Integration of Heterogeneous Sensor Nodes by Data Stream Management

Michael Daum, Frank Lauterwald, Martin Fischer, Mario Kiefer, and Klaus Meyer-Wegener

Abstract *Wireless Sensor Network*s (WSNs) will be an important streaming data source for many fields of surveillance in the near future, as the price of WSN technologies is diminishing rapidly, while processing power, sensing capability, and communication efficiency are growing steadily. Data-stream analyses should be distributed over the entire network in a way that the processing power is well utilized, the sensing is done in a semantically reasonable way, and communication is reduced to a minimum as it consumes much energy in general. Surveillance experts of different domains need technical experts in order to deploy those distributed data stream analyses. Data-stream queries often realize data-stream analyses. Especially surveillance scenarios that base on *Sensor Data Fusion* (SDF) will need the integration of heterogeneous data sources produced by potentially heterogeneous sensor nodes.

This chapter overviews existing WSN middleware solutions, *Stream Processing System*s (SPSs), and their integration. An approach that maps a global data-stream query to distributed and heterogeneous sensor nodes and SPSs opens a path to solve the problems mentioned above. Integration is achieved in two ways: semantic integration is done implicitly by the partitioning and mapping using rules that retain the semantics of the global query through the entire distribution and deployment process; technical integration is achieved during mapping and deployment with the help of the knowledge about platforms and connections.

Michael Daum
Chair for Computer Science 6 (Data Management), University of Erlangen-Nürnberg, Germany
e-mail: michael.daum@cs.fau.de

Frank Lauterwald
Chair for Computer Science 6 (Data Management), University of Erlangen-Nürnberg, Germany
e-mail: frank.lauterwald@cs.fau.de

The original publication is available at www.springerlink.com (DOI: 10.1007/978-3-642-13965-9)

1 Objectives

*Wireless Sensor Network*s (WSNs) consist of nodes that are widely distributed. These sensors are the data source for data stream processing in many scenarios. Data stream processing is an option for distributed processing that is more efficient than sending all data to a central processing unit. As data volume grows rapidly, the central processing approach becomes more and more impracticable. Data processing has to be distributed in a way that operators like filtering and aggregation can help to reduce communication already in the vicinity of the sensors. We expect complex scenarios with many distributed data sources as well as complex event processing and complex sensor data fusion in the near future. These scenarios will add new operators to the well-known set of operators.

Yet programming these scenarios should be simplified in the sense that users may write declarative queries instead of procedural code in some programming language. These queries combine expressions of operators in a very compact notation. The given scenarios may add new operators. From our point of view, sensor network nodes are *Stream Processing System*s (SPSs) with limited capacities. As the SPSs are heterogeneous, their query languages are heterogeneous, too. There are approaches that configure WSN nodes by using SQL-based query languages; others use graph-based query languages that correspond to the data flow. At least, most WSN nodes can be individually programmed in programming languages that are similar to C. We subsume all kinds of systems that process streams under the term SPS and call each instance of an SPS a node. Besides sensor network nodes, Data Stream Management Systems (DSMSs) are also SPSs. They are often coupled with WSNs to process the data streams produced by sensors. This requires an integration of stationary SPSs and WSNs. The query languages of DSMSs are more powerful and they are not limited with respect to power supply and communications. The heterogeneity of query languages is caused by the heterogeneity of different configurable data sources. A directed graph of distributed nodes forms the processing network that can process a query. The deployment of a query has to consider topology, performance, and the nodes' heterogeneity. This abundance of crucial aspects will surely overwhelm domain experts who just want to observe a scenario by sensors. We envision the domain experts to define abstract queries without considering platform-specific constraints of the nodes, topology, etc. Only the query and the sensor data matter. For this purpose, this chapter follows the *Model Driven Architecture* (MDA) approach with our middleware prototype *Data Stream Application Manage*r (DSAM). This is facilitated by a central repository that stores the metadata of the observed scenario. It manages all data sources and offers a semantic description of sensor data.

1.1 Introductory Example

As a fictitious introductory example (Fig. 1), we assume that biologists want to use WSNs for the surveillance of the behavior of animals. The biologists want to find out in which area the observed animals are under stress and where they go to for recreation. A query determines the area of recreation as the animal's position at 10 min after the last stress event. It uses sensor nodes that can communicate with each other and may have different sets of sensors, different locations, and different sets of installed modules. An example is depicted in Figure 1. `Node1` measures *skin conductivity level* (SCL) with sensor `S1` and *body temperature* (TEMP) with sensor `S2`. `Node2` can communicate with the base station and has higher energy capacity than `Node1`. `S3` connected to `Node2` delivers *position* data.

```
(S1,S2,S3,TIME:$1.filter("SCL>8"),$2.filter("TEMP>38"),MERGE()),
(S3,TIME:MERGE()):
   JOIN("$2.TIME-$1.TIME > 10min && $2.TIME-$1.TIME > 12min",
      windows(size="1", size-by="TUPLES")):
POR
```

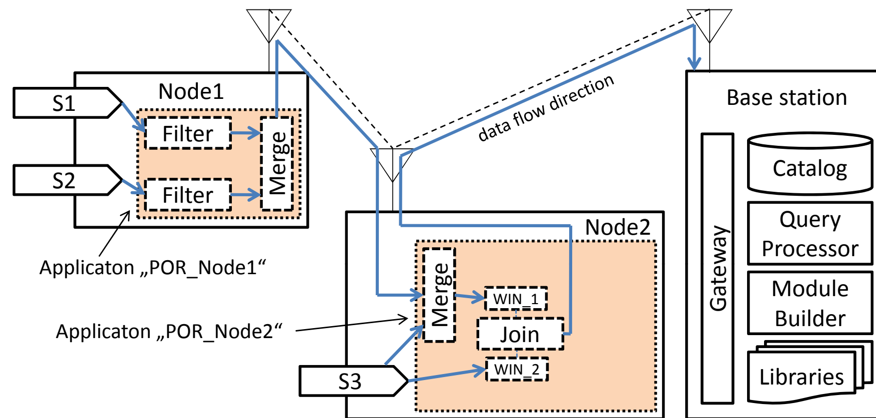

Fig. 1 Example query for animal surveillance

The locations of input and output streams, sensors, schema information, and topology are part of the catalog. In our example, we have three sensors that should be merged if:

- The animal's TEMP is higher than 38 °C and its SCL is greater than 8 Mho.
- The last position of a stress situation and the position of recreation (10 min later) are of interest.
- The observation at the points of recreation lasts at least 2 min.

The sample query has three input streams and one output stream. Further, the query has two subqueries in the input stream list. The first subquery has three abstract operators: one filter operator that selects all interesting SCL values from the

first stream, another filter operator that does the same for TEMP values from the second stream, and a ternary merge operator that merges all three streams after filtering the first and the second. The merge operator waits for all input events and creates one element including all inputs. The second subquery simply adds the current time to the sensor value. In our main query, the last items of the two subqueries are joined if the temporal condition is fulfilled. The last items are realized by sliding windows. The main difference between a merge and a join operator is that merge operators use input values only once in the resulting events. We will explain the query language in Sect. 3.4.

Users like behavioral scientists want to describe their needs using an abstract query language without considering the sensor network's topology in order to describe a query in a formal way. Further, there are different ways of defining partial queries and configuring WSN nodes. This leads to a top-down approach that uses an abstract query definition and does query partitioning, mapping, and deployment automatically. We will explain the partitioning of queries in Sect. 4. Handling distributed heterogeneous nodes will be described in Sect. 5.2.

1.2 The MDA approach

The *Model Driven Architecture* (MDA)[1] is a top-down approach for developing software systems. Its basic idea is mapping a *Platform Independent Model* (PIM) that is described in a *Domain Specific Language* (DSL) to code that can be executed. [21] gives a good overview of the ideas of the MDA approach. The PIM may e.g. be described in UML as it is in many business software systems. Alternatively, other OMG language standards and non-OMG languages can be the description language for the PIM, too.

A PIM is mapped to one or more *Platform Specific Model*s (PSMs) (Fig. 2). The information about the bridges between the PSMs is generated by the "first transformation". The "second transformation" is the generation of code that can be run on the platforms. The "code bridges" can be derived directly from the knowledge about the "PSM bridges".

Some of the main goals of MDA are the lowering of development costs by using software generators and a better manageability by using higher levels of abstraction. The same goals will be relevant for deploying queries and applications to a large number of heterogeneous WSN nodes. For domain experts, it will be relevant *what* the sensor network should do. They are neither interested in platform-specific restrictions of nodes nor in the programming or integration of nodes. In the introductory example (Fig. 1), the query describes the problem of the domain experts that is deployed to two nodes.

This chapter draws the parallels between the MDA and this approach. An abstract (generic, SPS independent) query and the semantic description of sensor data corre-

[1] `www.omg.org/mda`

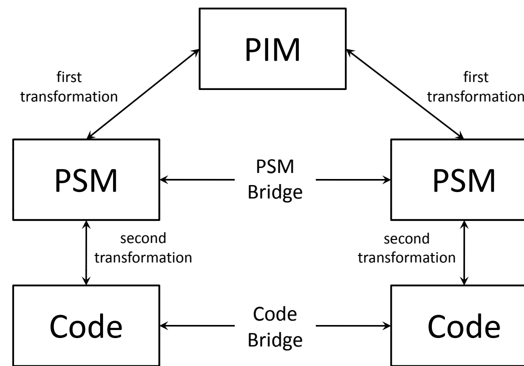

Fig. 2 Mapping between Models in the MDA approach [21]

spond to the Platform-Independent Model (PIM). The abstract query is partitioned and deployed to platform-specific nodes. To the best of our knowledge, considering platform-specific restrictions, topology, and cost estimations in the partitioning process together is a new approach in the field of WSNs and stream processing. The resulting set of partial queries and configurations is platform-specific and corresponds to the Platform-Specific Model (PSM) of the MDA approach. In the last step of generating platform-specific code, this approach supports the integration of heterogeneous platforms as the platform characteristics of the nodes are known.

