire 1.T	0.00009
Send avg(1.T,2.T)	0.62173	Send avg(3.T,1.T)	0.31087
Acquire 3.T	0.00009	Acquire 2.T	0.00009
Send avg(1.T,2.T,3.T)	1.24347	Send avg(3.T,1.T,2.T)	0.31087
Total Cost:	2.17634		0.9328698

Table 6. Cost of the query plans shown in Figure 9.

To provide efficient support to query execution we have defined, analyzed, and discussed the aspects related to data modeling, query algebra, and query optimization. Our approach offers many alternatives for processing a query and provides several opportunities for query optimization. In particular our approach allows the selection of an optimal query execution plan for a given query, according topology of the network, data statistics, and types of transducers. We show that accurate query optimization may provide a reduction of the query execution cost of some orders of magnitude.

The proposed approach maintains separate the aspects related to communication, data acquisition, data representation, and data processing, and it gives the opportunity to experiment new strategies related to each of these aspects without affecting the entire system design. In particular a new communication protocol, a new type of transducer, a new query processor can be used still being able to use the features of the other components.

References

1. Crossbow Technology Inc., http://www.xbow.com.
2. MaD-WiSe: Management of Data in Wireless Sensor networks. http://mad-wise.isit.cnr.it.
3. TinyOS. http://www.tinyos.net/.
4. D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.
5. D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.
6. I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communication Magazine*, 40(8):102–114, 2002.
7. G. Amato, P. Baronti, and S. Chessa. Connection-oriented communication protocol in wireless sensor networks. Technical Report 2005-TR-10, ISTI-CNR, 2005. http://dienst.isti.cnr.it/Dienst/UI/2.0/Describe/ercim.cnr.isti/2005-TR-10.
8. C. Antonio, C. Tamalika, and C. Stefano. Bounds on Hop Distance in Greedy Routing Approach in Wireless Ad Hoc Networks. *International Journal on Wireless and Mobile Computing*. To appear.
9. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.
10. M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *SIGMOD Conference*, pages 13–24, 2005.
11. F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez. Fad, a powerful and simple database language. In P. M. Stocker, W. Kent, and P. Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 97–105. Morgan Kaufmann, 1987.
12. P. Baronti, P. Pillai, V. Chook, S. Chessa, A. Gotta, and Y. F. Hu. Wireless sensor networks: a survey on the state of the art and the 802.15.4 and zigbee standards. Techical Report ISTI-2006-TR-18, ISTI-CNR, 2006. http://dienst.isti.cnr.it/.
13. P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Mobile Data Management, Second International Conference, MDM 2001, Hong Kong, China*, volume 1987 of *LNCS*.
14. A. Caruso, S. Chessa, S. De, and A. Urpi. Gps free coordinate assignment and routing in wireless sensor networks. In *Proceedings of IEEE INFOCOM*, 2005.
15. E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.
16. A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB*, pages 588–599, 2004.
17. A. El-Hoiydi. Aloha with preamble sampling for sporadic traffic in ad hoc wireless sensor networks. In *IEEE International Conference on Communications*, pages 3418–3423, 2002.
18. L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.
19. C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th annual ACM/IEEE international conference on mobile computing and networking, Boston, MA, USA*, pages 56–67, 2000.
20. ISTI-CNR, Via G. Moruzzi, 1, 56124, Pisa, IT. *SensorViz/MaD-WiSe*, version 1.3 edition, July 2006. http://mad-wise.isti.cnr.it/manual_13.pdf.

21. D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.
22. Y.-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *VLDB*, pages 492–503, 2004.
23. S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *In 18th International Conference on Data Engineering, ICDE 2002, San Jose, CA, USA*, pages 555–566, 2002.
24. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA*, 2002.
25. S. 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.
26. C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD Conference*, pages 563–574, 2003.
27. J. Polastre, J. Hill, and D. E. Culler. Versatile low power media access for wireless sensor networks. In *SenSys*, pages 95–107, 2004.
28. P. Seshadri, M. Livny, and R. Ramakrishnan. Seq: A model for sequence databases. In *ICDE*, pages 232–239, 1995.
29. Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.