Extending Query Languages for In-Network Query Processing

George Valkanas, Dimitrios Gunopulos
Dept. Informatics & Telecommunications
University of Athens, Greece
{gvalk, dg}@di.uoa.gr

Ixent Galpin, Alasdair J. G. Gray, Alvaro A. A. Fernandes
School of Computer Science
University of Manchester, United Kingdom
{ixent,a.gray,alvaro}@cs.man.ac.uk

ABSTRACT

Sensor networks have become ubiquitous and their proliferation in day-to-day life provides new research challenges. Sensors deployed at forest sites, high performance facilities, or areas striken by environmental, or other, phenomena, are only a few representative examples. More recently, mobile sensor networks have made their presence and are rapidly growing in numbers, such as the successful ZebraNet project or PDAs and smartphones. Nevertheless, such networks have mainly been used for data acquisition and data are being processed externally instead of in-network. Basic research problems that arise in the in-network setting include how to adjust in a timely and efficient manner to changing conditions and network topology. In this paper, we present a methodology, based on declarative query processing to alleviate the aforementioned problems, by making the deployment and optimization of a data analysis application as automatic as possible, which also helps execution in mobile environments. Our proposed solution focuses on extending a state-of-the-art sensor network platform, SNEE, with built-in data analysis capabilities.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data Mining

General Terms

Design, Languages

Keywords

Sensor Networks, Data Analysis, In-Network Processing

1. INTRODUCTION

Sensor Networks have received considerable attention recently, as they provide manifold benefits. Monitoring the surroundings with light, particle, motion, temparature, humidity, RFID, camera and other sensor types are merely some examples of what we have nowadays at our disposal. Such data can be used to automatically control high performance buildings (e.g. hotels, power plants), track environmental changes, monitor the health of patients, etc.

Rapid improvements in hardware technologies and wireless communications allow us to deploy large-scale networks of sensing devices, as their cost also decreases over time. However, low cost imposes resource limitations on the nodes and introduces challenges for data collection, aggregation and especially (data) analysis and mining tasks. These constraints, inherent in the domain, need to be tackled efficiently to best exploit the network's resources and potential. An example of a sensor network is shown in Fig.1. Node *0* is called the *sink*, where data are collected, and edges between nodes signify parent-child relations in the communication routing tree (edge direction is the opposite to data exchange direction).

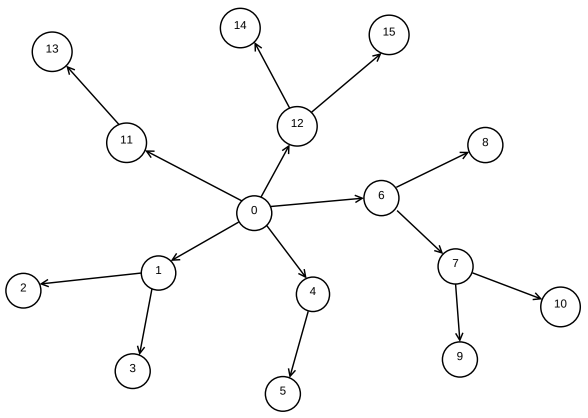


Figure 1: A Sensor Network example

A particularly interesting type of such networks is *mobile sensor networks*, which have become ubiquitous and their proliferation in day-to-day life provides new research opportunities. Smartphones and PDAs, cars, sensor networks deployed at sea, etc. are only a few representative examples. Mobility is inherent in life, as we move between places, e.g. *drive* to work, *go* for a walk, *travel* etc, and is not only related to human activities (e.g. fluid flow, wind etc). Another interesting aspect of such networks is that they *exist*, without (always) an actual need to *deploy* them. Take smartphones for example, owned by different users, most likely strangers to each other. New generations of such devices are equipped with sensing boards and can self-organize in *ad-hoc* networks through WiFi or bluetooth connectivity. Therefore, mobile sensor networks emerge as a new paradigm for modelling and using information. *Mobility*, of course, is what sets these networks apart from classic ones.

Figures 2(a) and 2(b) show the sensor network of Fig. 1 in two consecutive snapshots, specifically illustrating the fact that network structure is not static. We have marked in red the movement direction of each node (nodes 1 and 2 are assumed stationary in the first occasion). In Fig. 2(b), blue lines between two nodes signify the formation of a *new* communication link, which did not exist in the previous timestamp. The sensor network is still rooted at node 0, yet we should note that even such a node may be mobile.

However, despite being rich sources of information, their current application direction is limited to data acquisition. Data are sent to a central, external, location where they are stored and processed, rather than being analyzed in the sensor network. Basic research problems that arise in the in-network setting include how to adjust in a timely and efficient manner to changing conditions and network topology. Re-routing data dissemination is a fundamental step towards this direction [10, 15], but is not the only one to take. A highly optimized execution plan of a query, decided at some point in time, may not be as efficient at another timestamp, due to topology or connectivity changes. *Operator placement trees*, i.e. where each query operator will be hosted and executed, need to be updated as well. These requirements are posed by node *mobility* and come in addition to resource constraints, previously presented.

