



WatERP

Water Enhanced Resource Planning
“Where water supply meets demand”

GA number: 318603

WP 3: Data Management
D3.4: WDW Final Prototype

V1.0 20/05/2015

www.waterp-fp7.eu

Document Information

Project Number	318603	Acronym	WatERP
Full title	Water Enhanced Resource Planning “Where water supply meets demand”		
Project URL	http://www.waterp-fp7.eu		
Project officer	Grazyna Wojcieszko		

Deliverable	Number	3.4	Title	WDW Final Prototype
Work Package	Number	3	Title	Data Management

Date of delivery	Contractual	M30	Actual	M32
Nature	Prototype <input checked="" type="checkbox"/> Report <input type="checkbox"/> Dissemination <input type="checkbox"/> Other <input type="checkbox"/>			
Dissemination Level	Public <input checked="" type="checkbox"/> Consortium <input type="checkbox"/>			

Responsible Author	Johannes Kutterer	Email	johannes.kutterer@disy.net
Partner	DISY	Phone	(+49) 721 1 6006-286

Abstract (for dissemination)	This document provides an overview of the architecture of the water data warehouse and the integration of the pilot sites. It explains the protocols and components that are being used for storing and exchanging both ontological information and observation results between the pilot site, water data warehouse and SOA-MAS. Further, it describes how data mining and geospatial reasoning have been implemented and what considerations have been made about performance, stability and extensibility.
Key words	Water data warehouse, OGC services, WPS, WFS-T, WMS, GeoSPARQL, SOS, WaterML2.0, triple store, RDF

Glossary of Acronyms

ACA	Agència Catalana de l'Aigua (trans. Catalan Water Agency)	PIM	Pilot Integration Manager
CUAHSI	Consortium of Universities for the Advanced of Hydrologic Sciences, Inc.	RDF	Resource Description Framework
DMZ	DeMilitarized Zone	REST	REpresentational State Transfer
HIS	Hydrologic Information System	SOS	Sensor Observation Service
Hydro DWG	OGC Hydrology Domain Working Group	SensorML	Sensor Markup Language
OWL	Web Ontology Language	SWE	Sensor Web Enablement
MAS	Multi Agent System	SWKA	Stadtwerke Karlsruhe
O&M	Observations and Measurements	WaterML	Water Markup Language
ODM	Observation Data Model	WDTF	Water Data Transfer Format
OGC	Open Geospatial Consortium	WDW	Water Data Warehouse
		WFS	Web Feature Service
		WMS	Web Map Service
		XML	eXtensible Markup Language

Executive Summary

This document describes the final prototype of the WatERP Water Data Warehouse (WDW), including internal functions and interfaces for the pilot integration. It explains the management functions and contains an installation description. The document summarizes the descriptions of the first and the second prototype (WatERP deliverables D3.1, D3.2 and D3.3) and extends the content with the latest changes. The document is structured as follows:

Chapter 1 gives a short overview of the WDW architecture and the role of the WDW within this context. Chapters 2 – 4 describe the different layers that together constitute the WDW. Chapter 2 explains the core functionalities of the Data Warehouse Layer, namely (2.1) the WDW service for overall WDW management and storage of time-series data; (2.2) the Basic Data Mining Operations which both implement ETL (extract-transform-load, to be precise: “transform” in this case) processes of the Data Warehouse Layer and provide efficient preprocessing steps to support data analytics in other WatERP workpackages; and (2.3) the triple store which realizes geospatial and ontological reasoning on observation metadata. Chapter 3 explains the WDW Interface Layer used to access data and knowledge stored in the WDW. Chapter 4 sketches the Pilot Integration Layer which is needed to couple an existing data infrastructure with the WDW. Both the general approach and the concrete example instantiations for the two WatERP pilots (SWKA, ACA) are discussed. Chapter 5 provides an instruction how to set up a WDW installation. Chapter 6 summarizes and concludes the document.

To understand this document better the following deliverables would be useful to be read.

Number	Title	Description
D1.2	Generic functional model for water supply and usage data	Report describing the approach that will be used in the WatERP project to represent the processes required to match supply with demand across the water supply distribution chains. It includes processes and decisions involved in the pilot cases.
D1.3	Generic Ontology for water supply distribution chain	Description of the generic ontology that was developed within WatERP project. This deliverable introduce into the incorporation of human-made interactions inside natural water paths in order to better understanding of the decisions to be adopted. Furthermore, data provenance and Linked Open Data Cloud (LOCD) mechanism are also introduced.
D1.4.1	Inference and Simulation Engine Conceptual Design	This deliverable depicts the architecture of the Decision Support System including a behavioural definition of the inference and simulation engine.
D2.1	External System Integration requirement	Comprehensive review of generic water supply - distribution chain was undertaken in order to understand general requirement for system integration and interoperability to be performed within the WatERP project development.
D7.3.1	Implementation of WDW	Description of the implementation on the pilot site. Provides an insight into the pilot environment and the activities to adapt the sensor infrastructure to

		the WDW.
D7.3.2	Implementation of WDW	Description of the implementation on the pilot site. Provides an insight into the pilot environment and the activities to adapt the sensor infrastructure to the WDW.

Please note that some of these documents are project-internal deliverables. Hence, we repeat them in some parts, in order to have a self-contained document with this public deliverable.

Table of contents

1. INTRODUCTION	12
2. DATA WAREHOUSE LAYER	17
2.1 WDW SERVICE	17
2.1.1 <i>Functional Overview</i>	17
2.1.2 <i>Technical Overview</i>	18
2.1.3 <i>Database Design and Persistence</i>	20
2.1.4 <i>Internal Processing</i>	25
2.1.5 <i>Management Service</i>	30
2.1.6 <i>WDW Service Performance</i>	38
2.1.7 <i>Integration of Rserve</i>	41
2.2 BASIC DATA MINING OPERATIONS	43
2.2.1 <i>Integrated R Infrastructure within WDW</i>	44
2.2.2 <i>Description of Basic Data Mining Operations</i>	48
2.3 TRIPLE STORE	62
2.3.1 <i>Triple Store Technical Overview</i>	62
2.3.2 <i>Geospatial and Ontological Indexing</i>	64
2.3.3 <i>Triple Store Performance</i>	66
3. INTERFACE LAYER	74
3.1 SOS/WATERML2	74
3.2 MANAGEMENT SERVICE	75
3.3 WMS	76
3.4 WFS/WFS-T	77
4. PILOT INTEGRATION MANAGER	79
4.1 GENERAL MODULE DESIGN CONSIDERATIONS	79
4.2 SWKA INTEGRATION	82
4.2.1 <i>Load Sensor Metadata</i>	86

4.2.2	Configure New Sensor	86
4.2.3	Trigger Import / Insert Observation.....	87
4.3	ACA INTEGRATION	88
4.3.1	WaterOneFlow	88
4.3.2	General Processing Steps.....	90
4.3.3	Implementation Details for the ACA PIM.....	93
5.	INSTALLATION	95
5.1	INSTALLATION OF THE TRIPLE STORE	95
5.2	SOS SERVER INSTALLATION.....	100
5.3	GEOSEVER INSTALLATION.....	103
5.4	PILOT INTEGRATION MANAGER INSTALLATION.....	104
5.5	WDW SERVICE INSTALLATION	106
5.6	WPS4R INSTALLATION	108
6.	SUMMARY AND CONCLUSIONS	109
7.	APPENDIX	114
7.1	APPENDIX I: OBSERVATION DATA MODEL (ODM) OF THE CUAHSI HIS	114
7.2	APPENDIX II: WATERONEFLOW SERVICE	115
7.3	APPENDIX III: BASIC DATA-MINING SCRIPTS WITH WPS4R ANNOTATIONS	116
7.3.1	Script <i>bdm_auto_cf.v1.R</i>	116
7.3.2	Script <i>bdm_categorical.v1.R</i>	117
7.3.3	Script <i>bdm_cross_cf.v1.R</i>	117
7.3.4	Script <i>bdm_mv_filter.v1.R</i>	119
7.3.5	Script <i>bdm_outliers.v1.R</i>	120
7.3.6	Script <i>bdm_smooth.v1.R</i>	121
7.3.7	Script <i>bdm_spatial_interpolate.v1.R</i>	122
8.	REFERENCES	124

Table of figures

FIGURE 1: WATERP ARCHITECTURE FROM THE INTEGRATION POINT OF VIEW	12
FIGURE 2: WDW COMPONENT LAYERS.....	15
FIGURE 3: INTERACTIONS OF THE WDW SERVICE WITH OTHER WDW ELEMENTS.....	17
FIGURE 4: CONVERSION FROM WATERML2 TO THE DATABASE STRUCTURE	19
FIGURE 5: CONVERSION FROM DATABASE STRUCTURE TO WATERML2.....	20
FIGURE 6: WDW ENTITY-RELATIONSHIP DIAGRAM	23
FIGURE 7: ADD SENSOR DIAGRAM.....	26
FIGURE 8: ADD PROCESSING-SEQUENCE DIAGRAM	26
FIGURE 9: POLL SOS OBSERVATIONS SEQUENCE DIAGRAM.....	27
FIGURE 10: PERFORM PROCESSING SEQUENCE DIAGRAM.....	28
FIGURE 11: MANAGEMENT SERVICE XML SCHEME	30
FIGURE 12: DATABASE CONTENT AFTER SENSOR CREATION	32
FIGURE 13: GEOSERVER CONTENT AFTER SENSOR CREATION.....	32
FIGURE 14: DATABASE CONTENT AFTER PROCESSING CREATION.....	34
FIGURE 15: GEOSERVER CONTENT AFTER PROCESSING CREATION.....	34
FIGURE 16: 52NORTH SOS SERVER RESPONSE TIMES	39
FIGURE 17: MEMORY USAGE WHEN CREATING RESPONSES WITH LARGE TIME SERIES	40
FIGURE 18: RESPONSE TIMES WITH LINEAR PROGRESSION	40
FIGURE 19: R INFRASTRUCTURE INTEGRATED WITHIN WDW	46
FIGURE 20: A SIMPLE WPS4R ANNOTATED RSCRIPT	47
FIGURE 21: WPS REQUEST IN XML FORMAT (FILE "REQUEST.XML").....	47
FIGURE 22 : A SIMPLE WPS CLIENT WRITTEN IN PYTHON AND EXECUTION OUTPUT	48
FIGURE 23: OVERVIEW OF THE TRIPLE STORE ARCHITECTURE AND SUPPORTED INTERFACES.....	63
FIGURE 24: INTEGRATION OF OWLIM WITHIN THE TRIPLE STORE ENABLES ONTOLOGICAL REASONING	65
FIGURE 25: TRIPLE STORE RESPONSE TIMES GRAPH.....	70
FIGURE 26: COMPARISON OF QUERY 1	71
FIGURE 27: COMPARISON OF QUERY 2	72
FIGURE 28: COMPARISON OF QUERY 3	73

FIGURE 29: SAMPLE WMS REQUEST	76
FIGURE 30: INTEGRATION OF EXISTING PILOT INFRASTRUCTURE	79
FIGURE 31: INTEGRATION OF AN EXISTING SOS INFRASTRUCTURE	80
FIGURE 32: GENERAL MODULE DESIGN OF PIM	82
FIGURE 33: DATABASE TABLE FROM PILOT SITE SWKA.....	83
FIGURE 34: PILOT SITE SWKA ONTOLOGY	84
FIGURE 35: FLOW CHART OF CLIENT APPLICATION	84
FIGURE 36: FLOW CHART DIAGRAM OF FINAL CLIENT APPLICATION FOR THE SWKA INSTANTIATION.....	85
FIGURE 37: SIMPLIFIED WATERONEFLOW DATA STRUCTURE	89
FIGURE 38: GENERAL PROCESSING STEPS FOR ACA INTEGRATION	91
FIGURE 39: ACA POLLING	93
FIGURE 40: POLLING OF A SINGLE SENSOR	94
FIGURE 41: WDW DEPLOYMENT DIAGRAM.....	95
FIGURE 42: TEST SOS SERVER AFTER INSTALLATION - STEP 1	101
FIGURE 43: TEST SOS SERVER AFTER INSTALLATION - STEP 2	101
FIGURE 44: GEOSERVER STORES	103
FIGURE 45: GEOSERVER STORES LIST	103
FIGURE 46: GEOSERVER NEW DATA SOURCE WIZARD	104
FIGURE 47: GEOSERVER DATABASE CONNECTION CONFIGURATION	104
FIGURE 48: ILLUSTRATION OF WDW ARCHITECTURE	109
FIGURE 49: OBSERVATION DATA MODEL CUAHSI HIS.....	114
FIGURE 50: WATERONEFLOW SERVICE DIAGRAM	115

Table of tables

TABLE 1: POLLED_SENSOR TABLE COLUMNS	21
TABLE 2: PROCESSING TABLE COLUMNS	22
TABLE 3: COLUMN DESCRIPTION OF THE PROCESSING PARAMETER TABLE	22
TABLE 4: RAW OR TRANSFORMED TABLE COLUMNS	22
TABLE 5: DATABASE DIALECTS SUPPORTED BY HIBERNATE	24
TABLE 6: SQL FILES PER DATABASE DIALECT	25
TABLE 7: PARAMETERS FOR ADDSENSORREQUEST	31
TABLE 8: PARAMETERS FOR ADDPROCESSINGREQUEST	33
TABLE 9: 52NORTH SOS SERVER RESPONSE TIMES	38
TABLE 10: SYSTEM PARAMETERS FOR R SCRIPTS.....	41
TABLE 11: TRIPLE STORE RESPONSE TIME ON DATASETS OF DIFFERENT SIZES	69
TABLE 12: WATERONEFLOW METHOD DESCRIPTION	90
TABLE 13: COLUMNS OF SENSOR TABLE IN ACA PILOT INTEGRATION MANAGER	91
TABLE 14: CONFIGURATION PARAMETERS	105
TABLE 15: WDW SERVICE CONFIGURATION PARAMETERS	107

Table of listings

LISTING 1: EXAMPLE FOR ADDSENSOR REQUEST	31
LISTING 2: EXAMPLE FOR ADDSENSOR RESPONSE	31
LISTING 3: EXAMPLE FOR ADDPROCESSING REQUEST	33
LISTING 4: EXAMPLE FOR ADDPROCESSING RESPONSE	34
LISTING 5: NEWLY CREATED SOS SENSOR FOR A PROCESSING	36
LISTING 6: EXAMPLE FOR GETSENSORLIST RESPONSE	37
LISTING 7: EXAMPLE OF GETSENSOR RESPONSE	37
LISTING 8: SAMPLE R SMOOTHING SCRIPT	42
LISTING 9: SAMPLE R AGGREGATION SCRIPT	42
LISTING 10: ADDPROCESSINGREQUEST FOR SAMPLE AGGREGATION SCRIPT	43
LISTING 11: OWLIM SAIL SPRING BEAN CONFIGURATION	66
LISTING 12: SPRING OWLIM SAIL WRAPPER CLASS	66
LISTING 13: GEOSPARQL QUERY "POINT QRY"	68
LISTING 14: GEOSPARQL QUERY "POLYGON QRY"	68
LISTING 15: SPARQL QUERY TO KNOW THE INFLUENCE IN THE 'ABRERATRANSFORMATION'	71
LISTING 16: SPARQL QUERY TO KNOW ASSOCIATED OBSERVATION WITH 'SVHGAUGE STATION'	72
LISTING 17: SPARQL QUERY TO KNOW THE FEATURE OF INTEREST OBSERVED BY THE 'DWP2FORECASTEDENERGYCONSUMPTION'	72
LISTING 18: SAMPLE GETOBSERVATION REQUEST	74
LISTING 19: SAMPLE WFS RESPONSE	77
LISTING 20: WFS-T TRANSACTION	78
LISTING 21: OUTPUT IN THE FORM OF WML 2.0	85
LISTING 22: RESPONSE FROM SOS FOR SENSOR REGISTRATION	86
LISTING 23: REQUEST TO REGISTER A SENSOR	87
LISTING 24: REQUEST TO SOS FOR INSERTOBSERVATION	88
LISTING 25: RESPONSE FROM SOS	88
LISTING 26: TEST SOS SERVER AFTER INSTALLATION - STEP 3	102
LISTING 27: SAMPLE ACA-PIM CONFIGURATION	106
LISTING 28: WDW SERVICE CONFIGURATION EXAMPLE	107

1. Introduction

Figure 1 gives an overview over the different layers of the WatERP architecture and especially how the layers interact in order to offer needed knowledge to the water managers. A more detailed description of the WatERP architecture has been given in D2.3-“Open Interface Specification” chapter 3-“WatERP architecture”. At summary, the architecture is formed by three layers: (i) Pilot Data Integration that is focused on transferring the pilot’s information to the WatERP platform in WaterML2 format; (ii) WatERP framework managed by the SOA-MAS architecture and where the pilot’s information is enhanced by metadata information (WatERP ontology) and served for the building blocks (WDW); and (iii) Building Blocks Integration that is focused on discovering building blocks need and provide required information to execute the different processes (DSS, DMS, etc.).

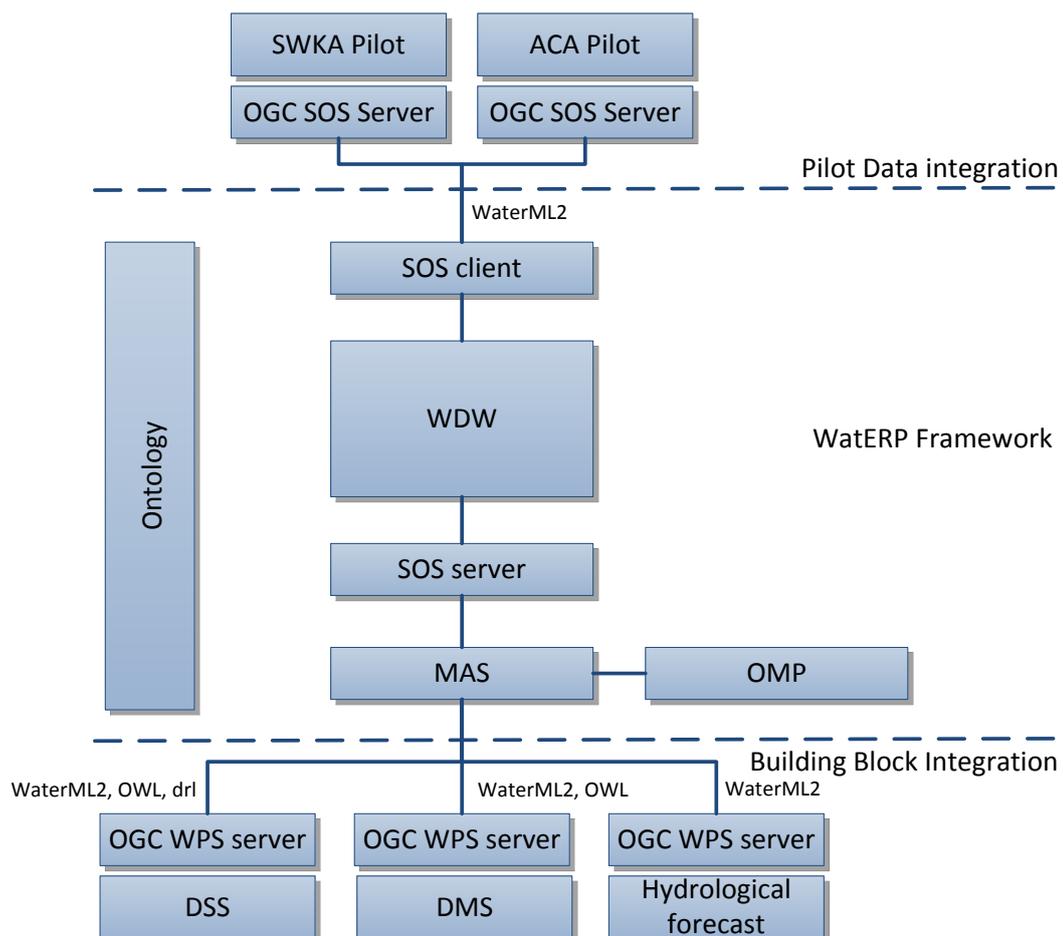


Figure 1: WatERP architecture from the integration point of view

The water data warehouse (WDW) plays an important, central role in order to gather, store and provide pilot's information to the rest of building blocks and to the water managers (through the OMP). In detail, its main function is to work as a reliable and durable data basis for all other components, providing both data needed for analyses or other functions and automated routines to incorporate new data sources and datasets at any time.

In order to create a completely interoperable solution, all data transfer and data access functions have been realized using OGC¹ standards or candidate standards. This guarantees flexible adaptation and expansion of the system setup as well as easy integration of new components, also making provisions for the evolution of those standards. These mentioned standards comprise the Sensor Observation Service (SOS), WaterML2, Web Map Services (WMS), Web Feature Service (WFS) and WFS-Transactional (WFS-T). The Sensor Observation Service (SOS)² together with WaterML2³ is used for all time series data exchange within the WatERP project – it is used both for data import in the WDW and for data release. The WMS⁴, WFS⁵ and WFS-T services have been implemented as WDW interfaces along with SPARQL⁶/GeoSPARQL⁷ interfaces in order to facilitate and provide to the water manager innovative semantics-based forms of data access.

All software code of the WDW has been implemented in Java, making the WDW-Server implementable on all systems being able to run a PostgreSQL⁸ installation. Internal realization of the WDW has been based on well-established software-design patterns and available libraries that grants for a maximum reusability potential. PostgreSQL as widely used, open source, free to use database system has been a sustainable choice of technology, facilitating the reuse of WatERP project results by easy implementation and maintenance at no further costs. As SOS-server, the 52North SOS-Server⁹ implementation has been chosen, taking into account its extensive documentation, open sources and good experiences by many users like CSIRO Tasmanian ICT Centre, granting for technological reuse in the SMART Aquifer Characterization Program¹⁰.

Internally, the prototype is structured in three layers (see Figure 2), realizing different aspects of the WDW. The lowest level labeled **pilot sites** realizes data integration through a Pilot Integration Manager

¹ OGC: <http://www.opengeospatial.org/>

² Sensor Observation Service: <http://www.opengeospatial.org/standards/sos>

³ OGC WaterML: <http://www.opengeospatial.org/standards/waterml>

⁴ Web Map Service: <http://www.opengeospatial.org/standards/wms>

⁵ Web Feature Service: <http://www.opengeospatial.org/standards/wfs>

⁶ SPARQL: <http://www.w3.org/TR/rdf-sparql-query/>

⁷ GeoSPARQL: <http://www.opengeospatial.org/standards/geosparql>

⁸ PostgreSQL: <http://www.postgresql.org/>

⁹ 52° North SOS: <http://52north.org/communities/sensorweb/sos/>

¹⁰ SMART: www.smart-project.info

(PIM), the intermediary **data warehouse layer** organizes all checks on data, aggregations and data optimizations, implements the persistent data storage as well as querying and reasoning algorithms, while the **interface layer** serves all external data requests using OGC-standards. All layers depend on the ontological information describing the hydrological domain, being accessible through uSeekM¹¹, providing geospatial (GeoSPARQL) search capabilities to Sesame¹² through an IndexingSail¹³.

Data integration has been based on ontological information. Whenever a new entity containing datasets is being registered in the ontology, it can be processed by the pilot integration manager and being registered for data access with the WDW-Service. If this entity already offers SOS WaterML2 services, these are being used. If there is no SOS WaterML2 service available, the dataset(s) of this entity are being registered as sensors within the pilot own SOS-server, and the pilot data is being read from their sources, transformed to WaterML2 in XML format and fed into the SOS-server.

Data aggregation, smoothing and checks within the data warehouse layer are also based on ontological information, in combination with additional metadata offered by all data consuming entities within the WatERP setup. Depending on the type of entity represented and metadata-information on timescales needed for this type of entity, input data is being loaded in regular intervals from the sources and further transformed according to the required processing. Each processing level is registered as sensor to be served with the WDW interface layer, along with the raw data undergoing no modification.

Data access through OGC compliant services is being realized by the WDW interface layer, once again making use of the ontological information within the triple store. Each pilot sensor is being registered with the WDW SOS-server with any given number of derived observation data, depending on the number of processings that are being requested for this specific entity. Additionally, corresponding WMS and WFS services have been implemented.

The components in charge of managing and providing data for the analysis clients through the multi agent system can be divided into three layers as depicted in Figure 2. These components correspond with (i) pilots sites where ACA and SWKA information is selected and stored in SOS by PIM process;(ii) Data warehouse layer that is aimed of storing and cleaning data from PIM process;(iii) Interface layer that is focused on making data accessible for the rest of WatERP architecture. Hence, the next sections provide a detailed description of the layers and how they are implemented.