1.3 Data Stream Application Manager

This chapter describes our efforts to address the research challenges that arose while building *Data Stream Application Manage*r (DSAM). DSAM is a prototype that provides a central manager of data stream applications. Its main goal is the integration of heterogeneous SPSs. As it is not practicable to send all data to a central site, DSAM supports distributed query processing (also called *in-network query processing* in WSNs). DSAM achieves this integration by using the MDA approach. This facilitates deploying code to heterogeneous nodes as well as creating adapters between the nodes.

From a stream processing point of view, a WSN in total is a configurable data source that behaves like an SPS (e.g. TinyDB [26]) and single WSN nodes can be seen as SPSs with limited capacity and computing power [14]. The introductory example shows the deployment of a query on two nodes. The base station symbolizes DSAM in a simplified way; DSAM consists of a central management component and distributed deployer components that can interact with WSNs. The presentation of DSAM's architecture is not in the focus of this chapter as we rather focus on its concepts.

Integration of heterogeneous data sources has been analyzed in other fields as well, e.g. in the form of wrappers and mediators for distributed databases [34]. The usage of wrappers for syntactic integration is similar for databases and data streams. Mediators are application-dependent components that represent domain-knowledge and as such are not part of DSAM. It is however possible to use predefined queries as mediators within DSAM.

1.4 Existing Approaches for the Integration of Sensor Networks

There are several projects that support *in-network query processing* in WSNs and their integration with SPSs. This section presents a survey of the most related projects compared to DSAM. The following sections will offer more detailed comparisons of DSAM and those projects in the specific contexts.

The *Global Sensor Networks* (GSN) project [5] offers a middleware that integrates heterogeneous sensor networks at network level. Each kind of sensor network is abstracted by a *virtual sensor*. A virtual sensor produces exactly one data stream. There is a wrapper for each supported sensor platform that transforms data from the different formats to the internal schema of GSN. SStreaMWare [20] integrates heterogeneous sensors by offering *Open Services Gateway initiative* (OSGi) services. It supports SQL-like queries. A sensor query service receives queries and distributes subqueries as tree-based query plans to different gateways that offer a gateway query service. Each gateway query service organizes a further decomposition of subqueries and sends the smallest units of subqueries to the proxies of the diverse sensors. The heterogeneous sensors are integrated by a proxy query service that offers a generic interface.

TinyDB [26] and Cougar [17, 35] are middlewares that facilitate SQL-like queries in a network of WSNs. Borealis [1] is an SPS that supports distributed stream processing by grouping operators on distributed Borealis nodes. Queries are defined by box-and-arrow diagrams. Grouping and distribution of operators has to be done manually. REED [4] realizes the integration of the WSN-application TinyDB [26] and Borealis. In [25], operators are pushed from Borealis to TinyDB manually. The results motivate us to make more efforts in *Cross Border Optimization* (CBO), i.e. distributing queries to both WSNs and SPSs.

Tab. 1 compares the different approaches of using WSNs as configurable data sources for stream processing with DSAM. The compared systems have similarities in their architectures. There are distributed nodes (GSN containers, SStreamWare's gateways, Borealis, etc.) with full capabilities of operators that integrate WSN nodes having lower capabilities. TinyDB is an exception: it manages only one WSN and has less capabilities compared to the other solutions that use full-fledged stream processing systems at higher level. The biggest differences of the approaches are the ways of integrating WSNs and propagating partial queries; Sect. 5.1 offers a detailed comparison. DSAM supports a direct propagation of partial queries to WSN nodes by using a top-down approach. The top-level query language of DSAM for global

queries is graph-based. DSAM can integrate systems that use SQL-like query languages by query mapping. The main focus of DSAM is integrating existing WSNs and higher-order stream processing systems. The challenges are query partitioning, query mapping, and integration. The main advantage of DSAM is its ability to integrate already existing systems and configurable data sources.

	GSN	**SStreamWare**	**Borealis/REED**	**TinyDB**	**DSAM**
Architecture	(distributed) autonomous GSN containers	distributed gateways and a central manager	distributed Borealis nodes	distributed homogeneous TinyOS applications	distributed middleware nodes and a central manager
Query Definition	SQL-based	SQL-based	graph-based	SQL-based	graph-based
Capabilities of QL	full	full	full	limited	full
Query Propagation to WSN	manual configuring of WSN and definition of virtual sensors	hidden by *Proxy Query Service* (PQS) PQS receives partial query; propagation unknown, but theoretically possible	manual partitioning, automatic "neighborhood optimization" pushing of selections and joins to REED/TinyDB	automatic	automatic code generation/ query mapping
Data Processing	processing of temporary relations derived from heterogeneous data sources	hierarchical flow from the PQSs to the *Gateway Query Service* (GQS) within the gateway and from all gateways to a central component	distributed processing of Borealis nodes having operators that can receive data from a WSN	distributed in-network stream processing	distributed in-network stream processing
Technical Integration	distributed adapters	implementation of PQSs	distributed adapters	*not required*	distributed adapters
Semantic Integration	temporary relations	automatic service lookup and binding service create global data schema	data stream schemas	virtual table "sensors"	data stream schemas and mapping
Metadata Management	definitions of virtual sensors	hierarchical lookup services	distributed catalog	distributed tables synchronized to the root node	global view on static and volatile metadata
Stream Processing Engine	GSN	SStreaMWare gateways and control site	Borealis	TinyOS applications and gateway	third party (Borealis, STREAM at the moment)
Support of Potential Users	virtual tables for SQL-queries	automatic service discovery	box- and arrow-diagrams	virtual table "sensors" and TinySQL	holistic top-down approach

Table 1 Comparison of existing approaches and DSAM

2 Stream Model and Metadata

Data-stream technologies can be a powerful tool for the integration of sensor networks as sensor data is streaming by nature. Their main promise is to ease the definition of complex queries by providing standardized tools and languages. The heterogeneity of sources however poses some integration problems. This section addresses two of these problems: The common definition of what comprises a stream and the required metadata that allows the global definition of queries.

The integration of heterogeneous streaming data sources needs a common definition of a global stream model that encloses most of the existing and relevant stream models. The global stream model is the basis for the global query language as the different stream models are for the different query languages. In complex distributed scenarios, both heterogeneity and topology of sensor nodes have to be considered. Automatic operator placement furthermore needs performance characteristics of nodes, communication paths, etc. There are different solutions in the literature that deal with metadata in distributed stream processing systems and sensor networks.

2.1 Stream Model

Deploying global stream queries to a heterogeneous network of nodes requires a common understanding of what comprises a stream. This section compares existing stream models, which are then used as a basis for a global stream model. There are some characteristics common to most stream models: Streams are "append-only relations with transient tuples" [7]. Transient tuples are either discarded after processing or explicitly stored. In data stream systems, tuples must have a well-defined schema; otherwise they cannot be processed. The SPS cannot guarantee any further properties for the arriving data stream tuples, because they are created externally to the system.

An SPS may use different stream models internally and externally. The external stream model describes the properties a stream needs to have so that an SPS can process it, while the internal stream model is used inside the SPS. The external stream models determine which and how SPSs may be integrated with each other, while the internal stream models influence the operator semantics (Sect. 3.1).

In the examination of existing stream models, we regard the following criteria as relevant for a classification:

- Timestamps and validity
  Most SPSs use timestamps to denote when an event happened. Timestamps may be generated by the data sources (external timestamps) or by the SPS upon arrival of an item (internal timestamps). A second timestamp may be added to denote how long an item is valid. Without this information, it may be necessary to denote the validity of items in the query definition. Both variants may be combined (e.g.

even if an item has an expiry date, the system may discard items at some other time earlier or later which is given in the query definition). Timestamps can be part of either the user data or the metadata. In the first case, queries may access (and possibly even alter) timestamps. In the second case, timestamps can only be used by the SPS internally.

- Uniqueness of items
  Stream models differ in their guarantees for uniqueness of items. It is usually not possible to guarantee uniqueness of user data (a sensor node may return the same temperature value several times). If uniqueness of items is required, this has to be done via timestamps. In the absence of timestamps or if timestamps do not guarantee uniqueness (e.g. because of insufficient granularity), a simple sequence of numbers may be used. Uniqueness is usually more of a concern for the mathematically precise definition of semantics than it is for actual query processing.

In addition to stream models, existing SPSs also differ in their delivery semantics. The following criteria may be used to distinguish systems.

- Lost or duplicated items
  Systems may react differently to lost or duplicated items. Unless there is a notion of a “next” item (e.g. by means of sequence numbers), it is not possible to detect if tuples have been lost.
- Ordering
  Another interesting difference between SPSs is the issue of ordering. For a data source with either timestamps or sequence numbers, it is possible to wait for out-of-order items and reorder them. This can be realized by different SPS-specific techniques like sorting with slack parameters [2], heartbeats [6], punctuations [24], or k-constraints [8]. If more than one data source is connected to an SPS, reordering depends on synchronized clocks.

2.1.1 Comparison of Stream Models

This section compares the models of several SPSs in order to identify their similarities and differences.