Trying to address all of these challenges at once is a daunting task. More importantly, writing software components, which run in a distributed and changing environment, under unknown or unforseen circumstances, further complicates the issue. Optimizing hand-written source code for large, or even medium scale network sizes, both in terms of processing time and power consumption, to increase node and, consequently, network longevity is error prone and also impractical.

In this paper we propose a methodology to address these challenges. The fundamental idea is to use a declarative approach to specify, optimize and deploy data analysis techniques in a sensor network. Our proposed solution focuses on extending a state-of-the-art sensor network platform, Streaming NEtwork Engine (**SNEE**), with built-in data analysis capabilities. The SNEE query execution engine optimizes queries posed in SNEEql, a language that is an SQL variant. We show how to extend SNEEql to pose data analysis queries using a declarative description, and how to extend SNEE to deploy such queries in the sensor network. Pursuing this approach greatly automates the process of query deployment and execution. We argue that this type of automation is very beneficial in the mobile setting because it allows the user to quickly respond to changes in the topology of the network. When changes occur, the user has to simply reissue a query, and let the query execution engine re-optmize and re-deploy the query in the network.

To summarize, our main contribution in this paper is that we present the extensions required to add support for in-network data analysis and mining tasks within an existing state-of-the art sensor network platform. A fundamental property of our approach is that we design the analysis tasks in such a way that they can be expressed in data analysis queries that can be optimized by the existing infrastructure without modification.

Our methodology for in-network query processing, is very useful for evaluating in-network data analysis tasks when we use mobile sensor networks. This is because we take advantage of the fact that the used engine already allows for automatic code generation, operator placement and deployment of executables from queries posed in a declarative language.

The rest of the paper is organized as follows: Section 2 sets the background of existing work related to our proposed methodology. Section 3 contains basic concepts of the components we use and build upon. Section 4 introduces our proposed methodology to address the discussed problems, followed by Section 5 which makes explicit the necessary changes to reach our goal. Section 6 concludes the paper and sets future steps.

2. RELATED WORK

Despite the fact that wireless sensor networks are a rather recent field, their application potential is such that they have triggered considerable research. Here, we will mostly focus on query processing aspects, as this is our primary goal.

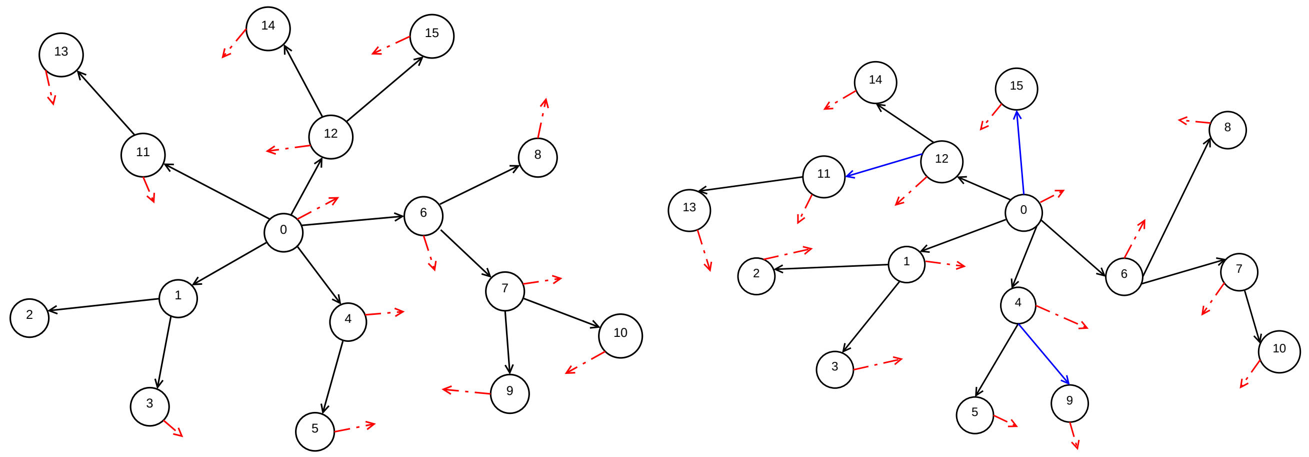
There are basically two lines of work when it comes to sensor networks and query processing: one is to view and manipulate the network as a stream management system, where nodes simply report their values to the sink node, and values are processed in a centralized environment. The second one is *in-network* processing, where queries are (partly or entirely) evaluated by the nodes of the network.

