Declarative In-Network Sensor Data Analysis

George Valkanas[1], Ixent Galpin[2], Alasdair J. G. Gray[3], Alvaro A. A. Fernandes[3], Norman W. Paton[3], and Dimitrios Gunopulos[1]

[1] Dept. of Informatics & Telecommunications, University of Athens, Greece
{gvalk,dg}@di.uoa.gr
[2] Universidad Jorge Tadeo Lozano, Colombia
ixent.galpin@utadeo.edu.co
[3] School of Computer Science, University of Manchester, United Kingdom
{a.gray,alvaro,norm}@cs.man.ac.uk

Abstract. Bridging data analysis techniques with classic query processing has long been of interest in the database community. Most approaches, however, are usually developed with a specific domain in mind, e.g. relational, streaming etc., use their own query language, or focus on specific techniques. In this paper, we propose a simple, yet effective, extension to standard or commonly used declarative processing languages to support data mining. Our approach is independent of a particular domain, and by utilizing a *query refactoring* technique, optimization issues are taken care of by the underlying query processing engine, which is already in place and knows best the setting's particularities. Therefore, our approach promotes ease of programmability, development, and use of the data mining techniques, with minimal modifications in the query processing stack. We demonstrate our technique through an experimental evaluation, using our prototype system *SNEE-A*, that runs in-network data analysis given a sensor network deployment, a setting with several critical constraints.

1 Introduction

Bridging classic query processing with data analysis and mining techniques has long been of interest in the database community [15, 17, 24, 27]. Following the widespread attention that data streams and sensor networks have received in the past few years, due to their potential benefits, e.g., environmental monitoring, automated facility control etc., several approaches have been proposed [21, 29, 32]. The main advantages are an in-place infrastructure, and a higher ease of programmabing data mining techniques.

However, such approaches typically suffer from the following shortcomings: i) they propose their unique query language, ii) focus on specific algorithms, and iii) are developed having a single domain in mind. Most rely on User Defined Functions (UDFs), which have been generally criticized for being "black-boxes" and not optimization-friendly [12, 23]. Sub-optimal plans, however, impact differently one domain from the other, making these techniques unfit to port between domains. Sensor networks, for instance, are constrained on resources, face a distributed setting and dynamic environment, and are substantially different from a relational database setting. A poor execution plan that runs in a relational database simply increases running time of the query, leaving the user to wait. On the contrary, a poor plan destined to run in a sensor network could completely drain nodes from their energy and render the network useless. Therefore, it is important to optimize the code for each setting of application, but doing so manually is error prone and requires significant technical effort.

To successfully integrate query processing and data analysis, we identify the following desiderata:

i) The primary objective is to support data analysis and mining tasks, such as clustering, classification, outlier detection, etc., in a consistent way across domains, e.g. relational databases, distributed databases, or streaming sensor networks.
ii) *Efficiency* is still a major concern, although its definition is highly dependent on the domain. For instance, relational databases are interested in reducing response time, whereas for a sensor network energy consumption is a first-class citizen.
iii) Ease of programmability and development of data analysis techniques is an additional goal, as it increases a programmer's productivity. Note that the integration of new techniques needs to satisfy the *efficiency* constraint we mentioned above.
iv) Maximize adoption and ease of use by the end-users.

To address these concerns, in this paper we propose an approach that allows users to express data mining tasks through a high-level declarative language, e.g. SQL. Our **contributions** in this paper can be summarized as follows:

i) We give an approach to define and execute data analysis techniques using declarative queries. The main advantages are speed in development and deployment, a simple syntax and a system that takes care of correctness implications.
ii) We conceptualize data analysis techniques as *intensional* extents, i.e. sources whose data do not have to be acquired or stored, as opposed to *extensional* ones, and derive a flexible framework where we can combine these two types within the same query.
iii) We propose a *query refactoring* approach, motivated by the idea that several data mining algorithms can be expressed as algebraic operators. Therefore, unlike UDFs, we can leverage the optimizers that query processing engines already have.

We showcase our approach through two entirely different data analysis techniques, applied in the in-network setting and provide experimental evidence of our prototype system SNEE-A, a sensor network query processing engine that supports data mining using the discussed methodology. Our example techniques are:
Online correlation: If we know there is a correlation between the values of two measured variables, e.g. temperature and humidity, is it possible to predict humidity knowing only temperature readings and can this be done efficiently?
Outlier detection: Given a sensor network deployment that measures humidity and temperature, we want to be informed of anomalies in readings.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 shows the necessary language extensions to integrate data analysis with standard query processing, whereas Section 4 the needed modifications to the query engine. Experiments are in Section 5, and Section 6 concludes the paper.

2 Related Work

We first focus on general approaches that bring together data mining tasks and classic query processing, and then focus in more specific approaches for the in-network setting, as our application domain is such.

Unifying query execution engines with data mining techniques under a common query language has been a research topic since the mid 1990s. Support for association rule mining [15, 17, 26] and classification [24, 27] at the language level has been examined. These approaches, however, were developed for the relational setting and were too narrow on their supported algorithms. For example, both classification techniques dealt with decision trees. More importantly, though, they all employ UDFs to achieve their goals. UDFs have long been critized as effectively being "black boxes" and not optimization-friendly [12, 23]. However, the impact of non-optimizable query operators may largely vary from setting to setting. For instance, a poor execution plan in the relational setting results in longer execution times and lower user satisfaction. On the other hand, poor optimization for in-network processing is detrimental, because it can drain node energy very quickly, rendering the sensor network practically useless.