STREAM

STREAM [6] has a relational stream model. Each stream S is a bag (i.e. multiset) of items $< s, \tau >$. $\tau \in T$ is an ordered set of discrete timestamps. The timestamp is not part of the stream’s schema. There can be more than one item with the same timestamp. As a stream consists of a multiset of items, uniqueness is not guaranteed. STREAM defines a relation as a multiset, too. Timestamps are external and the items are supposed to arrive in the correct order.

PIPES

PIPES [23] distinguishes between *raw*, *logical*, and *physical streams*. A *raw-stream item* (e,t) is an event that comes from an external data source and occurred at time t. System timestamps can be added to an event if no timestamp comes from the data source; when tuples arrive at the system, the input streams are implicitly ordered by system time. Items with external timestamps are expected to arrive in correct order. Otherwise an extra component reorders the items.

Logical-stream items (e,t,n) have multiplicity n of elements e at a definite point in time t. The logical stream definition is used for defining the logical algebra.

Physical streams are an outstanding concept of PIPES. *Physical-stream items* $(e,[t_S,t_E))$ have two discrete timestamps. t_S is the first point in time that the event e is valid at and is identical to the timestamp of the raw stream. t_E is the first point in time at that the event e is not valid anymore; it is added by the window operator depending on the window size (windows are explained in Sect. 3.2). *Physical streams* are important for invalidation of tuples in sliding windows.

From our point of view, only the *raw stream definition* is relevant to us, as it is the stream definition at the entry point of PIPES.

Aurora

Aurora [2] has a stream model $(TS,A_1,\ldots,A_n)$ that is similar to the stream models of STREAM and PIPES. The timestamp is hidden from queries. Borealis [1] is the successor of Aurora. In Borealis the tuple of a data stream has the following schema[2]:

Key				Attribute			
Tuple-ID	Tuple Type	t_g	t_l	A_1	A_2	$\ldots$	A_n

Queries can only use the attributes $A_1,\ldots,A_n$. t_g is the global timestamp that is added by the system upon arrival of an item. t_l is only used internally for measuring *Quality of Service* (QoS) parameters. Uniqueness is guaranteed by the tuple-ID. In this model, the timestamp t_g is not the only relevant attribute for the correct order of tuples. Ordering can be required for each user data attribute. Operators in Aurora and Borealis can deal with disordered tuples by using so-called *slack* parameters for ordering. As Aurora and Borealis support *load shedding*, they can deal with tuple losses.

Cayuga

Cayuga [13] has a stream model that is founded on expressive pub/sub systems [12]. Events $< \overline{a};t_0,t_1 >$ have a starting point in time t_0 and an endpoint in time t_1. $\overline{a}$ is just an abbreviation for a relational tuple. Instantaneous events have identical

[2] Code analysis of Borealis Version: Summer 2008

timestamps t_0 and t_1. The temporal order is defined by t_1, i.e. event e_1 is processed before e_2 if $e_1.t_1 \leq e_2.t_1$.

GSN

GSN [5] uses timestamped tuples, too. The management of the timestamp attribute (TIMEID) is implicit, i.e. all arriving tuples are timestamped automatically by a local clock. Timestamps can be used like normal attributes. More timestamps can be used by the query definition, e.g. both the implicit timestamp of arrival and timestamps emitted by the data sources can be used in the same query.

2.1.2 Towards a Common Stream Model

This section describes our efforts to develop a common stream model that needs to be more general than the described models. This common stream model should be mappable to different models without semantic loss. The disadvantage of a common stream model is the loss of some interesting concepts of special-purpose stream models.

We will explain the decisions we made regarding the dimensions of classification mentioned above and discuss some preliminary ideas on how mapping between this stream model and the ones used by different SPSs is possible.

Timestamps

We assume that each item has at least one timestamp. This timestamp can be added if the data source does not provide it. Thus, there is no semantic guarantee about the timestamp. It may be the creation time of an item or the arrival time at the first system under our control. We prefer the former. Further concepts like duration in Cayuga can only be supported by using normal user data attributes. As queries often have to work with timestamps, we are convinced that timestamps should behave like normal attributes and be accessible to the query.

Uniqueness

As most systems do not require items to be unique and uniqueness may be difficult to enforce, our common stream model does not make any guarantees regarding uniqueness of items.

Lost and Duplicated Tuples

It may be impossible to determine if tuples are lost within a sensor network. Thus, our stream model does not guarantee completeness. Currently, the user has to understand the implications of using lossy transmission protocols. We assume that no duplicated tuples exist[3].

Ordering

In our stream model, items that arrive at a node have to be in correct order. This assumption is made by most of the analyzed systems. When timestamps are created by the system, this is always the case. Otherwise, existing solutions for sorting are used.

2.2 Metadata

Metadata is a crucial aspect in the integration of heterogeneous nodes. It supports both domain experts and the partitioning process.

We distinguish *static* and *dynamic* metadata. Static metadata usually remains constant (e.g. capabilities of nodes), while dynamic metadata may change while a query is running (e.g. data rates).

Different kinds of metadata are used for query optimization, description of data sources, etc. Many of them are similar in most of the related systems. However, these systems usually differ in the storage of metadata and in the monitoring of dynamic metadata. Metadata in DSAM is described in a level of detail as far as it is relevant in the further sections.

2.2.1 Metadata of WSN Middleware and SPSs in Existing Solutions

TinyDB

TinyDB is a middleware for TinyOS nodes that supports queries similar to SQL [26].

There is a virtual table *sensors* that can be queried by TinySQL. *Sensors* consists of attributes like nodeid, light, temperature, etc.

Each node in TinyDB has a local catalog containing its local attributes, events, and user-defined functions. Each attribute has the following metadata:

- Power: Cost to sample this attribute (in J)

[3] This refers only to duplicates created by network protocols e.g. re-sending of tuples that were not acknowledged. Several tuples with identical values are still allowed.

- Sample time: Time to sample this attribute (in s)
- Constant?: Is this attribute constant-valued (e.g. id)?
- Rate of change: How fast the attribute changes (units/s)
- Range: Dynamic range of attribute values (pair of units)

The local catalog is copied to the root node for query optimization.

Global Sensor Networks

The *virtual sensor definition* is an XML-file containing the stream's name, address, output structure, input streams, and SQL queries that led to the output stream [5]. The GSN middleware enables the access to other data sources by using different wrappers, e.g. TinyOS, cameras, and RFID readers. The GSN middleware manages all virtual sensor definitions centrally. For integration, the output structure of a virtual sensor's stream and its name are relevant. The technical integration is realized by the available wrappers that are provided by the GSN middleware.

Borealis

Borealis has a local catalog within its query processor and a global catalog [1]. The local catalog holds all metadata that belongs to a local query fragment that is processed by the query processor. As Borealis is a distributed SPS, the global catalog stores all information about distributed query fragments. All dynamic metadata is gathered by monitoring components.

PIPES

PIPES uses static and dynamic metadata and propagates dynamic metadata through the query graph [9]. This is necessary for adaptive query processing.

SStreaMWare

SStreaMWare [20] suggests a management system that offers all relevant metadata [19]. This includes topology, energy, CPU, memory, etc. The management system centralizes all dynamic metadata.

2.2.2 Metadata in DSAM

This section discusses the metadata catalog used by DSAM. As requirements for metadata may change, its main focus is on extensibility. We discuss only the elements that are necessary for integration of different SPSs and partitioning of global

queries. There are extensions for a more detailed description of data sources and sensor data that are necessary for both semantic descriptions and for cost estimation; their description is beyond the scope of this chapter.

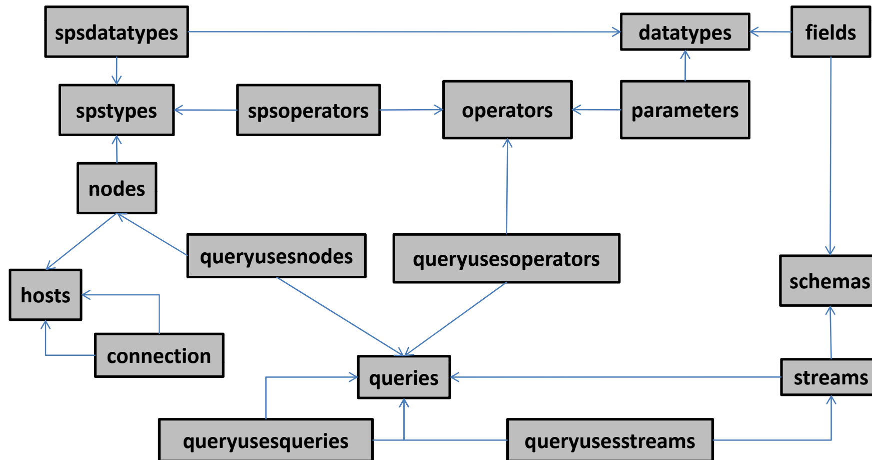

Fig. 3 Metadata of DSAM

DSAM uses the same conceptual schema for both static and dynamic metadata, i.e. all data can be accessed in the same way. Fig. 3 gives an overview of the catalog; the arrowheads denote the references. The whole scenario contains a number of nodes that are deployed to hosts that are connected to each other. Here, only direct connections are stored; transitive connections are considered by the query processing.

Nodes furthermore have an SPS type. An SPS type contains all information that is needed for the technical integration during the code generation process. The list of available operators is necessary for the partitioning.

The `streams` table stores all streams, i.e. input streams, output streams, and streams between nodes. Internal streams of partial queries that run on a node need not be stored in the catalog.

Both global queries and partial queries are stored in the `queries` table while their relationship is stored in `queryusesqueries`. From a conceptual point of view, queries at PIM- and queries at PSM level are stored in the same way. The generation of partial queries creates internal streams among the partial queries. These internal streams are the relevant information for the integration of heterogeneous nodes and correspond to the PSM bridges (Fig. 2).