Regarding the first category, sensor networks are simply a data acquisitional platform. Under this perspective, data are locally sensed and propagated to a central node, where they are processed in the form of *streams*. STREAM (Stanford Stream Data Manager) [1] is one of the first systems developped particularly for this setting, viewing streams as relations and processing information using well known database utilities. Another important contribution of STREAM was CQL (Continuous Query Language) [2], an SQL extension, enhanced with a synopsis and a relation-to-stream operators. A more recent approach is SMM (Stream Mill Miner) [21], which focuses on stream mining. It uses UDAs (User Defined Aggregates), which are extensible and expressive enough to support a wide range of functionalities. It is evident that these works do not focus on sensor network issues, such as network optimization and lifespan. Hence, our current work does not directly compare with these ones.

One of the earliest works that fall under the second category is the Cougar project [23], which introduced database concepts into the world of sensor networks, as well as some in-network aggregation. Cougar distinguishes between nodes according to the type of operation that they perform: some simply sense the environment, others perform processing on read values, while query optimization is handled by the remaining ones.

Madden *et al.* developped and presented one of the most well-known in-network query processing frameworks, TinyDB [18]. Sensor readings are represented by a relational table and optimization is limited to operator reordering. TinyDB also introduced ACQP (Acquisitional Query Processing), a query language suited for sensor networks, but with rather limited expressiveness due to the aggregation-based view that the project takes.

Cost-based [3, 5, 9] and trade-off [22] optimization have



(a) Initial topology (b) Changed topology, due to mobility

Figure 2: Consecutive snapshots of a mobile sensor network

also been examined for in-network processing. Zadorozhny *et al.* [24] examine how to maximize the number of concurrent communications, in order to increase throughput. It is clear, though, that these works are outside the context of a fully-fledged system which can be easily used to fullfil user requirements. Nevertheless, they can prove very useful when integrated with an actual system, which will use them as building blocks for more comprehensive optimization.

Operator placement and query routing trees [3, 6, 7, 8] are other fields of research for in-network optimization. Especially suited for changing environments and situations, where there is not a central node with complete view of the network, are adaptive operator placement techniques. In such cases, adaptive processing [10], monitoring and feedback are essential. Recent developments in the area of operator placement include the technique proposed in [6], which picks the next best node to host an operator *immediately*. This technique is also known as *placement update*. The approach, called *dFNS*, casts the operator placement problem to that of finding the 1-median problem in graphs. It gurantees to find the new optimal node to host an operator in a distributed manner. Moreover, the technique keeps the number of exchanged messages to a minimum, therefore it does not hinder network efficiency. To achieve this, the algorithm *i*) identifies the special case where no flooding of the network is required, *ii*) minimizes the flooding radius, in case flooding is unavoidable and *iii*) uses variable speed flooding and eves-dropping. It is also interesting to note that queries with k-ary operators can be treated efficiently by solving *k-1* binary placement operator problems.

As illustrated by the aforementioned works, a more holistic approach to the sensor network processing problem also requires the use of a more declarative query language, such as an SQL variant. An expressive language for in-network processing is SNEEql [4]. Queries posed in SNEEql are then optimized by an execution engine, SNEE [12, 14], taking into account node availability, energy consumption, network topology and other parameters which affect the network's efficiency. Once the query has been optimized, the engine is also responsible for over-the-air deployment and query monitoring. The authors present the steps during query optimization of SNEEql queries and present experimental evidence of their system's performance. We will return to SNEE in the following section, as it plays a fundamental role in our proposed methodology.

A newly presented system, which adopts a hybrid approach between in-network processing and stream management systems is *AnduIN* [17]. *AnduIN* uses a variation of CQL as its declarative query language. It also supports data analysis techniques through UDF (User Defined Functions). However, UDFs are practically black boxes to the optimizer, and this could lead to poor performance of execution plans and severely impact the network. "Black-box" functionality does not fit well the mobile environment either, as optimizing for such conditions becomes a responsibility of the algorithm's designer. Our proposed methodology for both data analysis and standard query processing does not suffer from this drawback.

Regardless of the approach taken to tackle in-network processing, all of the previously presented techniques are primarily suited for *static* environments, where both node topology and parameters which affect query efficiency remain the same during the execution of the query. Therefore, mobile sensor networks have emerged as the new paradigm to tackle this problem. This new type of network has still a long way to go, as current use is mainly limited to data acquisition. The value and importance of such networks has been successfully demonstrated by the ZebraNet project [16], also exhibiting an actual wild-life mobile sensor network deployment. Problems arising in the mobile setting, as identified by ZebraNet, vary from hardware such as weight and infrastructure, to software, such as storage, efficient routing and data dissemination etc., only to name a few.

