Integration of Heterogeneous Sensor Nodes by Data Stream Management

Michael Daum, Martin Fischer, Mario Kiefer, Klaus Meyer-Wegener
Dept. of Computer Science, University of Erlangen-Nuremberg, Germany
{md,simmfisc,simakief,kmw}@i6.cs.fau.de

Abstract

In this paper, we present DSAM, a data stream application manager. DSAM supports the deployment of global queries to distributed and heterogeneous sensor nodes and Stream Processing Systems (SPSs). We provide a graph-based global query language DSAM-AQL. The Abstract Query Language (AQL) is a declarative stream-oriented query language that abstracts from topology and distribution of Wireless Sensor Networks (WSNs) and the heterogeneity of their nodes. Query processing and distribution is supported by a central catalog that manages all stream schemas, nodes, (initial) topology, SPSs, queries, performance characteristics of both network connections and nodes, and expressiveness of nodes' supported query languages and query definitions. DSAM supports query partitioning, query mapping, deployment, and monitoring of distributed data stream applications. We focus on DSAM's infrastructure and the procedures of deployment of global queries.

1 Introduction

Wireless Sensor Networks (WSNs) consist of nodes that are widely distributed. These sensors are the data source for data stream processing in many scenarios. Having no central storage, data stream processing becomes a challenge of distributed processing that is more efficient than sending all data to a central processing unit. At the moment most data stream scenarios of academic data stream prototypes are quite small and manageable. They focus on their presented data stream prototype with its query language and processing model. We expect complex scenarios with many distributed data sources such as complex event processing and complex sensor data fusion in the near future.

From our point of view, sensor network nodes are Stream Processing Systems (SPSs) with limited capacities so the query language is less expressive than e.g. SQL [4]. We subsume all kinds of systems that process streams under the term SPSs. An instance of an SPS that is deployed on a host is called a node. A node is a homogeneous unit of a directed graph of distributed nodes that can process a partial query.

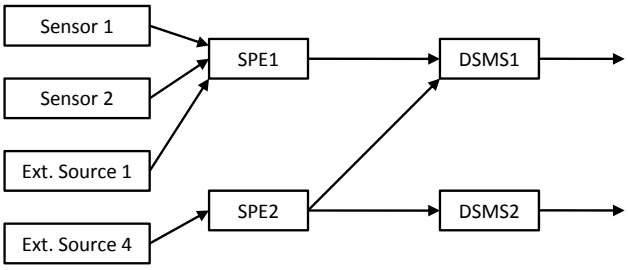

Figure 1. Scenario of heterogeneous stream components

We focus on the challenge of bringing those mentioned systems together. The contribution of this paper is the presentation of a concept of deploying global queries to heterogeneous and distributed sensor nodes and SPSs. This includes the infrastructure of DSAM and a short description of participating components and procedures. In Sec. 2 we motivate our abstract language for global queries, its partitioning to distributed heterogeneous nodes, and mapping of query partitions to query languages of concrete SPSs. The modules and high-level architecture of DSAM are described in Sec. 3 and provide a deeper insight into the concepts of DSAM. As we separate DSAM-AQL and metadata we have to present a conceptual data stream schema. Aside from query partitioning, query mapping is the core of our approach that we show in Sec. 4. Some projects that we mention in Sec. 5 are inspiring our work.

2 Partitioning of DSAM-AQL

DSAM processes global queries and configures the nodes, i.e. it partitions global queries and maps a partial query to either operator assemblies or special destination languages, and deploys it on the adequate node. This helps the integration of heterogeneous stream-emitting and stream-processing nodes, in order to deploy large queries. Some sensors and data sources may emit data that is processed by different types of SPSs (Fig. 1). We assume

 DOI: 10.1109/MDM.2009.92 The original publication is available at http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5088998

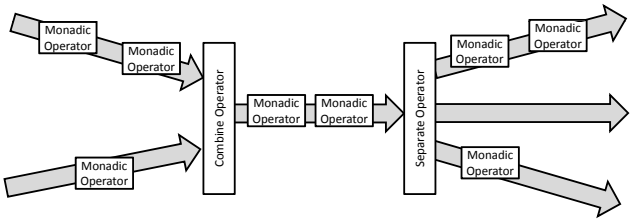

Figure 2. Graphical representation of an AQL subquery

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.