The work in [25] employs SQL queries to perform K-Means clustering through the use of triggers and vendor-specific SQL scripting extensions. However, the objective of that work is not to integrate data mining with a query language, but rather to use (relational) database technologies to perform K-Means clustering. Also note that the utilities used therein (i.e., triggers) operate differently in relational and streaming environments.

Moreover, when moving from the classic relational domain to more complex ones, e.g. streaming environments, the declarative language itself is constrained in expressiveness. To overcome this, most proposed systems and techniques introduced their own declarative query language, which is usually an SQL variation for that setting, such as CQL [7] (Continuous Query Language) and variants [18], ACQP (Acquisitional Query Processing) [22] and SNEEql [9] (SNEE query language). ESL, proposed in [21], employs User Defined Aggregates (UDAs), a subset of UDFs, thereby inheriting their drawbacks. These languages, however, for the most part, do not focus on data analysis and mining support. MMDL [29], an ESL extension, is a step forward in this direction for the streaming setting. However, as pointed out in [32], memory requirements of UDAs (therefore ESL and MMDL) cannot be clearly estimated from their syntactic structure, which does not fit well the resource-limited sensor network setting.

We now briefly discuss query execution engines for sensor network, as our application domain of choice is such. Typically, there are two lines of work in this area: i) Gather all sensed data to a central node, the *sink* and perform operations in a centralized environment or ii) view the network as a distributed processing query engine, and (partially or entirely) evaluate queries in-network.

Systems that fall under this category include STREAM (Stanford Stream Data Manager) [6], Aurora [2], Borealis [1], TelegraphCQ [20] and the more recent SMM (Stream Mill Miner) [29]. SMM is the only one among them to target specifically at data mining support. It uses UDAs, thus inheriting their drawbacks which we already discussed.

Works under the second category include the Cougar project [34], which introduced database concepts in sensor networks, as well as some in-network aggregation. Madden *et al.* developed one of the most well-known in-network query processing frameworks, TinyDB [22]. Sensor readings are represented by a relational table, optimization is limited to operator reordering and the same load is distributed among the nodes in the participating set, disregarding their position in the network topology. Despite their novelty in in-network query processing, none of them considers data mining tasks.

SNEE (Sensor NEtwork Engine) [13], is a sensor network query execution engine, optimizing queries submitted in a declarative language, *SNEEql*. SNEE considers multiple parameters that affect network efficiency, e.g. network topology, node availability, energy consumption of operators. By default, SNEE optimizes node power consumption, and maximizes network longevity. Quality of service requirements may also be imposed (e.g. delivery constraints), which effectively alter the optimization goal.

A hybrid approach is adopted by the recently presented *AnduIN* [18]. *AnduIN* uses a declarative, streaming language variant and supports data analysis techniques through UDFs at the query level. Another difference is that we model data analysis as algebraic operators and leverage the execution engine's optimizer, whereas *AnduIN* uses UDFs and evaluates code performance offline, through simulations.

Regarding in- and out of network custom data analysis and mining techniques, there is a large body of literature [4], not to mention for classic settings. However, these are stand alone solutions and not integrated with a query processing engine, which we aim for. It has also been discussed that they sometimes contradict established notions of relational databases [11], let alone streaming environments. Furthermore, in these cases, optimization issues are a responsibility of the algorithm's designer, despite the existence of optimizers in the processing engines, which we would like to take advantage of.

3 In-Network Data Analysis with a Declarative Language

Towards fulfilling our goals, we follow a holistic methodology that involves:

a) extending the declarative language appropriately, so that data analysis techniques are supported at the query language level.
b) implementing them as extensions to the query optimization stack, building on the contribution that they can be denoted by intensional extents.

For ease of discussion, we will use SNEE and SNEEql [9] as the query execution engine and language respectively. We chose SNEE due to its well-defined and modular query optimization stack, that extends the classical two-phase optimization approach from distributed query processing [19], as well as for the various optimization goals it supports. SNEEql is a declarative query language for sensor networks inspired by expressive classical stream query languages such as CQL [7]. Nevertheless, we stress that our findings apply in similar approaches where declarative languages are applicable.

3.1 Extending SNEEql

To support data analysis tasks at the declarative level, we manipulate them as any other *extensional* extent (i.e. relation, stream), leaving the query language syntax intact. As a distinction, We refer to them as *intensional* extents, i.e. sources of information for which it is not necessary that their tuples are acquired or stored.