While most solutions offer a central metadata catalog, some also make extensive use of local metadata. DSAM only relies on the central catalog. As we want to integrate different third-party SPSs, it is not feasible to use local metadata. The main feature of DSAM's catalog is the management of heterogeneous SPS types that allows to integrate new SPS types and operators.

3 Definition of Queries

Up to now there are no standards for query languages that can be used as a global query language. In order to deploy global queries to a heterogeneous network of nodes, a global query language is required. At the beginning, this section explains concepts of existing data stream queries that are relevant for queries in WSNs. Afterwards this section explores different types of proposed stream query languages and then introduces *Abstract Query Language* (AQL), the query language used by DSAM.

3.1 Operators

A data-stream operator receives items from one or more input streams, creates items, and delivers them to one or more output streams. In general, operators are application-independent and their output can be used as an input for the next operator in a directed graph of operators. The literature distinguishes between blocking (join, aggregate, sort, etc.) and non-blocking operators (selection, projection, simple functions, etc.) [7]. Non-blocking operators can directly create output items when an item is received. Blocking operators have to wait and collect a sequence of items from one or more input data streams in order to create an output item. Another classification of operators distinguishes between stateful and stateless operators [1]. In most cases, blocking operators are also stateful operators.

Most SPSs support stateless relational operators like selection, projection, and renaming with no relevant semantic differences. These operators can easily be implemented for WSN nodes with low capacities, too. Stateful operators can only be deployed on WSN nodes if the size of the state does not exceed the available capacity. In a distributed scenario, it is helpful to deploy selection and aggregation operators to WSN nodes as this reduces the amount of communication and saves energy.

For the definition of the relevant sequence of items, most SPSs use window definitions. Window definitions are used for e.g. sort, join, and aggregate. The window definition describes the operator's evaluation. If extra-regular evaluation is needed, most SPSs can use a slide parameter additionally to a window definition. Slide parameters bring the aspect of continuousness into queries as they define the cycle of evaluation. There are relevant differences in definition and implementation of windows as the next section will show.

3.2 Window Definition and Time Semantics

Stateful operators differ in their window definitions as there are *operator windows* and *source windows*. Operator windows are windows that may be defined for each

operator [1, 2]. In contrast, source windows are defined for input streams. They are mostly used by SPSs that define their semantics on the relational data model [6, 23].

In heterogeneous networks, it might be relevant to map both window definitions to each other. GSN [5] does not have this problem as window definitions are part of the virtual sensor definition; window definitions behave like source windows. The definition of windows is directly connected to a stream (*virtual sensor definition)*. By the window definition, each input stream is evaluated and stored in a temporary relation. GSN uses the input-triggered approach, i.e. all temporary relations are updated when a new item arrives, and all consumers of this virtual sensor are notified. This is possible as the joining of data sources is done in the GSN containers.

In most systems, windows are time-based, i.e. an item is valid for a period of time. Other possibilities are count-based, i.e. an item invalidates when a certain number of items has arrived. Value-based is more general than both time-based and count-based. Value-based means that any attribute can be used for the decision of expiry of an item.

Queries having operator windows can be easily mapped to SQL queries having source windows by using subqueries. Mapping source windows is trickier as the global definitions of source windows cannot be mapped to local operator-window definitions. E.g. this problem occurs for a source-window definition that includes the last 10 tuples. If the first operator is a filter operator and the second operator should calculate an aggregate function, an operator window of the aggregate operator with size 10 would not accord with the source window. The size of the operator window would depend on the selectivity of the filter operator in order to achieve the same semantics as the source operator. A workaround is using a value-based window and an ID or an auxiliary attribute that enumerates the stream elements. Expiring elements can be calculated by the enumeration.

There are differences in the semantics of window definitions that rely on implementation details. Due to the characteristics of blocking operators, the arrival of an item does not necessarily produce a result. And due to query descriptions, it is not the arrival that is relevant but a sequence of arrivals or a period of time. There are different possible points in time for creating results.

- *Input triggered*: The operator is triggered by the arrival of an item
- *Local-clock triggered*: The operator is triggered by a clock
- *Pull-based*: The operator is activated by its successor

Input triggered can lead to imprecise evaluation of slide parameters as results are only created when items arrive; systems like PIPES rely on incessantly sending data-stream sources for precise semantics. Local-clock triggered means an operator sleeps, but it can wake up by itself. Pull-based is very precise but loses the advantages of SPSs as they act like *Database Management System*s (DBMSs) and may have high latencies. A solution for improving the quality of input-triggered evaluation is negative tuples. A special operator at the entry of the SPS sends a negative tuple if the according tuple expires [18].

An efficient solution for source windows in distributed environments like WSNs could be the approach of the physical stream in PIPES. Here, a sensor node would

add an expiration timestamp to each tuple. This approach would however need a homogeneous implementation of operators using windows in different SPS types. As these implementations are heterogeneous, we suggest using either source windows for each node or operator windows.

3.3 Types of Query Languages

Recent years have seen the introduction of several query languages. Almost every SPS defines its own query language. We distinguish three families of query languages: programming languages, SQL-like languages and graph-based languages.

3.3.1 Programming Languages

The user writes code that produces the desired result. Examples for this family are e.g. nesC [15] (TinyOS), and Java (SunSpot). Programming languages are unmatched in extensibility and expressive power. They are usually easily mappable to systems that are freely programmable like sensor nodes - only a suitable compiler is needed. However, they cannot be mapped to SPSs that provide their own high-level languages. Programming languages are difficult to use for domain experts who are not programmers. Furthermore, it is difficult to automatically split a single program in order to deploy its parts to different systems. While this may be mitigated by special language constructs, the use of such constructs places an additional burden on the application developer. For these reasons programming languages are of limited value for the definition of global queries.

3.3.2 SQL-like Languages

Some SPSs, namely STREAM, Nile [18], TelegraphCQ [11], and PIPES use SQL-like queries. As they are historically based on database technologies and have descriptive query languages similar to SQL, these systems are often called Data Stream Management Systems (DSMSs) in analogy to DBMSs. The DSMS translates the query into a query graph. The whole query is in focus of interest and there is no explicit definition of operators. As the relational approach is recycled, the query language maps streams to temporary relations. The user adds window definitions to the streaming data source in order to define a set of consecutive tuples. As a query does not have any further window definitions, we call these windows “source windows”. *Negative tuples* realize source windows in Nile and STREAM. PIPES uses its physical schema.

The set of supported operators is limited to the relational operators due to the query language definition. *User-Defined Aggregate*s (UDAs) can extend the set of available aggregate functions. One weakness of SQL-like languages might be the

set-oriented data model. The temporal sequence of data stream items matters both in stream processing and in query definition. Especially sensor-data processing and *Sensor Data Fusion* (SDF) need a set of special stream operators that should be part of the set of core operators.

SQL-like languages allow the user to describe the relevant result of a query without thinking about how it is obtained. This allows the system to apply various optimization techniques as it is not restricted in how it computes the results. SQL-like languages play a major role in relational database systems. Before being executed, SQL-like queries are translated to a query graph (more specifically: a tree) where the nodes consist of operators and the edges represent the data flow among operators. This makes partitioning easy: any subset of operators may be placed on a system. The fixed structure and small set of available operators makes it simple to map SQL-like languages to different target languages. This fixed structure is also their greatest weakness: queries must be representable as trees and made up of a relatively small set of simple operators. While some SQL-like languages may be extended by new aggregate functions, it is not possible to add new operators. This would be difficult as each type of operator has its fixed position in an SQL-like statement and new operators do not have a "natural" position where they belong (though usually the only reasonable place for new operators would be in the FROM-clause). An example for an operator that might be relevant for SDF is `RESAMPLE` in Aurora [2]; this operator has no analogy in SQL-like query languages.

Perhaps for this lack of extensibility and expressiveness, available commercial products (e.g. Streambase[4], System S [16]) use graph-based languages instead of SQL-like ones.

3.3.3 Graph-based Query Languages

Graph-based query languages reflect the logical flow of data items through a network of connected operators. They define a query as a graph where the nodes represent operators and the edges represent data flow among operators. Existing graph-based languages differ in a couple of ways. The languages are usually closely tied to a certain system; thus some of the differences (especially syntactic ones) are actually properties of the implementing system and not of the languages themselves. *Boxes and arrows* are used by process-flow and work-flow systems. [10] describes the idea of *box-and-arrow* query graphs. The query graph represents the data flow. It is directed and acyclic in the case of Aurora. Borealis and Streambase are founded on Aurora. System S [16] proposes a descriptive graph-based query language.

Basically, graph-based query models need directed graphs that do not have to be acyclic; e.g. Borealis can handle cycles in queries.

Each node is individually configurable in a query graph and represents an operator. An operator may change the data stream's schema. The operators' configuration

[4] www.streambase.com

determines the inner schemas. Inner schemas are either derived automatically or have to be declared by the user.

The set of supported operators is conceptually unbounded. Most graph-based query languages support relational operators and an additional set of special stream operators. Those special operators are enormously important in real data-stream scenarios with sensor-data fusion. Defining a core set of relevant stream operators is still an open problem.

Graph-based query languages are extensible: All that is needed is a new operator implementation and an extension of the language grammar that allows this operator to be used. Such languages may be partitioned and mapped just like SQL-like languages. Their expressive power depends on which graphs may be represented in a given language. Cycles, multiple output streams or multiple inner streams are examples of constructs that cannot be represented in SQL-like languages but possibly in graph-based ones.