3. OUR APPROACH

We propose to adopt a holistic approach that relies on extending an existing state-of-the-art infrastructure for the static network counterpart of the problem, to address the above issues at large scale. Such an approach abstracts the entire network and handles many of its particularities, thereby allowing the user to focus on more application-oriented issues. More specifically, we add support for in-network processing of data analysis and mining techniques, e.g. classification, outlier detection, clustering, etc, both at the query language level as well as at the query execution level. Relying on declarative queries for in-network data analysis, which can be efficiently optimized and allow for automatic deployment of executable code, can prove very useful for the mobile setting.

We extend SNEE [14] with the required functionality, that fullfils our motivating goals. SNEE currently supports static topologies as it was initially designed for this type of setting. Query optimization is simpler in this case, but not trivial, since it it involves fewer parameters. Yet its architecture is modular and well-defined [13], adopting and extending established notions from the database domain, e.g. parsing, translating etc. This enables us to enhance it with the desired functionality in a fairly straightforward way.

Queries are expressed in SNEEql, an SQL-syntaxed language particularly suited for sensor networks and streaming environments[1].

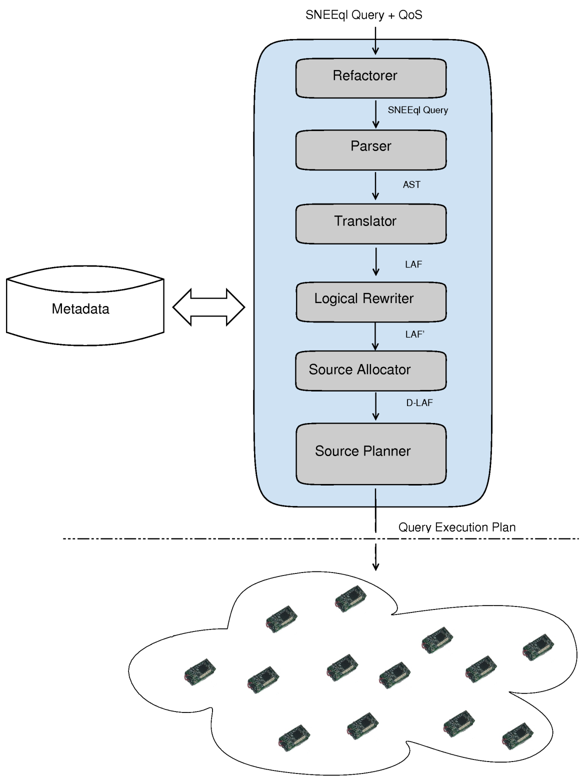


Figure 3: SNEE Optimization steps

The steps that SNEE takes to optimize a SNEEql query are portrayed in Fig. 3. In addition to the query itself, the user may impose some *Quality of Service* requirements. Examples of such requirements are delay tolerance in response time, node energy consumption etc. In the absence of explicit *QoS*, default ones are assumed. This provides greater flexibility as multiple types of sensor networks and application domains can be modelled and satisfied using this architecture. Once the optimization phases have taken place, executable code that will run on the participating nodes is created, compiled and deployed, either through wired interfaces or via over-the-air programming. Of course, when it comes to the mobile setting, over-the-air is the only viable solution.

[1]SNEEql and SNEE also support out-of-network optimization and processing, for relational tables as well.

4. EXTENDING SNEE FOR IN-NETWORK DATA ANALYSIS

With regards to our first goal, i.e. supporting in-network data analysis tasks, our proposed methodology includes the following:

a) enhance the SNEEql language with data analysis and mining capabilities so that they are supported at the declarative level.

b) implement them within the execution engine, taking advantage of the existing query stack.

In addition to addressing our first objective, we would also like to make the transition from total lack of data analysis to full integration as smooth as possible for the users. We have, hence, adopted an approach where we manipulate analysis techniques as if they were any other normal extent (e.g. relation, stream). As an immediate consequence, the language syntax for queries does not need to change, yet we can have support for the desired functionality.[2] Moreover, users can request the creation of data analysis and mining tasks through `CREATE` statements, much like when creating a new table in relational databases. For instance, to create a linear regression classifier over tuples within a 20 minute window from a `TropicalForestData` extent, the user may write the query shown in Fig. 4. The user can then refer to the classifier in following query statements, using `TropDataLRF` as the extent's name.

```
CREATE CLASSIFIER [linearRegression, humidity]
TropDataLRF FROM (
SELECT temperature, humidity
FROM TropicalForestsData[FROM NOW-20 MIN TO NOW]
);
```

Figure 4: Creating a Linear Regression Classifier.