Users create data analysis and mining tasks through `CREATE` statements, like creating a view in relational databases, which alters SNEE's metadata to accommodate the new extent. To support this functionality, we extend SNEEql's data definition language (DDL), utilizing a hierarchical decomposition of data analysis categories and their techniques. Figure 1 shows the updated DDL syntax. Tokens in bold are reserved terms, while the rest are replaced by the corresponding rule. Unmatched tokens refer to specific algorithms and their respective parameters, e.g. the value k for k-Means.

```
DDLIntExtent ::= createClause fromClause;
createClause ::= CREATE dattype [ datsubtype, datparams ]  identifier
fromClause   ::= FROM ( fromItem )
dattype      ::= CLASSIFIER | CLUSTER | SAMPLE |
                 ASSOCIATION_RULE | OUTLIER_DETECTION |
                 PROBFN | VIEW
datsubtype   ::= linearRegression |  knn | d3 | kmeans | ...
datparams    ::= paramListItem, dataparams | paramListItem
identifier   ::= Any valid identifier
fromItem     ::= Either an extent in the schema, or a sub-query
```

Fig. 1. Syntax for Defining an Intensional Extent.

```
Schema:
  AmazonForest:stream (id:int, time:ts, temperature:float)
  TropicalForestData:stream (id:int, time:ts, temperature:float, humidity:float)
```

Fig. 2. Example schema of two streams expressed in SNEEql.

3.2 Online correlation

Assume, for instance, the two extensional stream extents of Fig. 2, one for the amazon forest that reports temperature values, and a more general tropical forest stream that reports temperatures and humidity values.

Figure 3 shows the creation of a linear regression classifier over *TropicalForestData*, using tuples within a 20 minute window to construct it. We can then use that classifier in subsequent queries with *TropForestLRF* as the extent's name, as shown in Fig. 4. Here we wish to predict humidity values from the *AmazonForest* extent given its current temperature (this is what 'NOW' refers to). 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*". This makes our approach easy to understand for users who are familiar with SQL but not data analysis techniques.

Conceptually, when an intensional extent variable appears in an equality condition in the WHERE clause, what happens is akin to variable binding in logic languages, e.g. Datalog [3], after all necessary semantic checks have successfully completed. Our *query refactoring* approach makes extensive use of these value bindings.

Note that *TropForestLRF* is constantly updated, as it is an *intensional* extent, built over the *TropicalForestsData* stream. As data is acquired from that extent, the classifier is updated as well. More generally, intensional extents inherit the acquisitional properties of extensional ones, upon which they are built.

```
CREATE CLASSIFIER [linearRegression, humidity]
TropForestLRF FROM (
    SELECT RSTREAM temperature, humidity
    FROM TropicalForestData[FROM NOW-20 MIN TO NOW]
);
```

Fig. 3. Creating a Linear Regression Classifier.

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

Fig. 4. Using the TropForestLRF intensional extent.

```
CREATE OUTLIER_DETECTION [D3, 5, 0.15] d3od
FROM (
    SELECT RSTREAM temperature
    FROM AmazonForest[FROM NOW-20 MIN TO NOW]
);
```

Fig. 5. Creating a D3 outlier detection extent.

```
SELECT RSTREAM AF.temperature
FROM   AmazonForest[NOW] AF, d3od od
WHERE  AF.temperature = od.temperature;
```

Fig. 6. Using the d3od intensional extent.

3.3 Outlier detection

Our approach for enabling in-network processing of data analysis tasks can also handle more complex constructs, such as the D3 outlier detection algorithm [28]. Detecting outliers is useful for several reasons, e.g. event indication, identification of faulty hardware, or as a first step to data cleaning.

D3 uses sampling and the Epanechnikov kernel density estimator, to approximate the distribution of sensed data. It reports data as outliers if they have low probability to have been drawn from the same underlying distribution that created their (multi-dimensional) neighboring data. D3 requires two parameters: a neighborhood *range* and a *probability* threshold. Figure 5 shows how to create a D3 intensional extent over the temperature values of *AmazonForest* from the last 20 minutes, where $range = 5$ and $probability = 15\%$. Figure 6 shows a query using that extent to check whether the most recent tuple is an outlier. As we can observe from it, intensional extents can also be part of a self-join query.

4 Query Refactoring

In this section, we present how an existing query execution infrastructure, i.e. SNEE, can be modified, so that it supports data analysis techniques. When a query is submitted, we check with SNEE's metadata whether it contains intensional extents or not. The occurrence of an intensional extent triggers its substitution by a templated subplan, which performs its algorithmic computations. We collectively refer to this process as *query refactoring* [31]. In essense, we reformulate an initially posed query into an equivalent one, that is also expressed in the same declarative language (SNEEql).

This approach has the added advantage that data analysis techniques are no longer black boxes but can leverage the engine's optimizer, without altering query semantics. Query refactoring only affects the parts of the query related to the intensional extent, leaving the rest intact. To better illustrate the needed modifications, Fig. 7a)-b) show SNEE's optimization stack with and without the query refactoring module, respectively.

The output of the query refactoring process depends on the intensional extent used. This process is transparent to the user, who will simply write the initial query. It follows that we need not perform any changes to the query language level to support data analysis in such a way. On the downside, our approach is limited by the expressive power of SNEEql. Provided that the algorithmic description of a data analysis technique can be expressed in the query language, it can then be incorporated in our approach easily. We now demonstrate how query refactoring is applied to the two examples from Section 3.

