

SemSorGrid4Env

FP7-223913



Deliverable

D2.2v2

Implementation and Deployment of Data Management and Analysis, and the Query Processing Components – Phase II

Alasdair J G Gray, Ixent Galpin, Alvaro A A Fernandes, and Norman W Paton
University of Manchester

Alexis Kotsifakos, George Valkanas, and Dimitrios Gunopulos
National and Kapodistrian University of Athens

February 23, 2011



Executive Summary

One of the features of the SemSorGrid4Env architecture is the ability for users to pose declarative queries in order to receive the data that they require. That is, the user need not worry about where, when, or how data is gathered, they need only state their data needs. Work Package 2 (WP2) aims to support this vision by providing an implementation of the streaming data service through which users can pose declarative queries in SNEEqL. It is also intended to enhance the querying capabilities through data analysis functionality being made available directly in the query.

Sensor networks fall into two different categories. The first type of sensor network has fixed functionality with a stream of data being generated at a fixed rate. In order to query this stream of data, the stream must be processed outwith of the sensor network, called *out-of-network* evaluation. The second type of sensor network has the ability to respond to user requests for data by performing *in-network* query evaluation. This is where the functionality of the sensor network, i.e. when and where readings are taken, is controlled by the queries, and requires the placement of query processing code directly in the sensor network.

The SNEEqL query language is capable of querying streams generated by both kinds of sensor networks, as well as stored relational data sources. Prior to this version of the deliverable, there were independent evaluation engines for in-network and out-of-network evaluation over streaming data only. The prototype developed for this deliverable provides a common interface and compilation stack for queries over both kinds of sensor network as well as streaming and stored sources. The query compiler/optimiser generates plans for the out-of-network evaluation engine or generates TinyOS/nesc code for in-network evaluation. The SNEEqL query language and the SNEE query evaluation engine have also been extended to enable the declarative definition of data analysis operators over sensor network sources.

In this deliverable we present details of the implementation of the SNEE query evaluation engine for execution over streaming and stored sources, the mechanisms for expressing data analysis techniques as queries and their evaluation, together with the interfaces by which it will be made available as a Web service.

The major changes between this and the previous version of this document are as follows:

Section 1: Generally updated to reflect the new version of the prototype.

- System Objectives updated to those stated for the second release of the sensor data management platform, including data analysis functionality.
- System Description updated to reflect the position in the architecture and the new functionality offered.

Section 2: Updated to reflect the current state of the code and the use of the Maven build system.

Section 3: Updated to reflect that the sensor network data management platform can be used over a variety of data sources.

Section 4: Updated to show how to use a sample client in conjunction with the sensor network data management platform in a variety of configurations.

Section 5: Added section describing the testing strategy followed during the development of the sensor network data management platform.



Note on Sources and Original Contributions

The SemSorGrid4Env consortium is an inter-disciplinary team, and in order to make deliverables self-contained and comprehensible to all partners, some deliverables thus necessarily include state-of-the-art surveys and associated critical assessment. Where there is no advantage to recreating such materials from first principles, partners follow standard scientific practice and occasionally make use of their own pre-existing intellectual property in such sections. In the interests of transparency, we here identify the main sources of such pre-existing materials in this deliverable:

- Section 1.2.1 is based on material in [GGF⁺10b];
- Section 1.2.2 draws on material developed in [GGF⁺09, GBG⁺10, GBJ⁺09, BGFP08];
- Section 1.2.3 draws on material developed in [GGF⁺09, GGF⁺10a].



Document Information

Contract Number	FP7-223913	Acronym	SemSorGrid4Env
Full title	SemSorGrid4Env: Semantic Sensor Grids for Rapid Application Development for Environmental Management		
Project URL	www.sensorsgrid4env.eu		
Document URL	www.sensorsgrid4env.eu/files/deliverables/wp2/D2.2v2.pdf		
EU Project Officer	Antonios Barbas		

Deliverable	Number	D2.2v2	Name	Implementation and Deployment of Data Management and Analysis, and the Query Processing Components – Phase II					
Task	Number	T2.3	Name	Implement and deploy the SemSorGrid4Env SNDM platform					
Work package	Number			WP2					
Date of delivery	Contract	28 February 2011	Actual	28 February 2011					
Code name	D2.2v2		Status	draft <input type="checkbox"/>	final <input checked="" type="checkbox"/>				
Nature	Prototype <input checked="" type="checkbox"/> Report <input type="checkbox"/> Specification <input type="checkbox"/> Tool <input type="checkbox"/> Other <input type="checkbox"/>								
Distribution Type	Public <input checked="" type="checkbox"/> Restricted <input type="checkbox"/> Consortium <input type="checkbox"/>								
Authoring Partner	UNIMAN								
QA Partner	UPM								
Contact Person	Alasdair J G Gray			Email	A.Gray@cs.man.ac.uk	Phone	+44 161 275 6132	Fax	+44 161 275 6204
Abstract (for dissemination)	<p>Streams of data generated by sensor networks can be queried in one of two ways. The first involves sensors periodically take readings and streaming these out of the network for out-of-network evaluation by a stream processing engine. The second involves placing query processing capabilities into the network for in-network evaluation. Examples of both types of sensor network are found in the use cases of the SemSorGrid4Env project. The SNEEqL query language is capable of querying streams generated by both kinds of sensor networks, as well as relational stored data sources, although previously only the in-network query evaluator had been implemented.</p> <p>The prototype developed for this deliverable provides a SNEE query evaluation engine for SNEEqL queries posed over both kinds of sensor networks as well as relational stored data sources. The query language and evaluation engine have also been enriched to enable the expression and execution of some data analysis techniques as queries. We present details of the implementation, together with the interfaces by which it is made available as a Web service.</p>								
Keywords	Data Management, Query Processing, Data Analysis								

Version log/Date	Change	Author
0.1 / 7 December 2009	Initial Draft	A. Gray
0.2 / 8 December 2009	Revised text	I. Galpin
0.3 / 10 December 2009	Revised system description. Added details of source code and data source. Included quick start guide. Included WSDL.	A. Gray
0.4 / 10 December 2009	Revised text	I. Galpin
0.5 / 17 December 2009	Revisions responding to QA comments.	A. Gray
0.6 / 18 December 2009	Minor revisions. Updated Java API	A. Gray
1.0 / 18 December 2009	Finalised document for submission.	A. Gray
1.1 / 11 February 2011	Initial revised version	I. Galpin, A. Gray, and G. Valkanas
1.2 / 23 February 2011	Response to QA comments	I. Galpin, A. Gray, and G. Valkanas

Project Information

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number FP7-223913. The Beneficiaries in this project are:

Partner	Acronym	Contact
Universidad Politécnica de Madrid (Coordinator)	UPM 	Prof. Dr. Asunción Gómez-Pérez Facultad de Informática Departamento de Inteligencia Artificial Campus de Montegancedo, sn Boadilla del Monte 28660 Spain #@ asun@fi.upm.es #t +34-91 336-7439, #f +34-91 352-4819
The University of Manchester	UNIMAN 	Prof. Norman Paton Department of Computer Science The University of Manchester Oxford Road Manchester, M13 9PL, United Kingdom #@ carole@cs.man.ac.uk #t +44-161-275-61 95, # +44-161-275 62 04
National and Kapodistrian University of Athens	NKUA 	Prof. Manolis Koubarakis University Campus, Ilissia Athina GR-15784 Greece #@ koubarak@di.uoa.gr #t +30 210 7275213, #f +30 210 7275214
University of Southampton	SOTON 	Dr Kirk Martinez University Road Southampton SO17 1BJ United Kingdom #@ dder@ecs.soton.ac.uk #t +44 23 80592418, #f +44 23 80595499
Deimos Space, S.L.	DMS 	Mr. Agustín Izquierdo Ronda de Poniente 19, Edif. Fiteni VI, P 2, 2º Tres Cantos, Madrid – 28760 Spain #@agustin.izquierdo@deimos-space.com #t +34-91-8063450, #f +34-91-806-34-51
EMU Limited	EMU 	Dr. Bruce Tomlinson Mill Court, The Sawmills, Durley number 1 Southampton, SO32 2EJ, United Kingdom #@ bruce.tomlinson@emulimited.com #t +44 1489 860050, #f +44 1489 860051
TechIdeas Asesores Tecnológicos, S.L.	TI 	Dr. Jesús E. Gabaldón C/ Marie Curie 8-14 08042 Barcelona, Spain #@ jesus.gabaldon@techideas.es #t +34.93.291.77.27, #f +34.93.291.76.00



Contents

1	Introduction	1
1.1	System Objectives	1
1.2	System Description	1
1.2.1	SemSorGrid4Env Data Tier Services	2
1.2.2	SNEE Query Processing Engine	3
1.2.3	Support for Data Analysis	5
2	Source Code	8
2.1	Code Structure	8
2.2	External Application Dependencies	8
2.3	Configuration Files	9
3	Source Data	10
3.1	Streaming Data Service	10
3.2	Stored Data Service	10
3.3	Local Sensor Network	10
4	Installation and Execution	11
4.1	Setting Up SNEE Environment	11
4.1.1	Development Tools	11
4.1.2	Obtaining SNEE	11
4.1.3	Compiling the Sample Client	13
4.2	Running the SNEE Sample Client	14
4.2.1	Running an Out-of-Network Query	15
4.2.2	Running an In-Network Query	15
4.2.3	Running Data Analyses	16
5	Test Strategy	19
5.1	Unit Testing	19
5.2	Integration Testing	19
A	Interface Operations for Streaming Data	20
A.1	Query Interface	20
A.2	Data Access Interface	21
A.3	Subscription Interface	22
B	Java APIs	24
B.1	SNEE API	24
B.2	ResultStore API	24
C	Sample SNEE Client	26