Most importantly, in contrast with the approach in [17], we try to formulate data analysis tasks as algebraic operators, so that, in practice, they are not black boxes, but they leverage the engine's optimizer, using existing cost models. Such an approach not only increases network efficiency, but also developers' productivity, as they do not need to go through the time-consuming iterative process of "optimize-first, execute-next" used in AnduIN, especially for queries that can already be handled efficiently. Furthermore, expressing and decomposing data analysis to algebraic operators also allows for more flexibility on the operator placement problem. On the other hand, *AnduIN* treats each data analysis implementation technique as a separate operator, which needs to be handled individually as a whole. This could sever the network's efficiency.

[2]Though SNEEql syntax does not change, SNEE's parser functionality needs to be augmented, as will be explained in the following sections.

Our line of work is based on what we call *query refactoring* and is in part motivated by the work in [11]. In essense, we reformulate an initially posed query into an equivalent one that uses algebraic operators, which can be appropriately optimized. The query that is eventually optimized and executed is the refactored one. The results and semantics of both queries are identical. What changes is the actual query that needs to be optimized and will be deployed in the sensor network, as opposed to the approach in [17].

Figure 5 shows an example of a SNEEql query with an *intensional* extent, i.e. a source of information for which it is not necessary that its tuples are acquired or stored. Intensional extents may be materialized, e.g. views, or they may be *virtual*, as in our example. In this example we assume that we have some (historic) data from tropical forests at our disposal, with measurements (among others) regarding temperature and humidity readings, and we have built a linear regression classifier on top of them. Using this classifier, we would like to predict the humidity value of Amazon Forest (which is a tropical forest), given its current temperature (this is what "[NOW]" refers to). Here, we assume that there is a direct correlation between humidity and temperature levels in tropical forests and it makes sense to use such a classifier.

```
SELECT RSTREAM AF.temperature, LRF.humidity
FROM   TropForestLRF LRF, AmazonForest[NOW] AF
WHERE  AF.temperature = LRF.temperature;
```

Figure 5: Using a Linear Regression Classifier.

Therefore, our example uses `TropForestLRF`, which is the classifier, to predict the humidity level of tuples received from data input stream *AmazonForest*. Incorporating intensional extents in such a way also has a natural interpretation in terms of query semantics: "Give me the humidity value of a tuple from (virtual) relation `TropForestLRF`, for which the temperature is equal to the current sensed temperature from `AmazonForest`".

Using the stored metadata, extent `TropForestLRF` is recognized as an intensional one during parsing (phase 1), and specifically as a linear regression classifier. At that point the query is refactored to the one shown in Figures 6 and 7. Note that this is in fact a single query, but we have split it in two parts to increase readability. The query in Fig. 7 replaces the "`(ab_COMP)`" string in Fig. 6.

Figure 8 shows the Query Operator Tree of the refactored query form, using the algebra defined in [4]. All of the operators can be optimized through the existing infrastructure, since none of them is a black box. As a direct consequence of this fact, these operators can take advantage of the operator placement update mechanism described in the previous section, which makes our approach suitable for the mobile setting as well.

In addition to being interesting and useful on their own right, data analysis and mining tasks can prove very helpful for the mobile setting. Uncovering hidden patterns within mobile nodes can provide significant benefits: value correlation is useful in energy and storage savings, outlier detection could be an indication of interesting events etc. To illustrate with an example, one of ZebraNet's challenges was to choose a data propagation scheme so that data will reach the (mobile) sink. Instead of flooding the network, they opted for a

```
SELECT RSTREAM AF.temperature, a * AF.temperature + b
FROM   AmazonForest[NOW] AF, (ab_COMP) LRF;
```

Figure 6: Refactored Query using a Linear Regression Classifier.

```
SELECT RSTREAM
    (r.n*r.sxy - r.sx*r.sy) / (r.n*r.sxx - r.sx*r.sx) as a,
    (r.sy*r.sxx - r.sx*r.sxy) / (r.n*r.sxx - r.sx*sr.x) as b
FROM (
    SELECT RSTREAM
        COUNT(t.temperature) as n,
        SUM(t.temperature) as sx,
        SUM(t.humidity) as sy,
        SUM(t.temperature*t.humidity) as sxy,
        SUM(t.temperature*t.temperature) as sxx
    FROM (
        SELECT RSTREAM temperature, humidity
        FROM TropForest[FROM NOW-20 MIN TO NOW]
    ) t
) r;
```

Figure 7: SubQuery for (ab_COMP).