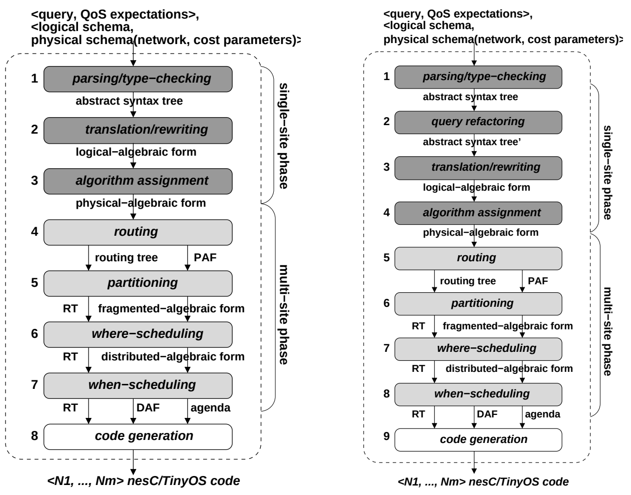


Fig. 7. SNEE optimization stack: (left) original stack, (right) query refactoring approach for data analysis techniques.

4.1 Linear Regression

We showcase the use of templated code in query refactoring through a general SNEEql query with a Linear Regression classifier. Assume an extent *LRFSource* with attributes *X* and *Y*, as shown in Fig. 8. Attribute *X* is the independent variable and *Y* is the dependent one. Values in bold are placeholders for actual extents and attribute names.

Training this classifier is equivalent to computing coefficients (a, b) based on *LRF-Source*, i.e., the source over which the extent was created. Moreover, these need to be appropriately updated as new readings are acquired, as discussed in Section 3. Both of these goals can be achieved through the templated SNEEql subplan of Fig. 9.

Figure 10 shows a SNEEql query using LRF, where the last argument in the `SELECT` clause is the dependent variable of the classifier. The position of both vari-

```
CREATE CLASSIFIER [linearRegression, Y] LRF
FROM (
  SELECT RSTREAM X, Y
  FROM LRFSource
);
```

Fig. 8. Creation of a templated 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*r.sx) as b
FROM (
       SELECT RSTREAM COUNT(t.X) as n,
           SUM(t.X) as sx, SUM(t.Y) as sy,
           SUM(t.X*t.Y) as sxy, SUM(t.X*t.X) as sxx
       FROM (
           SELECT RSTREAM X, Y
           FROM LRFSource
       ) t
) r;
```

Fig. 9. Templated subquery for computing (a, b) values

```
SELECT RSTREAM $s_1$, $s_2$, ..., $s_n$, lri.Y
FROM $e_1$, $e_2$, ..., $e_m$, LRF lri
WHERE $w_1$ $OP_1$ $w_2$ $OP_2$ ... $OP_{l-1}$ $w_l$
```

Fig. 10. General form of a query using the *LRF* extent

```
SELECT RSTREAM $s_1$, $s_2$, ..., $s_n$, lri.a * Z + lri.b
FROM $e_1$, $e_2$, ..., $e_m$, (LRF_ab) lri
WHERE $w_1$ $OP_1$ $w_2$ $OP_2$ ... $OP_{l-2}$ $w_{l-1}$
```

Fig. 11. General form of a refactored query using Linear Regression

ables is insignificant; it is at the end to ease readability. Projected attributes from other extents may also appear in the `SELECT` clause. The OP_is in the `WHERE` clause are standard boolean operators, e.g., *AND*, *OR*, combining boolean expressions (the w_is).

Through SNEE's metadata, we see that *LRF* is an intensional extent, at which point query refactoring comes into play. Briefly explained, we need to do the following:

- Locate Z in the `WHERE` clause, such that ***lri.X***=Z or Z=***lri.X***. Let us assume that this is w_l in Fig. 10.
- Replace all occurrences of **lri.Y** with **lri**.$a * Z +$ **lri**.b.
- Remove w_l from the `WHERE` clause, as it will not be used anymore.
- *LRF* becomes a placeholder for the subplan of Fig. 9, and is substituted accordingly. We briefly refer to it as **LRF_ab**.

Upon completing these steps we obtain the templated form of the refactored query, shown in Fig. 11. Applying this process to the query in Fig. 4, where *TropForestLRF* is the classifier, we obtain the mappings in Table 1. Combined with the above templates, we obtain both the subplan for computing values (a, b) (Fig. 12), which substitutes **LRF_ab**, and the overall refactored SNEEql query (Fig. 13). Note that all of these operators are directly optimizable through the existing infrastructure.

4.2 D3 Outlier Detection

Figure 14 shows the refactored query of the one in Fig. 6. The **STDEV** operator computes the standard deviation of the temperature tuples in the window specified in the FROM clause, which was provided when creating the extent. Note that this is just a convenient way of writing the computation of standard deviation, which can be also expressed through additional subqueries. Furthermore, this type of notation allows us to use more efficient, approximate algorithms [8], if we see fit. Recall that, during creation, the range was set to 5 and the probability threshold to 15%. The range is used to compute the closed form of the Epanechnikov integral, whereas the probability filters

Table 1. Template variables mapping for the query in Fig. 4.

Template	Mapping

LRFSource	SELECT RSTREAM temperature, humidity
	FROM TropicalForestsData[FROM NOW-20 MIN TO NOW]

LRF	TropForestLRF
X	temperature
Y	humidity
Z	AF.temperature

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

Fig. 12. Refactored Query of Fig. 4.