2.1 AQL

The Abstract Query Language (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, combine operators, and separate operators. The membership of an abstract operator to one of these classes is determined by its number of input and output streams. Monadic operators have one input and one output stream e.g. filter, map, or aggregate operator. Operators of the combine operator class are determined by one output stream and a minimum of two input streams e.g. union or join operator. A member of the separate operator class has one input stream and multiple output streams, e.g. split operator.

The following listings 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>
...
```

Queries can use nested subqueries as source streams. Each subquery can unite different streams and separate the streams once (Fig. 2). This pattern of subqueries allows arbitrarily complex directed graphs. In Sec. 4.3 we give a small example. The following excerpt of an AQL-query shows two streams `S1` and `S2` that are combined and split into two streams. There is a subquery that adds time information to stream `S2`. The subquery only consists of the combine operator `Merge`. The top-level query has three monadic operators, the combine operator `Union` and the separate operator `Split`. The decorator `$2` assigns the `Filter` to the second input stream, i.e. the resulting streams of the subquery; decorator `#2` manipulates the items of the second output stream `S4`.

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

Sources and sinks have distinct addresses that are saved in the metadata catalog (Sec. 3). At the moment we support some basic abstract operators like filter (selection), map (functions on attributes), join, union, merge (synchronization of streams with concatenation of fields), and split (conditional creation of multiple streams).

Window definitions are necessary for aggregations and joins. We distinguish between global *source windows* and local *operator windows*[1]. Mapping between *source windows* and *operator windows* is tricky but possible by insertion of additional ID fields (map operator).

2.2 Partitioning

We orient our query processing to query processing in distributed database systems (Fig. 3) that is researched reasonably well [5]. The classical steps "query parsing" and "rule-based optimization" can be adapted from distributed database systems to the context of stream processing. The step "creation of enumerated plans" additionally has to consider the different possibilities of nodes' distribution, heterogeneity, data flow, and communication paths. A metadata catalog is essential for query processing as it contains all information about data sources, topology of nodes, and even the set of available operators. Costs for each plan have to be estimated. There are conflicting objectives like minimizing CPU load and energy consumption on a node, maximizing the data quality (reducing load-shedding), minimizing latency etc. Processing costs can influence CPU load and energy consumption, the costs of transmission of data stream items are energy and time. In most database systems, the number of input and output operations is a metric for costs. In distributed data stream systems, metric and objective strongly depend on the concrete scenario. We weight relevant objectives and combine cost estimators that use different cost models depending on the objective, the data processing system, and the objective's level of detail depending on the relevance.

Further, we distinguish between local cost models and a global cost model. Local cost models consider various levels of detail and depend on the particular model of the underlying node. A deeper presentation of the cost models and their use is beyond the scope of this paper.

The catalog holds a list of available nodes that can run an abstract operator. We assume some operators that are

[1] We call window definitions of data sources *source windows* (windows in query languages similar to SQL) and window definitions that configure stateful operators *operator windows* (windows in e.g. Borealis). Local *operator windows* can be implemented more efficiently in WSNs.

Figure 3. Distribution of global queries

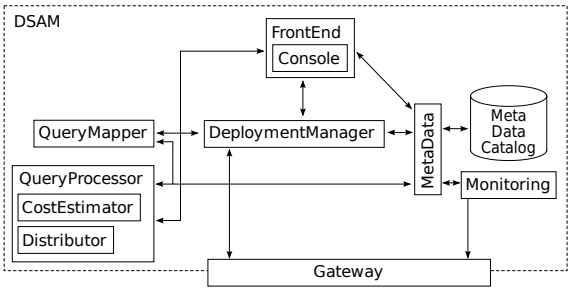

Figure 4. DSAM Architecture

not available on every node. E.g. some special operators for data fusion or User-Defined Operators (UDOs) might be installed on few single nodes. This leads to structural constraints of a query's deployment:

- Input streams are available at a definite place/node
- Some nodes can realize some abstract operators that others can not
- Nodes have individual capacity and performance behavior
- Nodes' connections have reachability constraints and performance behavior

At the moment, we don't apply any heuristics in the "creation of enumerated plans" process. Looking on the objective latency, cost estimation will prefer a solution with minimal traffic and a reasonable clustering of operators on nodes as data flow between nodes is much more expensive than data flow inside of nodes. The resulting query partitions are mapped to the corresponding language. We describe mapping and deployment in Sec. 4.

3 Architecture of DSAM