List of Figures

1.1	The interfaces offered by the SemSorGrid4Env data tier services.	2
1.2	Component diagram of the SNEE query processing engine.	3
1.3	SNEE query compilation stack.	4
1.4	Example SNEEqI schema and query.	4
1.5	PAF for the example query.	5
1.6	SNEE query compilation stack with support for data analysis.	6



List of Tables

5.1	Unit test coverage	19
A.1	The Query Interface.	20
A.2	Pull Stream Service Operations.	21
A.3	Push Stream Service Operations.	22
B.1	SNEE Java API.	24
B.2	SNEE ResultStore API.	24



1. Introduction

The SemSorGrid4Env Work Package 2 (WP2) objective is to design and implement a data management middleware for the SemSorGrid4Env architecture [GGF⁺10b]. The implementation should be capable of providing query-based access to acquisitional and non-acquisitional data streams and stored data sources (e.g. databases), as well as data analysis functionality. The resulting implementation will support the view-based semantic data integration being developed in WP4 as well as direct access by applications, mashups and application services.

Deliverable D2.1 [GGF⁺09] identified the functional requirements placed on the data management middleware by the other work packages of the SemSorGrid4Env project and produced a development plan for the data management implementation. This deliverable provides details of the implemented query processing platform over sensor network, streaming, and stored sources, as well as the extensions for providing data analysis functionality. Deliverable D1.4v2 (April 2011) will provide details of how the query processing platform is used as a service by other parts of the SemSorGrid4Env architecture.

The remainder of this chapter provides an overview of the SNEE query processing engine and the interfaces by which it is exposed as a service in the SemSorGrid4Env architecture, as well as the extensions to both the query language and execution platform for incorporating the data analysis techniques (DATs). Chapter 2 provides specific details of the code of the prototype released with this deliverable, while details of the data sources used are provided in Chapter 3. Chapter 4 provides instructions for deploying the prototype, and Chapter 5 presents the testing strategy used during the development of the software.

1.1 System Objectives

The objectives for the prototype sensor network data management platform developed for the SemSorGrid4Env project are to:

- Provide a complete specification of the WSDL to define the query, data access, subscription, and subscription manager interfaces as defined in the SemSorGrid4Env architecture [GGF⁺10b];
- Implement a SemSorGrid4Env streaming data service capable of executing SNEEqI queries over a variety of data sources including in-network query processing, streaming and stored data sources;
- Enable declarative specification of data analysis techniques as extensions to the SNEEqI query language and enable their execution by the SNEE query engine.

See Deliverable D2.1 [GGF⁺09] for full details of the motivation for this work.

1.2 System Description

The phase 1 release of the sensor network data management platform (Deliverable D2.2v1) focused on service-oriented access to out-of-network SNEEqI query processing over streams of data. This version of the sensor network data management platform provides service-oriented access to SNEEqI query processing over data streams, generated either by an external data source or a locally connected sensor network capable of in-network query processing, and stored data services. This

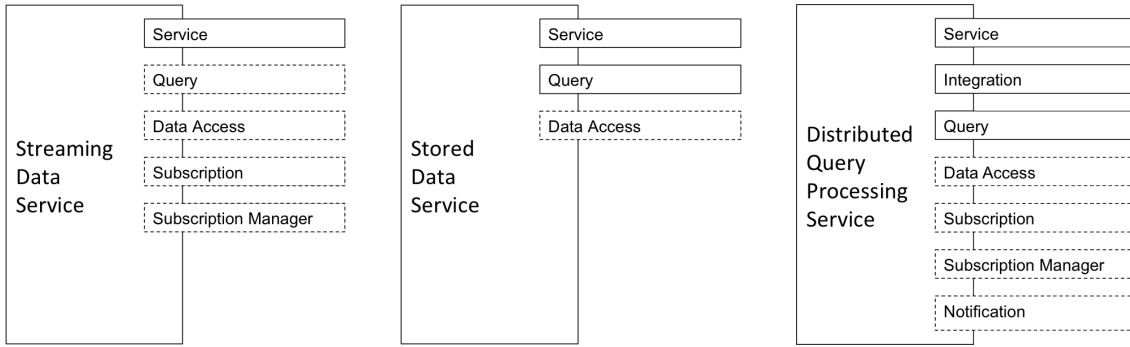


Figure 1.1: The interfaces offered by the SemSorGrid4Env data tier services.

version is also capable of accepting declarative descriptions of data analysis needs and evaluating these over a streaming data source. In this section, we describe the role of the SNEE Streaming Data Service within the SemSorGrid4Env architecture [GGF⁺10b]. We then describe the SNEEq query processing capabilities that will be made available through the Web service. Finally, we describe the query language and execution engine extensions for incorporating data analysis techniques.

1.2.1 SemSorGrid4Env Data Tier Services

The SemSorGrid4Env architecture [GGF⁺10b] defines three data tier services: (1) Streaming Data Service for accessing data streams; (2) Stored Data Service for accessing stored data (e.g. database); (3) Distributed Query Processing service for posing queries over more than one data source. Figure 1.1 shows the three types of data service and the interfaces that they offer. Note that interfaces shown with a dotted line are optional, although it is expected that a streaming data service will offer at least one of **data access** and **subscription**. The **service** interface is common to all SemSorGrid4Env Web services and provides a mechanism by which to discover the specific features and capabilities of the service. More details can be found in Deliverable D1.3v2 [GGF⁺10b].

The core of both the Streaming Data Service and Distributed Query Processing Service for streaming data is offered by the **query**, **data access**, and **subscription** interfaces. The **query** interface provides the mechanism by which a query can be passed to the service. The **data access** interface provides the mechanism by which streams can be pulled, i.e. periodically polling for data, from the service, and the **subscription** interface provides the mechanisms by which a data stream can be pushed, i.e. data sent as it is generated, from the data service to a client. As such, it is expected that the SNEE query evaluation engine can provide an implementation of both types of service. In the case of the distributed query processing service, SNEE is able it to access both streaming and stored data services as external data sources (see Sections 1.2.2 and Chapter 3).

The SemSorGrid4Env architecture document D1.3v2 [GGF⁺10b] defines the **query** and **data access** interfaces to provide the operations of the Web services data access and integration (WS-DAI) standard [AAK⁺06]. However, the WS-DAI standard does not fully specify the operations, leaving these to be defined by the relevant realisations. For example, there is a relational realisation of the WS-DAI standard which provides SQL specific operations for stored data access [ACK⁺06], which is used by the stored data service. As such, suitable extensions of the WS-DAI interfaces and operations for streaming data need to be specified.

The full specification of the **query**, **data access**, **subscription**, and **subscription manager** interfaces can be found in [GGFP09], which provides the data types, parameters, and operations for querying and accessing data streams, using the WS-DAI Core operations and extending them where necessary. A summary of the operations offered by each interface are provided in Tables A.1, A.2, and A.3 of Appendix A. The document [GGFP09] also provides the full WSDL specification of the service interface. The interfaces can again be further specialised for specific query languages, e.g. SNEEq.

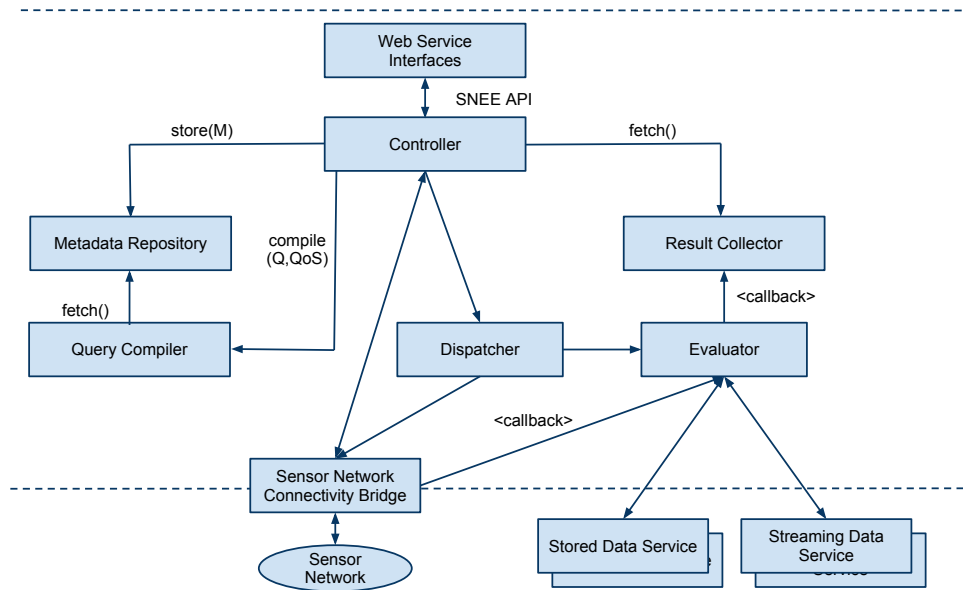


Figure 1.2: Component diagram of the SNEE query processing engine.