```
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*r.sx) 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   TropicalForestsData[FROM NOW-20 MIN TO NOW]
   ) t
) r;
```

Fig. 13. Subquery of (ab_COMP) in Fig. 12

points which are outliers, in the WHERE clause. We have marked the parameters with bold to distinguish them from the same values used as part of other expressions. We omit the query operator tree for the D3 refactored query due to space limitations.

4.3 Extensions

Query refactoring is a general technique, that can be applied to all settings with declarative query languages, e.g., relational, streaming, which is another advantage of our methodology. Nevertheless, expressing data analysis techniques as SNEEql queries is non-trivial in its own right. The fact that merging classical query processing with data mining has been an active reasearch topic for many years is indicative of its complexity. Additionally, given that intensional and extensional extents can now be interleaved, several query refactorings are possible, leaving room for additional optimizations.

Additional extensions include how to efficiently materialize such data mining models and reuse them. Clearly, this is not always possible. For example, materialization

```
SELECT RSTREAM od.temperature
FROM (
  SELECT x.temperature,
        ( 1/COUNT(y.temperature) ) * ( (1/4)^1 ) *
         SUM( (3 * 2 * 5 / q3.b1) -
         ( ( (x.temperature - y.temperature + 5) / q3.b1 )^3 -
           ( (x.temperature - y.temperature - 5) / q3.b1 )^3 ) ) as probability
  FROM (
    SELECT SQRT(5)*q1.sigma*(q2.rsize^(-1/5)) as b1
    FROM (
      SELECT STDEV(temperature) as sigma
      FROM AmazonForest[FROM NOW-20 MIN TO NOW SLIDE 20 MIN]
    ) q1,
    (
      SELECT COUNT(temperature) as rsize
      FROM AmazonForest[FROM NOW-20 MIN TO NOW SLIDE 20 MIN]
    ) q2
  ) q3,
  AmazonForest[now] x, AmazonForest[FROM NOW-20 MIN TO NOW SLIDE 20 MIN] y
  WHERE abs( (x.temperature - y.temperature) / q3.b1 ) < 1
  GROUP BY x.temperature
) od
WHERE od.probability < 0.15;
```

Fig. 14. Refactored query of D3 outlier detection algorithm.

is meaningful in a static environment like a relation database, but in a streaming environment, where the classifier is constantly updated as new data points arrive, such an alternative may be indifferent. What *is* interesting in both settings, however, is how to precompile and save the query operator tree of a data mining task, to subsequently integrate it in a new query, thus performing incremental optimizations. Such issues fall outside the scope of this paper but can serve as future research directions.

5 Experimental Evaluation

Efficiency in sensor networks is almost synomymous with energy consumption, which includes both CPU and radio energy consumption. To gain better insights, we also measured the number of transmitted messages and bytes. These two aspects are crucial in determining radio energy consumption. We evaluated all approaches using Avrora [30], a sensor network simulator, that provides accurate per-node statistics. All sources were written in nesC 2.0/TinyOS 2.x [14, 16] for MicaZ motes.

We will limit our experimental discussion to linear regression, due to lack of space, but also as a result of its widespread adoption as a data analysis method. Note, however, that we obtained similar results for the outlier detection technique (Fig. 22).

We experimented with various topologies and Table 2 summarizes some of their structural properties. The topologies are not directly comparable as their structural properties differ. For instance, an 8-node star-like network will behave differently from an 8-node chain. Because of this, extrapolating the results to other topologies should be performed with caution. Given a static topology during initialization, we construct a minimum-hop routing tree rooted at the sink, using one of several existing algorithms [5, 10, 33]. SNEE also uses the topology to find the best query routing tree, and does so during the routing stage of query optimization (Step 4 in Fig. 7(a)). As such, we have excluded the cost of building the routing tree from the graphs displayed below.

Given that we apply our method in a streaming setting, we experimented with various window, slide and acquisition intervals, and we will be using the following caption notation in the experimental figures for convenience: W:w, S:s, A:a, to signify them respectively. Varying the window and slide parameters gave similar results, so we omit these figures. Experiments were run for 300 seconds of simulated execution time. As the queries are periodical by nature, we can scale up the results for longer periods.