Graph-based query languages differ in their usability, which can be partly attributed to their different focus - who is expected to write (and read) queries and how often. Some systems provide graphical editors that make it possible to visually create queries - which is usually the easiest way for domain experts.

3.4 AQL - The Query Language of DSAM

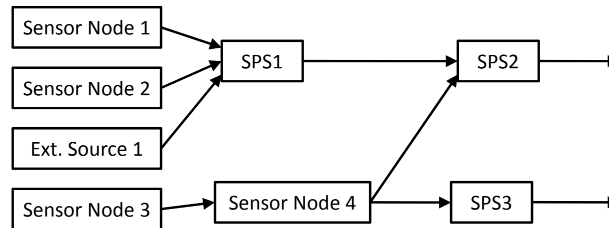

Fig. 4 Scenario of heterogeneous stream components

DSAM processes global queries and configures the nodes, i.e. it partitions global queries, maps partial queries to either operator assemblies or special destination languages, and deploys them on the adequate nodes. This supports the integration of heterogeneous stream-emitting and stream-processing nodes, in order to deploy large queries. Some sensors and data sources emit data that is processed by different types of SPSs (Fig. 4). We assume that users want to describe their needs in form of a query by using a uniform query language without considering the topology of sensor nodes. AQL is SPS-independent and used as global query language. It is completely descriptive by describing just data sources, data sinks and abstract operators. There are three classes of abstract operators: *monadic operators*, *combining operators*,

and *separating operators*. The membership of an abstract operator in one of these classes is determined by its number of input and output streams. Monadic operators have one input and one output stream like the filter, map, or aggregate operators. Operators of the combining-operator class are characterized by one output stream and a minimum of two input streams e.g. the union or join operators. A member of the separating-operator class has one input stream and multiple output streams, e.g. the split operator.

Figs. 5 and 6 show an excerpt of AQL's syntax and how the three operator classes are placed in a query.

```
query    := <source_list> ":" <fragment> ":" <sink_list>
subquery := "(" <source_list> ":" <fragment> ":" (<sink_list>)? ")"
fragment := (("$"<digit>"."<monadic_operator>",")*
            <combine_operator>)? "," (<monadic_operator>",")*
            (<separate_operator> (",#"<digit>"."<monadic_operator>)*)?
source_list := <source> ("," <source>)*
source      := identifier | <subquery>
...
```

Fig. 5 Syntax of AQL

Queries can use nested subqueries as source streams. Each subquery can unite different streams and separate the streams once. This pattern of subqueries allows arbitrarily complex directed graphs. The following excerpt of an AQL query shows two streams `S1` and `S2` that are combined and split into two streams. Sources and sinks have distinct addresses that are stored in the catalog (Sect. 2.2.2). There is a subquery that adds time information to stream `S2`. The subquery only consists of the combining operator `Merge`. The top-level query has three monadic operators, the combining operator `Union` and the separating operator `Split`. The decorator `$2` assigns the `Filter` to the second input stream, i.e. the resulting streams of the subquery.

```
S1,(S2,TIME:MERGE():):$2.Filter("S2.a<3"),
Union(),Filter(...),
Split(#1="g>5", #2=...):S3 ,S4
```

Fig. 6 Combining, monadic, and separating operators in AQL

AQL supports both *source windows* and *operator windows*. Mapping between *source windows* and *operator windows* is tricky but possible by insertion of additional fields (map operator).

When we designed AQL, we wanted a language that is both extensible and SPS-independent, so we chose a graph-based language. The set of operators can be easily extended and queries can be designed by an intuitive GUI tool. In order to remain SPS-independent, AQL defines a set of *abstract operators*. An abstract operator can

be mapped to different operator implementations on different SPSs as long as they adhere to the specification of the abstract operator. The set of allowed mappings is stored in the catalog. Some SPSs use advanced operator semantics for standard operators; e.g. Aurora can add a sorting to some operators that need tuples in a certain order. As the specification of an abstract operator is the lowest common denominator of the operator implementations that it is mapped to, these special cases are not supported up to now. It is possible to define an abstract operator that behaves exactly like an advanced operator of an SPS. However, this reduces SPS-independence as such an operator cannot be mapped to other SPS types.

4 Partitioning

Partitioning refers to splitting a global query into parts that may be individually deployed to distributed nodes. Deployment consists of two tasks. First, it has to be decided what is to be deployed where. Second, a technical infrastructure is required that can automatically implement this deployment. We refer to the first task as “deployment decision” and to the second one as “deployment implementation” or simply “deployment”.

Especially in heterogeneous environments, partitioning and deployment decision are difficult to separate because constraints on possible deployments affect possible partitionings. Thus, we discuss both problems in this section and subsume them under the term “partitioning”. Related literature calls this problem “operator placement” decision [22, 27]. We will discuss the technical realization of the deployment in Sect. 5.

Partitioning is a crucial step as it greatly influences the quality and the performance of a distributed query. As quality and performance requirements are query-dependent, a highly flexible partitioning process is required. Furthermore, partitioning has to consider many constraints, e.g. regarding availability of input streams, capacities of nodes, etc.

Metadata is essential for partitioning as it contains all information about data sources, topology of nodes, and even the set of available operators. During cost estimation, performance characteristics and knowledge about streams is used to choose a “best” plan according to some objectives. There are conflicting objectives like minimizing CPU load and energy consumption on a node, maximizing the data quality (reducing load-shedding), minimizing latency, etc.

In most database systems, the number of input and output operations is a reasonable metric for costs. In distributed data stream systems, metric and objective strongly depend on the concrete scenario. A solution is weighting estimated costs for all relevant objectives [27].

In the remainder of this section, we discuss how other projects deal with partitioning before we explain the solution we have chosen for DSAM.

4.1 Operator Placement Decisions in Existing Work

TinyDB [26] has the same query on each node, the GSN middleware [5] focuses on the integration of different sensor data streams. In both approaches, no operator placement decision is necessary.

Stream Based Overlay Network (SBON) [27] provides an approach for operator placement decisions in a network of homogeneous nodes of distributed SPSs. The cost model uses a blended metric, i.e. it tries to optimize two or more metrics (e.g. latency, load, etc.) at once.

The approach maps the metrics to a virtual cost space. This virtual space is a continuous mathematical and multi-dimensional metric space. Euclidean distances correspond to the optimized cost metric. All nodes have positions in this virtual cost space. Nodes can be producers, consumers, and processing nodes that can be used for placing an operator. The virtual cost space is modeled after physical rules. The model equates costs with spring pressure. Relaxation models the overlay network having massless bodies (operators) connected by springs (operator links). The optimization goal is spring relaxation that places the operator on a node in a way that spring pressure is minimized.

[30] assumes sensor network nodes with low capabilities acquiring the data and a hierarchy of nodes having increasing network bandwidth and computational power. They provide an algorithm that offers a globally optimal solution for the placement of expensive conjunctive filters and joins.

[36] considers the problem of placing both operators and intermediate data objects. Intermediate data objects are objects containing data that are neither sensor data nor final data. A set of queries is optimized so that the total cost of computation, memory consumption, and data transmission is minimized. One of the most relevant differences to other approaches is that they require only information exchange between neighbors.

4.2 Operator Distribution in DSAM

In DSAM, the task of the operator distribution process is both to split a global query into several so-called partial queries and to decide which partial query should be deployed on which node. Thus, the partial queries are the unit of distribution.

The deployment process is facilitated by an extensive metadata catalog. The catalog holds a list of available nodes that can run an abstract operator. We assume that some operators are not available on each node. For example, some special operators for data fusion or *User-Defined Operator*s (UDOs) might be installed on a few nodes only. This leads to structural constraints on a query's deployment:

- Input streams are available at a certain place/node
- Some nodes can execute some abstract operators that others can not
- Nodes have individual capacity and performance behavior

- Nodes' connections have reachability constraints and performance behavior

The partitioning process can be realized in an arbitrarily complex way, and partial queries may be of different granularity. For the sake of simplicity, we first assume individual operators as unit of distribution. Naively, each operator may be placed on an arbitrary node. For n operators and m nodes there are n^m possible distributions, one of which has to be chosen. This number is too large for an exhaustive search. On the other hand, there are some additional restrictions. Some operators may not be available on all nodes. Nodes and communication paths have limited capacity. Furthermore, only a few distributions are good enough to be considered at all. In mathematics, this problem is known as *Generalized Assignment Problem* (GAP). In a GAP, tasks have to be assigned to agents in a fashion that minimizes some global cost criterion. The costs for a task may vary among agents. Each agent has a capacity that must not be exceeded. We model operators as tasks and nodes as agents. Unfortunately, GAP is an NP-complete problem, thus it can only be solved approximately in acceptable time by using heuristics.

For the discussion of possible solutions to the GAP, the following definitions are necessary:

$$T = \{ t \mid t \text{ is an SPS type } \} \quad (1)$$

$$N_t = \{ n_t \mid n_t \text{ is an SPS node and } t \in T \} \quad (2)$$

$$N = \bigcup_{t \in T} N_t \quad (3)$$

$$O_t = \{ o_t \mid o_t \text{ is an operator type and } t \in T \} \quad (4)$$

$$O = \bigcup_{t \in T} O_t \quad (5)$$

$$C = \{ c \mid c \text{ is a cost model } \} \quad (6)$$

Further, we have the global query graph that is a directed graph $G = (V, E)$ containing *vertices* V and *edges* E. Vertices represent abstract operator instances. $M \subseteq N$ defines the set of SPS nodes that are available for a query.