¹¹ uSeekM: <https://dev.opensahara.com/projects/useekm/>

¹² Sesame: <http://www.openrdf.org/>

¹³ IndexingSail: <https://dev.opensahara.com/projects/useekm/wiki/IndexingSail>

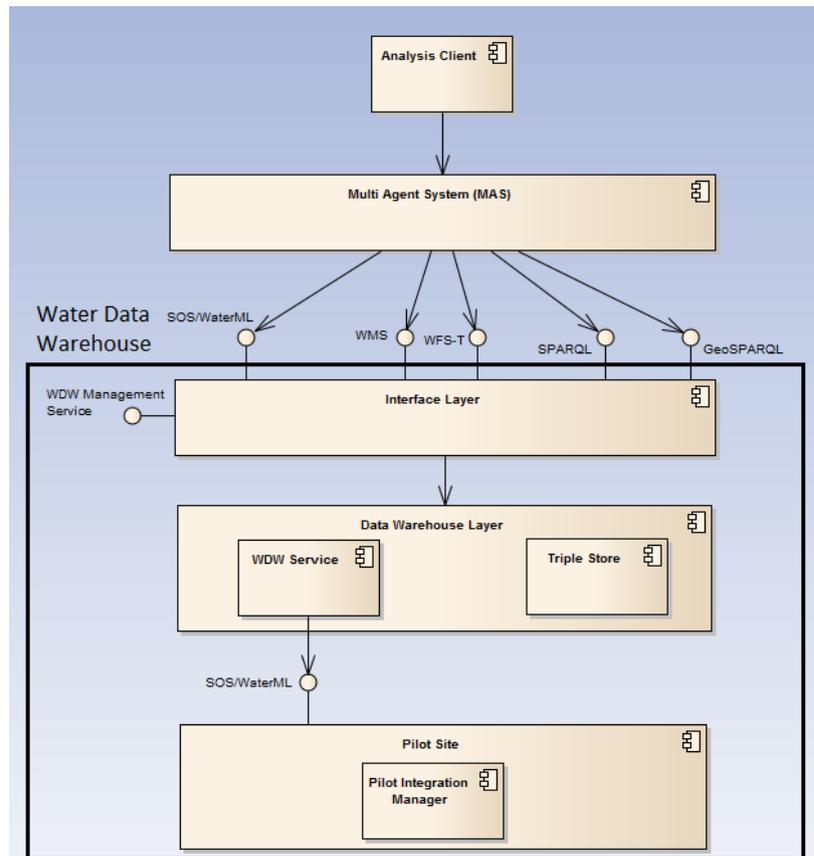


Figure 2: WDW component layers

Referring to these layers depicted in Figure 2, this document is structured as follows:

- Chapter 2 presents the core functionalities of the central Data Warehouse Layer
 - o Section 2.1 explains the WDW service for overall WDW manmanagement and storage of time-series data and sketches the basic mechanism for embedding R scripts into the WDW infrastructure.
 - o Section 2.2 elaborates on the Basic Data Mining Operations which employ R scripts for the implementation of data pre-processing steps for enhancement of data quality and for efficient support of data analytics in other WatERP workpackages.
 - o Section 2.3 describes the realization of the triple store for storage and processing of semantic knowledge thus supporting geospatial and ontological reasoning on observation metadata within the WDW.
- Chapter 3 explains the WDW Interface Layer used to access data and knowledge stored in the WDW. It contains the descriptions of the SOS/WaterML2 interface for observation data transmission, the WDW Management for controlling the WDW, and the OGC-compliant WFS/WFS-T and WMS interfaces (the SPARQL/GeoSPARQL interface for semantic queries is covered by Section 2.3 about the triple store.).

- Chapter 4 sketches the Pilot Integration Layer which is needed to couple an existing data infrastructure with the WDW. Both the general approach and the concrete example instantiations for the two WatERP pilots (SWKA, ACA) are discussed.

After these technical chapters, Chapter 5 provides an instruction how to set up a WDW installation and Chapter 6 summarizes and concludes this document.

2. Data Warehouse Layer

The Data Warehouse Layer consists of two main areas for data persistency. On one hand, a “conventional” (object-relational) geo database is used for managing time-series data (see Section 2.1 “WDW Service”). On the other hand, a triple store holds the pilot-specific implementation of the hydrological ontology thus allowing for semantic reasoning about observation metadata (see Section 2.3 “Triple Store”). Finally, an important amendment of the Data Warehouse Layer is the possibility to use R scripts for data pre-processing and time-series manipulation (see Section 2.2 “Basic Data Mining Operations”).

2.1 WDW Service

2.1.1 Functional Overview

The diagram depicted in Figure 3 shows the interactions of WDW Service with other elements of the WDW implementation.

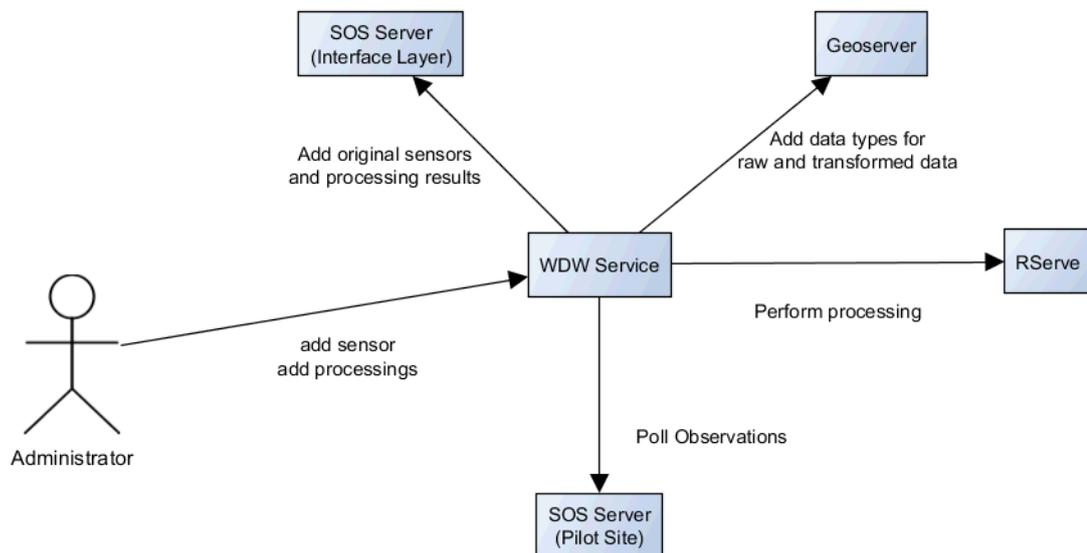


Figure 3: Interactions of the WDW Service with other WDW elements

Through an administration interface an Administrator can add sensors to the WDW Service. Thereby, the administrator provides the source where the sensor observation results can be retrieved and the parameters to query the sensor. The poller configuration is then adjusted and the database prepared to store the sensor raw data.

Through the same interface a processing can be defined. Here it is possible to specify an R script together with a list of parameters that are passed to the script.

For the sensors added to the WDW Service, the measured data is polled in defined intervals. The raw data is persisted, the requested processing steps are performed and the results are also stored. Both raw data and the processing results are published via Geoserver¹⁴ based WMS / WFS services and available to the corresponding WDW-MAS agent.

For each processing added to the WDW Service a sensor is added to a SOS Server at the interface layer to publish the transformed observation results as SOS/WaterML2 data. The creation of the metadata of the sensor representing the processing results is based on the WaterML2 description of the original sensor. *SensorId* and the *unit* of measured data are defined when the processing is added to the WDW Service.

2.1.2 Technical Overview

2.1.2.1 Basics

As Java is a platform and operating system independent programming language, which is also used in other open source reference implementations of OGC standards such as Geoserver and the 52°North SOS Server, the WDW Service is also implemented in Java.

The System is packaged as a Web Archive (WAR)¹⁵. It requires a servlet container to run the WDW Service. For the installation of the prototype, the Apache Tomcat¹⁶ has been chosen as it is a proved, reliable and very common open source reference implementation in the Java Servlet Specification. For setting up the project itself, Apache Maven is used as state of the art project management framework. This way the project can easily be integrated into a Continuous Integration build and testing system such as Hudson¹⁷, Jenkins¹⁸, CruiseControl¹⁹ or Apache Continuum²⁰.

Inside the project the Spring-Framework²¹ is used as dependency injection framework. Dependency injection is a pattern which allows the programmer to inject objects into a class by using a container that is externally configured (often by an XML file), instead of letting the class directly instantiate the objects. As stated by Razina E. and Janzen D. (2007):

“Maintainability of a software product is a big problem that often consumes 60% to 80% of the software life cycle. This problem is familiar to software developers and has existed for years, with no sign of relief

¹⁴ Geoserver: <http://geoserver.org>

¹⁵ Oracle Java Servlet Technology: <http://www.oracle.com/technetwork/java/index-jsp-135475.html>

¹⁶ Apache: Apache Tomcat: <http://tomcat.apache.org/>

¹⁷ Hudson: <http://hudson-ci.org/>

¹⁸ Jenkins: <http://jenkins-ci.org/>

¹⁹ CruiseControl: <http://cruisecontrol.sourceforge.net/>

²⁰ Apache Continuum: <http://continuum.apache.org/>

²¹ Spring Framework: <http://spring.io/>

in sight. Even though a complete solution to this problem does not exist, ways to measure code and predict maintainability do exist. Some of the measures that predict maintainability are coupling and cohesion metrics. ... In order to truly measure maintainability and the effects of dependency injection we would have to construct a more controlled study. However, a trend of lower coupling in projects with higher dependency injection percentage (more than 10 %) was evident.

2.1.2.2 WaterML2 Integration

To process WaterML2, the XML content is first converted into an internal XML format. This way any future WaterML2 version can be easily adapted and processed in parallel with Version 1.1 and 2.0 without much impact on the existing java code. The conversion between WaterML2 and the internal data format is implemented in XSLT²² which means that the java code has no direct dependency to the WaterML2 scheme. In a second step the internal format is converted into the database schema. By this, the internal database structure is highly separated from the external interfaces making the architecture most flexible from the external interfaces it supports.

Figure 4 describes how the WaterML2 XML bytes are passed to the *GetObservationResponseParser* which uses XSLT to convert them into the Results binding object. In the following step the binding objects are converted into *ObservationResultsEntities* that are persisted in the database.

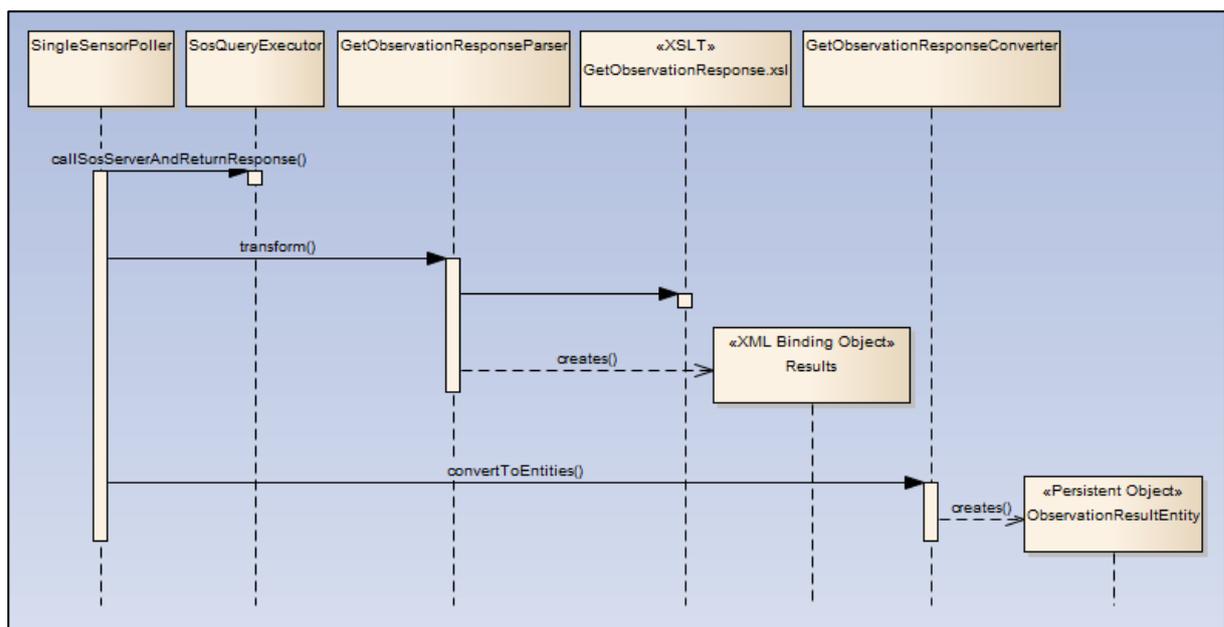


Figure 4: Conversion from WaterML2 to the database structure

The same is being performed when sending the measured data to the SOS Server in the interface layer. Figure 5 illustrates how the persisted data is transformed to the internal binding object

²² XSL Transformations (XSLT): <http://www.w3.org/TR/xslt20/>

ObservationData and, afterwards, an XSLT Transformation is processed and the result directly sent to the SOS Server.

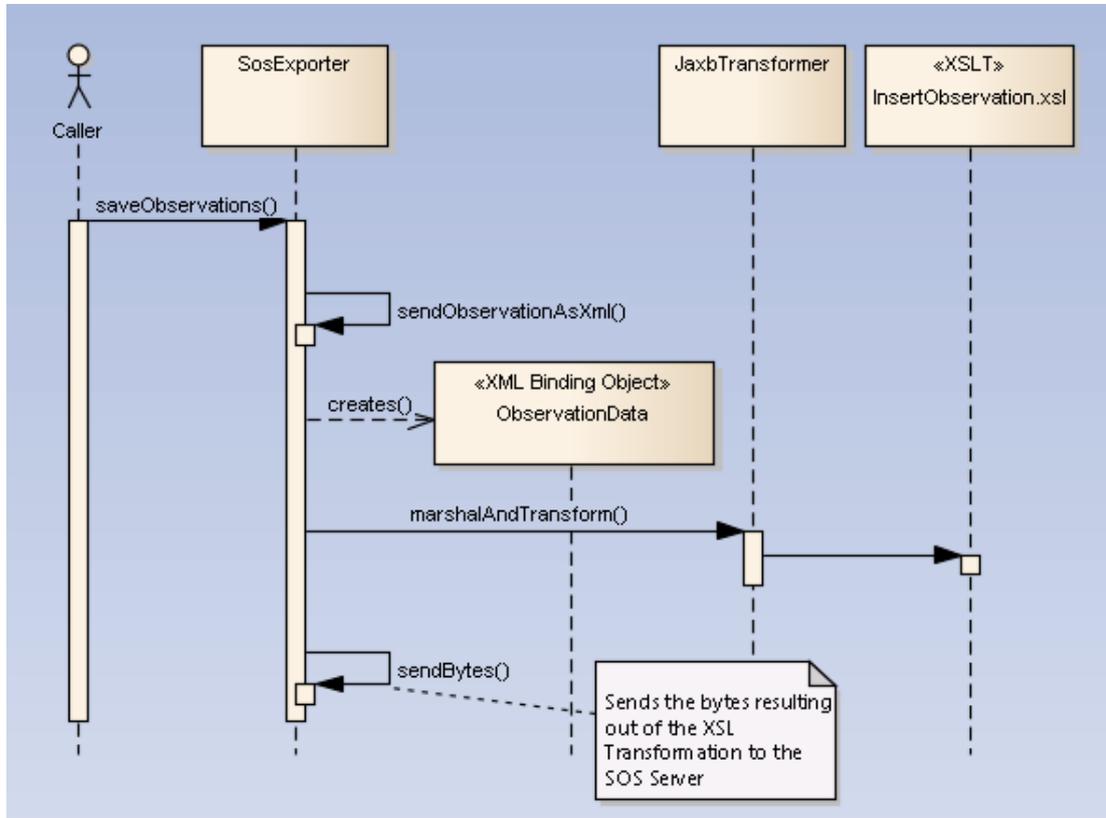


Figure 5: Conversion from database structure to WaterML2

2.1.3 Database Design and Persistence

Table 1 describes the setup of the generic database scheme used to store the sensor data within the WDW Service database. The configured sensors are stored in the *polled_sensor* table. For each sensor added to the WDW Service a table is automatically added to store the raw sensor data as it is loaded from the SOS Server.

Column	Description
Id	Primary key
sos_server_url	URL of the SOS server that provides the sensor's observation results
raw_data_table	Name of the table where the sensor data is stored
offering	Offering to specify the data within the SOS Server. An Observation Offering groups collections of observations produced by one procedure, e.g., a sensor system, and lists the basic metadata for the associated

	observations including the observed properties of the observations.
offering_name	Offering Name to specify the data within the SOS Server. This is used to create sensor descriptions for processing results.
foi	Feature of Interest to specify the data within the SOS Server. A Feature is an abstraction of real-world phenomena [OGC 10-004r3/ISO 19156].
sensor_id	Sensor-ID to specify the data within the SOS Server.
observed_property	Observed property to specify the data within the SOS Server. An observed property is a Facet or attribute of an object referenced by a name [OGC 10-004r3/ISO 19156] which is observed by a procedure.
highest_sos_timestamp	Highest timestamp read from the SOS Server.
next_polling_timestamp	Next polling timestamp.
start_polling_timestamp	Optional time for new sensors that limits the historical data that should be imported into the WDW
polling_interval_minutes	Requested polling interval in minutes.
unit	Measured unit of the observation results (UCUM ²³ Code).
srs_name	Name of the location's spatial reference system
location	Point coordinate that describes the location of the feature of interest
initialized	Indicates if the sensor has already been successfully polled and the infrastructure has been set up for that sensor. This includes defining the sensor in the SOS server as well as adding WFS and WMS types.

Table 1: Polled_sensor table columns

The processing steps which are requested by the clients are inserted into the *processing* table which is described in Table 2. For each entry, a separate table is generated to store the transformed data.

Column	Description
id	Primary key.
observation_id	Foreign key to the related polled sensor.
sensor_id	ID of the SOS-Sensor where the transformed sensor data is stored.

²³ Unified Code for Units of Measure: <http://unitsofmeasure.org/trac/>

sensor_description	Textual description for the sensor where the transformed data is stored.
resultTable	Name of the table where the transformed sensor data is stored.
rscript	Path of the R script that performs the processing.
unit	Measured unit of the transformed data.
initialized	Indicates if all initialisation steps like adding the sensor to WMS, WFS and SOS have been performed.

Table 2: Processing table columns

Every processing can have an unlimited number of user parameters. These parameters are passed to the R script together with a fixed set of system parameters that identify the data that has to be processed. This way the same R script can be reused for different sensors allowing pass sensor specific processing parameters. Table 3 lists the database columns of the parameter table.

Column	Description
id	Primary key.
processing_id	ID of the processing this parameter is attached to.
name	Parameter name.
value	Parameter value.

Table 3: Column description of the processing parameter table

The raw data tables and the transformed data tables all have the same structure as shown in Table 4.

Column	Description
identifier	ID of the sensor within the ontology.
timestamp	Timestamp of the time measured or transformed.
value	Measured or transformed value.

Table 4: Raw or transformed table columns

The entity-relationship diagram in Figure 6 describes the WDW database scheme.

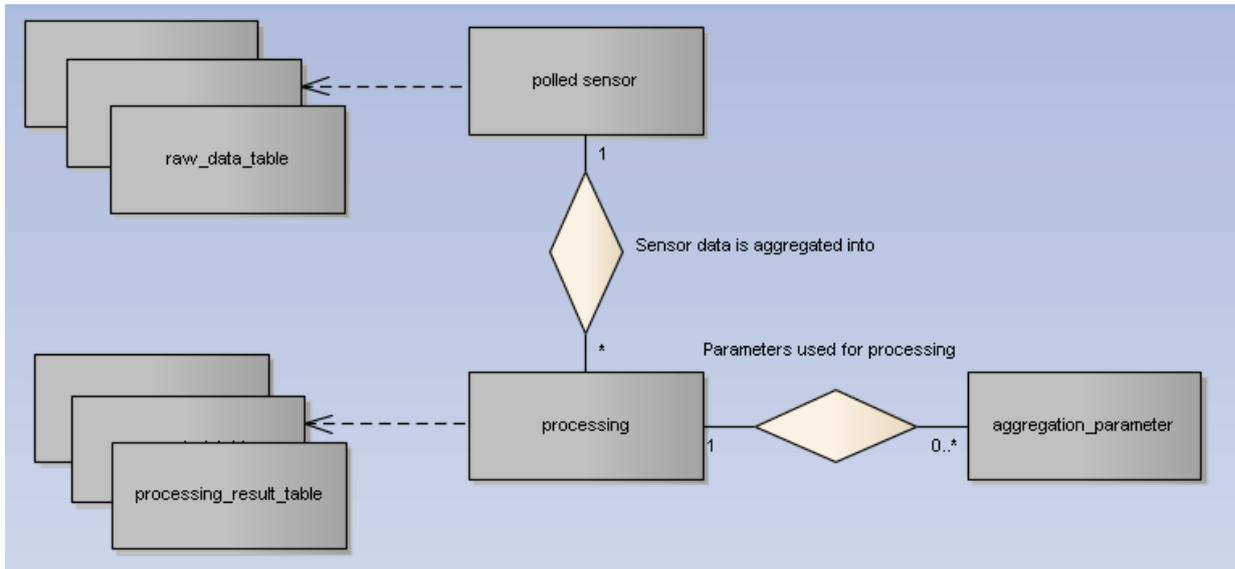


Figure 6: WDW entity-relationship diagram

The main concern on implementation of the persistence layer was to be database-independent. For the first integration the target database will be PostgreSQL. But if the WDW-Service will be used afterwards in the pilots' environment, it is very likely that it will be run on other database implementations as well. Therefore, for standard operations on the fixed tables *polled_sensor* and *processing*, Hibernate²⁴ is used to access the database. For the dynamic tables where measured and transformed values are stored, Hibernate can't be used because it is based on fixed table names. Therefore, the statements to create the tables or insert, delete and select data are kept external in SQL-Files. They are stored within a folder which name is the name of the Hibernate database dialect. Currently, PostgreSQL and HypersonicSQL²⁵ are supported, but other database dialects can be added without changing the Java code.

The dialects shown in Table 5 are supported by Hibernate and can easily be adopted²⁶:

Database	Dialect
DB2	DB2Dialect
DB2 AS/400	DB2400Dialect
DB2 OS390	DB2390Dialect

²⁴ Hibernate: <http://www.hibernate.org/>

²⁵ HypersonicSQL: <http://hsqldb.sourceforge.net/index.html>

²⁶ Hibernate Configuration: <http://docs.jboss.org/hibernate/core/3.3/reference/en-US/html/session-configuration.html#configuration-optional-dialects>

PostgreSQL	PostgreSQLDialect
MySQL	MySQLDialect
MySQL with InnoDB	MySQLInnoDBDialect
MySQL with MyISAM	MySQLMyISAMDialect
Oracle (any version)	OracleDialect
Oracle 9i	Oracle9iDialect
Oracle 10g	Oracle10gDialect
Sybase	SybaseDialect
Sybase Anywhere	SybaseAnywhereDialect
Microsoft SQL Server	SQLServerDialect
SAP DB	SAPDBDialect
Informix	InformixDialect
HypersonicSQL	HSQLDialect
Ingres	IngresDialect
Progress	ProgressDialect
Mckoi SQL	MckoiDialect
Interbase	InterbaseDialect
Pointbase	PointbaseDialect
FrontBase	FrontbaseDialect
Firebird	FirebirdDialect

Table 5: Database dialects supported by Hibernate

Currently, five SQL files have to be configured for each dialect (see Table 6):

File Name	Description
createDataScheme.sql	Creates a new table. '{0}' represents the actual table name. Example: <i>CREATE TABLE {0} ("IDENTIFIER" text, "TIMESTAMP" timestamp, "VALUE" numeric)</i>

addGeometryColumn.sql	<p>Adds a geometry column to an already existing table that has been created by createDataScheme.sql.</p> <p>Example:</p> <pre>SELECT AddGeometryColumn ("{0}", "geom", {1}, "POINT", 2);</pre>
deleteData.sql	<p>Deletes a row from a timestamp on. '{0}' represents the actual table name.</p> <p>Example:</p> <pre>DELETE FROM {0} WHERE "TIMESTAMP" >= ?</pre>
insertData.sql	<p>Inserts a new row. '{0}' represents the actual table name.</p> <p>Example:</p> <pre>INSERT INTO {0} ("IDENTIFIER", "TIMESTAMP", "VALUE", "geom") values (?, ?, ?, GeomFromEWKT (?));</pre>
selectData.sql	<p>Selects all rows from a specified time on. '{0}' represents the actual table name.</p> <p>Example:</p> <pre>SELECT "IDENTIFIER", "TIMESTAMP", "VALUE" FROM {0} WHERE "TIMESTAMP" >= ?</pre>

Table 6: SQL files per database dialect

2.1.4 Internal Processing

This section describes the main processing steps of the WDW Service and their implementation.

2.1.4.1 Add Sensor

Figure 7 shows how a sensor is added by the administrator to the WDW Service.

Firstly, a check is being performed to ensure that the sensor had not been added before. If the check fails, an error is sent to the client. Otherwise, the sensor is added to the polled observations and the table to store the raw data is created in the database. The table is then added to the Geoserver so that it is published as WFS or WMS type. To do so, the WDW Service calls a Geoserver REST endpoint that exports a database table as a WFS/WMS layer. With this second prototype, the raw sensor data is also published via the SOS, as MAS and analysis clients shouldn't depend on infrastructure that is designed to bind clients to the WDW.