5.1 Handcrafted Algorithms

We implemented two baselines, both of which perform a depth-first traversal of the reverse routing tree. The sink is responsible for initiating a new tree traversal, close to the end of each acquisition interval, so that the result is reported in a timely manner.

Table 2. Structural properties of the topologies used in the experimental evaluation.

Size	Avg. Length	Max. Length	Leaf Nodes	Description
4	1.3	2	2	Tree
5	1	1	4	Star
8	2.14	3	3	Tree
9	1.75	3	4	Tree
11	1.7	2	7	Tree
12	2.18	4	6	Tree
20	1.79	4	9	Tree

In our first approach, NAÏVE, the sink probes nodes separately for data one after the other, receiving and aggregating the tuples. The second one, LC for "Local-Computation", traverses the tree and aggregates values in a postordered fashion, where each node is contacted by its parent only once within an epoch. On the contrary, SNEE optimizes queries based on a time-strict agenda. This allows nodes to contact each other at predefined times, using a push-based scheme. SNEE calculates these times by utilizing the routing tree and its optimization cost models. All approaches are graphically portrayed in Fig. 15 for a simple 4-node example. Labels denote the sequence in which node communication occurs, until we obtain the full result.

5.2 In-Network Performance

Radio Communication: Figure 16 shows the average, per-node, number of sent packets (y-axis) for the various acquisition intervals. On the x-axis, we plot the average network length, which is a better indicator of how the network is affected, than the network size. We clearly observe that in practically all cases SNEE-A's performance is superior to the handcrafted alternatives, due to its push-based scheme. All approaches perform similarly when average length is 1, due to the star-shaped topology, where each node sends directly to the sink. However, as the average network length grows higher, the difference among the techniques increases. The reason is that SNEE-A exhibits a steady performance across the topologies, which is expected as each node will send data only once during a single epoch. On the other hand, the custom techniques require the transmission of additional (control) messages to probe nodes for data.

The graphs in Fig. 17 show the average number of bytes sent in total by each node. SNEE-A and LC exhibit a steady behavior across all acquisition intervals, unlike Naïve, as a result locally aggregating the results. Even so, SNEE-A is still superior to the other two, due to the control messages that the handcrafted implementations rely on.

We can also see that the type of information to send depends on a combination of the sensing rate, the window size and the structural properties. For instance, for high sensing rates (Fig. 17(a)), it is preferrable to send aggregate information. However, as the sensing interval increases (Fig. 17(c)), fewer readings are taken during an epoch and sending the raw data becomes more efficient. Such optimizations are beneficial to a lot more queries when implemented within a query execution engine.

Radio Energy: Fig. 18 shows the average, minimum and maximum transmission energy consumption per node, for all techniques, clearly favoring SNEE-A. Given the push-based model and partial aggregations of SNEE-A, all three values are identical.

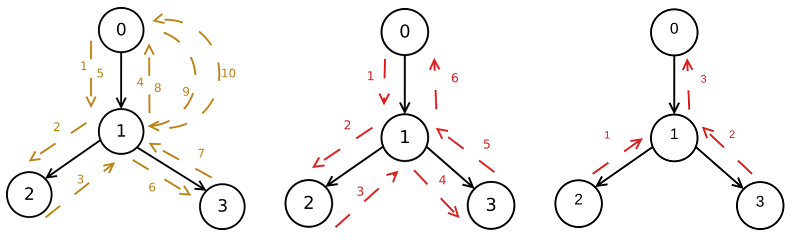


Fig. 15. Example of Linear Regression computation for Naïve, LC and SNEE-A approaches.

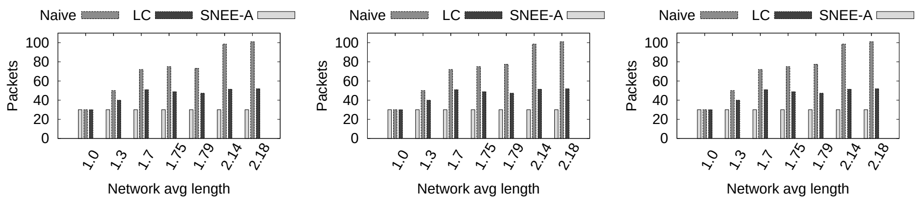

(a) W:10, S:10, A:2 (b) W:10, S:10, A:5 (c) W:10, S:10, A:10

Fig. 16. Average number of sent messages compared to the average network length for LR

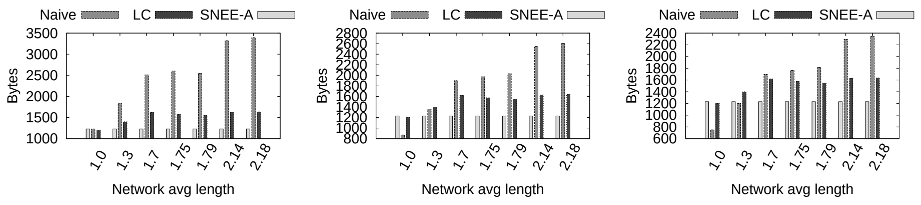

(a) W:10, S:10, A:2 (b) W:10, S:10, A:5 (c) W:10, S:10, A:10

Fig. 17. Average number of sent bytes compared to the average network length for LR

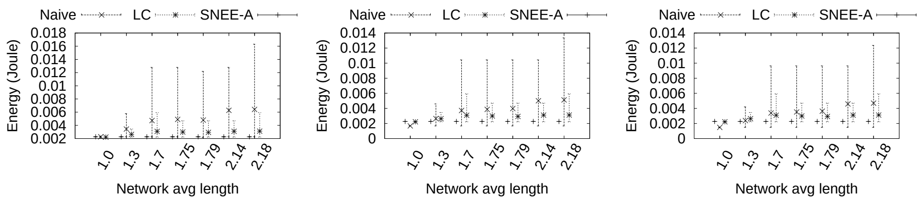

(a) W:10, S:10, A:2 (b) W:10, S:10, A:5 (c) W:10, S:10, A:10

Fig. 18. Transmission energy consumption compared to the average network length for LR

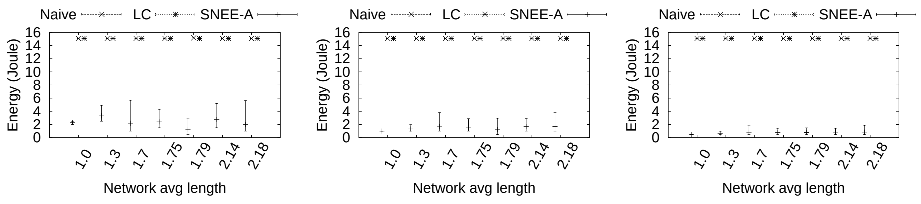
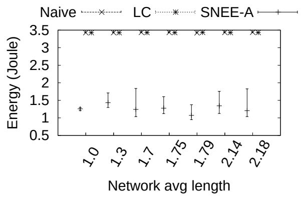

(a) W:10, S:10, A:2 (b) W:10, S:10, A:5 (c) W:10, S:10, A:10

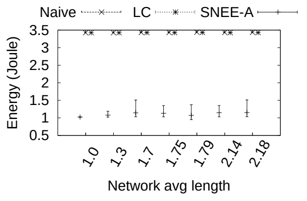
Fig. 19. Reception energy consumption compared to the average network length for LR