1.2.2 SNEE Query Processing Engine

The query processing functionality of the Streaming Data Service and the Distributed Query Processing Service described in the previous section are provided by the prototype implementation of the SNEE query processing engine. The SNEE query processing engine is capable of evaluating queries either over a local sensor network source using in-network query processing techniques [GBG⁺10] or over a set of external streaming and stored data services using data stream query processing techniques. The main components of the SNEE query processing engine are shown in Figure 1.2 and are explained below. Note, the dashed lines indicate the extent of one deployment of the Streaming Data Service which is configured to use any of the data sources at the bottom of the diagram (see Section 4).

The web service interfaces support external clients interacting with the SNEE query engine. The interfaces can be used for posing SNEEql queries and retrieving the results produced by the execution of long-running queries. The web service interface transforms service requests into suitable method calls through the SNEE API (Table B.1).

Upon receiving a new SNEEql query, the controller first sends it to the compiler/optimiser to check that it is a valid query for the available data sources and to generate a query execution plan (QEP). The compilation stack is shown in Figure 1.3. The compiler first parses and type-checks the query against the schema, and performs logical optimisation to generate a logical-algebraic form (LAF). The source allocator then partitions the LAF for evaluation by the different sources involved in the query. The appropriate source planner for each source is then used to generate the QEP for the query, which is returned to the controller.

Once the controller receives the QEP for the query, it passes this on to the dispatcher to manage the query execution. The dispatcher distributes each partition of the QEP to the sensor network connectivity bridge (SNCB) for execution in the sensor network, or to the evaluator for execution over a set of external data sources, as appropriate.

For queries that involve in-network query execution, the SNCB manages the evaluation of the query. It first generates appropriate nesC code images [GLvB⁺03] for each node in the sensor network based on their role in the QEP and the physical hardware. The SNCB then distributes

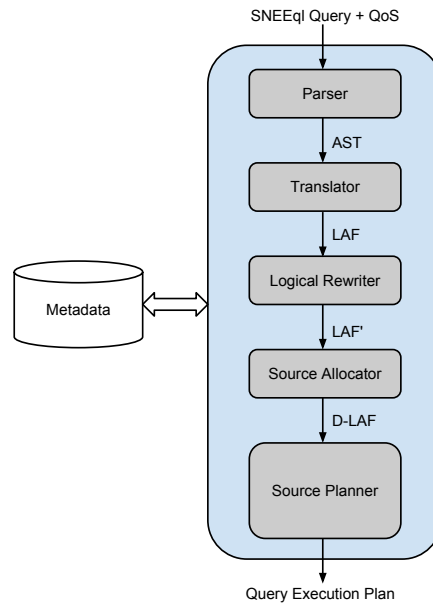


Figure 1.3: SNEE query compilation stack.

Schema	<code>wind:pushed (id:int, locx:int, locy:int, ts:time, speed:float, direction:float)</code>
Query	<code>SELECT RSTREAM avg(w.speed) FROM wind[FROM NOW - 1 HOUR TO NOW] w;</code>

Figure 1.4: Example SNEEql schema and query.

the code images using an over-the-air programmer based on Deluge [HC04]. The query is then started, with all nodes in a synchronised state, and results flow back through the SNCB to the evaluator.

The evaluator executes the parts of the QEP that involve external data services and the delivery of query answers to the result store. The evaluator instantiates the operators in the QEP and passes sub-queries down to the external service, where appropriate. The query is then continuously evaluated with results being passed onto the result collector for the query.

Query results can be gathered in one of two ways. The first relies on the client requesting data. When the controller receives a request for query answers, it consults the result collector for the query and sends back the appropriate answer tuples for the request. The second relies on the client subscribing to the answer stream for the query. The controller then sends notification messages to the client as query answers are generated.

An example SNEEql schema and query are shown in Figure 1.4. The schema describes a stream produced by a set of wind sensors. The query returns the average wind speed over the last hour every time a new tuple arrives. The resulting *physical-algebraic form* (PAF) is shown in Figure 1.5. Note the use of a time window, used to convert a stream into a window, over which the average is computed. A time window has three parameters, the *startTime*, *endTime* and *slide*, and returns the window containing tuples from the stream from *startTime* to *endTime* inclusive, every *slide* time units (in this case the default slide is used). Further details about the syntax and semantics of the SNEEql language is given by Brenninkmeijer *et al.* [BGFP08].

The SNEE query engine makes a Java API (Table B.1) available for other applications (e.g. the Web service implementation) to add and remove service sources, add and remove queries, poll for results from queries, or subscribe to answer streams.

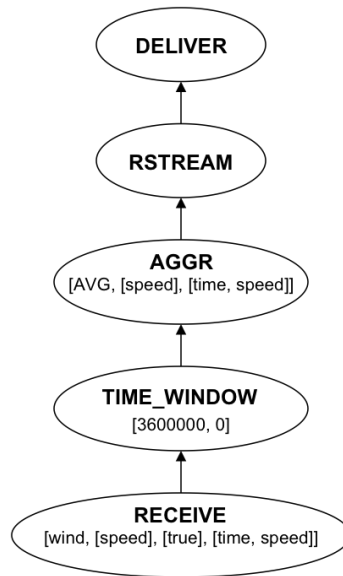


Figure 1.5: PAF for the example query.

1.2.3 Support for Data Analysis

As mentioned previously, one of our objectives is to seamlessly provide data analysis functionality within the SNEEqL language. By data analysis (or mining) we mean techniques such as clustering, classification, outlier detection etc. Our development plan [GGF⁺09] contains a comprehensive list of various DATs that may be developed and used in the context of sensor networks. Our motivation for this is to enable the composability of classical query algebraic operators (such as *project*, *join* etc.) with such DATs.

We have incorporated support for DATs using a pre-processing step, which we refer to as *query refactoring*, that precedes all stages shown in Figure 1.3. In essence, query refactoring involves identifying that a DAT is being used within the query, and transforming the query into an equivalent query using, as much as possible, operators available in the SNEEqL algebra. Note that, depending on the DAT encountered, the query is refactored to a different tree of SNEEqL operators. If it is not possible to represent certain functionality using SNEEqL operators, user-defined functions (UDFs) may be used. Figure 1.6 portrays the query stack for the enhanced version of SNEE. A software module, the `Refactorer`, has been added at the beginning of the execution stack, which provides the query refactoring functionality.

The SNEE stack with support for data analysis accepts a SNEEqL query and logical schema with minimal extensions to support data analysis functionality. Once the query is refactored into an equivalent query without these extensions, its compilation steps are the same as for the vanilla version. We now illustrate how a query that employs a linear regression classifier is refactored using this technique. To date, we have also implemented the D3 outlier detection algorithm [SPP⁺06].

We assume a logical schema as follows:

```
forest (ts:time, moisture:float, temperature:float)
forestLRF (ts:time, moisture:float, temperature:float)
forestMoisture (ts:time, moisture:float)
```

`forest` is a stream that provides data used to build the linear regression function. The `forestLRF` stream is intensional (i.e., it is not necessary for all its tuples to be acquired or stored), and uses the linear regression function to predict a value for `temperature` for the values in the domain

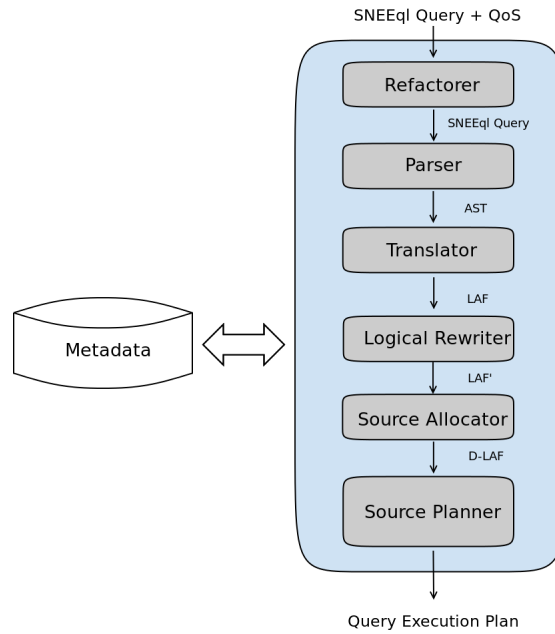


Figure 1.6: SNEE query compilation stack with support for data analysis.

of `moisture`. In other words, the `moisture` attribute is the input attribute, and `temperature` attribute the return value, of the linear regression function. `forestMoisture` is a stream that only consists of moisture readings, for which the linear regression function will be applied in order to predict values of temperature. This is done by joining `forestLRF` and `forestMoisture`, which obtains the predicted temperature based on the moisture values of the stream `forestMoisture`, as shown in the following SNEEq query:

```
SELECT RSTREAM fm.moisture, fr.temperature
FROM forestMoisture[NOW] fm, forestLRF fr
WHERE fm.moisture = fr.moisture
```

For the above query, the nested query that is the output of the query Refactorer is shown in two parts, due to space reasons. The inner query, which computes coefficients a and b for a linear function $y = ax + b$ using a refactoring technique based on the Least Squares Method, described in detail in [GGF⁺10a], is as follows:

```
SELECT ( S.n * S.s_xy - S.s_y * S.s_x ) / ( S.n * S.s_xx - S.s_x * S.s_x ) as a,
       ( S.s_y * S.s_xx - S.s_x * S.s_xy ) / ( S.n * S.s_xx - S.s_x * S.s_x ) as b
FROM (
  SELECT COUNT(*) as n,
         SUM( S.moisture ) as s_x,
         SUM( S.temperature ) as s_y,
         SUM( S.moisture * S.moisture ) as s_xx,
         SUM( S.moisture * S.temperature ) as s_xy
  FROM (
    SELECT moisture, temperature
    FROM forest [FROM NOW-1 HOUR TO NOW SLIDE 1 HOUR]
  ) S
) S
```

Note that the linear regression function is built based on the last hour's worth of data from the `forest` stream. The outer query is:



```
SELECT RSTREAM fm.moisture, rf.a * fm.moisture + rf.b  
FROM forestMoisture[NOW] fm, (INNER_QUERY) rf
```

With this approach, the data analysis technique is expressed using SNEE operators, avoiding the need for UDFs, which often are viewed as ‘black boxes’ in the eyes of the query optimizer. This is beneficial as it enables the optimizer to make more effective optimization decisions during QEP generation.