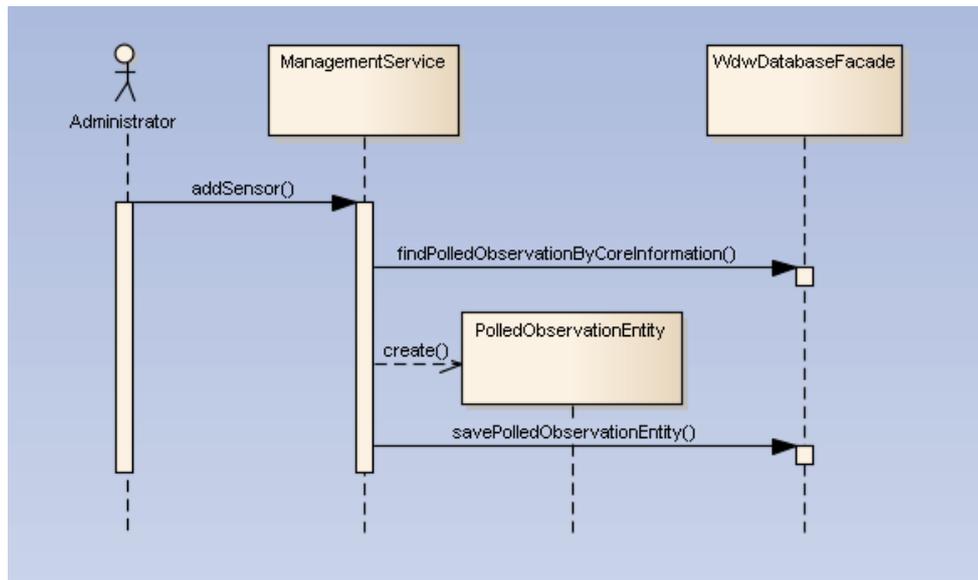


Figure 7: Add sensor diagram

2.1.4.2 Add Processing

Figure 8 shows how the administrator adds a processing for a registered sensor.

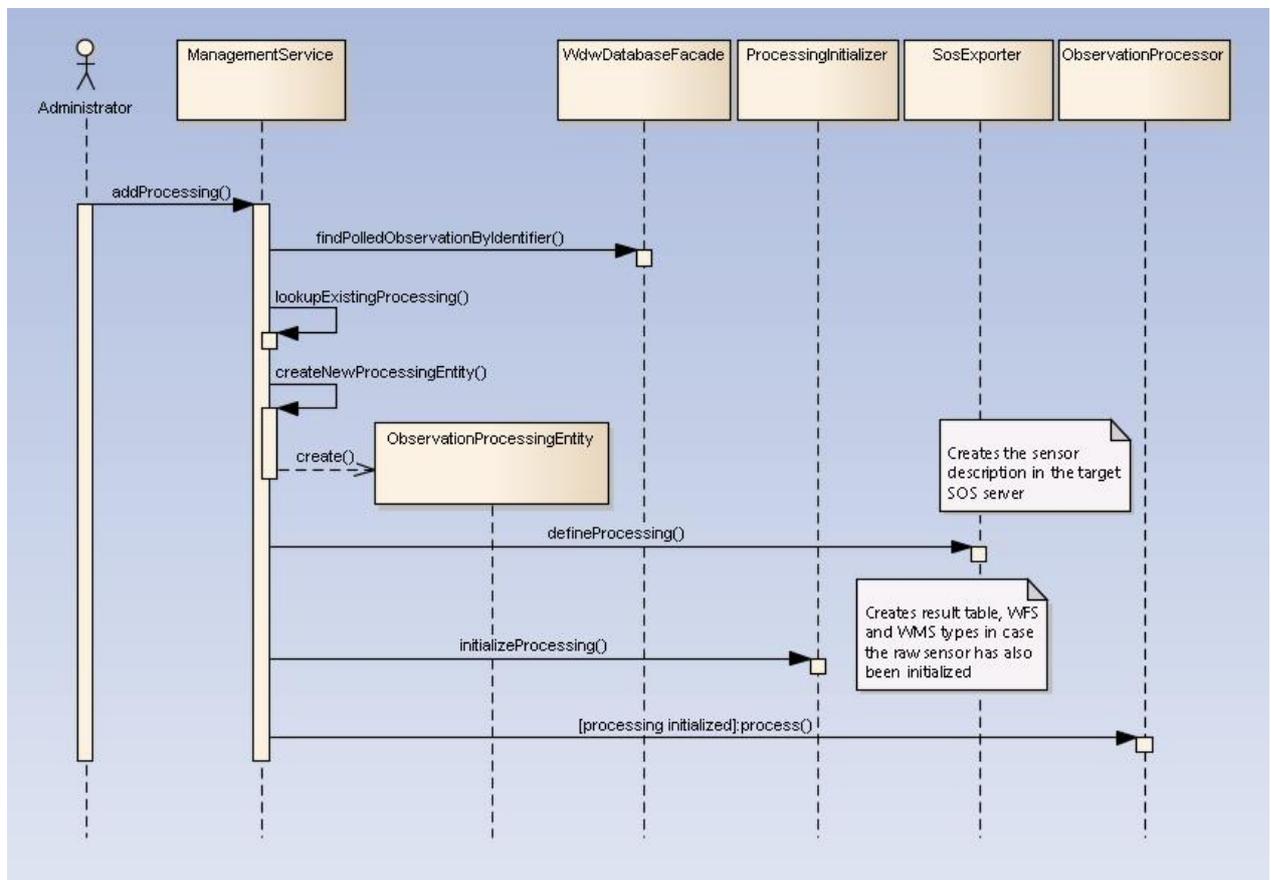


Figure 8: Add processing-sequence diagram

First, the existence of the sensor for which a processing should be added is checked in the database. If the sensor hasn't been added before, an error is sent to the client. Otherwise it's checked if that processing has already been added. In case the processing sensor id is found, the processing stops. Otherwise, the new processing is added to the processing table and the new sensor is added to the SOS server on the interface layer. Finally, it is checked if the processing initialization can already be finished. The processing can fully be performed if the base sensor has already been fully initialized. In this case the database table is created to store the transformed observation results and the new database table is added as a layer to the Geoserver to be accessible via WMS and WFS.

If the initialization has been successfully performed, the processing is triggered to process all data currently available for the underlying sensor.

2.1.4.3 Observation Result Polling

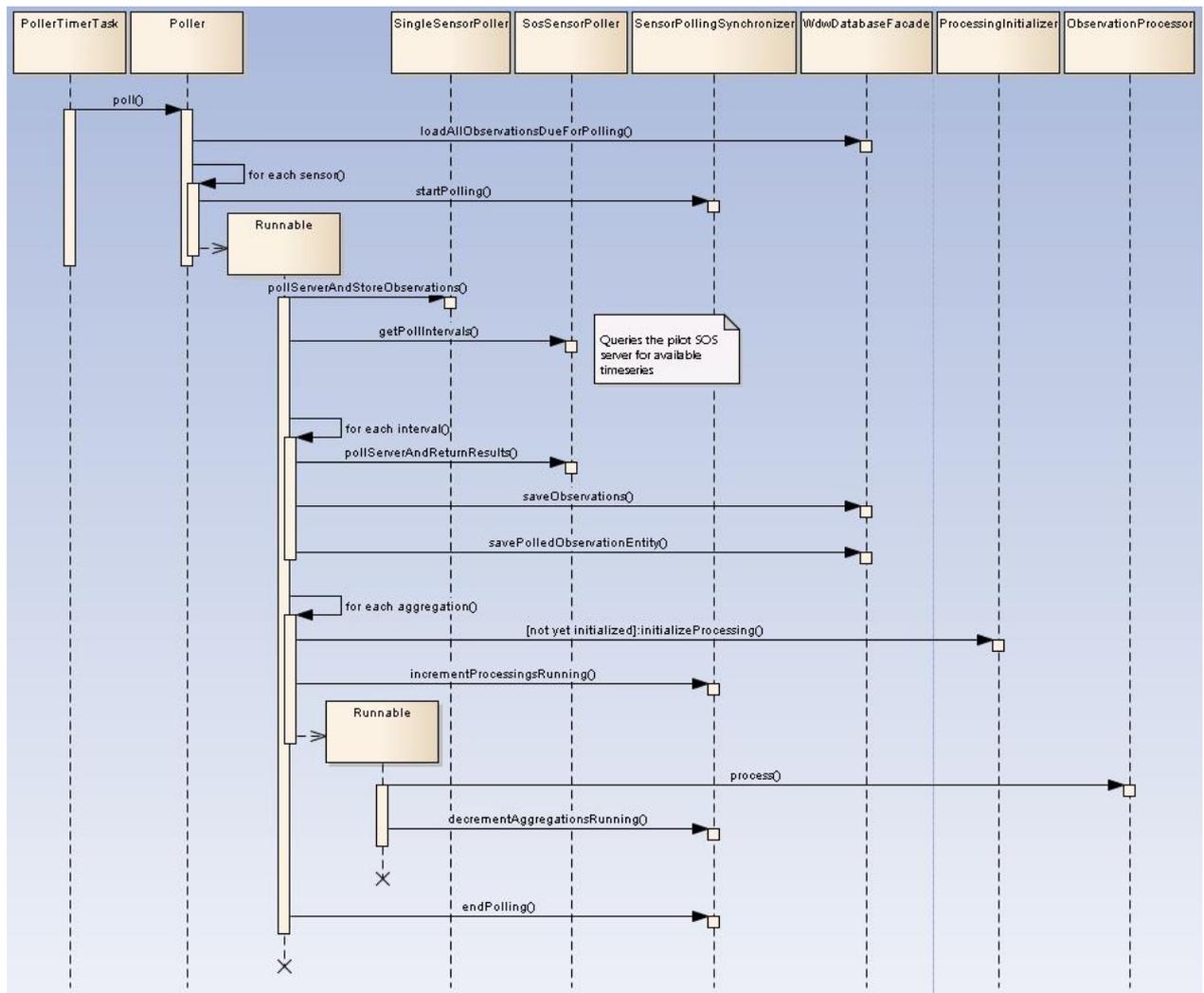


Figure 9: Poll SOS observations sequence diagram

In a configurable time interval, the WDW Service checks if there are sensors to poll for new observation results (Figure 9). For every sensor there can be a different polling interval. First, all the sensors are loaded which are due for polling.

The polling of the sensors is done in parallel threads. For each sensor, a Runnable is created and passed to an executor that will run up to a specific number of threads in parallel. The capacity of the thread executor can be adjusted without altering the code by changing the spring XML configuration.

For each sensor, first a list of time series is created to avoid querying too many time-series elements at a time. Thereby, the poller avoids destabilizing the SOS server or the WDW Service. Next, for each time-series interval, a query is sent to the SOS Server to retrieve the observation results. The response is converted to the internal persistence format and stored into the raw data table for that sensor. Afterwards, the processings are started. Again each processing is performed in a separate thread.

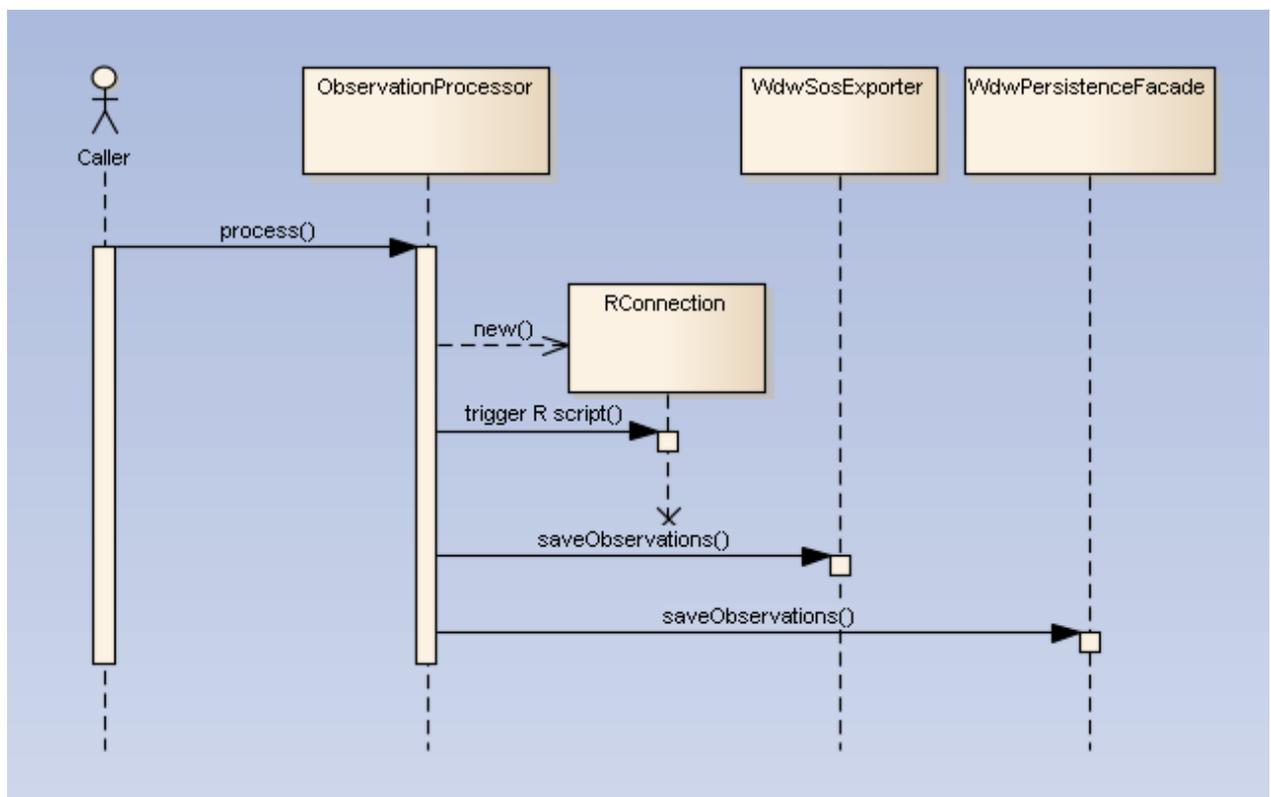


Figure 10: Perform processing sequence diagram

Due to the number of observations, short polling intervals and uncertainties related to network or database performance, it is possible that the polling of a sensor starts before the previous polling process has finished. Since this could cause serious problems with data integrity, the *SensorPolling-Synchronizer* is used to synchronize parallel processing of the same sensor. The current implementation is not capable of synchronizing the processing in a clustered environment as it only keeps its state in JAVA objects. To have a cluster wide synchronization, the state would have to be shared

between the servers running on separate servers. Alternatives to implement such a shared state could be to store the state in the database, use advanced techniques²⁷ to share object states. Of course, in a post-project exploitation phase, the WDW Service could be adapted such that clustering is possible..

Figure 10 shows that for each processing a connection to the RServe is created. Both system required parameters to access the source data and user parameters to make context-specific alterations to the R script, are passed to the remote session. Next, the R script is processed. The result arrays containing the timestamps and the values are retrieved and both stored into the Oracle table for WFS/WMS and inserted into the SOS server to be available over the SOA-MAS architecture. As the SensorML protocol only allows inserting and querying but not updating observation results, the WDW Service inserts the data via JDBC into the SOS Server database. Only the first value of a processing is sent via XML to the SOS Server. Then the SOS Server also updates other tables. For example, the *Feature of Interest* is inserted in case it is the first processing of this sensor. With this approach, minimum dependency to the actual SOS implementation is created. Only future changes observations persistence will affect the implementation of the WDW Service.

²⁷ Tech Spot Singleton in a cluster: <http://www.techspot.co.in/2009/07/singleton-in-cluster.html>

2.1.5 Management Service

The management service is an XML based REST service which is exported by the WDW Service. Figure 11 displays all XML elements and types used to configure the WDW Service via REST. The following subsections describe the different actions that can be triggered.

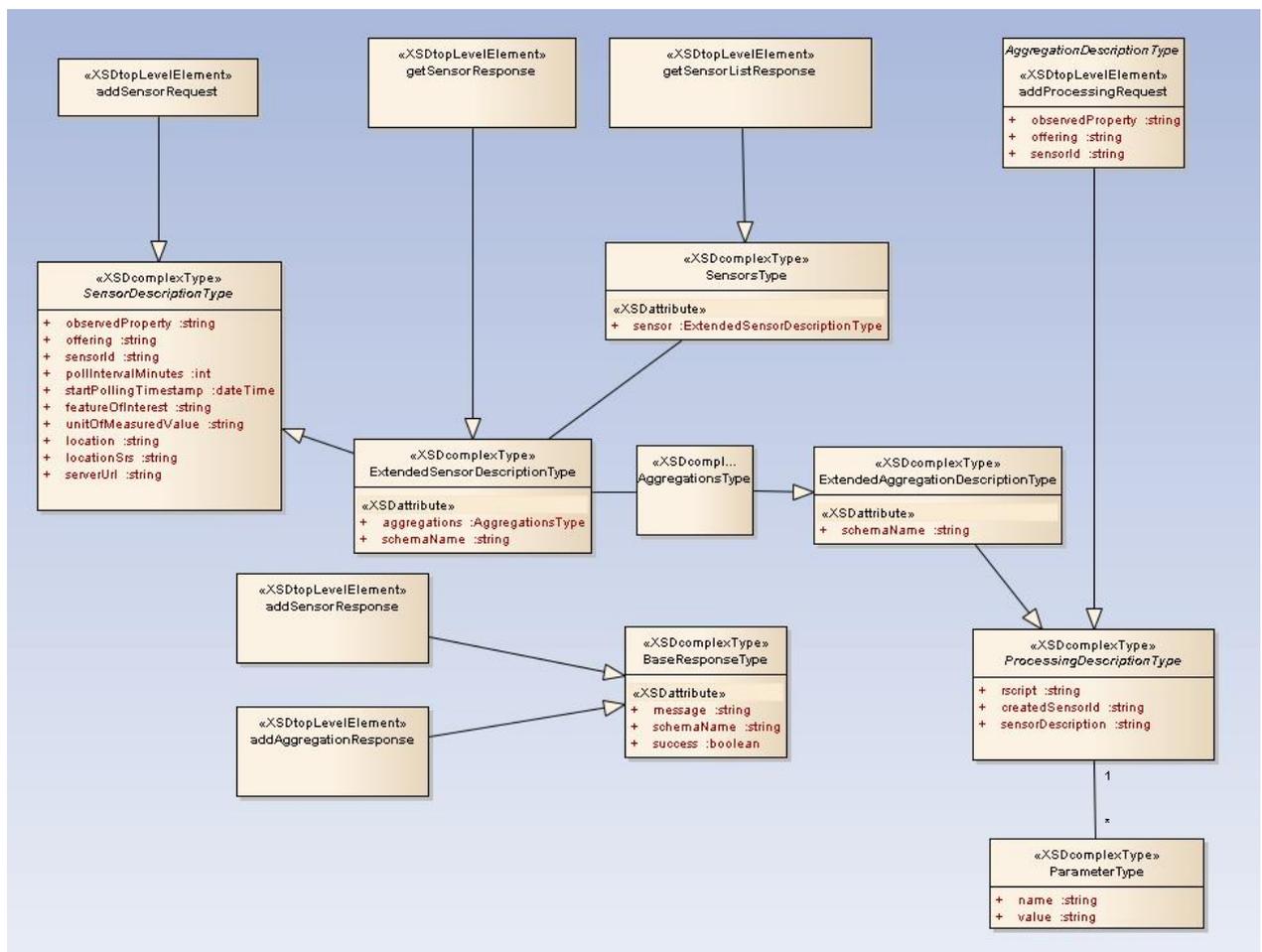


Figure 11: Management service XML scheme

2.1.5.1 Add Sensor

To add a sensor, the PIM must send a POST request to the management endpoint of the WDW Service passing an *addSensorRequest* message. Table 9 describes the parameters of the request. Listing 1 and Listing 2 show a sample request and response.

Parameter:

Name	Description
pollIntervalMinutes	Requested polling interval in minutes.
observedProperty	Observed property to specify the data within the SOS Server. An observed property is a Facet or attribute of an object referenced by a name [OGC 10-004r3/ISO 19156] which is observed by a procedure.
offering	Offering to specify the data within the SOS Server. An Observation Offering groups collections of observations produced by one procedure, e.g., a sensor system, and lists the basic metadata for the associated observations including the observed properties of the observations.
sensorId	Sensor-ID to specify the data within the SOS Server.
serverUrl	URL of the SOS Server where the observation results should be polled.

Table 7: Parameters for addSensorRequest

Example:

POST to *http://localhost:8080/wdw-service/management/sensor*

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:addSensorRequest xmlns:ns2="http://waterp/wdw/ManagementService">
  <observedProperty>urn:ogc:def:phenomenon:waterpressure</observedProperty>
  <offering>WATERPRESSURE_._urn:ogc:object:feature:Sensor:DEMO:sensor-1554</offering>
  <sensorId>urn:ogc:object:feature:Sensor:DEMO:sensor-1554</sensorId>
  <pollIntervalMinutes>1</pollIntervalMinutes>
  <ns2:sosSourceDescription>
    <serverUrl>http://localhost:8091/pilotsos/sos</serverUrl>
  </ns2:sosSourceDescription>
</ns2:addSensorRequest>
```

Listing 1: Example for addSensor request

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:addSensorResponse xmlns:ns2="http://waterp/wdw/ManagementService">
  <success>true</success>
  <schemaName>raw1395315704066</schemaName>
</ns2:addSensorResponse>
```

Listing 2: Example for addSensor response

Resulting Database State:

When adding a sensor to the WDW Service a row is added to the *poller_configuration* table and a *raw data table* is created to store the measured data (Figure 12).

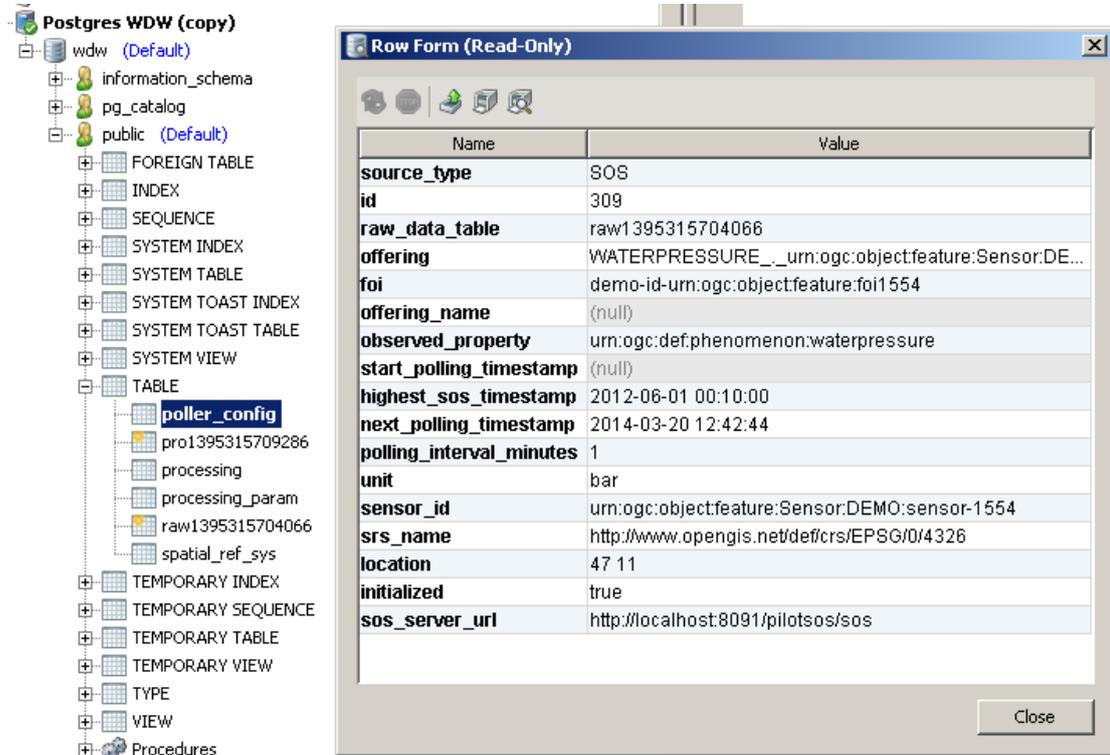


Figure 12: Database content after sensor creation

Resulting Geoserver content:

The raw data table is automatically added as a layer to the Geoserver to be available as WMS and WFS. To do so, the WDW Service calls a Geoserver REST endpoint that exports a database table as a WFS/WMS layer. Figure 13 shows the Geoserver web front-end that displays the raw data layer which was automatically created by adding a sensor to the WDW Service.

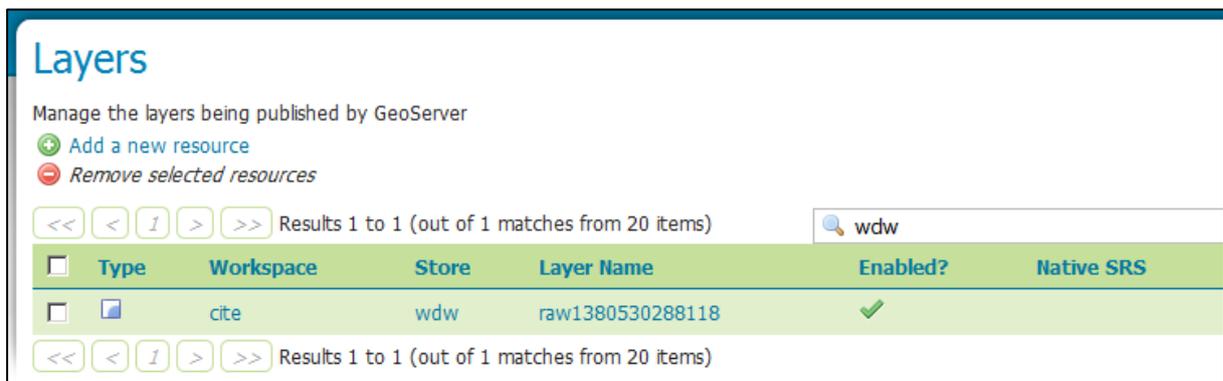


Figure 13: Geoserver content after sensor creation

2.1.5.2 Add Processing

To add a processing, the client must send a POST request to the management endpoint of the WDW Service passing an *addProcessingRequest* message. The message parameters are described in Table 8: Parameters for addProcessingRequest. Then, Listing 3 and Listing 4 show an example request and response.