This is not true for the custom implementations which use control messages, the number of which is affected by the network's structural properties. Figure 18 also shows the load distribution among the nodes, with SNEE-A, distributing it almost evenly.

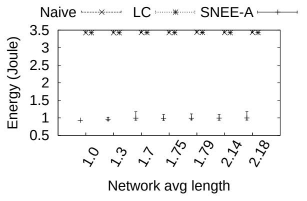
However, the factor that mostly affects radio energy consumption is the energy consumed while the radio is *on* waiting for messages, depicted in Fig. 19. The bespoke techniques have the radio switched *on* constantly, because it is impractical to manually compute when nodes will communicate. On the other hand, as a result of its agenda-



(a) W:10, S:10, A:2



(b) W:10, S:10, A:5



(c) W:10, S:10, A:10

Fig. 20. CPU energy consumption compared to the average network length for LR

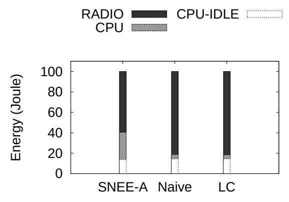
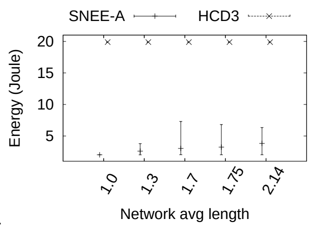
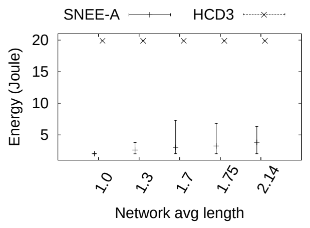


Fig. 21. Radio-CPU contribution(%) to power consumption.



(a) W:10, S:10, A:2



(b) W:10, S:10, A:10

Fig. 22. Average total energy consumption compared to the average network length for D3.

driven task execution, SNEE-A performs radio management, switching it *on* and *off* for each node independently of the others and achieves better performance.

Therefore, although we aimed for communication optimization as well, i.e., minimize transmission costs, that was not sufficient on its own. This validates our objective to utilize existing infrastructures and query engine optimizers.

CPU Usage: We finally turn our attention to the CPU consumption of the nodes. Figure 20 shows how the minimum, average and maximum CPU energy consumption is affected by the network's properties. Clearly, CPU follows a trend similar with radio reception, once again advantaging SNEE-A, for reasons we previously described.

These remarks are backed up by the graph in Fig. 21, showing the percentage with which the CPU and Radio components contribute to the total energy consumption, with an emphasis on the energy spent while the mote is idle. Even with SNEE-A's power management, the radio remains the dominant factor of consumption. However, the CPU is idle for proportionately less time compared to Naïve and LC. This implies that with SNEE-A, we make use of the resources of the node, when they are indeed required.

5.3 Ease of Programmability

One could argue that the handcrafted alternatives are inefficient because they do not use a push-based scheme. Firstly, as we already showed, most of the energy consumption is due to the radio being *idle*, which is a matter of *when* nodes communicate, rather than how they do so. Secondly, building manually such a push-based approach or even computing when nodes should communicate is impractical, as it involves accurate computation of processing and communication times for each node.

On the other hand, building a system or module that provides these accurate timings basically duplicates what the query engine already does. It is even less practical to rework this component when moving between settings (e.g., relational, streaming, distributed etc.). This brings us to another benefit of query refactoring: the time taken to write – and debug – the handcrafted code. For instance, SNEE-A has a clear advantage against the handcrafted alternatives, as i) the developer uses high-level (declarative) languages instead of low-level, and ii) the system autogenerates and deploys code for all nodes in the network, optimized for the requested goal. Finally, note that if we change the optimization goal, the bespoke techniques must be re-implemented, whereas for SNEE-A (and similar execution engines) it is a single parameter.

6 Conclusions and Future Work

In this paper we tackled the problem of integrating data analysis techniques with classic query processing through declarative languages. We proposed a query language extension, that incorporates major data mining categories, e.g. classification, outlier detection etc. We expressed data analysis tasks as intensional extents, and took advantage of existing query optimizers, with minimal modifications to the query execution stack. This also gives a natural interpretation in relational algebra terms. We implemented the above concepts in our prototype system SNEE-A, and compared its performance against handcrafted implementations.

We plan to incorporate additional techniques, to identify the limits of our approach and ways to overcome them using the framework we have presented herein. Interesting future directions include materialization of data mining models and support for incremental optimization.