2. Source Code

The source code for the SNEE query processing engine is available from the trunk of the code repository available at <http://code.google.com/p/snee/source/checkout>. The version with the extensions to support data analysis techniques is available at <https://snee.googlecode.com/svn/branches/data-analysis>.

2.1 Code Structure

The project uses Maven version 2.1 [Apa11] to compile source code and manage project dependencies. The code is packaged as `uk.ac.manchester.cs.snee.*`. The code is organised into the following Maven modules:

snee-api: contains interfaces and Java beans to enable external applications to interact with SNEE.

data-source: contains packages for interacting with external web services and consists of the following sub-modules:

data-source-core: contains common classes for interacting with external sources.

data-source-pull-stream: contains classes for interacting with external streaming data services.

data-source-wsdair: contains classes for interacting with WS-DAIR stored data services.

snee-core: contains implementations of core elements of SNEE.

sncb: contains the implementation of the sensor network connectivity bridge for interacting with a local sensor network.

compiler: contains the implementation of the compiler and evaluation engine.

snee-dat: contains the implementation of the refactorer module to support data analysis techniques (data analysis branch only).

2.2 External Application Dependencies

The SNEE query engine has the following external dependencies:

- `log4j.log4j.jar` version 1.2.15: Provides log message support;
- `org.apache.cxf.cxf-rt-frontend-jaxws.jar` version 2.2.8: Provides support for accessing external web services;
- `org.apache.cxf.cxf-rt-transport-http.jar` version 2.2.8: Provides support for accessing external web services;
- `net.tinyos.tinyos.jar` version 2.1: Provides the operating system for a local sensor network, only required for using in-network query processing;
- `antlr.antlr.jar` version 2.7.5: Provides support for parsing the query language.

There is an optional dependency on GraphViz [Gra09]. GraphViz can be used to generate diagrams representing the query plans generated. To enable this feature, the `compiler.generate_graphs` parameter must be set to true in the `snee.properties` file.



2.3 Configuration Files

The following describe the configuration files for SNEE. Except where declared as required, all parameters are optional. See Chapter 4 for examples of different configurations.

snee.properties contains settings for configuring the query engine.

- compiler.generate_graphs (boolean):** controls whether query plans are saved to disk.
- compiler.convert_graphs (boolean):** controls whether generated graphs are converted to graphics files.
- graphviz.exe (String):** declares the path for the GraphViz executable, required if **compiler.convert_graphs** is set to true.
- compiler.output_root_dir (String):** declares where generated output files should be stored.
- types_file (String) (required):** declares the location of the file that defines the type system.
- units_file (String) (required):** declared the location of the file that defines the units.
- results.history_size.tuples (int):** configures the size of storage to be used for query answers
- logical_schema (String):** declares the location of the file that defines the logical schema. The logical schema defines the streams and relations that may be queried. It is also used to define DATs (e.g., a linear regression function) that may be queried.
- physical_schema (String):** declares the location of the file that defines the physical schema, required if a logical schema is declared.
- cost_parameters_file (String):** declares the location of the file that defines the cost parameters for a sensor network, required for a local sensor network source.

log4j.properties contains settings that affect the logging behaviour of the system when it is run.

Note, for configuring SNEE as an in-network source the following files must be declared in the **snee.properties** file: logical schema, physical schema, and cost parameters. For using SNEE as a distributed query processing engine, the logical schema is read in from the external source. If two sources have the same extent declared, it will only be read from the first data source.



3. Source Data

The SNEE query engine can be configured to operate over a variety of data sources. The details of the types of data sources are provided in the following sections. Note that the data analysis functionality is independent of the types of data sources that will be used.

3.1 Streaming Data Service

The SNEE query engine can receive data from an external streaming data service that offers the Pull-Stream realisation of the Data Access interface. An example of such a data source is the Channel Coastal Observatory (CCO) streaming data service [CCO09] that has been developed to provide access to the CCO data feeds.

3.2 Stored Data Service

The SNEE query engine can query data from an external stored data service that offers the WS-DAIR realisation [ACK⁺06] of the Data Access interface. An example of such a data source is the Channel Coastal Observatory (CCO) stored data service [CCO09] that has been developed to provide access to the stored data held about the CCO data feeds.

3.3 Local Sensor Network

The SNEE query engine can query readings obtained from the sensors of the nodes that comprise a wireless sensor network via the SNCB. The code generator in the SNCB generates a program for execution on each node. Depending on the SNEEqI query, the program on each node requests readings from the Analog-to-Digital Converter (ADC) software component that wraps the appropriate hardware sensor. For example, if the SNEEqI query requests readings from a light sensor, the program will obtain readings from the ADC component corresponding to the light sensor. Note that, currently, the only mote hardware supported by the SNCB are TelosB motes manufactured by Crossbow¹ (and equivalent hardware).

¹This hardware has the following specification: CPU = MSP430 8MHz, RAM = 10 kB, Program Memory = 48 kB, Data Flash = 1 MB, Radio = CC2420. Detailed specifications can be found at http://www.willow.co.uk/TelosB_Datasheet.pdf.



4. Installation and Execution

This chapter provides details of installing and executing the SNEE query engine in a variety of configurations using a supplied sample Java client application. The sample client shows how a client application can use SNEE and is not intended as a robust client. A basic knowledge of compiling and running a Java application are assumed.

4.1 Setting Up SNEE Environment

4.1.1 Development Tools

It is assumed that the following tools have been installed and configured:

- Java Development Kit (version of 1.6): <http://www.java.com/>
- Graphviz (optional, but useful for debugging): <http://www.graphviz.org/>

To enable the use of GraphViz, the following settings need to be specified in the `snee.properties` file:

- `compiler.generate_graphs = true`
- `compiler.convert_graphs = true`
- `graphviz.exe = /usr/local/bin/dot1`

For running In-network queries, it is also necessary to install:

- TinyOS 2.1.1: <http://www.tinyos.net/>

Full details on the configuration of these tools, and platform-specific installation issues, are updated at <http://code.google.com/p/snee/wiki/SettingUpDeveloperEnvironment> as they become known. The wiki page contains full details of setting up a SNEE developers environment.

4.1.2 Obtaining SNEE

SNEE has been made available as a download from <http://code.google.com/p/snee/downloads/>. It is also available for inclusion as a dependency in a Maven project from the TechIdeas repository².

The most recent release (currently SNEE-1.5.3) should be downloaded and extracted. A new working directory containing all the jar dependencies, the source code for a sample client (listed in Appendix C), and some initial configuration files are contained in the compressed file. The structure of the new directory is as follows:

¹You should replace the path for GraphViz with the correct path for your machine.

²<http://ssg4env.techideas.net:8180/archiva/repository/internal/uk/ac/manchester/cs/snee/> deployed 22 February 2011. Note that developers require a user name and password to access the repository, obtainable from TechIdeas, which must be configured in your Maven setup. It is beyond the scope of this document to describe configuring a Maven project to use SNEE as a dependency.



- SNEE-1.5.3
 - etc
 - dqp.snee.properties
 - dqp-physical-schema.xml
 - inqp.snee.properties
 - inqp-logical-schema.xml
 - inqp-physical-schema.xml
 - inqp-cost-parameters.xml
 - inqp-query-parameters.xml
 - telosb-site-resources.xml
 - inqp-topology.xml
 - datqp.snee.properties
 - datqp-logical-schema.xml
 - datqp-physical-schema.xml
 - log4j.properties
 - Types.xml
 - units.xml
 - lib
 - activation-1.1.jar
 - antlr-2.7.5.jar
 - aopalliance-1.0.jar
 - asm-2.2.3.jar
 - commons-lang-2.4.jar
 - commons-logging-1.1.1.jar
 - cxf-api-2.2.8.jar
 - cxf-common-schemas-2.2.8.jar
 - cxf-common-utilities-2.2.8.jar
 - cxf-rt-bindings-soap-2.2.8.jar
 - cxf-rt-bindings-xml-2.2.8.jar
 - cxf-rt-core-2.2.8.jar
 - cxf-rt-databinding-jaxb-2.2.8.jar
 - cxf-rt-frontend-jaxws-2.2.8.jar
 - cxf-rt-frontend-simple-2.2.8.jar
 - cxf-rt-transport-http-2.2.8.jar
 - cxf-rt-ws-addr-2.2.8.jar
 - cxf-tools-common-2.2.8.jar
 - data-source-core-1.5.2.jar
 - data-source-pull-stream-1.5.2.jar
 - data-source-wsdair-1.5.2.jar
 - geronimo-activation1.1.spec-1.0.2.jar
 - geronimo-annotation1.0.spec-1.1.1.jar
 - geronimo-javamail1.4.spec-1.6.jar
 - geronimo-jaxws2.1.spec-1.0.jar
 - geronimo-stax-api1.0.spec-1.0.1.jar
 - geronimo-ws-metadata2.0.spec-1.1.2.jar
 - jaxb-api-2.1.jar
 - jaxb-impl-2.1.12.jar
 - jms-1.1.jar
 - log4j-1.2.15.jar



```
- mail-1.4.jar
- neethi-2.0.4.jar
- saaj-api-1.3.jar
- saaj-impl-1.3.2.jar
- sds-wsdl-0.1.1.jar
- snee-api-1.5.2.jar
- snee-compiler-1.5.2.jar
- snee-core-1.5.2.jar
- snee-sncb-1.5.2.jar
- spring-beans-2.5.6.jar
- spring-context-2.5.6.jar
- spring-core-2.5.6.jar
- spring-web-2.5.6.jar
- tinyos-2.1.jar
- wsdlair-wsdl-0.0.1-SNAPSHOT.jar
- wsdl4j-1.6.2.jar
- wstx-asl-3.2.9.jar
- xml-resolver-1.2.jar
- XmlSchema-1.4.5.jar
- src/uk/ac/manchester/cs/snee/client
- SampleClient.java
```