Parameter:

Name	Description
Rscript	Path of the R script to process.
createdSensorId	ID of the sensor that will store the transformed time series in the SOS server.
sensorDescription	Description of the sensor that will store the transformed time series in the SOS server.
Unit	Measured unit of the resulting time series.
observedProperty	Observed Property of the underlying sensor
Offering	Offering of the underlying sensor
sensorId	Sensor ID of the underlying sensor.
Parameter	List of key –value pairs that are passed to the R script.

Table 8: Parameters for addProcessingRequest

Example:

POST to <http://localhost:8080/wdw-service/management/processing>

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:addProcessingRequest xmlns:ns2="http://waterp/wdw/ManagementService">
  <rscript>rscript.r</rscript>
  <createdSensorId>urn:ogc:object:feature:Sensor:DEMO:sensor-1554/created
</createdSensorId>
  <sensorDescription>urn:ogc:object:feature:Sensor:DEMO:sensor-1554
</sensorDescription>
  <unit>kbar</unit>
  <parameter>
    <name>pname1</name>
    <value>pvalue1</value>
  </parameter>
  <observedProperty>urn:ogc:def:phenomenon:waterpressure
</observedProperty>
  <offering>WATERPRESSURE_._urn:ogc:object:feature:Sensor:DEMO:sensor-1554
</offering>
  <sensorId>urn:ogc:object:feature:Sensor:DEMO:sensor-1554</sensorId>
</ns2:addProcessingRequest>
```

Listing 3: Example for addProcessing request

Response:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:addProcessingResponse xmlns:ns2="http://waterp/wdw/ManagementService">
  <success>true</success>
  <schemaName>pro1395315709286</schemaName>
</ns2:addProcessingResponse>
```

Listing 4: Example for addProcessing response

Resulting Database State:

When a processing is added to the WDW Service, a row is added to the processing table and a table is created to store the transformed data (Figure 14).

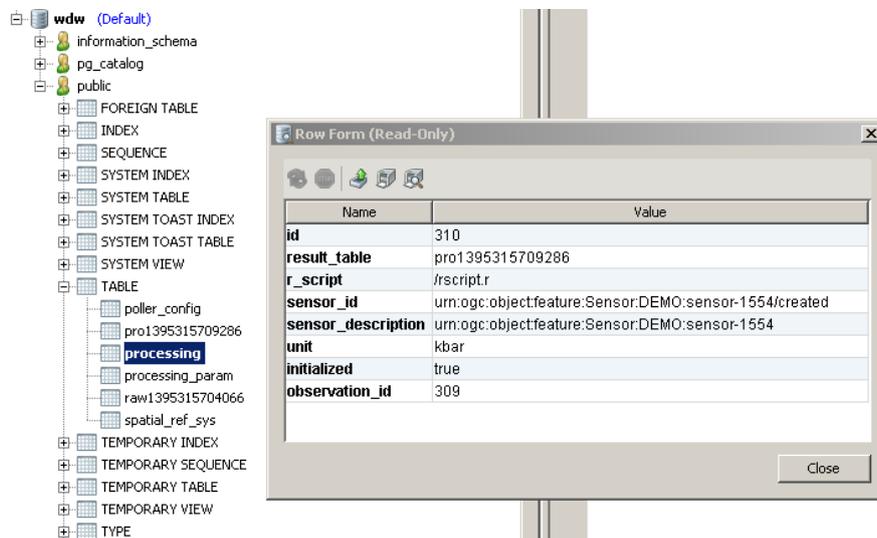


Figure 14: Database content after processing creation

Resulting Geoserver content:

Like new raw data tables, also the transformed data table is added to the Geoserver, and by this, it is accessible for WMS and WFS. Figure 15 shows the raw and the transformed data table as layers in the Geoserver web front-end.

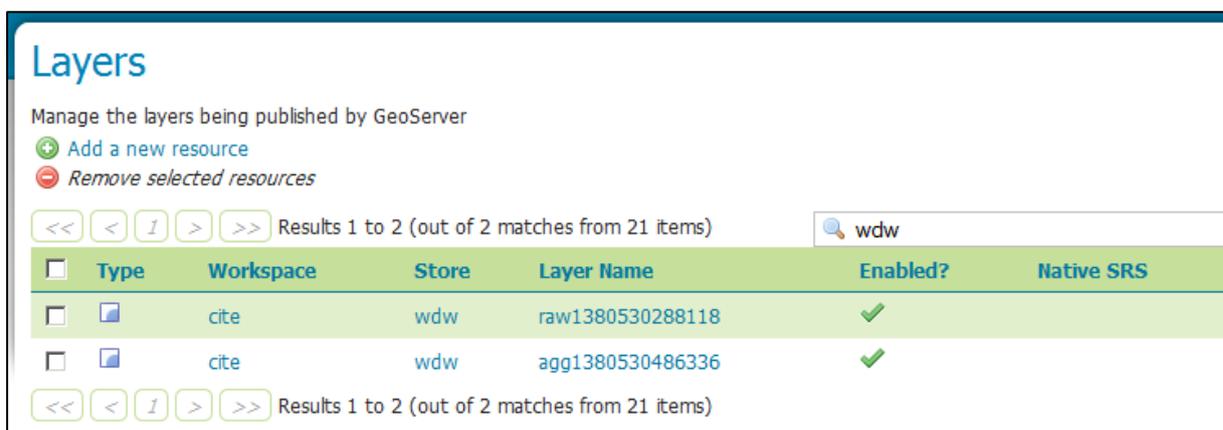


Figure 15: Geoserver content after processing creation

Resulting SOS Sensor:

The transformed data is added to an SOS Server as an additional sensor. Listing 5 shows the query result on the newly created SOS layer.

```
<?xml version="1.0" encoding="UTF-8"?>
<sos:GetObservationResponse xmlns:sos="http://www.opengis.net/sos/2.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:om="http://www.opengis.net/om/2.0"
  xmlns:gml="http://www.opengis.net/gml/3.2" xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:wml2="http://www.opengis.net/waterML/2.0"
  xmlns:sf="http://www.opengis.net/sampling/2.0"
  xmlns:sams="http://www.opengis.net/samplingSpatial/2.0"
  xsi:schemaLocation=" http://www.opengis.net/sos/2.0
  http://schemas.opengis.net/sos/2.0/sos.xsd http://www.opengis.net/om/2.0
  http://schemas.opengis.net/om/2.0/observation.xsd http://www.opengis.net/sampling/2.0
  http://schemas.opengis.net/sampling/2.0/samplingFeature.xsd
  http://www.opengis.net/samplingSpatial/2.0
  http://schemas.opengis.net/samplingSpatial/2.0/spatialSamplingFeature.xsd">
  <sos:observationData>
    <om:OM_Observation gml:id="o_1390233366510">
      <om:type xlink:href=
"http://www.opengis.net/def/observationType/waterML/2.0/measurementTVPTimeseriesObservation"
/>
      <om:phenomenonTime>
        <gml:TimePeriod gml:id="phenomenonTime_o_1390233366510">
          <gml:beginPosition>2012-06-01T00:01:00.000+02:00
          </gml:beginPosition>
          <gml:endPosition>2012-06-01T00:10:00.000+02:00</gml:endPosition>
        </gml:TimePeriod>
      </om:phenomenonTime>
      <om:resultTime xlink:href="#phenomenonTime_o_1390233366510" />
      <om:procedure>
        <wml2:ObservationProcess gml:id="process.o_1390233366510">
          <wml2:processType
            xlink:href="http://www.opengis.net/def/waterML/2.0/processType/Sensor"
            xlink:title="Sensor" />
          <wml2:processReference
            xlink:href="urn:ogc:object:feature:Sensor:DEMO:sensor-1554/created" />
        </wml2:ObservationProcess>
      </om:procedure>
      <om:observedProperty xlink:href="urn:ogc:def:phenomenon:waterpressure"
        xlink:title="" />
      <om:featureOfInterest>
        <wml2:MonitoringPoint gml:id="sf_1">
          <gml:identifier codeSpace="">demo-id-urn:ogc:object:feature:foi1554</gml:identifier>
          <gml:name codeSpace="" />
          <sf:type
            xlink:href="http://www.opengis.net/def/samplingFeatureType/OGC-
OM/2.0/SF_SamplingPoint" />
          <sf:sampledFeature xlink:href="urn:ogc:def:nil:OGC:unknown" />
          <sams:shape>
            <gml:Point gml:id="point_sf_1">
              <gml:pos srsName="http://www.opengis.net/def/crs/EPSG/0/4326">47 11</gml:pos>
            </gml:Point>
          </sams:shape>
        </wml2:MonitoringPoint>
      </om:featureOfInterest>
      <om:result>
        <wml2:MeasurementTimeseries gml:id="timeseries.3">
          <wml2:metadata>
```

```

<wml2:MeasurementTimeseriesMetadata>
  <wml2:temporalExtent xlink:href="#phenomenonTime_o_1390233366510" />
</wml2:MeasurementTimeseriesMetadata>
</wml2:metadata>
<wml2:defaultPointMetadata>
  <wml2:DefaultTVPMeasurementMetadata>
    <wml2:uom code="kbar" />
    <wml2:interpolationType
      xlink:href="http://www.opengis.net/def/timeseriesType/WaterML/2.0/continuous"
      xlink:title="Instantaneous" />
  </wml2:DefaultTVPMeasurementMetadata>
</wml2:defaultPointMetadata>
<wml2:point>
  <wml2:MeasurementTVP>
    <wml2:time>2012-06-01T00:01:00.000+02:00</wml2:time>
    <wml2:value>0.0045</wml2:value>
  </wml2:MeasurementTVP>
</wml2:point>
</wml2:MeasurementTimeseries>
</om:result>
</om:OM_Observation>
</sos:observationData>
</sos:GetObservationResponse>

```

Listing 5: Newly created SOS sensor for a processing

2.1.5.3 Query all Sensors

To query all sensors, the client has to send a GET request to the management endpoint without any special information. Listing 6 shows an example response.

Example:

GET to <http://localhost:8080/wdw-service/management/sensors>

Response:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:getSensorListResponse xmlns:ns2="http://waterp/wdw/ManagementService">
  <sensors>
    <sensor>
      <sensor>
        <observedProperty>urn:ogc:def:phenomenon:waterpressure</observedProperty>
        <offering>WATERPRESSURE_._urn:ogc:object:feature:Sensor:DEMO:sensor-1554</offering>
        <sensorId>urn:ogc:object:feature:Sensor:DEMO:sensor-1554</sensorId>
        <pollIntervalMinutes>1</pollIntervalMinutes>
        <featureOfInterest>demo-id-urn:ogc:object:feature:foi1554</featureOfInterest>
        <ns2:sosSourceDescription>
          <serverUrl>http://localhost:8091/pilotsos/sos</serverUrl>
        </ns2:sosSourceDescription>
        <unitOfMeasuredValue>bar</unitOfMeasuredValue>
        <location>47 11</location>
        <locationSrs>http://www.opengis.net/def/crs/EPSSG/0/4326</locationSrs>
        <highestTimestamp>2012-06-01T00:10:00.000+02:00</highestTimestamp>
        <schemaName>raw1395315704066</schemaName>
        <processings>
          <processing>
            <rscript>/rscript.r</rscript>
            <createdSensorId>urn:ogc:object:feature:Sensor:DEMO:sensor-1554/created</createdSensorId>

```

```

    <sensorDescription>urn:ogc:object:feature:Sensor:DEMO:sensor-
1554</sensorDescription>
    <unit>kbar</unit>
    <parameter>
      <name>pname1</name>
      <value>pvalue1</value>
    </parameter>
    <schemaName>pro1395315709286</schemaName>
  </processing>
</processings>
</sensor>
</sensors>
</ns2:getSensorListResponse>

```

Listing 6: Example for `getSensorList` response

2.1.5.4 Query one Sensor

To query one specific sensor the client has to send a GET request to the management endpoint and pass the identification of the sensor in the ontology that has been specified in the `addSensorRequest` as the identifier of the sensor. Listing 7 shows an example response.

Example:

GET to `/wdw-service/management/sensor?observedProperty=a&offering=b&sensorId=c`

Response:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:getSensorResponse xmlns:ns2="http://waterp/wdw/ManagementService">
  <observedProperty>b</observedProperty>
  <offering>b</offering>
  <sensorId>c</sensorId>
  <pollIntervalMinutes>1</pollIntervalMinutes>
  <featureOfInterest>demo-id-urn:ogc:object:feature:foi1554</featureOfInterest>
  <ns2:sosSourceDescription>
    <serverUrl>http://localhost:8091/pilotsos/sos</serverUrl>
  </ns2:sosSourceDescription>
  <unitOfMeasuredValue>bar</unitOfMeasuredValue>
  <location>47 11</location>
  <locationSrs>http://www.opengis.net/def/crs/EPSG/0/4326</locationSrs>
  <highestTimestamp>2012-06-01T00:10:00.000+02:00</highestTimestamp>
  <schemaName>raw1395315704066</schemaName>
  <processings>
    <processing>
      <rscript>/rscript.r</rscript>
    <createdSensorId>urn:ogc:object:feature:Sensor:DEMO:sensor-
1554/created</createdSensorId>
    <sensorDescription>urn:ogc:object:feature:Sensor:DEMO:sensor-1554</sensorDescription>
    <unit>kbar</unit>
    <parameter>
      <name>pname1</name>
      <value>pvalue1</value>
    </parameter>
    <schemaName>pro1395315709286</schemaName>
  </processing>
</processings>
</ns2:getSensorResponse>

```

Listing 7: Example of `getSensor` response

2.1.6 WDW Service Performance

It's critical that the WDW infrastructure can handle even huge time series. As one of the most critical parts of the WatERP infrastructure is the SOS server, especially this element must work most reliably. Therefore, a stress scenario has been created to ensure that the 52North SOS implementation in combination with WaterML2 meets the requirements. The strategy was to select an increasing amount of observation results from an SOS server installation. The data was queried in WaterML2 and CSV²⁸ format. Table 9 lists the response times in exact numbers and Figure 16 gives a graphical overview.

Number of observations	WaterML2 Response time (seconds)	CSV Response time (seconds)
270	0	0
498	0	0
1,633	0	0
7,604	1	0
19,712	6	0
30,281	13	0
72,169	148	4
128,329	290	5
187,369	475	7
203,209	449	7
340,144	1,360	10
427,984	2,004	12
990,223	7,908	26

Table 9: 52North SOS server response times

²⁸ http://en.wikipedia.org/wiki/Comma-separated_values

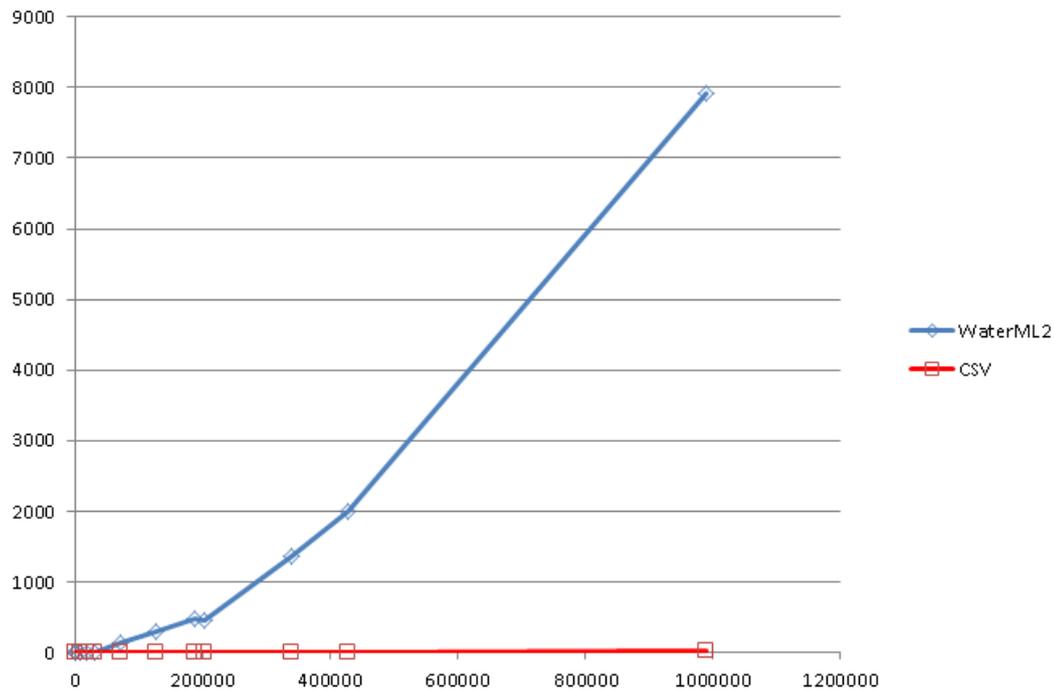


Figure 16: 52North SOS server response times

Figure 16 illustrates the difference between the response times that is bigger when more observations have to be returned. The reason for this gap is that the XML transformation to WaterML2 is extremely expensive. 52North uses Apache XMLBeans²⁹ to create the response document. XMLBeans allows the developer to work on XML binding object very much like JAXB³⁰ which is the most often used standard for XML bindings. The downside of this approach is that all objects are being kept in memory until the preparation of the entire XML document is completed. Then, the whole structure is serialised to XML and sent to the client. This means that an extreme amount of objects have to be kept for a long period of time binding a lot of memory. Besides the time the creation of the object tree requires, the occupation of so much storage also increases the frequency of garbage collection, especially triggering full garbage collections. Figure 17 shows the memory usage when 52North creates responses with large time series. The consumed memory is increasing steadily. Every time the curve falls it shows the effect of a garbage collection call. But as the amount of available free memory decreases, the virtual machine has to trigger the garbage collector in ever shorter intervals. The curve shows that when the size of the time series increases, the time spent on garbage collection increases over-proportionally.

²⁹ <http://xmlbeans.apache.org/>

³⁰ <http://www.oracle.com/technetwork/articles/javase/index-140168.html> Java Architecture for XML Binding

The problem is intensified by the fact that the 52North server is designed to handle several requests at a time requesting such big amounts of data. This can cause serious problems as several threads consuming such amounts of space will definitely cause the virtual machine to run out of memory.

In contrast, the CSV transfer works extremely fast and shows nearly no impact in the memory usage. This is because every row read from the database is directly transformed into an output line and sent back to the client.

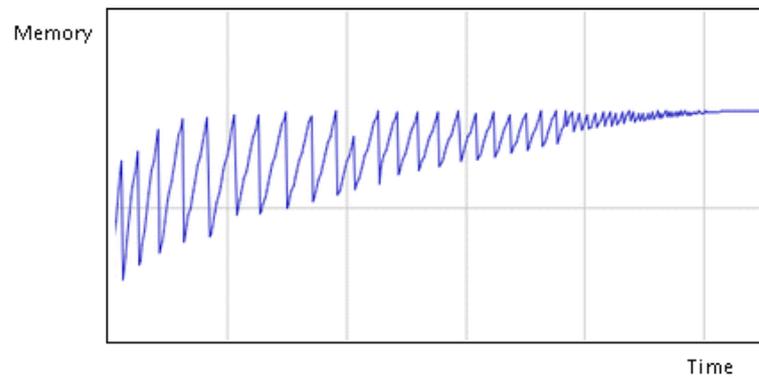


Figure 17: Memory usage when creating responses with large time series

There are two solutions to these problems. One is to use CSV instead of WaterML2. As WaterML2 is one of the central protocols for the WatERP setup this approach has not been further followed. The other solution is to restrict the size of the time series requested from the SOS server. Figure 18 shows the response times with linear progression which are the result if the client sends many small requests instead of one big request.

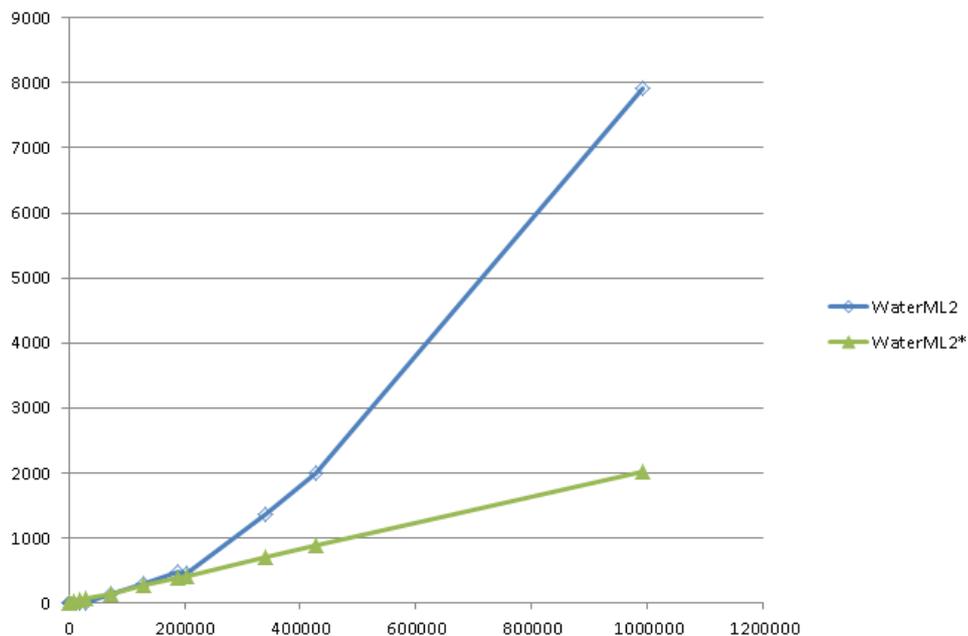


Figure 18: Response times with linear progression

Of course, sending many requests causes additional overhead but the problems caused by the long term allocation of memory can be avoided in that way. Specifically, the risk is eliminated that the SOS server runs out of memory by many long running requests being processed in parallel.

In consequence, the WDW Service offers a configuration parameter to restrict the time range that can be queried with a single request. A bigger time range is split into small intervals which are processed sequentially.

2.1.7 Integration of Rserve

As described above, for every polled sensor an infinite number of processings can be registered. Each processing refers to an R script which performs the processing together with a variable number of parameters that are passed to the script. The script result is then stored into the SOS server that the analysis clients can access via the SOA-MAS architecture.

To interact with the R server the parameters are passed over to the R session as variables. Then, the script is loaded and executed. In the end, the WDW reads output variables that contain the resulting time series. Table 10 shows the system parameters that are always exchanged between the WDW Service and the R script.

Parameter	Input/Output	Description
sosUrl	Input	SOS URL that provides the observation results
offering_name	Input	Offering that should be queried
procedure_name	Input	Procedure that should be queried
fromTime	Input	Lower range of the time interval that should be queried
toTime	Input	Upper range of the time interval that should be queried
resultValues	Output	Array of resulting values
resultTimestamps	Output	Array of timestamps of the resulting values

Table 10: System parameters for R scripts

Besides these parameters user defined parameters can be passed, too.

```

1  library("sos4R"); library("zoo")
2  watersos = SOS(sosUrl)
3  procedures <- sosProcedures(watersos)[[offering_name]]
4  contains_procedure <- procedures %in% grep(procedure_name, procedures, value=TRUE)
5  procedure <- subset(procedures, contains_procedure)
6
7  offering <- sosOfferings(watersos)[[offering_name]]
8  observedProperty <- sosObservedProperties(offering)[1]
9  period <- sosCreateTimePeriod(sos = watersos,
10     begin = as.POSIXct(fromTime), end = as.POSIXct(toTime))
11  observations <- getObservation(sos = watersos, observedProperty = observedProperty,
12     procedure = procedure, eventTime = sosCreateEventTimeList(period), offering = offering)
13  data <- sosResult(observations)
14  zoo <- zoo(x = data[[2]], order.by = data[[1]])
15  resultTimeSeries <- rollmean(x = zoo, k = periods)
16  resultTimestamps <- time(resultTimeSeries)
17  resultValues <- coredata(resultTimeSeries)

```

Listing 8: Sample R smoothing script

Listing 8 shows a sample R script that performs a simple smoothing of the observation results. Lines 2 – 13 load the required data from the source SOS server. These lines will be the same for most R scripts. Next, the observation results are transformed to time series using the “zoo” library³¹. After that, the smoothing is performed by using the *rollmean* function. The number of periods that are used by the rolling mean shows the usage of a user defined parameter. The variable “*periods*” is defined as parameter within the processing definition. Line 16 extracts the timestamps into the output parameter *resultTimestamps* and line 17 creates the output parameter *resultValues*.