We have different cost models for different SPSs. The costs for edges *comtranscost* (sending items between two operators) consist of *communication* and *transformation* costs. We assume no costs if two adjacent operators are assigned to the same node. In short, the costs of executing an operator instance v on an SPS node m are:

$$c_{mv} = load(m, v) + comtranscost_v \quad (7)$$

$load(m, v)$ calculates the costs for running an operator on a node. A further explanation of $load(m, v)$ is beyond the scope of this chapter.

With these definitions, we can adapt the GAP to the needs of DSAM's partitioning process:

$$x_{mv} \in \{0,1\}, \qquad \forall m \in M, \forall v \in V \tag{8}$$

$$\sum_{m \in M} x_{mv} = 1, \qquad \forall v \in V \tag{9}$$

$$\text{Minimize } \sum_{m \in M} \sum_{v \in V} c_{mv} x_{mv} \tag{10}$$

$$\sum_{v \in V} a_{mv} x_{mv} \leq b_m, \qquad \forall m \in M \tag{11}$$

x_{mv} denotes whether an operator instance v is running on node m. b_m is the capacity of a node, and a_{mv} is the load that an operator v causes if it is run on m. Equation 8 postulates that an operator instance is either deployed on a node (1) or not (0), while equation 9 simply states that each operator instance is deployed on exactly one node. Equation 10 defines the primary optimization goal: to minimize the total costs incurred by each operator instance running on its respective node. Equation 11 places further constraints on acceptable solutions: No node may be burdened with a total load that exceeds its capacity.

Each connected group of operators that run on the same SPS is combined to a partial query. We save all partial queries in the catalog of DSAM. Each partial query uses a set of input streams and has a set of output streams. In Fig. 7, the relevant excerpt of the catalog shows the dependencies among partial queries, global queries, and "inner" schemas. The schemas of output streams are defined by the partial queries (`fk_pub_query`).

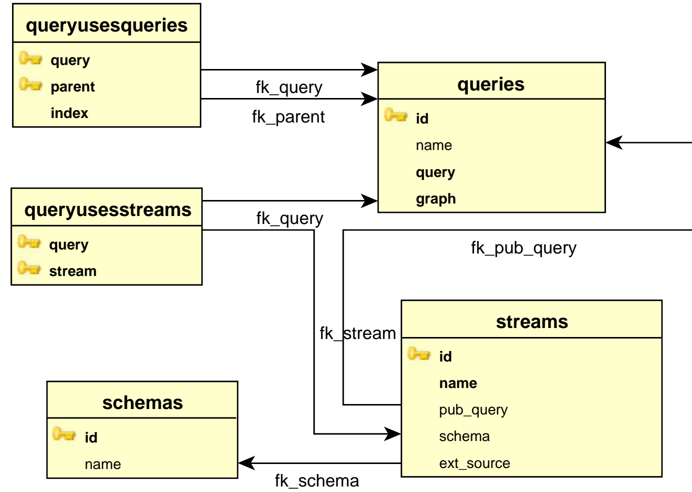

Fig. 7 Partial Queries in Metadata

The discussed idea relies on the ability to estimate costs for executing a partial query on a node as well as for transmitting results to the next node in the query graph. These estimations are derived from existing metadata. While topology, node types, and available operators are known, good estimations for data rates, selectivities, etc. are difficult to obtain.

In contrast to other solutions, DSAM's partitioning is more general as new systems or operators do not require any changes to the algorithm; it suffices to configure the cost estimators. Our current implementation allows balancing quality of operator placement and the time spent on finding a solution. This may be beneficial for large networks where an exhaustive search for a best solution is not feasible.

5 Integration of WSNs

The partitioning step of the previous section leads to partial queries. In top-down approaches such as DSAM, these partial queries have to be deployed to the distributed nodes. In other approaches, applications for WSNs and the query of the SPS are independently developed and deployed. In this point, most existing approaches and DSAM's approach differ.

All related approaches use WSNs as data sources, e.g. for surveillance. The processing within the WSN nodes is kept simple due to restricted capabilities. The common assumption is that the result of a WSN or rather a WSN query is a data stream that can be processed by SPSs.

5.1 Integration of WSNs in Existing Approaches

SStreaMWare

StreaMWare [20] has a three-level query processing. The *control site* is the central manager of SStreaMWare and provides a global *Sensor Query Service* (SQS). It communicates with *gateways* that might be widely distributed on different hosts. Each gateway has a set of *Proxy Query Service*s (PQSs). Each sensor network needs an implementation of a PQS in order to provide a uniform interface that is used by the GQS. The implementation of a PQS can be either an adapter or a proxy.

A PQS adapter communicates with a proprietary proxy of a WSN and can only use the functionality of this proxy. A PQS proxy communicates directly with the sensors. This enables data processing both within the sensor network and within the PQS.

Having a uniform interface, all PQSs can conceptually process the same kind of partial queries. As a minimum, they can process the relational selection and projection [20] within the PQS. PQSs provide a homogeneous view over heterogeneous data sources.

PQSs differ in the provided attributes. The management of provided sensors and attributes is an outstanding feature of SStreaMWare. An instance of a *Lookup Service* (LS) runs on the central manager and on each gateway. These LSs provide information about available sensors that can be used for queries dynamically without having a central registry.

GSN

GSN [5] uses the concept of *virtual sensors* in order to get a homogeneous view on heterogeneous sensors. Each virtual sensor definition describes a temporary relation. From a technical point of view, there are two groups of adapters: remote adapters and local adapters. Remote adapters refer to virtual sensors that are installed on remote GSN containers. Local adapters support different kinds of data sources. These adapters offer a streaming data source.

A query can only use the given temporary relations. In contrast to SStreaMWare and DSAM, top-down propagation of partial queries is not in the focus of GSN as the approach is bottom-up.

TinyDB

TinyDB [26] is a TinyOS application. TinyOS provides the technical integration, communication, etc. As TinyDB is installed on all nodes, the view to single nodes is quite homogeneous. Integration is done by the base station. It parses a query and propagates it to the sensor network. Query propagation forms a routing tree having the base station at the root. All sensor data is sent back in reverse direction of the query propagation.

Borealis/REED

Some approaches use TinyDB as a configurable data source. REED [4] integrates TinyDB and Borealis. Its integration framework [3, 29] uses wrappers that provide a standardized API for the sensor networks. Each query processor of a Borealis node has a proxy that gathers statistics about the sensor networks from the wrappers. These statistics include constraints that are necessary in order to reject impossible operator movements. Further, the proxy organizes the optimization, i.e. pushing operators to the sensor network.

5.2 Code Generation, Deployment, and Integration in DSAM

DSAM maps partial queries to the corresponding platform-specific query languages. Fig. 8 shows the whole deployment process of a global query on different kinds of nodes. The results of the partitioning process are partial queries that *can* be deployed on the according node, i.e. the node must provide all necessary operators. We sketch the mapping process of partial queries to target query languages, the generation of source code for SPSs that are just programmable and do not support query languages, and the deployment of partial queries. A short example supports the presentation of our concept.

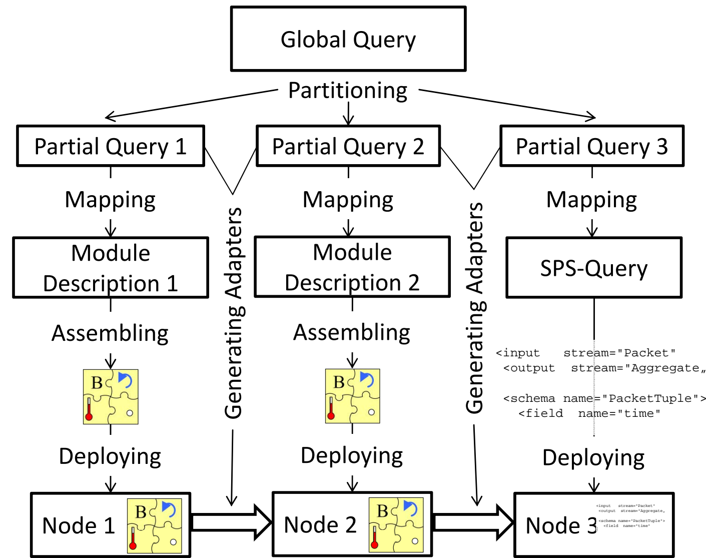

Fig. 8 Mapping of Queries

5.2.1 Mapping

The query mapping is a two-phase process and supports different target languages. Mapping corresponds to the second transformation of the MDA approach (Figs. 2, 8). It takes an abstract operator graph representing a partial query as input and generates queries in the specified query language as output. As preconditions for the mapping process, DSAM has to provide for each target language:

- Operator transformation rules
  Each rule transforms the syntax of an abstract operator into the syntax of the target language and adds metadata.
- Target language template
  A template defines the structure of a target language and can also make use of information from the catalog.

In the first phase we map an abstract operator graph to intermediate data structures. The second phase uses the query language template and constructs the target language queries from the intermediate data structures.

Each node is of a specific SPS-type and therefore supports a specific query language. A query-mapper component transforms the partial query, which will be deployed on a specific node. Fig. 9 gives an example for mapping a partial query to an SQL-based query language.