DSAM is the central manager of distributed data stream applications. It communicates with all nodes, i.e. deployment and monitoring are done either directly or by multihop approaches. We focus on query partitioning and query mapping; so we have planned DSAM as a centralized management system at first. In this section we briefly describe the main modules of DSAM that process queries and maps query partitions to the target systems, the catalog, that stores all relevant metadata and provides both static and volatile metadata, and an optional distributed part that makes common nodes providing an interface for deployment and monitoring to DSAM.

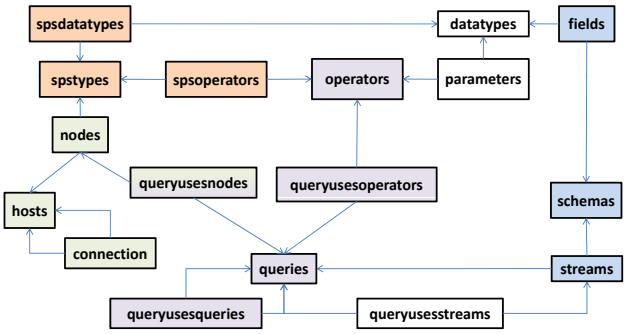

Figure 5. The tables of the catalog

3.1 Modules of DSAM

DSAM consists of five major components:

- `MetaDataCatalog`
  This central component of DSAM stores all necessary metadata used by all other components.
- `QueryProcessor`
  The query processor parses AQL queries and generates the corresponding abstract operator graph plus the operator SPE-node distribution. For the latter, there exists the subcomponent `Distributor` which uses information from the subcomponent `Cost-Estimator`. According to the operator SPE-node distribution the abstract operator graph is going to be split into separate subgraphs. Each of these subgraphs represents a query for a SPE-node.
- `QueryMapper`
  The main purpose of this component is the mapping of operator graphs to different target query languages (detailed description in Sec. 4.1).
- `Frontend`
  This component represents the user interface. At the moment we provide a simple text-based console.
- `Gateway`
  This component realizes monitoring and deployment of partial queries for both WSN-nodes and SPSs.

Fig. 4 depicts the DSAM architecture and the dependencies among its components. The modules `DeploymentManager` and `Monitoring` are explained in the following sections.

3.2 Catalog

All necessary information about nodes, streams, queries, available operators, and topology of queries is stored in a catalog (see Fig. 5). The `Queries` and the `QueryUsesQueries` relations hold all information on

global queries and their partial queries. A hierarchy of queries and partial queries is necessary because global queries may be split up into several partial queries and deployed on different nodes. The catalog furthermore supports a hierarchy with more than two levels (global query ↔ partial query), if queries have to be clustered or the splitting is done in a more fine grained way, which can be inevitable when queries get really big or a lot of queries have to be distributed among a few nodes.

The actual distribution of a global query is stored in the `QueryUsesNodes` table. Every node has an `spstype`. A `stream` entity represents a sequence of tuples with a specific `schema`. Every stream is either published by a query or an external source. External sources are stored with an URL. The URI of this URL is used to instantiate an appropriate adapter to parse tuples from external schemes. A stream may be consumed by different queries and a query can subscribe to more than one stream (`QueryUsesStreams`).

We use a set of generic `datatypes` to describe the schemes of our streams. Not all nodes support all data types and a mapping has to be performed. The information about the unsupported data types and their mapping to our generic types is stored in the `spsdatatypes` table. Some of the supported nodes run on 32bit or 64bit architectures only.

The supported operators of an SPS are held in the `SPSOperators` relation, which is an important information required by the distribution process. The abstract operators of AQL must have equivalents in the SPS. Otherwise the operator has to be deployed on an alternative node.

A distributed and deployed query has to be monitored for its resource consumption to enable redistribution and optimization. Volatile metadata about runtime properties of nodes will be gathered in the monitoring component and can be accessed through the metadata-interface. This information is important when the decision has to be made, whether a node has enough resource capacities to deploy more queries on these nodes.

4 Mapping and Deployment

Fig. 6 shows the whole deployment process of a global query on different kind of nodes.

The result of the partitioning process is 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 don't support query languages, and the deployment of partial queries.

A short example supports the presentation of our concept.

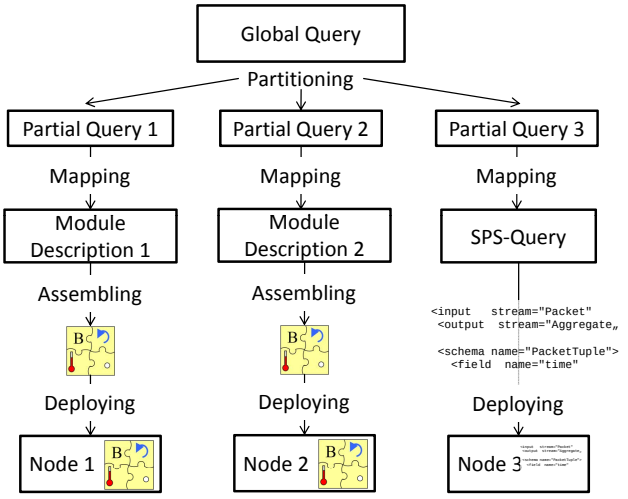

Figure 6. Mapping of Queries

4.1 Mapping

The query mapping is a two phase process and supports different target languages. It takes as input an abstract operator graph representing a partial query 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.

The deployment manager calls the query-mapper component after the initialization of all nodes used by a query, because some information are available at runtime only, e.g. stream ports. Each node corresponds to a specific SPS-type and therefore a specific query language. The query-mapper component transforms the partial query, which is going to be deployed on a specific node. Fig. 7 gives an example for mapping a partial query to an SQL-based query language.

4.2 Code Generation

Some stream processing components don't support query languages. Especially data sources like WSNs have limited capacities and can only be configured by individual software deployment. In [3], AQL is used for global

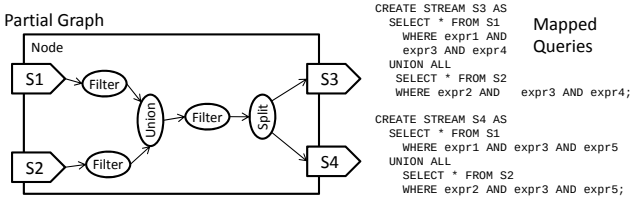

Figure 7. Query mapping example for SQL-based destination languages