Another example is shown in Listing 9.

```

1  library("sos4R"); library("zoo")
2  watersos = SOS(sosUrl)
3  procedures <- sosProcedures(watersos)[[offering_name]]
4  contains_procedure <- procedures %in% grep(procedure_name, procedures, value=TRUE)
5  procedure <- subset(procedures, contains_procedure)
6
7  offering <- sosOfferings(watersos)[[offering_name]]
8  observedProperty <- sosObservedProperties(offering)[1]
9  period <- sosCreateTimePeriod(sos = watersos,
10     begin = as.POSIXct(fromTime), end = as.POSIXct(toTime))
11  observations <- getObservation(sos = watersos, observedProperty = observedProperty,
12     procedure = procedure, eventTime = sosCreateEventTimeList(period), offering = offering)
13  data <- sosResult(observations)
14  zoo <- zoo(x = data[[2]], order.by = data[[1]])
15  resultTimeSeries <- aggregate(zoo, time(zoo) - as.numeric(time(zoo)) %/% periodInSeconds, mean)
16  resultTimestamps <- time(resultTimeSeries)
17  resultValues <- coredata(resultTimeSeries) * factor

```

Listing 9: Sample R aggregation script

³¹ <http://cran.r-project.org/web/packages/zoo/index.html>

The time aggregation is done in line 15 while the unit transformation is performed when filling the *resultValues* in line 17. The two parameters “*periodInSeconds*” and “*factor*” are required from the processing metadata. Listing 10 shows how the *addProcessingRequest* can provide the required parameters.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:addProcessingRequest xmlns:ns2="http://waterp/wdw/ManagementService">
  <rscrip>/aggregate.R</rscrip>
  <createdSensorId>urn:ogc:object:feature:Sensor:DEMO:sensor-1554/aggregated
  </createdSensorId>
  <sensorDescription>urn:ogc:object:feature:Sensor:DEMO:sensor-1554
  </sensorDescription>
  <unit>kbar</unit>
  <parameter>
    <name>periodInSeconds</name>
    <value>600</value>
  </parameter>
  <parameter>
    <name>factor</name>
    <value>.001</value>
  </parameter>
  <observedProperty>urn:ogc:def:phenomenon:waterpressure
  </observedProperty>
  <offering>WATERPRESSURE_._urn:ogc:object:feature:Sensor:DEMO:sensor-1554
  </offering>
  <sensorId>urn:ogc:object:feature:Sensor:DEMO:sensor-1554</sensorId>
</ns2:addProcessingRequest>
```

Listing 10: *addProcessingRequest* for sample aggregation script

2.2 Basic Data Mining Operations

Data preparation and pre-processing is a fundamental stage of data analysis. In many computer science fields, such as information retrieval, machine learning and data mining, pre-processing of raw data is required in order to prepare quality data for further analysis. Most data mining algorithms expect quality data as input, i.e. their input is assumed to be appropriately distributed, containing no missing or incorrect values etc. However, in real-life scenarios this assumption is not valid, especially when dealing with time series data usually coming from live observations (Reeves et al. 2009) or sensors (Stiefmeier et al. 2007), which are particularly subject to noise and outliers. This may lead to poor-quality input and disguising useful patterns that are hidden in the data, resulting in poor-quality outputs and low performance of the data mining algorithms in general and time series data mining algorithms in particular.

These problems are handled by data preparation and pre-processing. Data preparation comprises those techniques concerned with analysing raw data so as to yield quality data, mainly including data integration, data transformation, data cleaning, data reduction, and data discretization. Data preparation can be more time consuming than data mining, and can present equal, if not more, challenges than data mining (Yan et al. 2003). In practice, it has been generally found that data preparation takes approximately 80% of the total data engineering effort (Zhang et al., 2003).

In the context of the WDW prototype we designed and developed basic data management operations implementing data preparation and pre-processing techniques, as well as basic statistical functionalities, with an emphasis on time series data. These basic data mining operations can significantly improve the efficiency of the WatERP WP4/WP5 components which incorporate data analysis and mining algorithms and especially algorithms dealing with time series data mining, such as segmentation and trend analysis, pattern discovery, clustering and recognition, forecasting, time series similarity measures etc. (Esling and Agon, 2012); (Ratanamahatana et al, 2010); (Fu, 2011).

2.2.1 Integrated R Infrastructure within WDW

In order to develop the basic data mining operations implementing data preparation and pre-processing techniques, as well as basic statistical functionalities, we integrated within the WatERP WDW environment a software infrastructure based on the R³² environment (R Development Core Team, 2012). R is an open source project, providing a freely available and a high quality computing environment with thousands of add-on packages. The R statistical environment provides a powerful suite of tools for statistical analysis for use in many scientific fields. Its application is not limited to statistical research and applications only, but its modular design allows the use of R and its packages in areas such as data mining (Zhao, 2012), while in the area of computational time series analysis R has established itself as the choice of many researchers and practitioners (McLeod et al., 2011). R is extendable through packages, while there are around 4000 packages available in the CRAN repository³³, categorised in collections of packages for different tasks such as machine and statistical learning, time series analysis and analysis of spatial data.

The R environment provides two native interfaces for communication with external applications: a simple standard input/output model and R as a shared library model. Both of these interfaces are unsatisfactory for WatERP, because they assume a tight integration of external applications to the R environment and because of performance concerns. In our design we consider the separation of the R system from the client applications themselves very important. One reason is to avoid any dependence on the programming language of the application, since a native direct interface to R is usable from the C language only (R Development Core Team, 2012). The other reason is performance, because the goal is to provide the user with the desired results quickly, without the need of starting an R session from scratch.

Based on the aforementioned concerns we decided to expose functionalities implemented in R and integrate the R environment within the WDW infrastructure through OGC WPS, a standardized

³² The R Project for Statistical Computing: <http://www.r-project.org/>

³³ <http://cran.r-project.org>

geoprocessing service interface protocol. In order to achieve this we use WPS4R³⁴, a solution for creating WPS processes based on annotated R-scripts which was developed in the FP7 projects UncertWeb³⁵ and GeoViQua³⁶. WPS4R publishes annotated R-scripts as standardized WPS processes and generates relevant metadata. Scripts can be uploaded during runtime. They are stored by the WPS4R and made available immediately as processes for WPS clients. The scripts are executed by WPS4R using on the background Rserve which is an independent TCP/IP server for R. It has either a local or a remote connection to the WPS and could also run on a different server. WPS4R is a server which allows other applications to use facilities of R from various languages without the need to initialise R or link to the R library while it complements the family of interfaces between applications and R by providing a fast, language-independent and remote-capable way of using all facilities of R from other applications. Due to a clean separation between R (server) and the application (client), internal data manipulation on one side cannot affect the other side. Finally, client-side implementations for WPS communication are available for popular languages such as Python and Java, allowing clients implemented in different technologies to communicate with the R core where computations are done.

The architecture of the integrated R environment within the WDW infrastructure is depicted in Figure 19. As can be seen in the figure, the basic data mining operations are implemented on the basis of R functionalities and packages and are exposed to the client applications (i.e. to other WatERP components) through the WPS interface and the Multi Agent System (MAS) architecture. A typical sequence of interactions between a client application and a basic data mining operation is the following: A client retrieves sensor-observation data through the SOS interface, calls a basic data mining operation through the WPS interface providing the input data required by this operation, while the results of the operation are retrieved through the same WPS interface.

The role of the developed architecture is three-fold:

- a) to allow the design, implementation and provision of basic data mining operations which implement data preparation and pre-processing techniques, as well as basic statistical functionalities;
- b) to efficiently serve data-aggregation functionalities of the WDW-service, and
- c) to provide the infrastructure so that other WatERP Workpackages can run R scripts.

³⁴ <http://52north.org/wps4r>

³⁵ <http://www.uncertweb.org>

³⁶ <http://www.geoviqua.org>

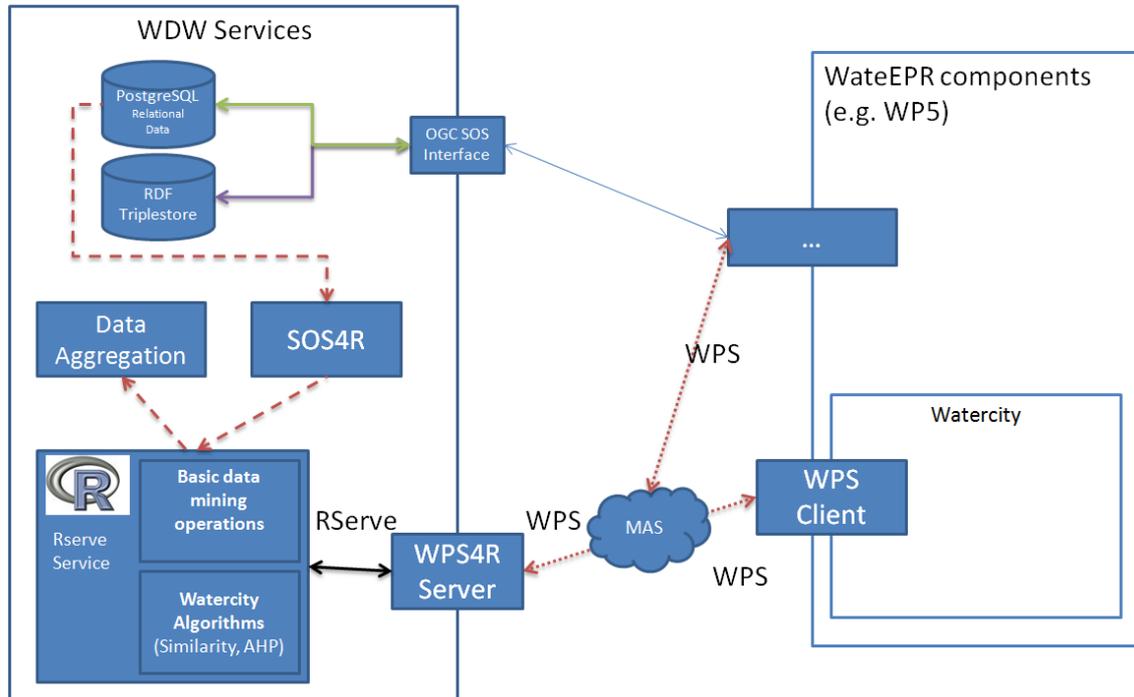


Figure 19: R infrastructure integrated within WDW

The sos4R package, a project of the geostatistics community of 52 North³⁷, has been incorporated in the architecture in case the R scripts need to use SOS server data. Sos4R is an R client for the OGC Sensor Observation Service, which enables R users to integrate (near real-time) sensor observations directly into R. In more detail, it allows to query data from standard-conform SOS instances using simple R function calls and doesn't require any knowledge about the Sensor Web. WaterCity (see WatERP deliverable D5.4) is an example of case c), i.e. of the case that a WatERP tool uses the WDW R infrastructure in a context different than that of basic data mining operations. WaterCity uses R scripts of the WDW R infrastructure through a Python WPS client. The R scripts are wrapped through WPS services and expose the following functionality, which is further detailed in deliverable D5.4:

- Ability to run the Analytic Hierarchy Process (AHP) method for calculating feature weights based on expert responses on questionnaires and pairwise feature comparisons from the literature.
- Calculation of similarity between WaterCity users by executing the mixed AHP-weighted user similarity method.
- Provision of water consumption data that are available in the WDW to WaterCity.

³⁷ <http://52north.org/communities/geostatistics/projects>),

Figure 20 depicts a simple WPS4R annotated R-script. This script exposes the R function “runif(n, min = , max =)”, which generates n uniform random numbers lying in the interval (min, max), as WPS service.

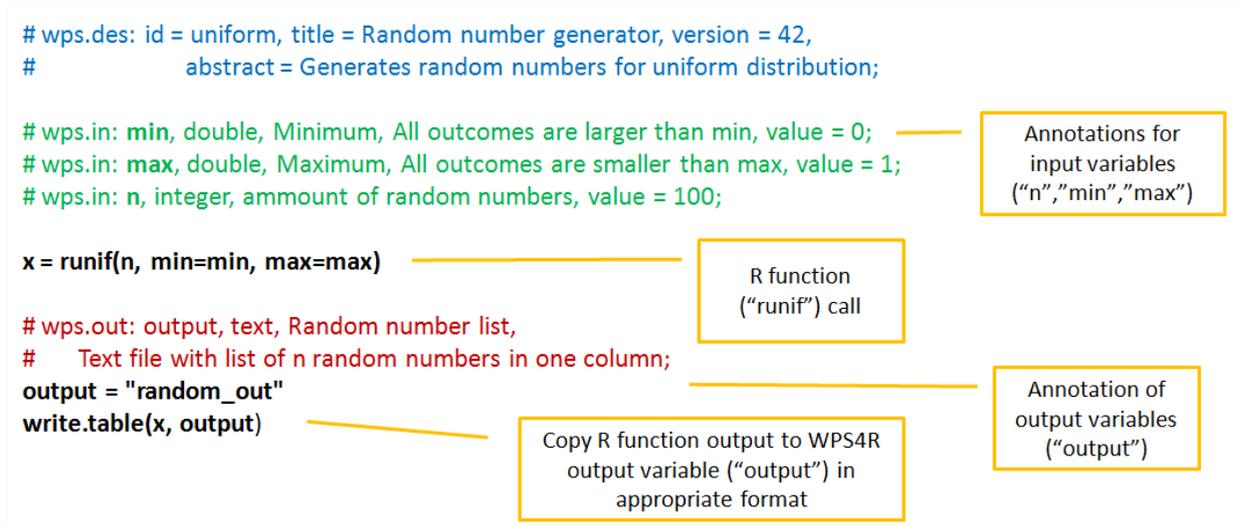


Figure 20: A simple WPS4R annotated Rscript

Figure 21 depicts a code snippet of a WPS request in XML format which can be used to call (via a WPS client) the R function “runif(n=5,min=10,max=100)” when the above annotated R script is installed in a WPS4R server.

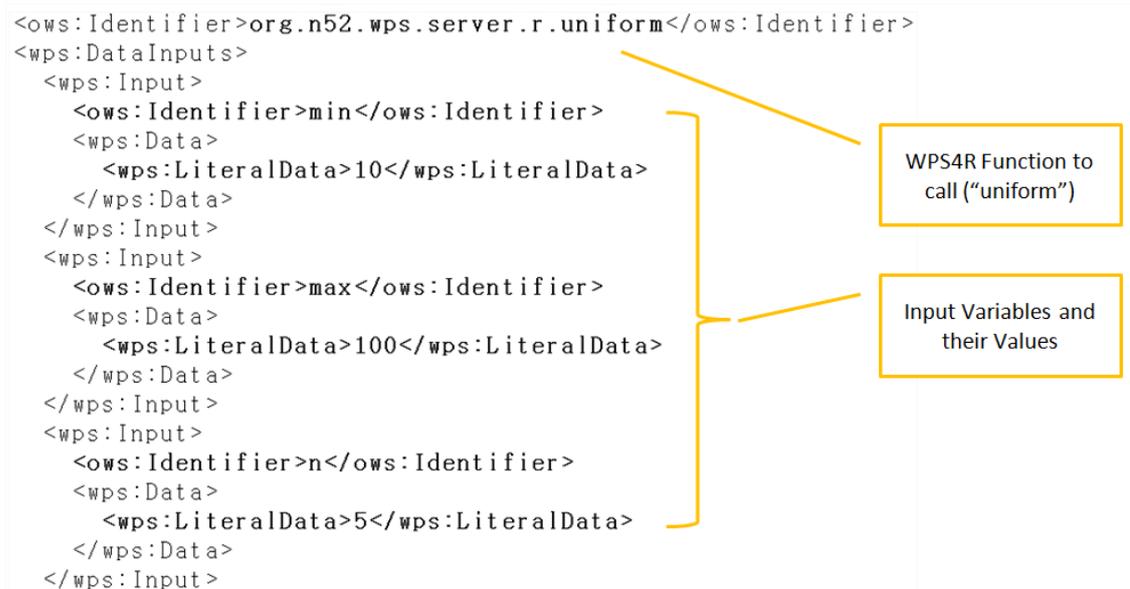


Figure 21: WPS request in XML format (file “request.xml”)

Finally, Figure 22 depicts a simple WPS client written in Python which calls the R function “runif”, along with the corresponding output.

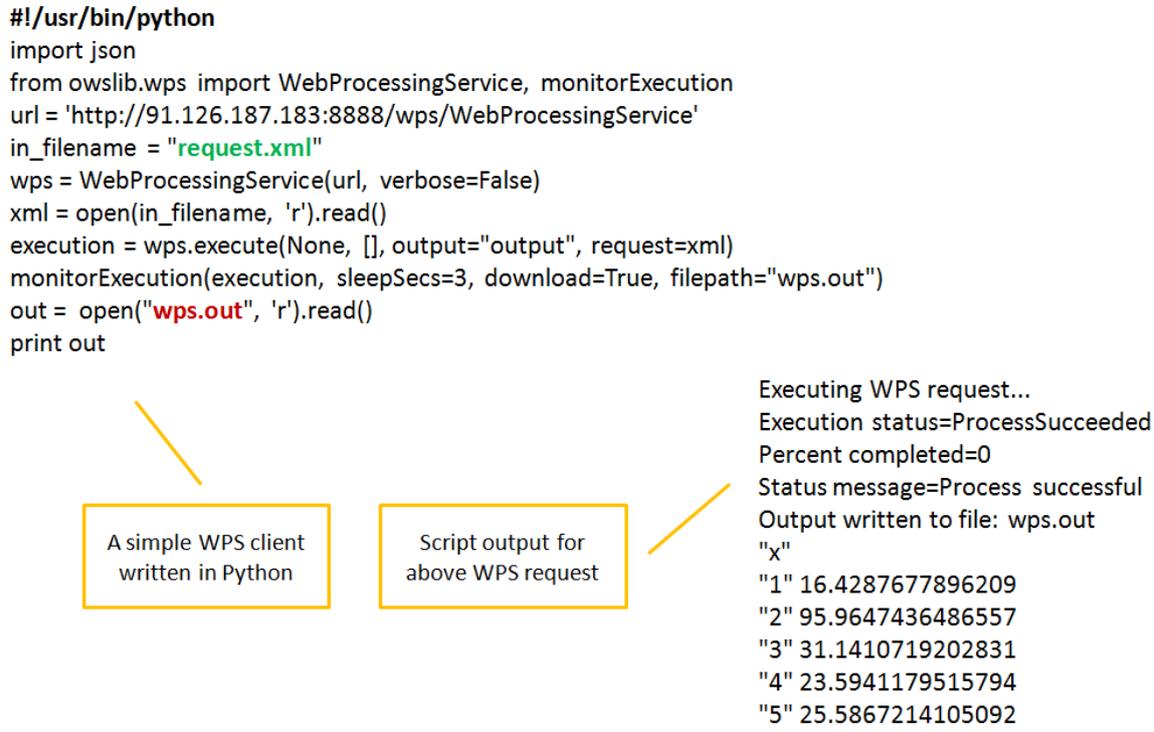


Figure 22 : A simple WPS client written in Python and execution output

2.2.2 Description of Basic Data Mining Operations

In this section we describe the basic data mining operations offered by the WatERP WDW. The selection of basic data mining operations to be implemented was based on the requirements of WatERP WP5 (Demand Management Systems) which is the main client of these operations. The requirements elicitation process was iterative and included three working meetings between the WP3 and WP5 teams. The set of basic data mining operations covers functionalities of missing values removal and interpolation, time-series smoothing, detection and replacement of outliers, conversion of numerical to categorical data, spatial interpolation, as well as correlation and covariance estimation. For each basic data mining operation presented below, the motivation (requirements) for its development is explained, its main features are described, while its main input and output parameters are listed along with some practical usage examples.

2.2.2.1 Missing values

Basic Data Mining Operation Name	Missing values (bdm_mv_filter and bdm_mv_interpolate)
----------------------------------	---

<p>Motivation / need for basic data mining operation</p>	<p>The WDMS uses sets of correlated time-series data (consumption, weather conditions, etc.) as input for statistical data analysis and demand forecasting. Missing values are common and if left unhandled or if even they are always handled by resorting to rejecting all (sets of) observations containing one or more missing values, then there is danger of reducing the accuracy of the forecast or other analysis, making graphs unintelligible, etc. Depending on the purpose of the data analysis, different methods of handling missing values are required. For example, in the case of average daily temperatures, there is some continuity along the time series and the most probable value for a day's average temperature is some type of interpolation between the previous and next day(s). In addition, for some processes, such as demand forecasting, that may need extensive calibration, it would be useful to be able to flag observations with missing data in order to search, manually or otherwise, for the causes of possible discrepancies and estimation errors.</p>
<p>Description of features</p>	<p>Two methods for dealing with missing values in the observations will be provided:</p> <ul style="list-style-type: none"> a) Function bdm_mv_filter returns one or more time series with missing values (observations) removed. b) Function bdm_mv_interpolate replaces missing values (NAs) by interpolation. If the input dataset has more than one column, the above functionality is applied to each column.
<p>Input Parameters</p>	<p>a) The identifier of the WPS process bdm_mv_filter is "org.n52.wps.server.r.bdm_mv_filter.v1". It has one input variable named "tsdata_json" which contains a json object (serialized as a string) of one or more time-series.</p> <p><u>Input example:</u></p> <pre><wps:Input> <ows:Identifier>tsdata_json</ows:Identifier> <wps:Data> <wps:LiteralData>[{"Date":"2009-12-31","A":4.9,"B":18.4,"C":32.6,"D":77}, {"Date":"2010-01-29","A":5.1,"B":17.7}, {"Date":"2010-01-31","A":5,"C":32.8,"D":78.7}, {"Date":"2010-02-26","A":4.8}, {"Date":"2010-02-28","A":4.7,"B":18.3,"C":33.7,"D":79}, {"Date":"2010-03-31","A":5.3,"B":19.4,"C":32.4,"D":77.8}, {"Date":"2010-04-30","A":5.2,"B":19.7,"C":33.6,"D":79}, {"Date":"2010-05-28","A":5.4,"D":81.7}, {"Date":"2010-05-30","C":34.5}, {"Date":"2010-05-31","A":4.6,"B":18.1}]</wps:LiteralData> </wps:Data> </wps:Input></pre>

	<p>b) The identifier of the WPS process bdm_mv_interpolate is “org.n52.wps.server.r.bdm_mv_interpolate.v1”. It has one input variable named “tsdata_json” which contains a json object (serialized as a string) of one or more time-series.</p> <p><u>Input example:</u></p> <pre><wps:Input> <ows:Identifier>tsdata_json</ows:Identifier> <wps:Data> <wps:LiteralData>[{"Date":"2009-12-31","A":4.9,"B":18.4,"C":32.6,"D":77}, {"Date":"2010-01-29","A":5.1,"B":17.7}, {"Date":"2010-01-31","A":5,"C":32.8,"D":78.7}, {"Date":"2010-02-26","A":4.8}, {"Date":"2010-02-28","A":4.7,"B":18.3,"C":33.7,"D":79}, {"Date":"2010-03-31","A":5.3,"B":19.4,"C":32.4,"D":77.8}, {"Date":"2010-04-30","A":5.2,"B":19.7,"C":33.6,"D":79}, {"Date":"2010-05-28","A":5.4,"D":81.7}, {"Date":"2010-05-30","C":34.5}, {"Date":"2010-05-31","A":4.6,"B":18.1}]</wps:LiteralData> </wps:Data> </wps:Input></pre>
<p>Output Parameters</p>	<p>a) The output of the WPS process bdm_mv_filter is a json object containing the filtered timeseries</p> <p><u>Output example:</u></p> <pre><ns:Output> <ns1:Identifier xmlns:ns1="http://www.opengis.net/ows/1.1">output</ns1:Identifier> <ns1:Title xmlns:ns1="http://www.opengis.net/ows/1.1">id</ns1:Title> <ns:Data> <ns:LiteralData dataType="xs:string"> [{"A":4.9,"B":18.4,"C":32.6,"D":77},{"A":4.7,"B":18.3,"C":33.7,"D":79},{"A":5.3,"B":19.4,"C":32.4,"D":77.8},{"A":5.2,"B":19.7,"C":33.6,"D":79}] </ns:LiteralData> </ns:Data> </ns:Output></pre> <p>b) The output of the WPS process bdm_mv_interpolate is a json object containing the interpolated timeseries</p> <p><u>Output example:</u></p> <pre><ns:Output> <ns1:Identifier xmlns:ns1="http://www.opengis.net/ows/1.1">output</ns1:Identifier> <ns1:Title xmlns:ns1="http://www.opengis.net/ows/1.1">id</ns1:Title> <ns:Data> <ns:LiteralData dataType="xs:string"></pre>

	<pre>[{"A":4.9,"B":18.4,"C":32.6,"D":77},{ "A":5.1,"B":17.7,"C":32.79,"D":78.59},{ "A":5,"B":17.74,"C":32.8,"D":78.7},{ "A":4.8,"B":18.26,"C":33.64,"D":78.98},{ "A":4.7,"B":18.3,"C":33.7,"D":79},{ "A":5.3,"B":19.4,"C":32.4,"D":77.8},{ "A":5.2,"B":19.7,"C":33.6,"D":79},{ "A":5.4,"B":18.25,"C":34.44,"D":81.7},{ "A":4.87,"B":18.15,"C":34.5},{ "A":4.6,"B":18.1}]</pre> <pre></ns:LiteralData></pre> <pre></ns>Data></pre> <pre></ns:Output></pre>
--	--

2.2.2.2 Smoothing

Basic Data Mining Operation Name	Smoothing (bdm_smooth)
Motivation / need for basic data mining operation	The time-series data, related to water demand and used by WDMS, exhibit periodicities, that can be analyzed into a number of systematic variation cycles (e.g. daily, weekly, seasonal), in addition to more random or event-dependent variations. In order to search for systematic periodicities, non-periodic variations need to be smoothed-out. Also, when correlating or comparing two time-series (or parts of) it may be needful to smooth-out random variation, in order to derive a meaningful relationship. The same technique will be useful in deriving a best estimate (i.e. an average) when the most probable value for a factor, which varies in time, is needed. For example, the WDMS economic tools would need to derive typical values for the current (or any other period's) water demand from a set of recent time-series data.
Description of features	<p>Function bdm_smooth will:</p> <ul style="list-style-type: none"> - Extract an underlying signal from time series - Perform robust (online) extraction of low frequency components (the signal) from a univariate time series by applying robust regression techniques to moving time windows
Input Parameters	<p>The identifier of the WPS process bdm_smooth is "org.n52.wps.server.r.bdm_smooth.v1"</p> <p>The input parameters are:</p> <ul style="list-style-type: none"> • tsdata_json : a json list of time-series data • tsstart : an integer indicating the start of the time series • tsend : an integer indicating the end of the time series • tsfrequency : an integer indicating the frequency of the time series • width : a positive integer defining the window width used for fitting • method : a comma-separated list of methods. Any combination of the following