4.1.3 Compiling the Sample Client

The sample client requires compilation before it can be run. First, we set the classpath to include the dependencies of SNEE³. (Note that care should be taken when copying and pasting the command from this document that spaces are not introduced.)

³The command shown is that for a linux/unix/Mac OSX environment. It should be adjusted accordingly for Windows.



```
SNEE-1.5.3> export CLASSPATH="$CLASSPATH:./lib/activation-1.1.jar:  
lib/antlr-2.7.5.jar:lib/aopalliance-1.0.jar:lib/asm-2.2.3.jar:  
lib/commons-lang-2.4.jar:lib/commons-logging-1.1.1.jar:lib/cxf-api-2.2.8.jar:  
lib/cxf-common-schemas-2.2.8.jar:lib/cxf-common-utilities-2.2.8.jar:  
lib/cxf-rt-bindings-soap-2.2.8.jar:lib/cxf-rt-bindings-xml-2.2.8.jar:  
lib/cxf-rt-core-2.2.8.jar:lib/cxf-rt-databinding-jaxb-2.2.8.jar:  
lib/cxf-rt-frontend-jaxws-2.2.8.jar:lib/cxf-rt-frontend-simple-2.2.8.jar:  
lib/cxf-rt-transport-http-2.2.8.jar:lib/cxf-rt-ws-addr-2.2.8.jar:  
lib/cxf-tools-common-2.2.8.jar:lib/data-source-core-1.5.2.jar:  
lib/data-source-pull-stream-1.5.2.jar:lib/data-source-wsdair-1.5.2.jar:  
lib/geronimo-activation_1.1_spec-1.0.2.jar:  
lib/geronimo-annotation_1.0_spec-1.1.1.jar:  
lib/geronimo-javamail_1.4_spec-1.6.jar:  
lib/geronimo-jaxws_2.1_spec-1.0.jar:  
lib/geronimo-stax-api_1.0_spec-1.0.1.jar:  
lib/geronimo-ws-metadata_2.0_spec-1.1.2.jar:  
lib/jaxb-api-2.1.jar:lib/jaxb-impl-2.1.12.jar:lib/jms-1.1.jar:  
lib/log4j-1.2.15.jar:lib/mail-1.4.jar:lib/neethi-2.0.4.jar:  
lib/saaj-api-1.3.jar:lib/saaj-impl-1.3.2.jar:lib/sds-wsdl-0.1.1.jar:  
lib/snee-api-1.5.2.jar:lib/snee-compiler-1.5.2.jar:lib/snee-core-1.5.2.jar:  
lib/snee-sncb-1.5.2.jar:lib/spring-beans-2.5.6.jar:  
lib/spring-context-2.5.6.jar:lib/spring-core-2.5.6.jar:  
lib/spring-web-2.5.6.jar:lib/tinyos-2.1.jar:  
lib/wsdair-wsdl-0.0.1-SNAPSHOT.jar:lib/wsd4j-1.6.2.jar:  
lib/wstx-asl-3.2.9.jar:lib/xml-resolver-1.2.jar:lib/XmlSchema-1.4.5.jar"
```

The client can then be compiled with the command:

```
SNEE-1.5.3> javac uk/ac/manchester/cs/snee/client/SampleClient.java
```

4.2 Running the SNEE Sample Client

The sample client can be run from the command line and takes the following parameters:

- The location of the SNEE properties file;
- The query;
- The duration for query execution, in seconds;
- The optional parameters associated with the query.

You can modify these arguments when you run the client to specify different SNEE property files, queries, and execution durations, as is shown in the following sections.

The basic form of the command line invocation is shown below.

```
SNEE-1.5.3> java uk.ac.manchester.cs.snee.client.SampleClient  
  <SNEE properties file>  
  <query>  
  <duration>  
  [<query-parameters-file>]
```



4.2.1 Running an Out-of-Network Query

Configuration

The sample DQP physical schema configuration file (`etc/dqp-physical-schema.xml`) for the sample client declares the use of the following external data services

- CCO Streaming Data Service: accessible at `http://webgis1.geodata.soton.ac.uk:8080/CCO/services/PullStream?wsdl`;
- CCO Stored Data Service: accessible at `http://webgis1.geodata.soton.ac.uk:8080/dai/services/`.

SNEE retrieves the logical schema directly from these sources⁴, thus no logical schema is declared. Note, it is also possible to add external data sources through the `addServiceSource` in the SNEE API (Table B.1), in that case there is no need to provide a physical schema file.

Running the Sample Client

To run a union query over two of the stream extents available from the CCO Streaming Data Service for a duration of 30 seconds, the application can be run with the following command.

```
SNEE-1.5.3> java uk.ac.manchester.cs.snee.client.SampleClient
"etc/dqp.snee.properties"
"(SELECT * FROM envdata_haylingisland) UNION
(SELECT * FROM envdata_sandownbay);" 30
```

4.2.2 Running an In-Network Query

Configuration

The sample INQP SNEE configuration file (`etc/inqp.snee.properties`) specifies the use of a logical and physical schema files.

In the logical schema file (`etc/inqp-logical-schema.xml`), an acquisitional stream named `SeaDefence` is defined. It has a single sensed attribute, `waterLevel`, of type integer:

```
<stream name="SeaDefence" type="pull">
  <column name="seaLevel">
    <type class="integer"/>
  </column>
</stream>
<stream name="SeaDefenceEast" type="pull">
  <column name="seaLevel">
    <type class="integer"/>
  </column>
</stream>
<stream name="SeaDefenceWest" type="pull">
  <column name="seaLevel">
    <type class="integer"/>
  </column>
</stream>
```

⁴The order of sources is important. Declarations of the same extent by later sources are ignored.



Note that the `SeaDefence` stream also implicitly includes the attributes `time`, which specifies the time, and `id`, which specifies node identifier, from which the sensor reading was taken.

In the physical schema file (`etc/inqp-physical-schema.xml`) specifies:

- A topology file, with a description of the sensor network. This is an optional parameter as SNEE has the capability to discover the topology of a sensor network.
- A site resources file, which describes the resources (e.g., memory) available at each node in the sensor network. This depends on the physical hardware being used.
- The identifier of the *gateway* node, i.e. , the node from which QEPs are disseminated from, and towards which query results are sent to.
- For each acquisitional stream in the logical schema, the identifiers of the nodes that provide sources to this stream;

The example configuration for the sample physical schema file is shown below:

```
<sensor_network name="wsn1">
  <topology>etc/inqp-topology.xml</topology>
  <site-resources>etc/inqp-telosb.xml</site-resources>
  <gateways>1</gateways>
  <extents>
    <extent name="SeaDefence">
      <sites>2,3</sites>
    </extent>
    <extent name="SeaDefenceEast">
      <sites>2</sites>
    </extent>
    <extent name="SeaDefenceWest">
      <sites>3</sites>
    </extent>
  </extents>
</sensor_network>
```

The above are used as metadata for query compilation.

Running the Sample Client

To run a join query over two of the stream extents available from a sensor network for a duration of 300 seconds, the application can be run with the following command.

```
SNEE-1.5.3> java uk.ac.manchester.cs.snee.client.SampleClient
  "etc/inqp.snee.properties"
  "SELECT e.seaLevel
  FROM SeaDefenceEast[NOW] e, SeaDefenceWest[NOW] w
  WHERE e.seaLevel > w.seaLevel;" 300 "etc/inqp-query-parameters.xml"
```

4.2.3 Running Data Analyses

Configuration

A sample DAT SNEE configuration file (`etc/datqp.snee.properties`) specifies the use of a logical and physical schema files.