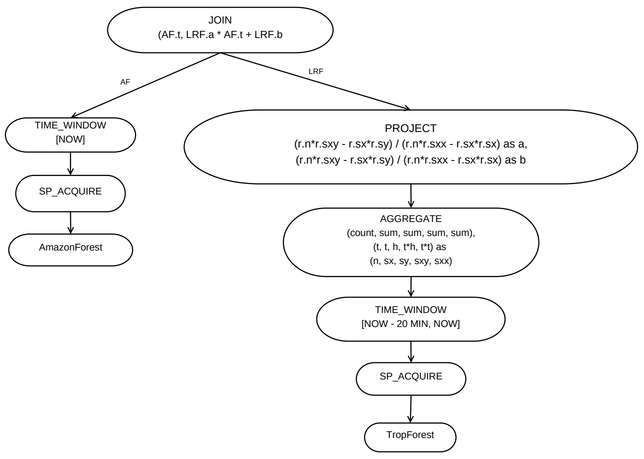

JOIN
(AF.t, LRF.a * AF.t + LRF.b
AF
LRF
TIME_WINDOW
[NOW]
SP_ACQUIRE
AmazonForest
PROJECT
(r.n*r.sxy - r.sx*r.sy) / (r.n*r.sxx - r.sx*r.sx) as a,
(r.n*r.sxy - r.sx*r.sy) / (r.n*r.sxx - r.sx*r.sx) as b
AGGREGATE
(count, sum, sum, sum, sum),
(t, t, h, t*h, t*t) as
(n, sx, sy, sxy, sxx)
TIME_WINDOW
[NOW - 20 MIN, NOW]
SP_ACQUIRE
TropForest

Figure 8: Query operator tree for the Linear Regression classifier

best-effort scheme, where nodes, upon reaching another set of nodes, would delegate data to the one that had reached a sink the most. Though this proved better than flooding, building an actual classifier, with additional features based on mobility patterns (trajectories), e.g. [19], could provide even better results. The classifier would run in-network and would be continuously updated with new values.

Our approach for enabling in-network processing of data analysis tasks can also handle more complex constructs, such as D3 [20]. D3 is an oulier detection algorithm, using *kernel density estimators* at its core and more specifically the Epanechnikov estimator. The algorithm approximates the distribution of sensed data through *sampling* and reports sensed tuples as outliers based on the probability that they have been drawn from the same underlying distribution, by employing Scott's rule. A significant advantage is that it can also handle multi-dimensional outliers, which makes it even more practical and useful.

Detecting outliers is useful for several reasons, e.g. *i*) indication of events, *ii*) identification of faulty hardware, etc. Conversely, outliers could impact decision making, therefore detecting them beforehand is a first step to data cleaning. A query using the D3 outlier detection algorithm is shown in Fig. 9, where we want to find outliers of the *AmazonForest* stream. Sensed tuples are coined outliers if their temperature and humidity values have less than a 15% probability to have been drawn from the underlying distribution. In this example we assume that the distribution has been computed over the AmazonForest stream itself, thereby outliers are tuples which deviate significantly from previously sensed ones.

```
SELECT RSTREAM AF.*
FROM   AmazonForest[NOW] AF, d3 od
WHERE  AF.temperature = od.temperature AND
       AF.humidity = od.humidity AND
       od.probability < 0.15;
```

Figure 9: Example query of D3 outlier detection.

Figure 10 shows the basic outline of the D3 algorithm, both for children and parent nodes. Note that the algorithm maintains a sample of the sensed tuples (line 9), approximates the data distribution and also computes the density around each tuple (line 28).

Refactoring the initial query is fairly elaborate - but still doable - so we avoid getting into too much technical detail. We should also mention at this point that sampling is handled and supported as a separate data analysis task. Therefore, our proposed approach is not monolithic either, as was the case for [18], but it can use simple analytical tasks as building blocks for more complex ones.

The static topology that SNEE relies on is a rather strict restriction, which we need to alleviate with a view towards mobile sensor networks. Another restriction that needs to be tackled is the fact that once the query execution tree has been created, it remains the same until the query is stopped by the user. Under changing conditions, this is not efficient, as a query plan that was good at deployment time may be moderate or even bad afterwards. This is especially true for long-running queries. The longer a query runs, the greater the need for query tree adaptation is. Using an execution engine that supervises and automates the steps of execution and deployment, given a network description, is much more preferable to a hand-written alternative, as the application developer is not required to deal with the network's intricacies. This holds true especially for large, or even medium size networks. Supporting in-network data analysis techniques through the same declarative query language and processing mechanism as conventional queries makes our proposed approach also suitable for the mobile setting.