Acknowledgements: This work has been funded by EU project SemSorGrid4Env (FP7-223913), INSIGHT and THALIS-DISFER, the UK EPSRC WINES Programme and a Heraclitus II Fellowship.

References

1. D. J. Abadi, Y. Ahmad, M. Balazinska, and U. Ç. et al. The design of the borealis stream processing engine. CIDR, pages 277–289, January 2005.
2. D. J. Abadi, D. Carney, U. Çetintemel, and M. e. a. Cherniack. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12:120–139, August 2003.
3. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
4. C. Aggarwal, editor. *Data Streams – Models and Algorithms*. Springer, 2007.
5. P. Andreou, D. Zeinalipour-Yazti, P. K. Chrysanthis, and G. Samaras. Workload-aware query routing trees in wireless sensor networks. MDM, pages 189–196, 2008.
6. A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: the stanford stream data manager (demonstration description). SIGMOD, 2003.
7. A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *VLDB J.*, 15:121–142, June 2006.
8. B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan. Maintaining variance and k-medians over data stream windows. PODS, pages 234–243, 2003.
9. C. Y. Brenninkmeijer, I. Galpin, A. A. Fernandes, and N. W. Paton. A semantics for a query language over sensors, streams and relations. BNCOD, pages 87–99, 2008.
10. G. Chatzimilioudis, A. Cuzzocrea, and D. Gunopulos. Optimizing query routing trees in wireless sensor networks. In *ICTAI (2)*, pages 315–322, 2010.
11. S. Chaudhuri. Data mining and database systems: Where is the intersection? *Data Engineering Bulletin*, 21, 1998.

12. S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *TODS*, 1999.
13. I. Galpin, C. Y. A. Brenninkmeijer, A. J. G. Gray, F. Jabeen, A. A. A. Fernandes, and N. W. Paton. Snee: a query processor for wireless sensor networks. *Distributed and Parallel Databases*, 29(1–2):31–85, 2011.
14. D. Gay, P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer, and D. E. Culler. The nesc language: A holistic approach to networked embedded systems. PLDI, pages 1–11, 2003.
15. J. Han, Y. Fu, W. Wang, K. Koperski, and O. Zaiane. Dmql: A data mining query language for relational databases. In *Proc. of the SIGMOD workshop on Research issues on Data Mining and knowledge discovery*, pages 27–33, 1996.
16. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. ASPLOS, pages 93–104, 2000.
17. T. Imieliński and A. Virmani. Msql: A query language for database mining. *Data Mining and Knowledge Discovery*, 3(4):373–408, 1999.
18. 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. *Distributed and Parallel Databases*, 29(1–2):151–183, 2011.
19. D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
20. S. Krishnamurthy, S. Chandrasekaran, O. Cooper, and A. D. et al. Telegraphcq: An architectural status report. *IEEE Data Engineering Bulletin*, 26(1):11–18, March 2003.
21. C. Luo, H. Thakkar, H. Wang, and C. Zaniolo. A native extension of sql for mining data streams. SIGMOD, pages 873–875, 2005.
22. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *TODS*, 30(1):122–173, 2005.
23. G. Mitchell, S. B. Zdonik, and U. Dayal. Object-oriented query optimization: What's the problem? Technical report, Providence, RI, USA, 1991.
24. A. Netz, S. Chaudhuri, J. Bernhardt, and U. M. Fayyad. Integration of data mining with database technology. In *VLDB*, pages 719–722, 2000.
25. C. Ordonez. Integrating k-means clustering with a relational dbms using sql. *IEEE TKDE*, 18:188–201, February 2006.
26. S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. *Data Mining and Knowledge Discovery*, 4(2):89–125, 2000.
27. K.-U. Sattler and O. Dunemann. Sql database primitives for decision tree classifiers. CIKM, pages 379–386, 2001.
28. S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos. Online outlier detection in sensor data using non-parametric models. VLDB, pages 187–198, 2006.
29. H. Thakkar, N. Laptev, H. Mousavi, B. Mozafari, V. Russo, and C. Zaniolo. Smm: A data stream management system for knowledge discovery. ICDE, pages 757–768, April 2011.
30. B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. IPSN, 2005.
31. G. Valkanas, D. Gunopulos, I. Galpin, A. J. G. Gray, and A. A. A. Fernandes. Extending query languages for in-network query processing. MobiDE, pages 34–41, 2011.
32. H. Wang and C. Zaniolo. Atlas: A native extension of sql for data mining. SDM, 2003.
33. Y. Yang, H.-H. Wu, and H.-H. Chen. SHORT: shortest hop routing tree for wireless sensor networks. *International Journal of Sensor Networks*, 2:368–374, July 2007.
34. Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31(3):9–18, 2002.