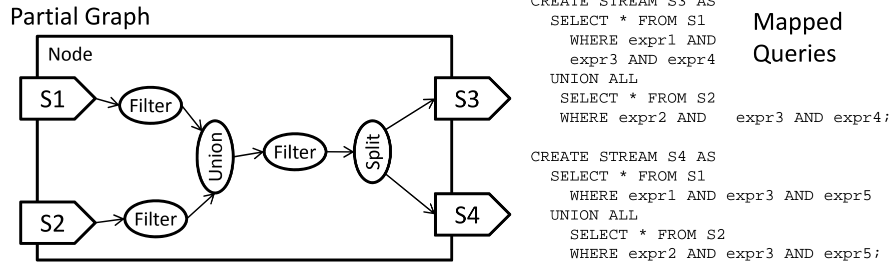

Fig. 9 Query mapping example for SQL-based target languages

```
APPLICATION("POR_Node2", stack_size, arg) {
  // Initialization of Streams
  LocalSensor POR_Node2_s3 = init_pos_sensor();
  InputStream POR_Node2_Node1_OUT_01 = init_input("node1");
  RemoteAddress POR_Client = "base";
  // Data structures
  struct POR_Node2_OUT_01 {
   int struct_size;
   [...]
   int time;
  };
  [...]
  // Query processing
  for(;;) {
   in_01 = getSensorData(POR_Node2_s3);
   in_02 = getInputStreamData(POR_Node2_Node1_OUT_01;);
   res_01 = merge(in_01,in_02,"")
   reorganizeWindow(win_01, res_01);
   res_02 = merge(in_01, time());
   reorganizeWindow(win_02, res_02);
   res_03 = join(&join_res_size_01, win_01, win_size_01, \
                win_02, win_size_02, join_cond_01);
   for (int i = 0 ; i < join_resultsize_01 ; i++) {
    send(POR_Client, res_03[i], sizeof(POR_Node2_OUT_03));
   }
   NutSleep(125);
  }
}
```

Fig. 10 Generated sample application code for a BTnode

5.2.2 Code Generation

Some stream processing components do not support query languages. Especially data sources like WSNs have limited capabilities and can only be configured by individual software deployment. In [14], AQL is used for global queries that are deployed on different BTnodes. A BTnode is a typical sensor node developed at the ETH Zurich, which is based on an Atmel ATmega128 micro controller. DSAM supports the invocation of code-generation modules. The result is code that uses a set of operators and system components. Let us revisit the example from Sect. 1.1 which tracked animals' stress levels.

Fig. 10 shows an excerpt of generated BTnode code for this scenario. The partial query has an external stream and a local sensor as input streams. It uses a join operator with operator windows and a merge operator. The catalog of DSAM knows which operators are available on a node. The resulting code can be linked and deployed in the usual way.

5.2.3 Deployment and Integration

Technical integration is achieved during mapping and deployment with the help of the knowledge about platforms and connections (Fig. 8). The addresses of inner streams between SPS and WSN nodes can be directly derived from the knowledge about partitioning. This knowledge helps configuring adapters or generating the source code (Fig. 10) of WSN applications.

For the deployment we assume direct contact between DSAM and the node or multi-hop deployment[5]. The deployment on BTnodes is described in [14].

The main differences between DSAM and other approaches are: DSAM exploits the MDA analogy in order to deploy global queries automatically to heterogeneous nodes. This makes it very flexible, as code can be deployed on target systems in a generic way. The target systems are not required to run any special software (as opposed to e.g. TinyDB). In the absence of such special software, a central manager is required. In this respect, DSAM is similar to SStreamWare or TinyDB as opposed to systems like GSN or Borealis that do not require a central manager.

6 Examples and Evaluation

To demonstrate the benefits of operator propagation to sensor networks, we present further examples and give the results of some measurements that support our approach.

6.1 Mapping to SQL-like query languages for WSNs

The fictitious example scenario for the usage of DSAM is a modern hospital that monitors vital signs of patients with the help of wireless sensors attached to a patient. Each of these wireless sensors emits a data stream with the patient's id and the vital signs. For tracking, all patients are equipped with a *R*adio *F*requency *ID*entification (RFID) chip that emits the patient's id to a reader. RFID readers are distributed over the hospital and form the global stream `POSITION` that contains

[5] At the moment we are using `Deployer` components for the SPSs and direct flushing for the WSN nodes.

PAT_ID, *Antenna_ID*, and *Time*. An SPS combines the `POSITION` stream and the streams emitted from the wireless sensors of each patient.

Physicians can configure which of a patient's vital signs they want to monitor.

6.1.1 Example Query

For an example query, we assume a filter criterion for the heartbeat rate. The minimum threshold for the heartbeat is 60 beats and the maximum threshold is 160 beats. We are only interested in the last position and the current vital signs of the patient. Therefore we define a count-based window of size 1 on the `POSITION` stream and a time-based sliding window of 1 second on the `PAT_SENSOR_1` stream. The additional partitioning ensures that the last position and the current vital signs of all patients are going to be recognized. These requirements result in the AQL query in Fig. 11.

```
PAT_SENSOR_1, POSITION:
  $1.Filter("Heartbeat <= 60 OR Heartbeat >= 160"),
  Join("$1.Pat_ID = $2.Pat_ID",
       window1(size = "1", size-by="SECONDS", partition = "Pat_ID"),
       window2(size = "1", size-by="TUPLES", partition = "Pat_ID")
  )
:PAT_STAT
```

Fig. 11 Health monitoring example

6.1.2 Example Mapping

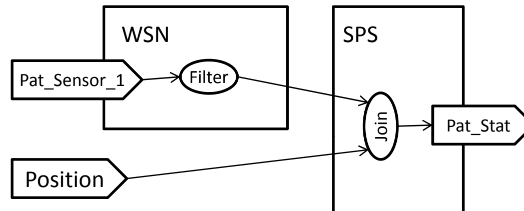

Fig. 12 Abstract operator graph with partitioning

The abstract operator graph for the above AQL query is shown in Fig. 12. From this abstract operator graph, the distributor component creates two partial queries. The distribution algorithm pushes the filter operator to the wireless sensor of the patient.

Partial query one is mapped to the target language of the wireless sensor and partial query two to the target language of the SPS used in the scenario. We assume SQL-based query languages for both nodes and obtain the following two queries as a result:

```
CREATE STREAM QUERY_STREAM_1 AS
  SELECT * FROM SENSORS
    WHERE Heartbeat <= 60 OR Heartbeat >= 160
CREATE STREAM PAT_STAT AS
SELECT * FROM
   QUERY_STREAM_1[PARTITION BY Pat_ID RANGE 1s],
   POSITION[PARTITION BY Pat_ID ROWS 1]
WHERE QUERY_STREAM_1.Pat_ID = POSITION.Pat_ID
```

Fig. 13 Query mapped to SQL-like language

Note: Internally all wireless sensors have only a stream called `SENSORS`. So we map stream `PAT_SENSOR_1` to `SENSORS`. This mapping has to be configured in the metadata catalog.

6.2 TinyOS application

Fig. 14 shows an excerpt of a runnable TinyOS application. This is a case study that is realized in TinyOS2 and is run on Intel Mote 2. All operators are realized as configurable components. The predicates have to be implemented in a way that can be interpreted by the operator instances. In the given example, all configured operators are wired.

The generator for TinyOS code is still work in progress and its use still requires some manual work. Yet, the results obtained so far absolutely meet our expectations. We expect DSAM to completely generate these wiring applications in the near future.

6.3 Measurements

[25] summarizes the results of [32]. In this earlier work, neighborhood-optimization - a concept of Borealis [1] - is applied to TinyDB as a neighbor of a Borealis node. Figure 15 shows the development of the estimated lifetime. The gap is caused by relocating the aggregate into the sensor. After the relocation the estimated lifetime increased enormously. Though this is not a result of DSAM, we expect similar improvements with DSAM.

```
#include "../../QueryProcessor.h"
#define COMPOSITION_TEST_1 Composition_Test_1

configuration TestAppC {}

implementation {
        components LedsC;
        components MainC;
        components TestC as App;
  components SensorC;

  ...
  /* stream components */
        components new StreamC() as SensorStream;
        components new StreamC() as FilterOutputStream1;
        components new StreamC() as FilterOutputStream2;
  ...
  /* operation components */
        // temp < 100
        components new FilterOpC("02:04:100", 1);
                // output field = room , temperature. group by room. order by room. window↩
                      size = 2. sliding window = 2. aggregate function = avg(temperature)
        components new AggregateOpC("05,02", 2, FIELD_NAME_ROOM, FIELD_NAME_ROOM, 2, 2, ↩
             AGG_FUNCTION_AVG, FIELD_NAME_TEMP);
        components new MapOpC("?:02+2:?:?:05+1", 5); // tid = tid, temp = temp + 2, light ↩
             = light, humidity = humidity, room = room + 1

        // predicate: left.room == right.room. output fields = left.room, left.temperature↩
             , right.light. left buffer size = 2, right buffer size = 2
        components new JoinOpC("05:01:05", 1, 2, 2, 1, 1, "L05;L02:R03", 3);

        /* data sender components */
        components new DataSenderC(1, ACTION_SEND_TO_CLIENT) as DataSender;
     ...
        /* wiring */

        /* SensorC */
        SensorC.StreamWriter -> SensorStream;

        /* FilterOpC */
        FilterOpC.InputStreamReader -> SensorStream;
        FilterOpC.OutputStreamWriter_1 -> FilterOutputStream1;
        FilterOpC.OutputStreamWriter_2 -> FilterOutputStream2;

        /* AggregateOpC */
        AggregateOpC.InputStreamReader -> FilterOutputStream1;
        AggregateOpC.OutputStreamWriter -> AggregateOutputStream;


        ...

        /* DataSender */
        DataSender.StreamReader -> JoinOutputStream;
        DataSender.AMSend -> AM.AMSend[AM_QUERY_RESULT];
        DataSender.Packet -> AM;

}
```

Fig. 14 TinyOS application

7 Future Work - Maintenance and Adaptation of Query Plans in Heterogeneous WSNs

Queries in WSNs may run for a significant amount of time during which the environment may change drastically. This may happen due to failing nodes, changing data rates, or changing topologies if WSN nodes are mobile or connections fail. In this case, it may be beneficial or even necessary to redistribute a global query. If it is possible to just stop the old query and distribute a new one, this problem is trivial with the help of an infrastructure like DSAM. However, taking a query offline during redeployment is often not feasible - continuous queries are expected to return results continuously. Especially in the presence of stateful operators (e.g. ag-

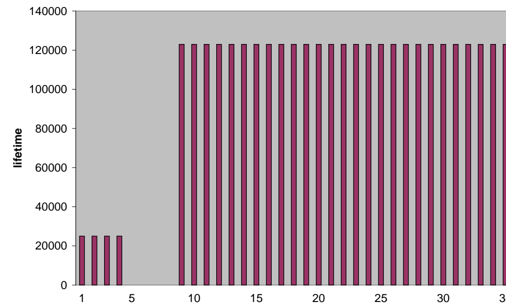

Fig. 15 Improvement of the lifetime score

gregates), intermediate results may be lost by redeployment. An interesting solution to this problem is hot redeployment, i.e. the deployment of the new version of a query while the old one is still running as well as a seamless switch to the new one. A technical solution for storing the state in BTnodes during redeployment can be found in [14].