5. IMPLEMENTATION

In this section we identify the key changes that we need to make to the existing infrastructure to achieve the desired functionality. This will provide a clearer perspective of how our methodology can be applied.

```
Algorithm D3 (Distributed Deviation Detection)
Let $W^w$ and $W^b$ be the sliding windows of leaf and parent
    nodes;
Let $R^w$ and $R^b$ be the samples on $W^w$ and $W^b$;
Let $\sigma^w$ and $\sigma^b$ be the standard deviations on $W^w$ and $W^b$;
Let $f$ be the fraction of the sample propagated from a child
    to its parent;

1.  procedure D3()
2.      assign one leaf node to each one of the input streams;
3.      configure all parent nodes in a hierarchy on top of leaf
    nodes;
4.      initiate ParentProcess() for each parent node;
5.      initiate LeafProcess() for each leaf node;
6.      return;

7.  procedure LeafProcess()
8.      when a new value S(i) arrives
9.      update $R^w$, $\sigma^w$;
10.     if (S(i) included in $R^w$)
11.         send S(i) to parent with probability $f$;
12.     if ( IsOutlier($R^w$, $\sigma^w$, S(i)) )
13.         report S(i) as an outlier;
14.         send S(i) to parent;
15.     return;

16. procedure ParentProcess()
17.     when a new message from a child node arrives
18.     if ( message is new outlier P )
19.     if (IsOutlier($R^b$, $\sigma^b$, S(i))
20.         report P as an outlier;
21.         send P to parent;
22.     if (message is new value from child l)
23.         update $R^b$ and $\sigma^b$
24.         if (the new value is included in $R^b$)
25.             send new value to parent with probability f;
26.     return;

27. procedure IsOutlier(sample R, stddev $\sigma$, point P)
28.     use $R$ and $\sigma$ to estimate N(P, r);
29.     if ( N(P, r) < t )
30.         mark P as an outlier;
31.     return;
```

Figure 10: Outline of D3 algorithm

It is easy to see that the query parser requires modification. Note that this change is due to the query refactoring approach alone. Moreover, a change in the stored metadata is needed, in order to distinguish between intensional and extensional extents. Hence, the new parser implements a superset functionality of the current one. The actual functionality of refactoring can be implemented in two ways: the first is to create a new query as the one in Fig. 6 and 7, which will be re-submitted. The second is to directly output an AST, ready to be translated, as explained in the previous section. The overhead is small in either occasion and not really significant, as it will be executed only once. Refactoring queries that do not include intensional extents has no effect. The query then proceeds with the rest of the optimization phases. The updated stack is presented graphically in Fig. 11.

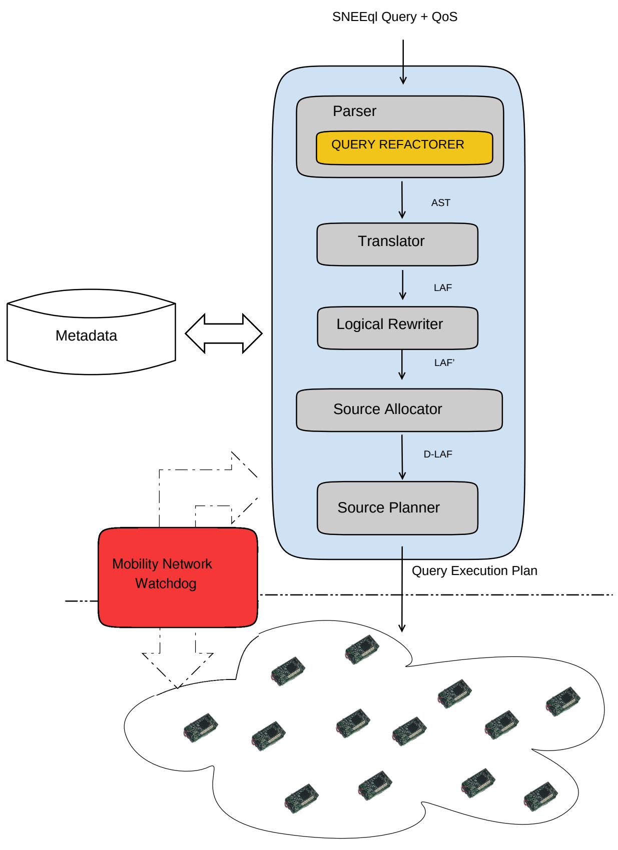


Figure 11: Updated query stack steps to support data analysis and mining tasks

6. CONCLUSIONS AND FUTURE DIRECTIONS

In this work we proposed a well-defined methodology to extend a state-of-the-art execution engine for sensor networks with data analysis capabilities. Our approach is such that it can also fit the mobile setting. Through careful inspection of the steps that a query undergoes, we identified the key phases and their respective software components that we need to enhance. We also presented the methods we plan on using to achieve this functionality. Immediate future steps include a concrete implementation of the aforementioned proposal and an extensive experimental evaluation.

Another future direction is that of a mobility "watchdog". Taking a more proactive approach, this software module would be responsible for tracking the sensor network for changes. Whenever drastic changes are observed in its properties (e.g. connectivity, node topology, routing trees), it could make the process of stopping and restarting the query fully- or semi-automatic. This is the simplest approach to take using the existing architecture. Though more complex approaches, such as operator updating might yield better results, they require substantial additional effort to implement within SNEE's optimization stack. The reason is that SNEE produces executable code which is destined to run under very tight *time* and *cost* constraints, making network efficiency and longevity some of its primary goals. As operator updating requires exchanging additional messages, this might prove harmful for the produced query execution plan eventually.