	<p>values is supported:</p> <ul style="list-style-type: none"> ○ "DR" Deepest Regression ○ "LMS" Least Median of Squares regression ○ "LQD" Least Quartile Difference regression ○ "LTS" Least Trimmed Squares regression ○ "MED" Median ○ "RM" Repeated Median regression, <p><u>Input example:</u></p> <pre> <wps:DataInputs> <wps:Input> <ows:Identifier>tsdata_json</ows:Identifier> <wps>Data> <wps:LiteralData>[25.96,25.63,24.78,26.24,26.42,26.84,26.99,24.75,26.76,37.87,34.76, 32.23,25.48,26.55,24.18,26.53,27.05,24.19,26.82,25.94,26.6,26.05,26.85,26.89,27.57,2 7.76,23.72 ,26.56,28.14,28.52,26.66,26.72,25.61,26.41,26.45,27.16,27,28.04,27.62,27.7,27.01,28. 19,27.77,28.47,29.76,28.4,26.38,29.96,27.75,29.46]</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>tsstart</ows:Identifier> <wps>Data> <wps:LiteralData>1</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>tsend</ows:Identifier> <wps>Data> <wps:LiteralData>50</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>tsfrequency</ows:Identifier> <wps>Data> <wps:LiteralData>1</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>width</ows:Identifier> <wps>Data> <wps:LiteralData>6</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>method</ows:Identifier> <wps>Data> <wps:LiteralData>RM,LMS,LTS,DR,LQD</wps:LiteralData> </wps>Data> </wps:Input> </wps:DataInputs> </pre>
Output	

Parameters	<p>The output of the WPS process is a JSON object (in the output variable named “output”) containing the smoothed time series. If multiple methods are asked the output json object contains one time-series per method. The output contains also a JSON list with the slope of each element of the smoothed time-series.</p> <p>Output example:</p> <pre> <ns:Output> <ns1:Identifier xmlns:ns1="http://www.opengis.net/ows/1.1">output</ns1:Identifier> <ns1:Title xmlns:ns1="http://www.opengis.net/ows/1.1">id</ns1:Title> <ns:Data> <ns:LiteralData dataType="xs:string"> ["LQD":[25.34,25.64,25.94,26.24,26.45,26.71,26.99,27.01,26.76,30.76,33.74,32.19,29.49,26.86,26.35,26.53,26.71,26.26,26.69,26.56,26.6,26.8,26.85,27.1,27.45,27.76,27.64,28.14,27.1,28.09,26.9,26.72,26.62,26.53,26.7,27.14,27.19,27.37,27.62,27.72,27.83,28.09,28.6,28.47,28.61,29.07,29.59,29.17,29.32,29.46], "RM":[25.37,25.65,25.93,26.22,26.46,26.7,26.7,27.30,30.87,30.76,32.65,32.04,29.39,27.02,26.54,26.27,26.63,26.24,26.54,26.49,26.29,26.7,26.85,27.14,27.32,26.97,27.57,27.76,27.54,26.8,26.9,26.68,26.56,26.59,26.59,27.03,27.16,27.34,27.51,27.7,27.72,28.28,33.28,39.28,39.28,67.28,18.28,89.28,82.28,74], "LMS":[25.28,25.58,25.88,26.18,26.46,26.77,27.06,26.88,25.7,28.49,35.05,32.08,29.26,26.71,26.54,26.89,26.84,26.6,26.83,26.62,26.26,26.74,26.74,27.21,27.45,27.77,27.86,28.05,27.54,27.96,26.95,26.43,26.56,26.48,26.64,27.06,27.03,27.29,27.46,27.78,27.7,28.14,28.05,28.19,28.33,28.81,29.18,29.25,29.31,29.38], "LTS":[25.33,25.62,25.92,26.21,26.45,26.76,27.02,26.84,25.96,28.97,34.57,32.15,29.35,26.75,26.4,26.71,26.77,26.59,26.74,26.59,26.25,26.8,26.81,27.26,27.4,27.78,27.81,28.08,27.53,27.82,26.93,26.34,26.56,26.49,26.64,27.06,27.06,27.26,27.54,27.74,27.75,28.13,28.14,28.23,28.33,28.94,28.35,29.45,29.52,29.59], "DR":[25.28,25.58,25.87,26.16,26.46,26.72,26.8,26.91,28.57,30.94,33.67,32.03,29.07,26.64,25.63,25.92,26.62,26.06,26.19,26.26,26.48,26.56,26.75,27.03,27.24,27.24,27.13,27.04,27.42,27.61,27.03,26.48,26.53,26.44,26.5,27.03,27.24,27.4,27.6,27.74,27.73,27.95,28.13,28.47,28.46,28.67,28.2,28.83,28.82,28.82], "LQD.1":[0.3,0.3,0.3,0.3,0.27,0.28,0.28,0.08,-0.11,1.26,-4.13,-2.82,-2.74,-2.68,0.35,0.23,0.05,-0.14,-0.04,-0.26,-0.1,0.05,0.24,0.47,0.47,0.36,0.25,0.19,-0.22,0.76,-0.64,-0.63,-0.08,-0.08,0.29,0.36,0.22,0.25,0.15,0.23,0.11,0.19,0.41,0.35,0.14,0.3,0.37,0.14,0.14,0.14], "RM.1":[0.28,0.28,0.28,0.28,0.29,0.27,0.15,0.15,1.98,1.98,-2.53,-2.78,-2.78,-2.37,0.17,0.18,0.1,0.05,-0.05,-0.11,0.21,0.1,0.3,0.33,0.33,0.04,0.13,0.38,0.38,0.38,-0.5,-0.5,-0.12,0.04,0.24,0.43,0.28,0.21,0.08,0.08,0.02,0.13,0.43,0.43,0.1,0.22,-0.22,-0.07,-0.07,-0.07], "LMS.1":[0.3,0.3,0.3,0.3,0.26,0.28,0.28,0.08,-0.11,1.05,-4.13,-2.82,-2.82,-2.68,0.35,0.35,0.1,-0.15,0.01,-0.2,0.12,0.01,0.33,0.24,0.44,0.36,0.25,0.19,1.6,-0.74,-0.53,-0.53,-0.08,-0.08,0.42,0.52,0.24,0.25,0.15,0.23,-0.05,0.19,0.14,0.14,0.14,0.37,0.07,0.07,0.07,0.07], "LTS.1":[0.29,0.29,0.29,0.29,0.27,0.27,0.27,0.07,-0.25,1.28,-3.97,-2.81,-2.81,-2.68,0.31,0.31,0.07,-0.14,-0.02,-0.18,0.22,0.02,0.3,0.25,0.45,0.34,0.22,0.17,1.6,-0.61,-0.59,-0.59,-0.07,-0.07,0.47,0.49,0.25,0.26,0.17,0.23,-0.03,0.18,0.15,0.1,0.1,0.38,-0.25,0.07,0.07,0.07], "DR.1":[0.29,0.29,0.29,0.29,0.32,0.28,0.1,0.11,1.75,2.37,-0.12,-2.8,-2.89,-2.58,-0.68,0.03,0.05,0.1,0.1,0.1,0.36,0.17,0.27,0.37,0.26,-0.04,-0.09,0.26,0.38,0.18,-0.55,-0.55,-0.12,-0.03,0.17,0.45,0.3,0.22,0.12,0.17,0.05,0.12,0.34,0.31,0.05,0.18,-0.13,0,0,0] </ns:LiteralData> </ns:Data> </ns:Output> </pre>
------------	---

2.2.2.3 Outliers

<p>Basic Data Mining Operation Name</p>	<p>Outliers (bdm_outliers)</p>
<p>Motivation / need for basic data mining operation</p>	<p>WDMS would need to handle outliers in time-series data, in more-or-less the same manner as missing values. That is, during model (e.g., forecasting) calibration (e.g., in Task 7.4 of WP7) outliers in the input data would need to be detected and investigated in order to explain for any discrepancies and large errors. If testing and calibration show that some outliers over some threshold would need specific smoothing, a replacement function should take care of that. WDMS, in actual (future) use, would need to identify outliers in the differences between the time series of results (e.g., forecasted demand) and the time-series of the actual values measured (e.g., future consumption).</p>
<p>Description of features</p>	<p>An outlier is an observation that lies an abnormal distance from other values in a random sample from a population.</p> <p>Function bdm_outliers will detect and replace outliers.</p>
<p>Input Parameters</p>	<p>The identifier of the WPS process bdm_outliers is "org.n52.wps.server.r.bdm_outliers.v 1".</p> <p>The input parameters are :</p> <ul style="list-style-type: none"> • tsdata_json : a JSON list of time-series data • tsstart: an integer indicating time series start • tsend: an integer indicating time series end • tsfrequency : an integer indicating time series frequency <p><u>Input example:</u></p> <pre><wps:DataInputs> <wps:Input> <ows:Identifier>tsdata_json</ows:Identifier> <wps>Data> <wps:LiteralData>[25.96,25.63,24.78,26.24,26.42,26.84,26.99,24.75,26.76,37.87,34.76, 32.23,25.48,26.55,24.18,26.53,27.05,24.19,26.82,25.94,26.6,26.05,26.85,26.89,27.57,2 7.76,23.72 ,26.56,28.14,28.52,26.66,26.72,25.61,26.41,26.45,27.16,27,28.04,27.62,27.7,27.01,28. 19,27.77,28.47,29.76,28.4,26.38,29.96,27.75,29.46]</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>tsstart</ows:Identifier> <wps>Data> <wps:LiteralData>1</wps:LiteralData> </wps>Data> </wps:Input></pre>

	<pre> <wps:Input> <ows:Identifier>tsend</ows:Identifier> <wps:Data> <wps:LiteralData>50</wps:LiteralData> </wps:Data> </wps:Input> <wps:Input> <ows:Identifier>tsfrequency</ows:Identifier> <wps:Data> <wps:LiteralData>1</wps:LiteralData> <wps:Data> </wps:Input> </wps>DataInputs> </pre>
<p>Output Parameters</p>	<p>The output of bdm_outliers is a JSON object that contains:</p> <ul style="list-style-type: none"> a) a list of indexes of the outliers and b) a list of the proposed replacements (in the same order with the indexes) <p><u>Output example:</u></p> <pre> <ns:Output> <ns1:Identifier xmlns:ns1="http://www.opengis.net/ows/1.1">output</ns1:Identifier> <ns1:Title xmlns:ns1="http://www.opengis.net/ows/1.1">id</ns1:Title> <ns:Data> <ns:LiteralData dataType="xs:string">{"index":[10,11],"replacements":[28.58,30.41]}</ns:LiteralData> </ns:Data> </ns:Output> </pre>

2.2.2.4 Categorical Data

<p>Basic Data Mining Operation Name</p>	<p>Categorical Data (bdm_categorical)</p>
<p>Motivation / need for basic data mining operation</p>	<p>Most of WDMS services (data analysis, forecasting, economic tools and profiling) besides continuous numeric values (e.g., demand, temperature) use as input categorical data (e.g., temperature ranges, precipitation levels, seasons, day types, etc.). Besides the obvious usefulness for handling categorical data for statistical analysis and reporting purposes (e.g., histograms), dynamic categorisation of data is needed while automatically re-calibrating the models used by WDMS to better describe meaningful relationships, e.g., between consumption and temperature; labelling, while forecasting water demand, a range of temperatures as ‘high’ (or ‘low’ for that matter) indicates that the re-calibration process has found dynamically a specific relation between a level of consumption and the level of temperature which is being labelled as high. Also, during various processes (e.g., examining demand variation or demand levels corresponding to various water availability scenarios, assessing forecasting</p>

	<p>errors, etc.) there is a need to label the factor under observation as 'low', 'high' or any other classification, in order to take some action.</p>
<p>Description of features</p>	<p>Conceptually, categorical variables are variables, which take on a limited number of different values.</p> <p>The function bdm_categorical will be able to encode a numeric vector as a set of categorical values. bdm_categorical will divide the range of input data into intervals and codify the values according to which interval they fall. The leftmost interval should correspond to level one, the next to level two and so on</p>
<p>Input Parameters</p>	<p>The identifier of the WPS process bdm_categorical is "org.n52.wps.server.r.bdm_categorical.v1".</p> <p>The input parameters are:</p> <ul style="list-style-type: none"> • data_json: a JSON object of vector data, • breaks_json: a JSON object of vector data. Either a numeric vector of two or more unique cut points or a single number (greater than or equal to 2) giving the number of intervals into which data is to be cut • labels_json: a JSON object containing the labels for the levels of the resulting category <p><u>Input example:</u></p> <pre><wps:DataInputs> <wps:Input> <ows:Identifier>data_json</ows:Identifier> <wps>Data> <wps:LiteralData>[-5,5,15,20,21,-10,1,11,31,23,32,42,60,4,123]</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>breaks_json</ows:Identifier> <wps>Data> <wps:LiteralData>[-20,10.5,30.5,200]</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>labels_json</ows:Identifier> <wps>Data> <wps:LiteralData>["low","medium","high"]</wps:LiteralData> </wps>Data> </wps:Input> </wps:DataInputs></pre>
<p>Output Parameters</p>	<p>Will return a vector values converted to level codes. Values which fall outside the range of breaks are coded as NA.</p> <p><u>Output example :</u></p> <pre><ns:Data></pre>

	<pre><ns:LiteralData dataType="xs:string"> ["low","low","medium","medium","medium","low","low","medium","high","mediu m","high","high","high","low","high"] </ns:LiteralData> </ns>Data></pre>
--	---

2.2.2.5 Spatial Interpolation

<p>Basic Data Mining Operation Name</p>	<p>Spatial interpolation or Kriging (bdm_spatial_interpolate)</p>
<p>Motivation / need for basic data mining operation</p>	<p>WDMS, especially forecasting, uses some geocoded data that, while they relate to a factor (e.g., temperature, water losses/evaporation), which can actually take continuous values throughout the geographical space, they are measured at a very small number of metering points (e.g., weather stations). When these data need to be correlated with points in space (e.g., demand nodes) which are some distance from the metering points, then a data conversion needs to be done, based on a spatial interpolation technique known as kriging. This holds also true in the reverse case that a set of results is available, referring to specific points in geo space (e.g., demand forecasted at specific demand nodes, water use profiles resulting from a number of smart meter distributed in space) would need to be spatially distributed in an even way, either to estimate a point value or to create a thematic map.</p>
<p>Description of features</p>	<p>The function bdm_spatial_interpolate will perform spatial prediction (interpolation) using a kriging algorithm (simple kriging, ordinary kriging, or universal kriging). Given a set of known values at specific locations, the goal is to estimate the unknown value at location that doesn't exists in the input dataset.</p>
<p>Input Parameters</p>	<p>The identifier of the WPS process bdm_spatial_interpolate is "org.n52.wps.server.r.bdm_spatial_interpolate.v1".</p> <p>The input parameters are:</p> <ul style="list-style-type: none"> • locs_json : a JSON array of observation locations • obs_json : a JSON array of observation values • grid_locs_json : a JSON array of prediction locations <p><u>Input example:</u></p> <pre><wps:DataInputs> <wps:Input> <ows:Identifier>locs_json</ows:Identifier> </wps:Data> </wps:LiteralData>[[15,28],[36,28],[21,31],[39,40],[23,33],[40,41],[24,35],[29,28]]</wps:Lit</pre>

	<pre> eralData> </wps:Data> </wps:Input> <wps:Input> <ows:Identifier>obs_json</ows:Identifier> <wps:Data> <wps:LiteralData>[8.9,27.7,39.6,24.6,10,6,39.1,18.3]</wps:LiteralData> </wps:Data> </wps:Input> <wps:Input> <ows:Identifier>grid_locs_json</ows:Identifier> <wps:Data> <wps:LiteralData>[[[13,20],[15,20],[17,20],[19,20],[21,20],[23,20],[25,20],[27,20],[29,20],[31,20],[33,20],[35,20],[37,20],[39,20],[41,20],[13,22],[15,22],[17,22],[19,22],[21,22],[23,2 2],[25,22],[27,22],[29,22],[31,22],[33,22],[35,22],[37,22],[39,22],[41,22],[13,24],[15,24],[1 7,24],[19,24],[21,24],[23,24],[25,24],[27,24],[29,24],[31,24],[33,24],[35,24],[37,24],[39,24] ,[41,24],[13,26],[15,26],[17,26],[19,26],[21,26],[23,26],[25,26],[27,26],[29,26],[31,26],[33, 26],[35,26],[37,26],[39,26],[41,26],[13,28],[15,28],[17,28],[19,28],[21,28],[23,28],[25,28],[27,28],[29,28],[31,28],[33,28],[35,28],[37,28],[39,28],[41,28],[13,30],[15,30],[17,30],[19,3 0],[21,30],[23,30],[25,30],[27,30],[29,30],[31,30],[33,30],[35,30],[37,30],[39,30],[41,30],[1 3,32],[15,32],[17,32],[19,32],[21,32],[23,32],[25,32],[27,32],[29,32],[31,32],[33,32],[35,32] ,[37,32],[39,32],[41,32],[13,34],[15,34],[17,34],[19,34],[21,34],[23,34],[25,34],[27,34],[29, 34],[31,34],[33,34],[35,34],[37,34],[39,34],[41,34],[13,36],[15,36],[17,36],[19,36],[21,36],[23,36],[25,36],[27,36],[29,36],[31,36],[33,36],[35,36],[37,36],[39,36],[41,36],[13,38],[15,3 8],[17,38],[19,38],[21,38],[23,38],[25,38],[27,38],[29,38],[31,38],[33,38],[35,38],[37,38],[3 9,38],[41,38],[13,40],[15,40],[17,40],[19,40],[21,40],[23,40],[25,40],[27,40],[29,40],[31,40] ,[33,40],[35,40],[37,40],[39,40],[41,40],[13,42],[15,42],[17,42],[19,42],[21,42],[23,42],[25, 42],[27,42],[29,42],[31,42],[33,42],[35,42],[37,42],[39,42],[41,42],[13,44],[15,44],[17,44],[19,44],[21,44],[23,44],[25,44],[27,44],[29,44],[31,44],[33,44],[35,44],[37,44],[39,44],[41,4 4],[13,46],[15,46],[17,46],[19,46],[21,46],[23,46],[25,46],[27,46],[29,46],[31,46],[33,46],[3 5,46],[37,46],[39,46],[41,46],[13,48],[15,48],[17,48],[19,48],[21,48],[23,48],[25,48],[27,48] ,[29,48],[31,48],[33,48],[35,48],[37,48],[39,48],[41,48]]</wps:LiteralData> </wps:Data> </wps:Input> </wps:DataInputs> </pre>
<p>Output Parameters</p>	<p>The output is a JSON vector of the predicted values.</p> <p><u>Example output:</u></p> <pre> <ns:Data> <ns:LiteralData dataType="xs:string"> <u>[20.78,20.78,20.98,21.28,21.53,21.66,21.68,21.67,21.7,21.81,21.98,22.12,22.19,22.16,</u> <u>22.08,19.96,19.94,20.39,21.05,21.57,21.76,21.69,21.56,21.56,21.78,22.15,22.47,22.58,</u> <u>22.49,22.29,18.52,18.31,19.38,20.93,21.93,22.12,21.78,21.32,21.19,21.64,22.43,23.14,</u> <u>23.33,23.02,22.6,16.29,15.14,17.97,21.41,23.19,23.07,22.01,20.86,20.28,21.26,22.86,2</u> <u>4.43,24.69,23.74,22.91,14.78,8.9,17.67,23.68,26.41,24.84,22.39,20.52,18.3,20.97,23.2</u> <u>1,25.89,26.17,24.17,23.06,16.88,16.43,20.85,27.86,33.48,26.22,16.21,19.20,76.21,66.2</u> <u>3.15,24.63,24.82,23.79,22.9,19.45,20.2,23.19,28.16,30.64,18.6,21.22,22.71,22.49,22.5</u> <u>5,23.04,23.56,23.61,23.13,22.53,20.94,21.74,23.49,25.43,24.62,22.3,28.07,25.81,24.04</u> <u>,23.24,23.08,23.14,23.07,22.67,22.08,21.64,22.25,23.31,24.62,26.41,31.65,32.9,27.65,</u> <u>24.86,23.59,23.12,23.06,23.02,22.47,21.43,21.91,22.36,23.1,24.19,25.89,28.13,28.48,2</u> <u>6.47,24.58,23.48,23.23,23.34,22.67,19.83,21.99,22.3,22.81,23.54,24.5,25.39,25.51,24.</u> <u>76,23.78,23.04,22.61,22.44,22.66,24.6,14.26,21.97,22.18,22.5,22.92,23.41,23.78,23.82</u> <u>,23.5,23.01,22.52,22.05,21.4,19.82,14.26,11.46,21.93,22.06,22.25,22.48,22.71,22.87,2</u> <u>2.88,22.72,22.44,22.09,21.63,20.83,19.25,16.8,15.86,21.88,21.96,22.06,22.18,22.29,22</u> <u>.36,22.36,22.26,22.09,21.84,21.46,20.86,19.93,18.93,18.56,21.84,21.89,21.94,22,22.06</u> <u>,22.08,22.07,22.01,21.9,21.73,21.47,21.09,20.62,20.19,20.02]</u> </pre>

	<pre></ns:LiteralData> </ns>Data></pre>
--	--

2.2.2.6 Correlation & Covariance

Basic Data Mining Operation Name	Correlation & covariance (Bdb_auto_cf, bdf_cross_cf)
Motivation / need for basic data mining operation	WDMS needs to correlate water demand to various demand drivers, as specified in the description of Tasks 5.1 and 5.4. This is useful as an accessory tool for water demand management. It is of particular usefulness for calibrating and optimising the water demand model(s) for a specific pilot site or case study, since it permits adding case-specific factors (variables) to the models and estimating their probable weighting factors. An auto-covariance/autocorrelation function would also permit WDMS to perform a quick projection of a time series, when it needs to predict a future value without resorting to some specialised forecasting model, e.g., extend a weather forecast by 1-2 days, provide a demand forecast for a node for which no other data (besides past consumption) is available, etc.
Description of features	<p>a) Function bdm_auto_cf will compute estimates of the auto-covariance or autocorrelation function of a time series</p> <p>b) Function bdm_cross_cf will compute the cross-correlation or cross-covariance of two univariate series.</p>
Input Parameters	<p>The identifier of the WPS process bdm_auto_cf is "org.n52.wps.server.r.bdm_auto_cf.v1".</p> <p><u>The input parameters of bdm_auto_cf are:</u></p> <ul style="list-style-type: none"> • tsdata_json : a JSON list of time-series data, • tsstart : an integer indicating time series start • tsend : an integer indicating time series end • tsfrequency : an integer indicating time series frequency • acftype : One of the strings "correlation" or "covariance" according to the desired operation type <p><u>Example input of bdm_auto_cf:</u></p> <pre><wps:DataInputs> <wps:Input> <ows:Identifier>tsdata_json</ows:Identifier></pre>

	<pre> <wps>Data> <wps:LiteralData>[25.96,25.63,24.78,26.24,26.42,26.84,26.99,24.75,26.76,37.87,34.76, 32.23,25.48,26.55,24.18,26.53,27.05,24.19,26.82,25.94,26.6,26.05,26.85,26.89,27.57,2 7.76,23.72,26.56,28.14,28.52,26.66,26.72,25.61,26.41,26.45,27.16,27,28.04,27.62,27.7 ,27.01,28.19,27.77,28.47,29.76,28.4,26.38,29.96,27.75,29.46]</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>tsstart</ows:Identifier> <wps>Data> <wps:LiteralData>1</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>tsend</ows:Identifier> <wps>Data> <wps:LiteralData>50</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>tsfrequency</ows:Identifier> <wps>Data> <wps:LiteralData>1</wps:LiteralData> </wps>Data> </wps:Input> <wps:Input> <ows:Identifier>acftype</ows:Identifier> <wps>Data> <wps:LiteralData>correlation</wps:LiteralData> </wps>Data> </wps:Input> <wps>DataInputs> </pre> <p>The identifier of the WPS process bdm_cross_cf is "org.n52.wps.server.r.bdm_cross_cf.v1".</p> <p><u>The input parameters of bdm_cross_cf are:</u></p> <ul style="list-style-type: none"> • ts1data_json : a JSON list of time-series data of the first time series • ts1start : an integer indicating time series start of the first time series • ts1end : an integer indicating time series end of the first time series • ts1frequency : an integer indicating time series frequency of the first time series • ts2data_json : a JSON list of time-series data of the second time series • ts2start : an integer indicating time series start of the second time series • ts2end : an integer indicating time series end of the second time series • ts2frequency : an integer indicating time series frequency of the second time series • ccftype, type = string, title = "correlation" or "covariance"
--	--

Example input of **bdm_cross_cf**:

```

<wps:Input>
  <ows:Identifier>ts1data_json</ows:Identifier>
  <wps:Data>
<wps:LiteralData>[25.96,25.63,24.78,26.24,26.42,26.84,26.99,24.75,26.76,37.87,34.76,
32.23,25.48,26.55,24.18,26.53,27.05,24.19,26.82,25.94,26.6,26.05,26.85,26.89,27.57,2
7.76,23.72,26.56,28.14,28.52,26.66,26.72,25.61,26.41,26.45,27.16,27,28.04,27.62,27.7
,27.01,28.19,27.77,28.47,29.76,28.4,26.38,29.96,27.75,29.46]</wps:LiteralData>
  </wps:Data>
</wps:Input>
<wps:Input>
  <ows:Identifier>ts1start</ows:Identifier>
  <wps:Data>
  <wps:LiteralData>1</wps:LiteralData>
  </wps:Data>
</wps:Input>
<wps:Input>
  <ows:Identifier>ts1end</ows:Identifier>
  <wps:Data>
  <wps:LiteralData>50</wps:LiteralData>
  </wps:Data>
</wps:Input>
<wps:Input>
  <ows:Identifier>ts1frequency</ows:Identifier>
  <wps:Data>
  <wps:LiteralData>1</wps:LiteralData>
  </wps:Data>
</wps:Input>
<wps:Input>
  <ows:Identifier>ts2data_json</ows:Identifier>
  <wps:Data>
<wps:LiteralData>[24.31,24.8,23.72,25.4,24.54,23.03,24.94,23.25,24.88,31.02,29.57,29
.87,30.12,31.34,28.81,31.62,30.14,30.18,21.98,14.13,14.88,13.02,25.6,33.65,34.62,33.
79,43.15,44.95,23.92,25.62,25.03,24,26.1,25.36,27.62,27.16,26.81,26.16,26.49,27.38,2
6.62,24.66,23.96,28.07,27.15,26.23,25.79,24.68,27.29,27.16]</wps:LiteralData>
  </wps:Data>
</wps:Input>
<wps:Input>
  <ows:Identifier>ts2start</ows:Identifier>
  <wps:Data>
  <wps:LiteralData>1</wps:LiteralData>
  </wps:Data>
</wps:Input>
<wps:Input>
  <ows:Identifier>ts2end</ows:Identifier>
  <wps:Data>
  <wps:LiteralData>50</wps:LiteralData>
  </wps:Data>
</wps:Input>
<wps:Input>
  <ows:Identifier>ts2frequency</ows:Identifier>
  <wps:Data>
  <wps:LiteralData>1</wps:LiteralData>
  </wps:Data>
</wps:Input>

```

	<pre><wps:Input> <ows:Identifier>ccftype</ows:Identifier> <wps:Data> <wps:LiteralData>correlation</wps:LiteralData> </wps:Data> </wps:Input></pre>
<p>Output Parameters</p>	<p>a) The output of bdm auto cf: is a JSON object containing the result of the operation.</p> <p><u>Example output:</u></p> <pre><ns:Output> <ns1:Identifier xmlns:ns1="http://www.opengis.net/ows/1.1">output</ns1:Identifier> <ns1:Title xmlns:ns1="http://www.opengis.net/ows/1.1">id</ns1:Title> <ns:Data> <ns:LiteralData dataType="xs:string"> [[0,1],[1,0.47],[2,0.13],[3,-0.14],[4,-0.1],[5,-0.06],[6,-0.05],[7,-0.14],[8,-0.25],[9,-0.14],[10,-0.08],[11,-0.07],[12,0.02],[13,0.04],[14,0.08],[15,0],[16,-0.05]] </ns:LiteralData> </ns:Data> </ns:Output></pre> <p>b) The output of bdm cross cf: is a JSON object containing the result of the operation.</p> <p><u>Example output:</u></p> <pre><ns:Output> <ns1:Identifier xmlns:ns1="http://www.opengis.net/ows/1.1">output</ns1:Identifier> <ns1:Title xmlns:ns1="http://www.opengis.net/ows/1.1">id</ns1:Title> <ns:Data> <ns:LiteralData dataType="xs:string"> [[-13,0.03],[-12,-0.28],[-11,-0.43],[-10,-0.62],[-9,-0.45],[-8,-0.18],[-7,0.06],[-6,0.13],[-5,0.14],[-4,0.18],[-3,0.14],[-2,0.2],[-1,0.08],[0,0.08],[1,0.05],[2,-0.03],[3,-0.13],[4,-0.16],[5,-0.11],[6,-0.1],[7,-0.11],[8,-0.16],[9,-0.16],[10,-0.06],[11,0.03],[12,0.12],[13,0.09]] </ns:LiteralData> </ns:Data> </ns:Output></pre>

2.3 Triple Store

In this section we describe the final version of the triple store. After a technical overview, implementation details about the triple store's ability to support both geospatial and ontological reasoning are discussed, while the results of technical performance tests are presented.

2.3.1 Triple Store Technical Overview

The triple store covers the requirements of other WatERP components (mainly WP4 and WP6) which need ontological reasoning along with geospatial reasoning functionality. The triple store developed in

the context of the first WDW prototype used the Sesame native store for semantic storage, which didn't support ontological reasoning capabilities. Therefore, we later incorporated in the triple store architecture the OWLIM module which in addition to providing a semantic storage, supports also for the semantics (reasoning) of RDFS, OWL 2 RL and OWL 2 QL.

The WDW triple-store is implemented using uSeekM and Sesame Java libraries along with the PostGIS spatial database, an extender for PostgreSQL object-relational database, and sesame-based OWLIM storage (instead of sesame native storage). An overview of the triple store architecture and supported interfaces is given in Figure 23.

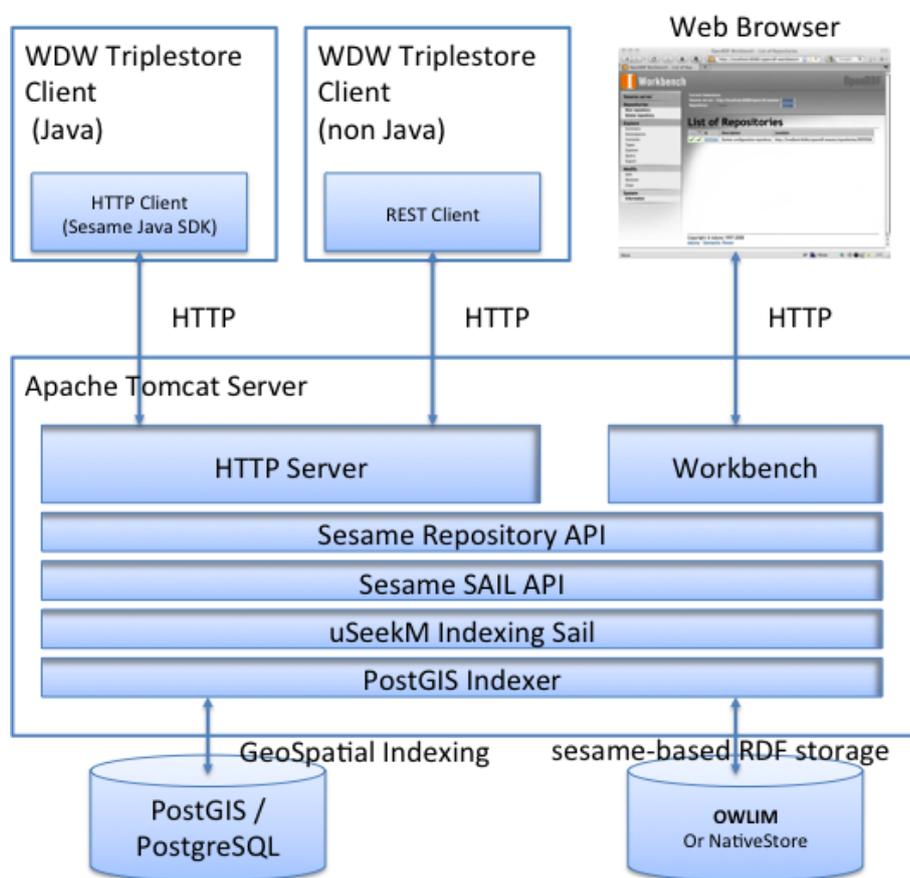


Figure 23: Overview of the triple store architecture and supported interfaces

The OpenSahara uSeekM is a Java library that adds spatial indexing and querying capabilities to Semantic Databases (also known as Triple-stores, Quad-stores, or RDF-databases) that use the Sesame Java interface. The uSeekM library indexes spatial data in a separate R-Tree based index. GeoSPARQL queries are intercepted by the library, and rewritten into a combination of queries against the spatial-index and the semantic database. It can work with many Semantic Databases that are based on the Sesame Sail layer. Sesame is an open-source Java framework and API for the implementation of RDF storage engines. In addition to offering ready to use native and database-backed Quad-stores, it also offers an extensible API to build other stores upon. The Sesame SAIL API can be used to extend

the functionality of RDF databases that have a Sail layer. The SAIL API (Storage And Inference Layer) is a low level system API for RDF stores and inference engines. It is used for abstracting from the storage details, allowing various types of storage and inference to be used.

The triple store exploits the aforementioned Sesame's extensibility and integrates OWLIM, a semantic database allowing ontological reasoning. Therefore, it can support both ontological and geospatial reasoning at the same time, covering all the current requirements of the WatERP platform, as explained in more details in the next section.

2.3.2 Geospatial and Ontological Indexing

The uSeekM library provides several SAIL wrappers such as *IndexingSail*, *SimpleTypeInferencingSail*, *SmartSailWrapper* and *PipelineSail*. In WDW we use the **IndexingSail** wrapper which adds geospatial (GeoSPARQL) search capabilities to Sesame.

An *IndexingSail* must be configured with *Indexers*. The *Indexers* make sure that when data is added or removed from the database, the required extra indices are updated. When an *IndexingSail* is queried, the Sail checks whether the query will benefit from the extra indexes. If so, the query is rewritten. Parts of the query are answered by the extra indexes, parts are answered by the original RDF database. The uSeekM library takes care of re-planning the query, and of joining the results from the different sources. It has advanced query planners for this that can be optimised for different underlying RDF Databases and use cases. The following *Indexers* are currently available as shown in Figure 24:

- **PostgisIndexer**: builds geometry and full-text indexes for statements. Queries can use GeoSPARQL (Perry and Herring, 2012) and Full Text Search features. This indexer uses a PostGIS database to build its indexes. Internally the PostGIS Indexer uses an R-Tree-over-GiST index for geospatial searches and a Generalized INverted (GIN) index for text search. Statements containing other types of objects can also be indexed. The additional indexes will be used to optimise performance when search results are joined with other statement patterns.
- **ElasticSearchIndexer**: builds resource based indexes (see Resource Based Search). Queries can use GeoSPARQL and Full Text Search features. Additionally, the resource-based indexes are used to speed-up queries by utilising the inverted-indexes and minimising the number of required joins for common query patterns. Geospatial data is indexed with quad-trees or geohashes. This indexer uses elasticsearch³⁸ (a Lucene³⁹ wrapper in an easily scalable architecture) to build its indexes.

³⁸ elasticsearch: <http://www.elasticsearch.org/>

³⁹ Apache Lucene: <http://lucene.apache.org/>

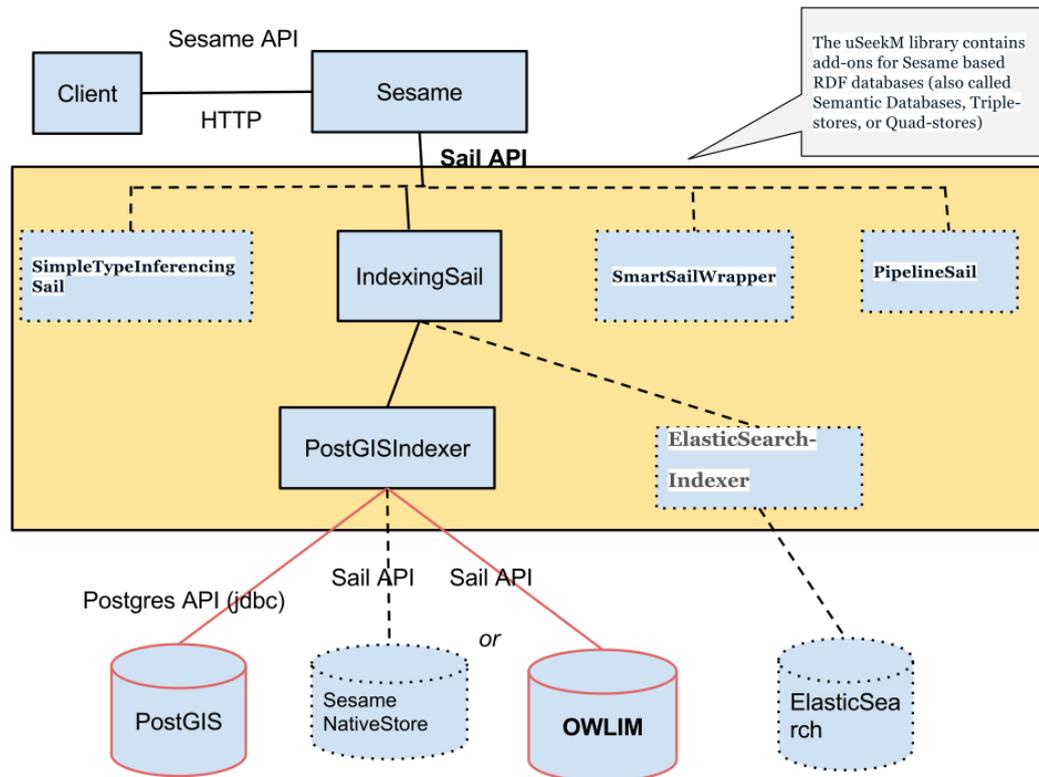


Figure 24: Integration of OWLIM within the triple store enables ontological reasoning

The triple store uses the *IndexingSail* with *PostgisIndexer* as depicted in Figure 24. With respect to semantic storage, as already explained, we used OWLIM instead of Sesame native store, with the aim to support ontological reasoning over the triple store contents. Therefore the WatERP triple store supports both geospatial and ontological reasoning, i.e. it can be queried either through geoSPARQL (in case the query involves geographical predicates) or through simple SPARQL otherwise. In case of geoSPARQL queries, the uSeekM Sesame SAIL (*IndexingSail*) intercepts and rewrites the query into a combination of queries against the spatial-index (*PostgisIndexer*) and the OWLIM semantic database and joins the results from the different sources, while for SPARQL queries the original OWLIM semantic storage answers the query, proving ontological reasoning capabilities.

In order to integrate OWLIM within the WatERP triple store, a wrapper class was developed which allows the correct initialisation of OWLIM as a Spring Bean. As already explained above, in order to use the *IndexingSail*, which adds geospatial (GeoSPARQL) search capabilities to Sesame, a semantic database that has a Sesame Sail layer is required. OWLIM provides a Sesame Sail layer but the current implementation doesn't allow the correct initialisation of OWLIM as a Spring Bean. The developed wrapper allows exactly this, enabling the correct operation of OWLIM's Sesame Sail layer and thus its integration within the WatERP triple store. Moreover, the OWLIM repository should be inserted in the configuration file *useekm-config.xml*. In more details, the administrator must put the

following beans and change "/temp/repositories" to the effective dataDir as depicted in the code snippet below.

```
<bean id="owlimSail" class="org.iccs.waterp.triplestore.SpringOwlimSail">
  <property name="properties">
    <map>
      <entry key="storage-folder" value="owlim-storage"/>
      <entry key="ruleset" value="owl-horst"/>
    </map>
  </property>
  <property name="dataDir">
    <bean class="java.io.File">
      <constructor-arg value="/temp/repositories"/>
    </bean>
  </property>
</bean>
<bean id="owlimRepository" class="org.openrdf.repository.sail.SailRepository">
  <constructor-arg ref="owlimSail"/>
</bean>
```

Listing 11: OWLIM Sail Spring Bean configuration

The code of the wrapper class “SpringOwlimSail” is depicted in the following code snippet:

```
package org.iccs.waterp.triplestore;

import java.util.Map;
import com.ontotext.trree.owlim_ext.SailImpl;

/**
 * Wrapper around {..SailImpl} to make it usable in Spring
 * @author npapag@mail.ntua.gr
 */
public class SpringOwlimSail extends SailImpl {

    public void setProperties(Map<String,String> properties) {
        for (String key : properties.keySet())
            setParameter(key, properties.get(key));
    }
}
```

Listing 12: Spring OWLIM Sail Wrapper Class

2.3.3 Triple Store Performance

The concept that was designed for data management (WDW) in WatERP, involves two separate repositories: a relational repository and a triple store. In order to avoid performance issues we decided to **store in the triple store pilot metadata only**, while the **time series data are stored in the relational schema** in order to avoid overloading of the WatERP triple store. As explained above, triple store data can be queried either through geoSPARQL (in case the query involves geographical predicates) or through simple SPAQRL otherwise.

In case of geoSPARQL queries, the uSeekM Sesame SAIL intercepts the query, so the performance of the triple store in that case is identical to the performance of uSeekM. No performance issues are

expected here for two reasons: First, the WatERP pilot metadata related to geographical information will mainly concern the geographies of the physical elements that are useful to define water resource management entities (Source, Transfer, Sink, Transformation, Storage), while no more than one hundred physical elements are expected in WatERP pilots. Second, uSeekM is the better performing RDF store among the stores who are fully compliant with GeoSPARQL, according to Geographica (Garbis et al., 2013), which is the first benchmark for evaluating geospatial RDF stores by using both real-world and synthetic data.

In case of SPARQL queries, the performance of the triple store will be identical to the performance of the underlying RDF repository. As we have developed a solution which extends uSeekM (an RDF repository that supports geospatial reasoning but not OWL reasoning) with OWLIM (with the aim to support both OWL and geospatial reasoning capabilities on the same repository as required by WP4 and WP6), the performance of the triple store will be identical to the performance of OWLIM for SPAQRL queries. OWLIM is a semantic repository which delivers best overall performance and scalability according to multiple independent evaluations conducted recently – see, e.g., (Voigt et al., 2012), (Thakker et al., 2010), (Kiryakov et al., 2005) and (Ontotext, 2014). Due to its very good performance, OWLIM has penetrated in the commercial sector, e.g. OWLIM is the semantic database used in the semantic publishing architecture behind the successful high-volume BBC World-Cup Website, BBC Sport Website and the BBC's 2012 Olympics online coverage (Kiryakov et al., 2010), (Sematrix, 2014). Also, it is used for data integration in the life sciences, e.g. in the LinkedLifeData.com platform, which is a public service consolidating 25 of the most popular biomedical databases and provides search and SPARQL query facilities over some 4 billion statements. Therefore, based on the above facts we argue that the WatERP triple store is able to handle SPAQRL queries about pilot metadata with good performance.

This claim was additionally confirmed based on the results of the technical performance testing. More specifically, we validated the performance of the triple store for both SPARQL (ontological reasoning) and GeoSPARQL (geospatial reasoning) queries based on real data as discussed in the following sections. Regarding ontological reasoning the correctness of the triple store's OWL-DL reasoning as well as its ability to generate new knowledge was validated. As far as geospatial reasoning is concerned, we tested the triple store's performance in terms of response time for the types of geospatial queries performed by WP4/WP6 based on the geospatial data available at the two WatERP pilots.

2.3.3.1 Geospatial reasoning testing

In this section, we present the results of the triple-store performance testing, which indicates that it can handle the required amount of geospatial data. The triple-store is expected to store according to the two pilot scenarios less than 5000 triples representing geospatial features. Such geospatial features will be mainly represented through point geometries in the WatERP pilots. A performance-testing scenario has

been created to ensure that the WatERP triple store meets the requirements of geospatial queries performed by the WP4/WP6.

The strategy was to perform two types of queries on geospatial features of increasing dataset size in the range between 497 and 25366 triples. The first GeoSPAQL query, depicted in Listing 13, retrieves all the geospatial features that are located at a specific point. A scenario requiring such type of query involves the user clicking on a specific point on the map. The second GeoSPAQL query, depicted in Listing 14, retrieves all the geospatial features located within a user defined polygon. Such type of query is submitted for example in order to allow the spatial user interface to look for entities of a particular type that fall within an explicit bounding box or polygon, related to the area displayed.

```
SELECT ?f ?fID ?fName
WHERE {
  ?f geo:hasGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .
  ?f rdfs:label ?fID .
  ?f georesource:name ?fName .
  FILTER (geof:sfContains(?fWKT, "POINT (32.4206171 34.7523534)"^^geontology:wktLiteral))
}
```

Listing 13: GeoSPARQL Query "Point QRY"

```
SELECT ?f ?fName ?fWKT
WHERE {
  ?f geo:hasGeometry ?fGeom .
  ?fGeom geo:asWKT ?fWKT .

  ?f georesource:name ?fName .
  FILTER (geof:sfWithin(?fWKT, "POLYGON (( 32.3206171 34.7023534, 32.5206171 34.7023534,
  32.3206171 34.8023534, 32.5206171 34.8023534, 32.3206171 34.7023534 ))"^^geo:wktLiteral))
}
```

Listing 14: GeoSPARQL Query "Polygon QRY"

Each query was executed five times per dataset of a specific size and the average time was computed and incorporated in the graphs depicted below. The time required for loading and indexing each dataset into the repository was also measured. All the tests were conducted and measured automatically using a Java-based tool that was developed for that purpose. Datasets of different size were generated based on the initial full dataset by randomly removing a configurable percent of triples. More specifically, the tool allows specifying the minimum and maximum percent of the initial dataset to be removed, as well as the step size (from 1% to 99%). The initial dataset was constructed by converting

the respective geometries of the ESRI shapefiles from OpenStreetMap project⁴⁰, into a format imposed by WatERP triple store, i.e. RDF/N-triples representing geometries in WKT format. More specifically we used the open-source tool TripleGeo (Patroumpas et al., 2014), a generic purpose utility based on the open source tool geometry2rdf (Geo.LinkedData.es Team, 2014).

The performance testing results are depicted in Table 11 and Figure 25 for a step size of ~500 triples and a range from 497 to 25366 triples. More specifically, Table 11 shows the response times in exact numbers, while Figure 25 provides a graphical overview.

Table 11: Triple Store Response Time on datasets of different sizes

Dataset Size	Indexing Time	Point QRY	Polygon QRY	Dataset Size	Indexing Time	Point QRY	Polygon QRY	Dataset Size	Indexing Time	Point QRY	Polygon QRY
497	124	98	109	8952	1872	1914	1897	17411	3744	3546	3458
994	265	228	239	9450	2356	2028	2033	17904	3915	3847	3556
1492	343	358	364	9948	2231	2002	2027	18405	3713	3822	3978
1990	452	462	478	10442	2184	2080	2267	18899	3931	3858	3775
2489	593	556	577	10940	2496	2386	2454	19400	4088	3739	3713
2983	687	488	665	11438	2418	2235	2314	19896	4056	3962	4217
3481	733	811	806	11937	2590	2590	2418	20394	4414	4295	3614
3979	1045	790	754	12435	1966	2636	2678	20886	4305	4476	4617
4477	1061	946	962	12932	2620	2860	2860	21382	3916	3823	4223
4975	1279	753	1061	13430	2730	2714	2901	21884	4227	4648	4274
5473	1108	1170	1102	13927	3057	2932	3083	22377	4961	4336	4768
5967	1264	1221	1253	14424	3027	3146	3005	22877	4680	4581	4700
6467	1014	1388	1440	14920	3166	3104	3349	23374	4836	5013	4820
6963	1482	1466	1492	15419	3229	3343	3416	23870	4851	4986	5200
7459	1575	1445	1534	15915	2932	3276	2974	24369	4633	5184	4986
7958	1779	1534	1773	16413	3323	3015	3717	24868	5553	5434	5626
8455	1810	1757	1831	16910	3853	3265	3827	25366	5179	5158	5246

As shown in Figure 25, the indexing and query response times increase proportionally with the dataset size (number of triples) for both types of queries. As already mentioned, the total number of geospatial features of the WatERP pilots will not exceed 5000 triples mainly represented through point geometries. In this range, indexing time as well as the response time of both queries is less than 1 second, as can be seen in Figure 25, which is considered adequate for the WatERP applications. Finally, it should be mentioned that the performance tests were performed on a VMWare host running Windows Server 2011 with allocated 4 GB of RAM and 2 threads on an Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz. It is expected that the presented performance results will be even better in a standalone installation on a stronger machine.

⁴⁰ <http://download.geofabrik.de/>

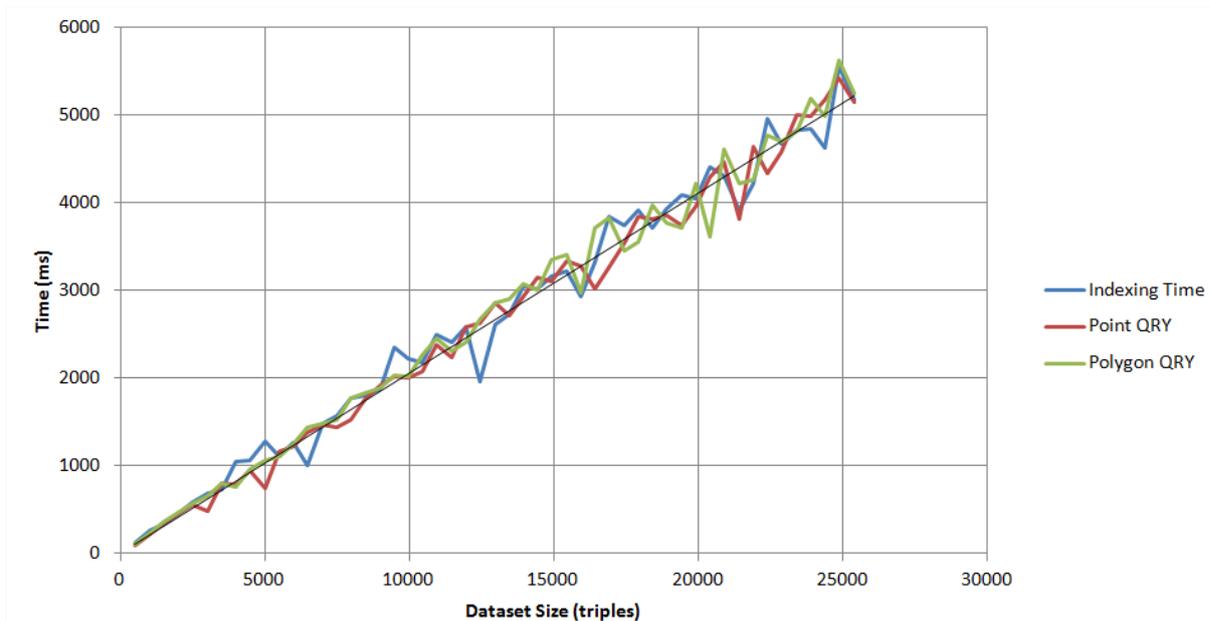


Figure 25: Triple store response times graph

2.3.3.2 Ontological reasoning testing

The main aim of this section is to evaluate the OWL-DL reasoning capabilities of the WatERP triple store repository in order to make explicit the implicit knowledge of the water supply and distribution chain for both pilots. That means, the testing of the repository has been focused on generating new knowledge aligned with the human-interactions in the water supply distribution chain that permit to support the decision making process in the water chain. Furthermore, the OWL-DL reasoning also permits to categorize the XML definitions (nodes) of the WaterML2 in order to share a common vocabulary between all WatERP building blocks. Then, the pilot's information is transformed into a common language defined and described in a knowledge base. By the use of the knowledge base, the information is machine understandable and interoperable because it is based on a standard vocabulary and terminology used throughout the water supply and distribution chain (based on SWEET, SSN, CUAHSI, etc.).

The performed testing over the developed triple store has been done over the pilot's instantiation presented on WatERP deliverable D1.4.1-"Extension of taxonomy and ontology to the pilots", Section 2 (page 14). This instantiation for the ACA and SWKA pilot cases contains information about the logical model, features of interest, observations, phenomena, procedures and observation results (time series) associated with the defined features of interest. Based on the information gathered from the pilots, some queries that need reasoning were executed on the WatERP triple store (OWL-DL reasoning).

The executed queries were compared against the local reasoning performed by Pellet (OWL-DL reasoning) in order to demonstrate that the ontological reasoning is done successfully.

1. Query 1: Which water resources are affected by the decisions made in TF4 (“AbreraTransformation”) water resource?

This SPARQL query is aimed at showing which logical model entities have direct and indirect relation to the decisions taken on the *AbreraTransformation* water resource of the ACA pilot case.