Automatic redeployment decisions pose some interesting additional problems: First, monitoring has to gather information about nodes' load, battery state, etc. Second, it has to be determined when a possible redeployment should be computed.

A periodic redeployment strategy computes a new partitioning in certain time intervals. It may be a good choice if the partitioning is relatively cheap to compute and a relatively high gain by redeployment is expected. Its major advantage is that it does not require additional monitoring data.

An on-demand-strategy triggers a redeployment computation in the case of certain events, e.g. the addition or removal of nodes or if some nodes are overloaded. This has the advantage of consuming CPU power for computation only when actually necessary. However, in some cases, it may be too late (e.g. when overloaded nodes are used as a trigger). It relies on the existence of external events that may be used as triggers.

A heuristic strategy is quite similar to the on-demand-strategy, but more general. While the latter relies on trigger events that require a redistribution, the former utilizes "rules of thumb" to determine when to compute a redistribution. To our best knowledge, no heuristics exist for solving this problem.

8 Conclusion

This chapter presented an approach for efficient query definition and deployment in heterogeneous WSNs and their integration with SPSs. Its main idea is the definition of global and abstract queries on streaming data by users that are not used to the programming of WSN nodes. For this purpose, we defined a data model for streaming data and considered an appropriate query language. All descriptive information

of data sources are stored in a catalog. The definitions of queries include the choice of the relevant data sources, the definition of a graph-based query, and the destinations of the query. Distribution of queries and their deployment are not left to the user but done automatically by the middleware. The prototype of DSAM can do the partitioning of data stream queries. It maps partial queries to traditional query definitions of existing SPS products and supports code generation for programmable WSN nodes.

In our earlier work [25], cross-border optimization between TinyDB and Borealis is presented. The measurements that are related to this paper but could not be presented due to space limitations showed that pushing operators to the WSN can extend the lifetime many times over. These results motivate us to integrate cross-border optimization in the query partitioning process.

Up to now most query languages do not consider time and quality constraints. Most SPSs are *best effort DSMSs* [28]. First approaches deal with quality constraints [33] by defining deadlines in SPSs. Especially traditional sensor technology has strict quality requirements for the processing of sensor data. Another question is how loss-tolerant an application is; e.g. load shedding is a relevant aspect in stream processing [31]. Beyond the definition of data stream queries, these quality requirements will be relevant for real scenarios. Partitioning and mapping will have to consider these requirements.

References

1. Abadi, D., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.H., Lindner, W., Maskey, A.S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: 2nd Biennial Conference on Innovative data Systems Research (CIDR) (2005)
2. Abadi, D., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A New Model and Architecture for Data Stream Management. VLDB Journal **12**, 120–139 (2003)
3. Abadi, D.J., Lindner, W., Madden, S., Schuler, J.: An Integration Framework for Sensor Networks and Data Stream Management Systems. In: 13th international conference on very large data bases (VLDB) (2004)
4. Abadi, D.J., Madden, S., Lindner, W.: REED: robust, efficient filtering and event detection in sensor networks. In: 31st Conference on Very Large Data Bases (VLDB) (2005)
5. Aberer, K., Hauswirth, M., Salehi, A.: Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. In: International Conference on Mobile Data Management (MDM) (2007)
6. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB Journal **15**, 121–142 (2006)
7. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and Issues in Data Stream Systems. In: Proceedings of 21st ACM Symposium on Principles of Database Systems (PODS 2002) (2002)
8. Babu, S., Srivastava, U., Widom, J.: Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries Over Data Streams. ACM Transactions on Database Systems (TODS) **29**, 545–580 (2004)

9. Cammert, M., Krämer, J., Seeger, B.: Dynamic metadata management for scalable stream processing systems. In: Proc. of First International Workshop on Scalable Stream Processing Systems (2007)
10. Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.: Monitoring streams - a new class of data management applications. In: Proceedings of the 28th international conference on Very Large Data Bases-Volume 28, pp. 215–226. VLDB Endowment (2002)
11. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F., Shah, M.: TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In: Proceedings of the 2003 CIDR Conference (2003)
12. Demers, A., Gehrke, J., Hong, M., Riedewald, M., White, W.: Towards Expressive Publish/-Subscribe Systems. In: EDBT 2006 (2006)
13. Demers, A., Gehrke, J., Panda, B.: Cayuga: A General Purpose Event Monitoring System. In: 3rd Biennial Conference on Innovative Data Systems Research (CIDR 2007), pp. 412–422 (2007)
14. Dressler, F., Kapitza, R., Daum, M., Strübe, M., Preikschat, W.S., German, R., Meyer-Wegener, K.: Query Processing and System-Level Support for Runtime-Adaptive Sensor Networks. In: Kommunikation in Verteilten Systemen (KIVS) (2009)
15. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. ACM SIGPLAN Notices **38**(5), 1–11 (2003)
16. Gedik, B., Andrade, H., Wu, K.L., Yu, P.S., Doo, M.: SPADE: The System S Declarative Stream Processing Engine. In: ACM SIGMOD Conference (SIGMOD) (2008)
17. Gehrke, J., Madden, S.: Query Processing in Sensor Networks. Pervasive Computing, IEEE **3**(1), 46–55 (2004)
18. Ghanem, T., Hammad, M., Mokbel, M., Aref, W., Elmagarmid, A.: Query Processing using Negative Tuples in Stream Query Engines. Tech. Rep. TR 04-030, Purdue University (2004)
19. Gürgen, L., Honiden, S.: Management of Networked Sensing Devices. In: Internation Conference on Mobile Data Management (MDM) (2009)
20. Gürgen, L., Roncancio, C., Labbé, C., Bottaro, A., Olive, V.: SStreaMWare: a Service Oriented Middleware for Heterogeneous Sensor Data Management. In: 5th International Conference on Pervasive Services (ICPS), pp. 121–130 (2008)
21. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley (2003)
22. Kossmann, D.: The State of the Art in Distributed Query Processing. ACM Computing Surveys (CSUR) **32**(4), 422–469 (2004)
23. Krämer, J.: CONTINUOUS QUERIES OVER DATA STREAMS-SEMANTICS AND IMPLEMENTATION. Ph.D. thesis, Philipps-Universitat Marburg (2007)
24. Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: Semantics and Evaluation Techniques for Window Aggregates in Data Streams. In: Proceedings of the 2005 ACM SIGMOD international conference (2005)
25. Lindner, W., Velke, H., Meyer-Wegener, K.: Data Stream Query Optimization Across System Boundaries of Server and Sensor Network. In: 7th International Conference on Mobile Data Management (MDM) (2006)
26. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: An Acquisitional Query Processing System for Sensor Networks. ACM Trans. Database Syst. **30**, 122–173 (2005)
27. Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., Seltzer, M.: Network-Aware Operator Placement for Stream-Processing Systems. In: 22nd International Conference on Data Engineering (ICDE 2006) (2006)
28. Schmidt, S.: Quality-of-service-aware data stream processing. Ph.D. thesis, Technische Universitt Dresden (2007)
29. Schuler, J.: Query Optimization in Data Stream Architectures. Master's thesis, University of Erlangen-Nürnberg (2004)

30. Srivastava, U., Munagala, K., Widom, J.: Operator Placement for In-Network Stream Query Processing. In: 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS 2005), pp. 250–258. ACM Press (2005)
31. Tatbul, E.N.: Load Shedding Techniques for Data Stream Management Systems. Ph.D. thesis, Brown University (2007)
32. Velke, H.: Query Optimization between Data Stream Management Systems and Sensor Network Query Systems. Master's thesis, University of Erlangen-Nürnberg (2005)
33. Wei, Y., Prasad, V., Son, S.: QoS Management of Real-Time Data Stream Queries in Distributed Environments. In: 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), pp. 241 – 248 (2007)
34. Wiederhold, G.: Mediators in the Architecture of Future Information Systems. IEEE Computer **25**(3), 38–49 (1992)
35. Yao, Y., Gehrke, J.: The Cougar Approach to In-Network Query Processing in Sensor Networks. SIGMOD Rec. **31**(3), 9–18 (2002)
36. Ying, L., Liu, Z., Towsley, D., Xia, C.: Distributed Operator Placement and Data Caching in Large-Scale Sensor Networks. In: 27th Conference on Computer Communications IEEE (INFOCOM 2008), pp. 977–985 (2008)