7. ACKNOWLEDGMENTS

The authors would like to thank Prof. Norman W. Paton, University of Manchester, for fruitful discussions that greatly contributed to the development of this work. This work has been supported by *SemSorGrid4Env* (FP7-223913) and MODAP European Commission projects.

8. REFERENCES

[1] STREAM: Stanford Stream Data Management Project, http://www-db.stanford.edu/stream.

[2] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15:121–142, June 2006.

[3] B. J. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. In *Proceedings of the 2nd International Conference on Information processing in Sensor Networks (IPSN)*, pages 47–62, 2003.

[4] C. Y. Brenninkmeijer, I. Galpin, A. A. Fernandes, and N. W. Paton. A semantics for a query language over sensors, streams and relations. In *Proceedings of the 25th British National Conference on Databases*, pages 87–99, 2008.

[5] C. Y. A. Brenninkmijer, I. Galpin, A. A. A. Fernandes, and N. W. Paton. Validated cost models for sensor network queries. In *Proceedings of the 6th International Workshop on Data Management for Sensor Networks (DMSN)*, pages 1–6, 2009.

[6] G. Chatzimilioudis, A. Cuzzocrea, and D. Gunopulos. Optimizing query routing trees in wireless sensor networks. In *ICTAI (2)*, pages 315–322, 2010.

[7] G. Chatzimilioudis, N. Mamoulis, and D. Gunopulos. A distributed technique for dynamic operator placement in wireless sensor networks. In *Proceedings of the 11th International Conference on Mobile Data Management (MDM)*, pages 167–176, 2010.

[8] G. Chatzimilioudis, D. Zeinalipour-Yazti, and D. Gunopulos. Minimum-hot-spot query trees for wireless sensor networks. In *Proceedings of the 9th International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, pages 33–40, 2010.

[9] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-based approximate querying in sensor networks. *VLDB J.*, 14(4):417–443, 2005.

[10] A. Deshpande, Z. Ives, and V. Raman. Adaptive query processing. *Found. Trends databases*, 1:1–140, January 2007.

[11] L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst.*, 25:457–516, December 2000.

[12] I. Galpin, C. Y. Brenninkmeijer, A. J. Gray, F. Jabeen, A. A. Fernandes, and N. W. Paton. Snee: a query processor for wireless sensor networks. *Distrib. Parallel Databases*, 29:31–85, February 2011.

[13] I. Galpin, C. Y. A. Brenninkmeijer, F. Jabeen, A. A. Fernandes, and N. W. Paton. An architecture for

query optimization in sensor networks. In *ICDE*, pages 1439–1441, 2008.

[14] I. Galpin, C. Y. A. Brenninkmeijer, F. Jabeen, A. A. Fernandes, and N. W. Paton. Comprehensive optimization of declarative sensor network queries. In *SSDBM*, pages 339–360, 2009.

[15] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11:2–16, 2003.

[16] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 96–107, 2002.

[17] D. Klan, M. Karnstedt, K. Hose, L. Ribe-Baumann, and K.-U. Sattler. Stream engines meet wireless sensor networks: cost-based planning and processing of complex queries in anduin. *Distrib. Parallel Databases*, 29:151–183, February 2011.

[18] 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, March 2005.

[19] C. Panagiotakis, N. Pelekis, and I. Kopanakis. Trajectory voting and classification based on spatiotemporal similarity in moving object databases. In *Proceedings of the 8th International Symposium on Intelligent Data Analysis (IDA): Advances in Intelligent Data Analysis VIII*, pages 131–142, 2009.

[20] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos. Online outlier detection in sensor data using non-parametric models. In *VLDB*, pages 187–198, 2006.

[21] H. Thakkar, N. Laptev, H. Mousavi, B. Mozafari, V. Russo, and C. Zaniolo. Smm: a data stream management system for knowledge discovery. In *Proceedings of the 27th International Conference on Data Engineering (ICDE)*, page (to appear), 11-16 April 2011.

[22] N. Trigoni, Y. Yao, A. Demers, J. Gehrke, and R. Rajaraman. Wavescheduling: energy-efficient data dissemination for sensor networks. In *Proceedings of the 1st International Workshop on Data Management for Sensor Networks (DMSN)*, pages 48–57, 2004.

[23] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31:2002, 2002.

[24] V. I. Zadorozhny, P. K. Chrysanthis, and P. Krishnamurthy. A framework for extending the synergy between mac layer and query optimization in sensor networks. In *Proceedings of the 1st Int Workshop on Data Management for Sensor Networks*, pages 68–77, 2004.