The logical schema file (`etc/datqp-logical-schema.xml`) reflects the examples shown in Chapter 1 in which three streams are defined, viz., `Forest`, `ForestMoisture` and `ForestLRF`. The latter is an intensional extent that represents a linear regression classifier. The definition of these streams in the logical schema is shown below:

```
<stream name="Forest" type="push">
  <column name="ts">
    <type class="timestamp"/>
  </column>
  <column name="moisture">
    <type class="float"/>
  </column>
  <column name="temperature">
    <type class="float"/>
  </column>
</stream>

<stream name="forestMoisture" type="push">
  <column name="timestamp">
    <type class="float"/>
  </column>
  <column name="moisture">
    <type class="float"/>
  </column>
</stream>

<stream name="forestLRF" isIntensional="true" type="push">
  <column name="ts">
    <type class="timestamp"/>
  </column>
  <column name="moisture">
    <type class="float"/>
  </column>
  <column name="temperature">
    <type class="float"/>
  </column>
  <datParams type="CLASSIFIER" subtype="LinearRegressionFunction">
    <derivedAttribute name="temperature"/>
    <source>SELECT moisture, temperature
      FROM Forest[FROM NOW-1 HOUR TO NOW SLIDE 1 HOUR]</source>
  </datParams> </stream>
```

In this example, a random tuple generator is used as a data source. The physical schema (shown below) is configured so that the random tuple generator provides data to the `Forest` and `ForestMoisture` extents. Note that the `forestLRF` stream, being intensional, does not appear in the physical schema as its tuples are derived rather than being acquired or stored.



```
<udp_source name="TupleGenerator">
  <host>228.5.6.7</host>
  <extents>
    <extent name="Forest">
      <push_source>
        <port>6702</port>
      </push_source>
    </extent>
    <extent name="ForestMoisture">
      <push_source>
        <port>6703</port>
      </push_source>
    </extent>
  </extents>
</udp_source>
```

The above are used as metadata for query refactoring and compilation.

Running the Sample Client

To run a simple query, such as retrieving sensed values for all attributes from the **Forest** stream, for 60 seconds, the following command can be issued:

```
SNEE-1.5.3> java uk.ac.manchester.cs.snee.client.SampleClient
  "etc/datqp.snee.properties"
  "SELECT * FROM Forest[NOW] f;" 60
```

The following command runs a sample query using the **forestLRF** classifier, as shown in the example in Chapter 1:

```
SNEE-1.5.3> java uk.ac.manchester.cs.snee.client.SampleClient
  "etc/datqp.snee.properties"
  "SELECT RSTREAM fm.moisture, fr.temperature
  FROM forestMoisture[NOW] fm, forestLRF fr
  WHERE fm.moisture=fr.moisture;" 60
```



5. Test Strategy

The testing strategy for the development of SNEE has followed standard practice in its use of unit testing and integration testing. These are briefly summarised in the following sections.

5.1 Unit Testing

SNEE is being developed following a test driven development strategy with the JUnit framework [JUn11]. The unit testing framework is integrated into the Maven build system, ensuring that all tests are run on a regular basis by developers. The coverage of the unit tests is shown in Table 5.1.

Module	Coverage (%)
snee-api	0
data-source-core	50
data-source-pull-stream	0
data-source-wsdair	16
snee-core	27
snee-sncb	0
snee-compiler	33

Table 5.1: Unit test coverage

Notes:

1. `snee-api` contains no tests since it declares interfaces and associated bean classes.
2. `data-source-pull-stream` consists of generated code for interacting with external web services.
3. `data-source-wsdair` consists of generated code for interacting with external web services.
4. `snee-core` consists of large amounts of legacy code for which tests do not exist. New classes in this module contain test cases.
5. `snee-sncb` consists of scripts and legacy code for interacting with the sensor network.
6. `snee-compiler` consists of legacy code for which tests do not exist.

5.2 Integration Testing

Each development cycle of SNEE is extensively tested with a range of integration tests. These involve several command line client applications which test the running of SNEE with a variety of queries using the different configurations.

For the purpose of integration testing, the code base contains a random tuple generator which can be used to simulate a data source. The use of the tuple generator ensures that SNEE functionality can be tested independently of external data services, such as the CCO, or the quirks of sensor networks. Integration tests are also run over the CCO streaming and stored data services as well as a locally connected sensor network.



A. Interface Operations for Streaming Data

A.1 Query Interface

Table A.1: The Query Interface.

Types	
DataResourceAbstractName	The abstract name associated with the data resource(s) represented as a URI.
DatasetFormatURI	The URI of a dataset format.
RequestDocument	An XML document that contains the request expression, i.e. the query and its parameters.
ResponseDocument	A document consisting of a dataset format URI and the encoded data.
ConfigurationDocument	An XML document containing the initial parameters for the indirect access resource.
PropertyDocument	An XML document conforming to a defined XML schema to describe the properties of the service.
Expression (xsd:string)	Any language statement.
QueryDuration (xsd:dateTime xsd:duration)	The duration that the query is expected to execute for, or an absolute time when the query will stop running.
Faults	
ServiceBusyFault	The service is already processing a message and concurrent operations are not supported.
NotAuthorizedFault	The consumer is not authorized to perform the requested operation or is not authorized to perform the requested operation at this time.
InvalidResourceNameFault	The data resource specified is unknown to the service.
DataResourceUnavailableFault	The data resource that is the target of the message is currently not available.
InvalidExpressionFault	The expression given as part of the request contains errors.
InvalidLanguageFault	The input dataset (usually the expression component of an incoming request) has an unrecognized language element.
InvalidDatasetFormatFault	The dataset format URI specified is not known to the service.
InvalidConfigurationDocumentFault	The configuration document specified is not valid.
Operations	
GetDataResourcePropertyDocument	<i>Returns the core property document values associated with the service implementing this message.</i>
Inputs	resourceName: DataResourceAbstractName
Output	properties: PropertyDocument
Faults	InvalidResourceName, DataResourceUnavailableFault, NotAuthorizedFault, ServiceBusyFault
DestroyDataResource	<i>Destroy the named data resource; future messages directed at the resource MUST yield an InvalidResourceNameFault.</i>
Inputs	resourceName: DataResourceAbstractName
Output	
Faults	InvalidResourceName, DataResourceUnavailableFault, NotAuthorizedFault, ServiceBusyFault
GenericQuery	<i>Directs a query document to a data resource.</i>
Inputs	resourceName: DataResourceAbstractName, datasetFormatURI?: anyURI, queryExpression: GenericExpression
Output	queryResponse: GenericQueryResponse
Faults	InvalidResourceNameFault, DataResourceUnavailableFault, InvalidDatasetFormatFault, InvalidExpressionFault, InvalidLanguageFault, NotAuthorizedFault, ServiceBusyFault

Continued overleaf

Table A.1: The Query Interface. (continued)

GenericQueryFactory		<i>Creates a relationship between a data resource which executes the query and the data service through which query answers are made available.</i>
	Inputs	resourceName: DataResourceAbstractName, responseInterfaceType?: QName, configurationDocument?: ConfigurationDocument, preferredTargetService?: EndpointReference, requestDocument: GenericExpression
	Output	resourceList: DataResourceAddressList
	Faults	InvalidResourceNameFault, DataResourceUnavailableFault, InvalidPortTypeQNameFault, InvalidConfigurationDocumentFault, InvalidExpressionFault, InvalidLanguageFault, NotAuthorizedFault, ServiceBusyFault

A.2 Data Access Interface

Table A.2: Pull Stream Service Operations.

Types		
DataResourceAbstractName		The abstract name associated with the data resource(s) represented as a URI.
DatasetFormatURI		The URI of a dataset format.
ResponseDocument		A document consisting of a dataset format URI and the encoded data.
PropertyDocument		An XML document conforming to a defined XML schema to describe the properties of the service.
MaximumHistoryDuration? (xsd:duration)		The maximum amount of time that the data resource will make a tuple available.
MaximumHistoryTuples? (xsd:int)		The maximum number of tuples that the data resource will make available.
Faults		
ServiceBusyFault		The service is already processing a message and concurrent operations are not supported.
NotAuthorizedFault		The consumer is not authorized to perform the requested operation or is not authorized to perform the requested operation at this time.
InvalidResourceNameFault		The data resource specified is unknown to the service.
DataResourceUnavailableFault		The data resource that is the target of the message is currently not available.
Operations		
GetDataResourcePropertyDocument		<i>Retrieves the PropertyDocument for the pull stream response resource.</i>
	Inputs	resourceName: DataResourceAbstractName
	Output	properties: PropertyDocument
	Faults	InvalidResourceNameFault, DataResourceUnavailableFault, NotAuthorizedFault, ServiceBusyFault
GetStreamItem		<i>Retrieves a specified count of stream items from the pull stream service from a specific point in the available history.</i>
	Inputs	resourceName: DataResourceAbstractName, datasetFormatURI?: anyURI, position?: xsd:dateTime xsd:int, count?: xsd:duration xsd:int, maxTuples?: xsd:int
	Output	streamDataset: StreamDataset

Continued overleaf

Table A.2: Pull Stream Service Operations. (continued)

	Faults	InvalidResourceName, DataResourceUnavailableFault, InvalidDatasetFormatFault, NotAuthorizedFault, InvalidWindowFault, MaxTuplesExceededFault, ServiceBusyFault
GetStreamNewestItem		<i>Retrieves the most recent stream items in the pull stream response resource up to the specified count.</i>
	Inputs	resourceName: DataResourceAbstractName, datasetFormatURI?: anyURI, count?: xsd:duration xsd:int, maxTuples?: xsd:int
	Output	streamDataset: StreamDataset
	Faults	InvalidResourceName, DataResourceUnavailableFault, InvalidDatasetFormatFault, NotAuthorizedFault, InvalidWindowFault, MaxTuplesExceededFault, ServiceBusyFault

A.3 Subscription Interface

Table A.3: Push Stream Service Operations.