```
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);
  }
}
```

Figure 8. Generated sample application code for a BTNode

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.

Fig. 8 shows an excerpt of generated BTNode code. The partial query has an external stream and a local sensor as input streams. It uses local window organization, merge operators, and a join operator. The catalog of DSAM knows the available operators. If all preconditions are fulfilled, the code generation produces valid code, i.e. code that can be linked and deployed in the usual way.

4.3 Example

The fictive 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 Radio Frequency IDentification (RFID) that emits the patient's id to a reader. RFID readers are distributed over the hospital and form the global stream `POSITION` that contains *PAT_ID*, *Antenna_ID*, and *Time*. An SPS combines the `POSITION` stream and the streams emitted from the wireless sensors of each patient.

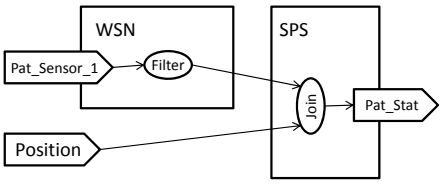

Figure 9. Abstract operator graph with partitioning

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

4.3.1 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. Our interests are only on the last position and the actual 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 actual vital signs of all patients are going to be recognized. These requirements result in the following AQL query:

```
PAT_SENSOR_1, POSITION:
  $1.Filter(Heartbeat <= 60 and Heartbeat => 160),
  Join(Predicate = '$1.Pat_ID = $2.Pat_ID',
       Window1(size = '1s', partition = 'Pat_ID'),
       Window2(size = '1', partition = 'Pat_ID')
  )
:PAT_STAT
```

4.3.2 Example Mapping

The abstract operator graph for the above example AQL query is shown in Fig. 9. Out of 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 for both a SQL-based query language and obtain as result the following two queries:

```
CREATE STREAM QUERY_STREAM_1 AS
  SELECT * FROM SENSORS
    WHERE Heartbeat <= 60 and Heartbeat => 60