Types	
DataResourceAbstractName	The abstract name associated with the data resource(s) represented as a URI.
PropertyDocument	An XML document conforming to a defined XML schema to describe the properties of the service.
Filter	A collection of expressions that evaluate which topics to permit.
Policy	A collection of policy statements.
NotificationMessage	A notification message as defined in the Notification Interface.
Faults	
ServiceBusyFault	The service is already processing a message and concurrent operations are not supported.
NotAuthorizedFault	The consumer is not authorized to perform the requested operation or is not authorized to perform the requested operation at this time.
InvalidResourceNameFault	The data resource specified is unknown to the service.
DataResourceUnavailableFault	The data resource that is the target of the message is currently not available.
ResourceUnknownFault	The resource is unknown to the service.
InvalidFilterFault	The filter was not understood or is not supported by the service.
TopicExpressionDialectUnknownFault	The dialect of the topic expression dialect in the filter is unknown to the service.
InvalidTopicExpressionFault	The topic expression in the filter is not valid.
TopicNotSupportedFault	The topic expression in the filter contains a topic that is not supported.
InvalidProducerPropertiesExpressionFault	The filter contained a property expression that is not valid.
InvalidMessageContentExpressionFault	The filter contained a message content filter that is not valid.
UnrecognizedPolicyRequestFault	The service does not recognize one or more policy request.
UnsupportedPolicyRequestFault	The service does not support one or more of the policy requests.
NotifyMessageNotSupportedFault	The service does not support the notify wrapper.
UnacceptableTerminationTimeFault	The specified termination time was not acceptable to the service.
SubscribeCreationFailedFault	The service failed to process the Subscribe message.
MultipleTopicsSpecifiedFault	More than one topic was referenced.
NoCurrentMessageOnTopicFault	There are no messages for the referenced topic.
Operations	
<i>Continued overleaf</i>	

Table A.3: Push Stream Service Operations. (continued)

GetDataResourcePropertyDocument		<i>Retrieves the PropertyDocument for the pull stream response resource.</i>
	Inputs	resourceName: DataResourceAbstractName
	Output	properties: PropertyDocument
	Faults	InvalidResourceNameFault, DataResourceUnavailableFault, NotAuthorizedFault, ServiceBusyFault
Subscribe		<i>Registers a consumer to receive a subset of the notification messages sent by the service. It results in a Subscription-Manager resource being created for the specific subscribe operation between the consumer and the push stream response resource. (This operation, its parameters and faults, are taken directly from WS-Notification).</i>
	Inputs	consumerReference: EndpointReference, filter?: wsnt:Filter, initialTerminationTime?: dateTime duration, subscriptionPolicy?: wsnt:Policy
	Output	subscriptionReference: EndPointReference, currentTime?: dateTime, terminationTime?: dateTime
	Faults	ResourceUnknownFault, InvalidFilterFault, TopicExpressionDialectUnknownFault, InvalidTopicExpressionFault, TopicNotSupportedFault, InvalidProducerPropertiesExpressionFault, InvalidMessageContentExpressionFault, UnrecognizedPolicyRequestFault, UnsupportedPolicyRequestFault, UnacceptableInitialTerminationTimeFault, NotifyMessageNotSupportedFault, SubscribeCreationFailedFault
GetCurrentMessage		<i>Allows a newly registered consumer to retrieve the most recent notify message.</i>
	Inputs	topicName: QName
	Output	message: NotificationMessage
	Faults	ResourceUnknownFault, TopicExpressionDialectUnknownFault , InvalidTopicExpressionFault, TopicNotSupportedFault, MultipleTopicsSpecifiedFault, NoCurrentMessageOnTopicFault

B. Java APIs

B.1 SNEE API

Table B.1: SNEE Java API.

Method Summary	
void	addServiceSource(String name, String url, SourceType interfaceType) Adds a service source to the set of available data for querying.
Collection<String>	getExtents() Return a list of the extent names available in the schema.
ExtentMetadata	getExtensionDetails(String extentName) Retrieve the metadata about a specified extent.
int	addQuery(String query, String parametersFile) Adds a query to the set of registered queries and returns the generated query identifier. It takes a query statement as input, and optionally any parameters associated with it, generates a query plan for its evaluation, and adds it to the set of registered query plans.
ResultStore	getResultStore(int queryId) Retrieve the ResultStore for a specified query if it exists.
int	removeQuery(int queryId) Removes a query from the set of registered queries. It takes a query identifier and stops the required query evaluation if it exists.
void	close() Close SNEE down gracefully.

B.2 ResultStore API

Table B.2: SNEE ResultStore API.

Method Summary	
void	add(Output data) Add an item of data to the result set.
void	addAll(Collection<Output> data) Add a collection of items of data to the result set.
int	size() Returns the current size of the result set.
ResultSetMetaData	getMetadata() Retrieves the number, types and properties of this StreamResultSet object's columns.
String	getCommand() Retrieves this StreamResultSet object's command property. The command property contains a command string, which must be a SNEEql query, that can be executed to fill the StreamResultSet with data. The default value is null .
void	setCommand(String cmd) Sets this StreamResultSet object's command property to the given SNEEql query.
List<ResultSet>	getResults() Return all of the results available for the specified query.
List<ResultSet>	getResults(int count) Return the specified number of results starting with the oldest available.
List<ResultSet>	getResults(Duration duration) Return the specified duration of results starting with the oldest available.

Continued overleaf

Table B.2: SNEE ResultStore API. (continued)

List<ResultSet>	getResultsFromIndex(int index) Return all of the results starting from the specified index value.
List<ResultSet>	getResultsFromIndex(int index, int count) Return the specified number of results starting from the index value.
List<ResultSet>	getResultsFromIndex(int index, Duration duration) Return the specified duration of results starting from the specified index value.
List<ResultSet>	getResultsFromTimestamp(Timestamp timestamp) Return all of the results starting from the specified timestamp.
List<ResultSet>	getResultsFromTimestamp(Timestamp timestamp, int count) Return the specified number of results starting from the specified timestamp.
List<ResultSet>	getResultsFromTimestamp(Timestamp timestamp, Duration duration) Return the specified duration of results starting from the specified timestamp.
List<ResultSet>	getNewestResults() Return the result items.
List<ResultSet>	getNewestResults(int count) Return the specified number of result items, counting back from the most recent.
List<ResultSet>	getNewestResults(Duration duration) Return the specified duration of result items, spanning back in time from the most recent.



C. Sample SNEE Client

```
1 package uk.ac.manchester.cs.snee.client;
2
3 import java.io.IOException;
4 import java.net.MalformedURLException;
5 import java.sql.ResultSet;
6 import java.sql.ResultSetMetaData;
7 import java.sql.SQLException;
8 import java.sql.Types;
9 import java.util.Collection;
10 import java.util.Date;
11 import java.util.List;
12 import java.util.Observable;
13 import java.util.Observer;
14
15 import org.apache.log4j.Logger;
16 import org.apache.log4j.PropertyConfigurator;
17
18 import uk.ac.manchester.cs.snee.EvaluatorException;
19 import uk.ac.manchester.cs.snee.MetadataException;
20 import uk.ac.manchester.cs.snee.ResultStoreImpl;
21 import uk.ac.manchester.cs.snee.SNEECompilerException;
22 import uk.ac.manchester.cs.snee.SNEEController;
23 import uk.ac.manchester.cs.snee.SNEEDataSourceException;
24 import uk.ac.manchester.cs.snee.SNEEException;
25 import uk.ac.manchester.cs.snee.common.SNEEConfigurationException;
26 import uk.ac.manchester.cs.snee.compiler.queryplan.expressions.Attribute;
27 import uk.ac.manchester.cs.snee.metadata.schema.AttributeType;
28 import uk.ac.manchester.cs.snee.metadata.schema.ExtentMetadata;
29
30 public class SampleClient implements Observer {
31
32     private static Logger logger =
33         Logger.getLogger(SampleClient.class.getName());
34
35     private SNEEController controller;
36
37     /**
38      * Configures the SNEE query engine according to the properties
39      * specified in the properties file.
40      *
41      * @param propertiesFile location of the snee configuration file
42      * @throws SNEEException
43      * @throws IOException
44      * @throws SNEEConfigurationException
45      * @throws MetadataException
46      * @throws SNEEDataSourceException
47      */
48     public SampleClient(String propertiesFile)
49         throws SNEEException, IOException, SNEEConfigurationException,
50         MetadataException, SNEEDataSourceException
51     {
52         if (logger.isDebugEnabled())
53             logger.debug("ENTER SampleClient()");
54         controller = new SNEEController(propertiesFile);
55         if (logger.isDebugEnabled())
56             logger.debug("RETURN");
57     }
58
59     /**
60      * Displays the extents available for querying
61      */
62     public void displayExtents()
63         throws MetadataException
64     {
65         Collection<String> extents = controller.getExtents();
66         for (String extent : extents) {
67             displayExtentSchema(extent);
68         }
69     }
70 }
```



```
69     }
70
71     private void displayExtentSchema(String extentName)
72     throws MetadataException
73     {
74         ExtentMetadata extent =
75             controller.getExtentDetails(extentName);
76         List<Attribute> attributes = extent.getAttributes();
77         System.out.println("Attributes for " + extentName + ":");
78         for (Attribute attr : attributes) {
79             String attrName = attr.getAttributeDisplayName();
80             AttributeType attrType = attr.getType();
81             System.out.print("\t" + attrName + ": " +
82                 attrType.getName() + "\n");
83         }
84         System.out.println();
85     }
86
87     private void printResults(List<ResultSet> results,
88                             int queryId)
89     throws SQLException {
90         System.out.println("***** Results for query " +
91             queryId + " *****");
92         for (ResultSet rs : results) {
93             ResultSetMetaData metaData = rs.getMetaData();
94             int numCols = metaData.getColumnCount();
95             printColumnHeadings(metaData, numCols);
96             while (rs.next()) {
97                 StringBuffer buffer = new StringBuffer();
98                 for (int i = 1; i <= numCols; i++) {
99                     Object value = rs.getObject(i);
100                     if (metaData.getColumnType(i) ==
101                         Types.TIMESTAMP && value instanceof Long) {
102                         buffer.append(
103                             new Date(((Long) value).longValue()));
104                     } else {
105                         buffer.append(value);
106                     }
107                     buffer.append("\t");
108                 }
109                 System.out.println(buffer.toString());
110             }
111         }
112         System.out.println("*****");
113     }
114
115     private void printColumnHeadings(ResultSetMetaData metaData,
116                                     int numCols) throws SQLException {
117         StringBuffer buffer = new StringBuffer();
118         for (int i = 1; i <= numCols; i++) {
119             buffer.append(metaData.getColumnLabel(i));
120             // buffer.append(": " + metaData.getColumnTypeName(i));
121             buffer.append("\t");
122         }
123         System.out.println(buffer.toString());
124     }
125
126     /**
127     * Callback mechanism for reacting to new query results
128     *
129     * @param observation
130     * @param arg the new query result
131     */
132     public void update (Observable observation, Object arg) {
133         if (logger.isDebugEnabled()) {
134             logger.debug("ENTER update() with " + observation + " " +
135                 arg);
136         }
137         // logger.trace("arg type: " + arg.getClass());
138         if (arg instanceof List<?>) {
139             List<ResultSet> results = (List<ResultSet>) arg;
140             try {
```



```
141         printResults(results, 1);
142     } catch (SQLException e) {
143         logger.error("Problem printing result set. ", e);
144     }
145 }
146 if (logger.isDebugEnabled()) {
147     logger.debug("RETURN update()");
148 }
149 }
150
151 /**
152  * Execute a SNEEqL query using the configured SNEE query execution
153  * engine.
154  *
155  * @param query SNEEqL query string to be executed
156  * @param duration length in seconds to execute the query
157  * @param queryParams location of the parameters file associated with the query
158  */
159 public void executeQuery(String query, String queryParameters,
160     double duration)
161     throws SNEECompilerException, MetadataException, EvaluatorException,
162     SNEEException, SQLException, SNEEConfigurationException {
163     if (logger.isDebugEnabled())
164         logger.debug("ENTER executeQuery() with query " + query +
165             " parameters " + queryParameters +
166             " duration " + duration);
167
168     System.out.println("Query: " + query);
169
170     int queryId = controller.addQuery(query, queryParameters);
171
172     long startTime = System.currentTimeMillis();
173     long endTime = (long) (startTime + (duration * 1000));
174
175     System.out.println("Running query for " + duration +
176         " seconds. Scheduled end time " + new Date(endTime));
177
178     ResultStoreImpl resultStore =
179     (ResultStoreImpl) controller.getResultStore(queryId);
180     resultStore.addObserver(this);
181
182     try {
183         Thread.currentThread().sleep((long)duration * 1000);
184     } catch (InterruptedException e) {
185     }
186
187     while (System.currentTimeMillis() < endTime) {
188         Thread.currentThread().yield();
189     }
190
191     List<ResultSet> results = resultStore.getResults();
192     System.out.println("Stopping query " + queryId + ".");
193     controller.removeQuery(queryId);
194
195     controller.close();
196     printResults(results, queryId);
197     if (logger.isDebugEnabled())
198         logger.debug("RETURN executeQuery()");
199 }
200
201 /**
202  * The main entry point for the Sample Client
203  * @param args
204  * @throws IOException
205  * @throws InterruptedException
206  */
207 public static void main(String[] args) {
208     // Configure logging
209     PropertyConfigurator.configure(
210         SampleClient.class.getClassLoader().
211         getResource("etc/log4j.properties"));
212     String propertiesFile = null;
```



```
213     String query = null;
214     Long duration = null;
215     String params = null;
216     //This method represents the web server wrapper
217     if (args.length < 3 || args.length > 4) {
218         System.out.println("Usage: \n" +
219             "\t\"location of SNEE properties file\"\n" +
220             "\t\"query statement\"\n" +
221             "\t\"query duration in seconds\"\n" +
222             "\t\"optional third argument stating location of query parameters file\"");
223         System.exit(1);
224     } else {
225         propertiesFile = args[0];
226         query = args[1];
227         duration = Long.valueOf(args[2]);
228         if (args.length == 4) {
229             params = args[3];
230         }
231     }
232
233     try {
234         /* Initialise the Client */
235         SampleClient client =
236             new SampleClient(propertiesFile);
237         /* Print the available extents */
238         client.displayExtents();
239         /* Execute the query */
240         client.executeQuery(query, params, duration);
241     } catch (Exception e) {
242         System.out.println("Execution failed. See logs for detail.");
243         logger.fatal(e);
244         System.exit(1);
245     }
246
247     System.out.println("Success!");
248     System.exit(0);
249 }
250
251 }
```



Bibliography

- [AAK⁺06] M. Antonioletti, Malcolm Atkinson, A. Krause, S. Laws, S. Malaika, Norman W. Paton, D. Pearson, and G. Riccardi. WS-DAI Specification, Version 1.0. Recommendation GFD.74, Open Grid Forum, 5 September 2006.
- [ACK⁺06] Mario Antonioletti, Brian Collins, Amy Krause, Simon Laws, James Magowan, Susan Malaika, and Norman W. Paton. WS-DAIR Specification, Version 1.0. Recommendation GFD.76, Open Grid Forum, 20 July 2006.
- [Apa11] Apache maven, 2011. <http://maven.apache.org/> accessed 11 February 2011.
- [BGFP08] Christian Y. A. Brenninkmeijer, Ixent Galpin, Alvaro A. A. Fernandes, and Norman W. Paton. A semantics for a query language over sensors, streams and relations. In *BNCOD*, pages 87–99, 2008.
- [CCO09] Channel coastal observatory, 2009. <http://www.channelcoast.org/> accessed 17 December 2009.
- [GBG⁺10] Ixent Galpin, Christian Y. A. Brenninkmeijer, Alasdair J. G. Gray, Farhana Jabeen, Alvaro A. A. Fernandes, and Norman W. Paton. SNEE: a query processor for wireless sensor networks. *Distributed and Parallel Databases*, 29(1-2):31–85, November 2010.
- [GBJ⁺09] Ixent Galpin, Christian Y. A. Brenninkmeijer, Farhana Jabeen, Alvaro A. A. Fernandes, and Norman W. Paton. Comprehensive optimization of sensor network queries. In *SSDBM*, pages 339–360, 2009.
- [GGF⁺09] Ixent Galpin, Alasdair J G Gray, Alvaro A A Fernandes, Norman W Paton, Alexis Kotsifakos, Dimitris Kotsakos, and Dimitrios Gunopulos. Data requirements, data management and analysis issues, and query-based functionalities. Deliverable D2.1, SemSorGrid4Env, August 2009.
- [GGF⁺10a] Ixent Galpin, Alasdair J. G. Gray, Alvaro A. A. Fernandes, Norman W. Paton, Alexis Kotsifakos, George Valkanas, and Dimitrios Gunopulos. Incorporation of data analysis functionality into the snee query processor. Technical report, University of Manchester and University of Athens, July 2010.
- [GGF⁺10b] Alasdair J. G. Gray, Ixent Galpin, Alvaro A. A. Fernandes, Norman W. Paton, Kevin Page, Jason Sadler, Kostis Kyzirakos, Manolis Koubarakis, Jean-Paul Calbimonte, Raúl Garcia, Oscar Corcho, Jesús E. Gabaldón, and Juan José Aparicio. SemSorGrid4Env architecture – phase II. Deliverable D1.3v2, SemSorGrid4Env, December 2010.
- [GGFP09] Alasdair J. G. Gray, Ixent Galpin, Alvaro A. A. Fernandes, and Norman W. Paton. WS-DAI-Streaming Specification. Technical report, University of Manchester, December 2009.
- [GLvB⁺03] David Gay, Philip Levis, J. Robert von Behren, Matt Welsh, Eric A. Brewer, and David E. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI*, pages 1–11, 2003.
- [Gra09] GraphViz, 2009. <http://www.graphviz.org/> accessed 10 December 2009.
- [HC04] Jonathan W. Hui and David E. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys*, pages 81–94, 2004.
- [JUn11] JUnit, 2011. <http://www.junit.org/> accessed 12 February 2011.
- [SPP⁺06] Sharmila Subramaniam, Themis Palpanas, Dimitris Papadopoulos, Vana Kalogeraki, and Dimitrios Gunopulos. Online outlier detection in sensor data using non-parametric models. In *VLDB*, pages 187–198, 2006.