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
```

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.

5 Related Work

Stream processing is a problem that is equally related to networking and data processing techniques. Cougar [4] and TinyDB [8] offer WSN-applications that distribute stream queries to WSNs.

Borealis [1] 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 [2] realizes the integration of WSN-application TinyDB [8] and Borealis. Cross Boarder Optimization (CBO) increases the performance of data stream queries. In [7], operators are pushed from Borealis to TinyDB manually. The results motivate us to make more efforts in CBO.

[10] proposes an energy/communication-aware cost model, that supports the placement decision of join operators.

In [9], a Stream Based Overlay Network (SBON) distributes operators to nodes. This approach is similar to our approach and focuses on distribution, i.e. operator placement decisions. We extend this approach by supporting heterogeneous nodes with different SPSs explicitly.

[11] proposes a distributed approach for operator placement decisions that works for query graphs with tree-structure. The Distributed Stream Management Infrastructure (DSMI) [6] supports distributed queries by clustering physical nodes hierarchically. An SQL-based query language is translated into a data flow graph. Each operator is mapped to a snippet of code that is sent to the appropriate node. Each node can parse those snippets and generate native code. The network consists of self-organizing homogeneous nodes without a central planner. So this approach is inapplicable in our scenario, because common heterogeneous nodes can't self-organize.

6 Conclusion and Further Work

We presented support for deployment of queries on heterogeneous distributed WSN and SPS nodes. This concept provides means for elegant and efficient user-centric query definitions. In this paper, we outlined the necessary steps from defining a global query to deployable query partitions. DSAM-AQL and a centralized catalog facilitate the definitions of abstract global queries. Our architecture is extensible as it is also applicable for heterogeneous sensor networks [3]. So, this approach can close the gap between WSNs and SPSs for query definitions. Future work includes the support of Intel Imote2 with TinyOS 2.0 and SunSPOTs with a customized Virtual Machine.

There are many heuristics for distribution of operators available in the literature. Our simple approach will not yet meet the demands of efficient query plan generation in huge scenarios. We will have to find better heuristics before doing simulation and measurements with a huge number of nodes.

In the future DSAM should share partial queries automatically between different queries. In addition to this, future work includes the operator migration between nodes that is necessary for efficient query optimization during runtime. "Changing Topology" and "Changing Data Source Properties" (Fig. 3) are serious and relevant issues in real distributed stream processing that need operator migration.

References

[1] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proceedings of the 2005 CIDR Conference*, 2005.

[2] D. J. Abadi, S. Madden, and W. Lindner. REED: robust, efficient filtering and event detection in sensor networks. In *31st international conference on Very Large Data Bases (VLDB 2005)*, 2005.

[3] F. Dressler, R. Kapitza, M. Daum, M. Strübe, W. Schröder-Preikschat, R. German, and K. Meyer-Wegener. Query Processing and System-Level Support for Runtime-Adaptive Sensor Networks. In *Kommunikation in Verteilten Systemen (KIVS 2009) (accepted)*, 2009.

[4] J. Gehrke and S. Madden. Query Processing in Sensor Networks. *Pervasive Computing, IEEE*, 3(1):46–55, 2004.

[5] D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2004.

[6] V. Kumar, B. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Resource-Aware Distributed Stream Management Using Dynamic Overlays. In *25th IEEE International Conference on Distributed Computing Systems (ICDCS 2005)*, 2005.

[7] W. Lindner, H. Velke, and K. Meyer-Wegener. Data Stream Query Optimization Across System Boundaries of Server and Sensor Network. In *7th International Conference on Mobile Data Management (MDM 2006)*, 2006.

[8] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30:122–173, 2005.

[9] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *22nd International Conference on Data Engineering (ICDE 2006)*, 2006.

[10] M. Stern, E. Buchmann, and K. Böhm. Where in the Sensor Network Should the Join Be Computed, After All? In *1st Ubiquitous Knowledge Discovery Workshop (UKD 2008)*, 2008.

[11] L. Ying, Z. Liu, D. Towsley, and C. Xia. Distributed Operator Placement and Data Caching in Large-Scale Sensor Networks. In *27th Conference on Computer Communications IEEE (INFOCOM 2008)*, pages 977–985, 2008.