```
SELECT ?wrDec
WHERE {
    WaterERPOntology:TF4 WaterERPOntology:hasLogicDependence ?wrDec
}
```

Listing 15: SPARQL query to know the influence in the ‘AbreraTransformation’

As a result of the execution, the results reveal that the repository offers the same information as the axioms inferred in the Protegé-Pellet reasoning process (see Figure 26). Then, direct logical dependence (blue colored line) and indirect logical dependence (green colored line) are returned to the water manager. By this kind of relation, the water manager is able to know that elements are affected for a decision on the *AbreraTransformation* water resource (TF4).

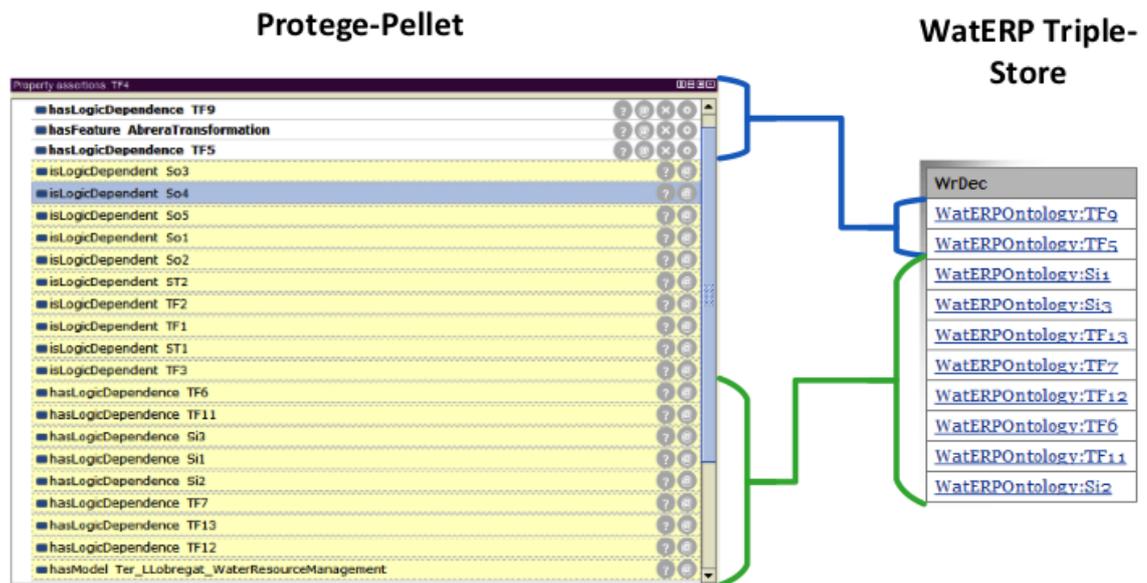


Figure 26: Comparison of Query 1

2. Query 2: Which kind of observation is associated with “SVHGaugeStation”?

This SPARQL query is aimed at showing the feature of interest type associated with the “SVHGaugeStation” defined observation.

```

SELECT ?obs
WHERE {
    WatERPontology:SVHGaugeStation rdf:type ?obs .
    FILTER(STRSTARTS(STR(?obs), "http://www.watERP-fp7.eu/WatERPontology.owl#") ||
    STRSTARTS(STR(?obs), "http://www.opengis.net/ont/geosparql#"))
}

```

Listing 16: SPARQL query to know associated observation with ‘SVHGaugeStation’

As a result of the query execution (see Figure 27), the WatERP triple store and the Protegé-Pellet local ontology provide similar information. In reference to the similar information, the WatERP triple store also returns the inferred hierarchy for the observation. In case of Protegé-Pellet, the returned information is known by direct inference (is_a internal property between concepts).

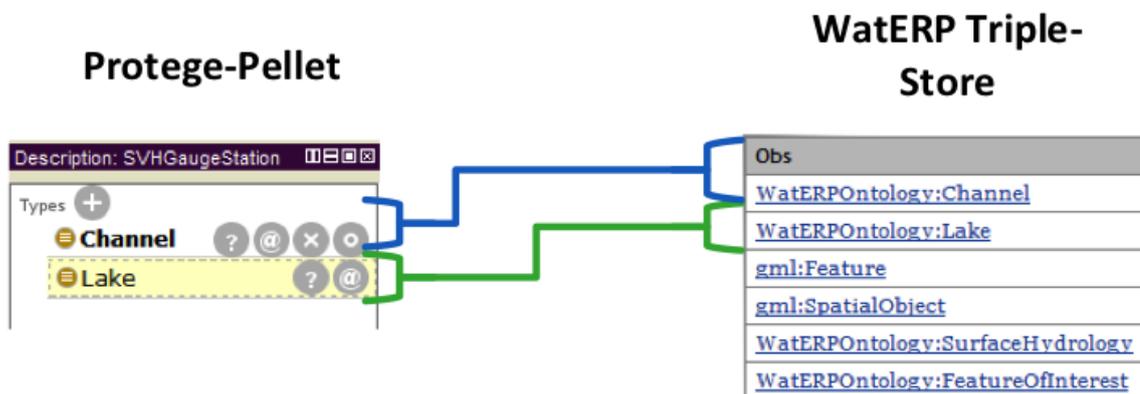


Figure 27: Comparison of Query 2

3. Which is the Feature Of Interest that observes the “DWP2ForecastedEnergyConsumption”?

This query is aimed at knowing which observations pertain to a certain Feature of Interest. This type of query is used by the DSS in order to know the observations that are aligned with a feature of interest or an offering.

```

SELECT ?foi
WHERE {
    WatERPontology:DWP2ForecastedEnergyConsumption WatERPontology:isObservedBy ?foi
}

```

Listing 17: SPARQL query to know the feature of interest observed by the ‘DWP2ForecastedEnergyConsumption’

As a result of the query execution (see Figure 28), both the local ontology reasoning and the WatERP triple store return the same result. The result of this specific example describes that the queried observation is observed by the DWP2 water work.

Protege-Pellet

WatERP Triple-Store

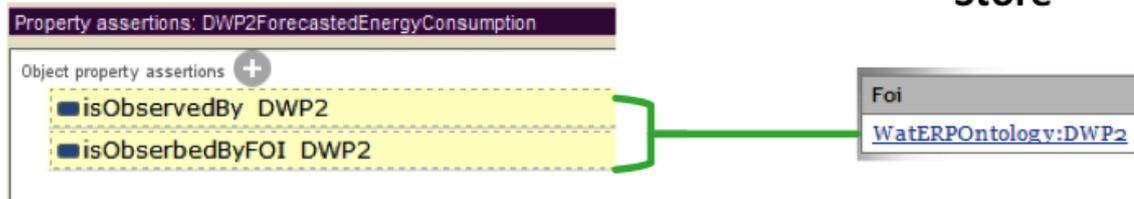


Figure 28: Comparison of Query 3

As a conclusion of the test, the WatERP triple store performs correctly the OWL-DL reasoning. Hence, the reasoning done over the pilot's information accomplish the requirements that are related to the impact of the decisions that affect to certain part of the logical model in the water supply distribution chain, the assurance of data provenance by the modeling of an observation and measurement process, and the inferred axioms in reference to missing categories causing by the WaterML2 population process.

3. Interface Layer

The WDW interface layer is responsible for all interactions with other WatERP functional units as well as other possible users of the prepared data. While communication within the WatERP project is realized through a SOA-MAS architecture, the interface layer itself consists of OGC-compliant standards. Additionally, a project-specific REST service has been realized which can be used by other WatERP functional units to commission additional views on WDW-data. While those views could also be generated within the functional unit itself, commissioning from the WDW serves the advantage of no additional logic within the functional unit as well as performance gains, since those views on data are being held precomputed within the WDW.

Implemented interfaces encompass the following OGC standards: SOS/WaterML2, WMS, WFS, WFS-T. With WaterML2 being the data exchange default format within WatERP, the SOS-Server implementation is by far the most important interface. Both WMS and WFS/WFS-T services are realized based on Geoserver, due to open source, free usage and widespread (dialect) support by any number of clients.

The reason for offering a number of different interfaces in addition to SOS/WaterML2, serving the same data, lies with the idea to always use the standard most feasible for any interaction. It also furthers interoperability with clients not being able to process SOS/WaterML2 data, keeping in mind later use of project results outside the scenarios of the two pilots.

3.1 SOS/WaterML2

This interface is being used by all WatERP functional units. It serves both raw data series from any sensor being available to the WDW as well as processing result views on these sensors.

Listing 18 shows a sample request body to obtain all observation results of a specific sensor.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.w3.org/2003/05/soap-envelope http://www.w3.org/2003/05/soap-envelope">
5   <env:Body>
6     <sos:GetObservation service="SOS" version="2.0.0"
7       xmlns:sos="http://www.opengis.net/sos/2.0" xmlns:fes="http://www.opengis.net/fes/2.0"
8       xmlns:gml="http://www.opengis.net/gml/3.2" xmlns:swe="http://www.opengis.net/swe/2.0"
9       xmlns:swes="http://www.opengis.net/swes/2.0" xmlns:xlink="http://www.w3.org/1999/xlink"
10      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
11      xsi:schemaLocation="http://www.opengis.net/sos/2.0 http://schemas.opengis.net/sos/2.0/sos.xsd">
12     <sos:procedure>urn:ogc:object:feature:Sensor:IFGI:ifgi-sensor-1
13     </sos:procedure>
14     <sos:responseFormat>http://www.opengis.net/waterml/2.0</sos:responseFormat>
15   </sos:GetObservation>
16 </env:Body>
17 </env:Envelope>
```

Listing 18: Sample GetObservation request

Additionally, this interface can be used to register virtual sensors and store time series for these sensors from any other functional unit within the WatERP project. Thus, calculations of scenarios and assumptions made from those units can be conserved within the WDW for later validation or other usages. In Subsection 2.1.5.1, it has already been shown how a sensor can be added to an SOS Server using SOS-T. Further below, in Subsection 4.2.3 “Trigger Import / Insert Observation” it will be described how sensor data is then stored for this sensor.

3.2 Management Service

The Management Service is based on REST technology. It takes requests in form of XML, storing this information as metadata for the configuration of the WDW Service process. Commissions can be made both for single sensor processing as well as for processings based on the type of entity. The management service has already been described in depth in Subsection 2.1.5 “Management Service”.

3.3 WMS

Using a WMS interface grants clients access to additional geographical data. This service is not a requirement for the functional uses of the WDW, yet it gives users potential access to cartographic background information, allowing for improved user acceptance and higher data reusability potential by offering new use cases apart from the WatERP pilots.

A small sample included in the Geoserver standard installation shows the basic request format. The full set of arguments can be looked up in the OpenGIS Web Map Service Client (WMS) Implementation Specification⁴¹.

Example: GET Request-URL:

<http://server/geoserver/sf/wms?service=WMS&version=1.1.0&request=GetMap&layers=sf:roads&styles=&bbox=589434.8564686741,4914006.337837095,609527.2102150217,4928063.398014731&width=512&height=358&srs=EPSG:26713&format=image%2Fgif>

Result:

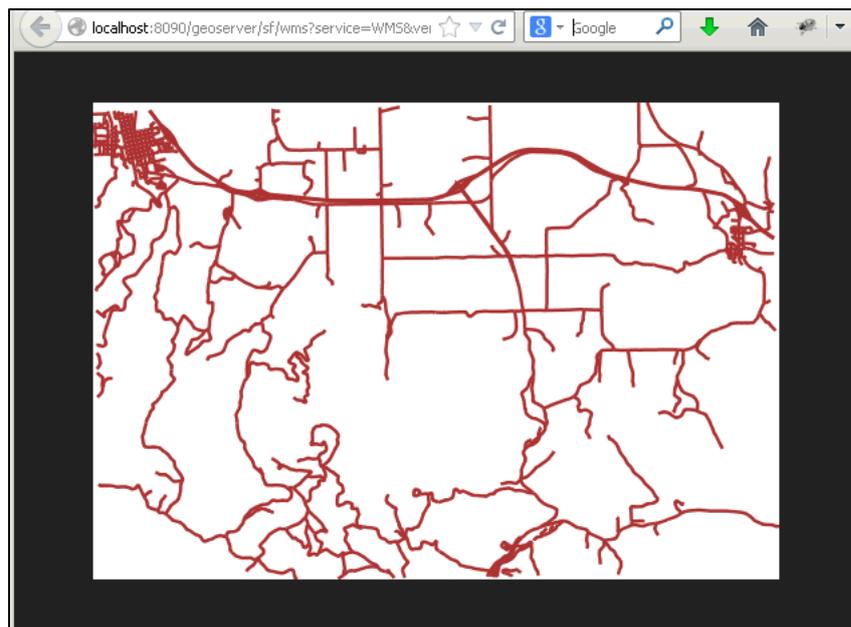


Figure 29: Sample WMS request

⁴¹ OpenGIS Web Map Service Client (WMS) Implementation Specification:

http://portal.opengeospatial.org/files/?artifact_id=14416

3.4 WFS/WFS-T

If maybe not for the pilots, there sure is a need anticipated for further project reusability to store additional context-relevant data not transportable by SOS/WaterML2. This information can be imported into the WDW using the transactional feature from the WFS. Additionally, SOS/WaterML2 transportable data is being offered using WFS, granting access to other clients either not being able to process WaterML2 or explicitly using WFS to avoid protocol overhead from WaterML2.

There is a great range on query parameters that are in detail explained in the OGC specification OpenGIS Web Feature Service 2.0 Interface Standard (also ISO 19142)⁴². A simple example will show the basic principle how to query a layer.

Example GET Request:

<http://localhost:8090/geoserver/cite/ows?service=WFS&version=1.0.0&request=GetFeature&typeName=cite:raw1379940028771&maxFeatures=50&outputFormat=GML2>

Response:

```

1 <?xml version="1.0" encoding="UTF-8"?><wfs:FeatureCollection
  xmlns="http://www.opengis.net/wfs" xmlns:wfs="http://www.opengis.net/wfs"
  xmlns:gml="http://www.opengis.net/gml" xmlns:cite="http://www.opengeospatial.net/cite"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.opengeospatial.net/cite http://localhost:8090/geoserver
/cite/wfs?service=WFS&version=1.0.0&request=DescribeFeatureType&
&typeName=cite%3Araw1379940028771 http://www.opengis.net/wfs http://localhost:8090
/geoserver/schemas/wfs/1.0.0/WFS-basic.xsd"><gml:boundedBy><gml:null>unknown</gml:null>
</gml:boundedBy><gml:featureMember><cite:raw1379940028771 fid="raw1379940028771.fid--
2736357b_14153cc5939_-7fcc"><cite:IDENTIFIER>demo-id-1</cite:IDENTIFIER>
<cite:TIMESTAMP>2013-09-17T12:40:26.899</cite:TIMESTAMP>
<cite:VALUE>11.101718454045</cite:VALUE></cite:raw1379940028771></gml:featureMember>
<gml:featureMember><cite:raw1379940028771 fid="raw1379940028771.fid--2736357b_14153cc5939_-
7fcb"><cite:IDENTIFIER>demo-id-1</cite:IDENTIFIER>
<cite:TIMESTAMP>2013-09-17T13:30:42.636</cite:TIMESTAMP>
<cite:VALUE>12.718776550698</cite:VALUE></cite:raw1379940028771></gml:featureMember>
<gml:featureMember><cite:raw1379940028771 fid="raw1379940028771.fid--2736357b_14153cc5939_-
7fca"><cite:IDENTIFIER>demo-id-1</cite:IDENTIFIER>
<cite:TIMESTAMP>2013-09-17T13:31:37.786</cite:TIMESTAMP>
<cite:VALUE>10.7571631306973</cite:VALUE></cite:raw1379940028771></gml:featureMember>

```

Listing 19: Sample WFS response

An additional layer can be added to WFS either through a web dialog or by calling a REST interface. Afterwards, the content of the layer can be altered by WFS-T. Possible transactional operations are *LockFeature*, *GetFeatureWithLock* and *Transaction* with Insert-, Update-, Replace- and Delete-Action. The OGC specification explains the possible operations in detail. For examples, the Geoserver provides a collection of sample requests. Here only a brief example showing the structure of a *Transaction* operation with an update-Action (Listing 20).

⁴² OpenGIS Web Feature Service 2.0 Interface Standard (also ISO 19142):

http://portal.opengeospatial.org/files/?artifact_id=39967

```
1 <wfs:Transaction service="WFS" version="1.0.0"
2   xmlns:topp="http://www.openplans.org/topp" xmlns:ogc="http://www.opengis.net/ogc"
3   xmlns:wfs="http://www.opengis.net/wfs">
4   <wfs:Update typeName="topp:tasmania_roads">
5     <wfs:Property>
6       <wfs:Name>TYPE</wfs:Name>
7       <wfs:Value>street</wfs:Value>
8     </wfs:Property>
9     <ogc:Filter>
10      <ogc:FeatureId fid="tasmania_roads.1" />
11    </ogc:Filter>
12  </wfs:Update>
13 </wfs:Transaction>
```

Listing 20: WFS-T Transaction

4. Pilot Integration Manager

This chapter describes how to integrate new data sources into the WatERP WDW. This is supported by the Pilot Integration Manager (PIM). We first discuss general considerations for designing the PIM and then present the concrete usage with the two WatERP pilot examples, SWKA and ACA.

4.1 General module design considerations

The pilot site is where the end users' (pilots') existing observation results have to be collected. For different end users, the measured data will probably be stored in different formats. In spite of accessing the data directly from the data warehouse (and thus making the WDW dependent from the pilot infrastructure), the pilot's data should be transferred through a standardized data format (WaterML2) and data-exchange protocol (SOS). Furthermore, SOS and WaterML2 have been chosen for pilot-data integration in order to decouple the data gathering from the data bases or systems that the pilots use. Additionally, any new sensor data could thus be integrated without any additional overhead, assuring both reusability and flexibility in data-integration management. Figure 30 describes the process for a pilot integration module which serves as an adapter to transform the existing sensor data into WaterML2 and then publish this information via an SOS server.

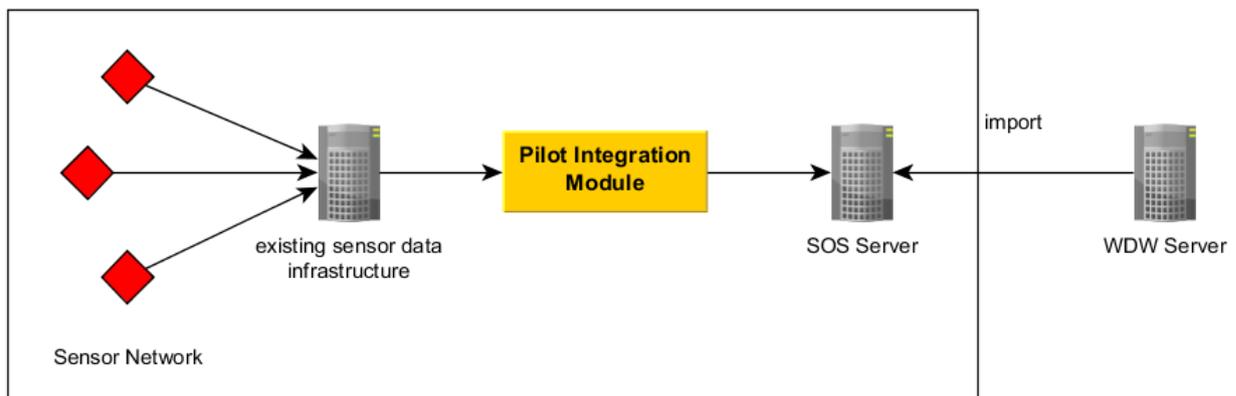


Figure 30: Integration of existing pilot infrastructure

In the case of an organization which directly feeds the SOS server from their sensor-network infrastructure (as described in OGC Sensor Web Enablement (SWE)⁴³), there would no a need for any kind of integration tool. As the sensor network would directly post their observation results to an SOS server instance, the WDW could directly access the existing SOS server that collects the observation results. Figure 31 shows the case when there is no need for any additional module because the sensor

⁴³ Sensor Web Enablement: <http://www.openeospatial.org/ogc/markets-technologies/swe>

network directly populates the SOS server that also serves as an interface to the WDW. However, even in this case, in order to be fully interoperable, the existing SOS infrastructure would need to use WaterML2 as data-exchange format (which is not necessarily the case, because the SOS protocol is, in principle, data-format agnostic).

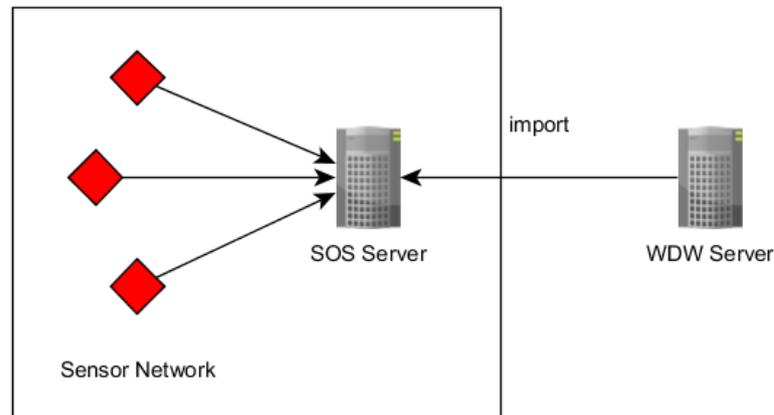


Figure 31: Integration of an existing SOS infrastructure

For all other cases (Figure 30) a separate integration module had to be implemented which is described in the following: Integration of an existing observation-data infrastructure has to be done in three different steps, namely (1) the data access, (2) the data mapping, and (3) the data import. The steps differ regarding if they are context specific:

Data Access:

A procedure to access the data has been defined. The main step is to identify data sources. At this point, three goals have been taken into consideration: (i) Access time; (ii) client data access and integration; (iii) stability of the existing infrastructure. Access time is one critical matter as one needs to provide the observation results as close as possible to real-time. For each data source/pilot it has to be discussed how close one can come to this goal. Another goal is that in case there is a way to access client data over a standard interface, the integration module should use it in order to improve the reusability of the implementation for future pilots. Last, the integration should not compromise the stability of the existing infrastructure.

When the investigation of concrete data sources/pilots started in the WatERP project, it became clear that the infrastructure to be integrated into the WDW is highly differing. At SWKA, the PIM must deal with a rather simple database table whereas at ACA, data had to be accessed through a WaterOneFlow service call. For the design of the PIM it became apparent that it would have to allow highly pilot-specific implementations of the data access. For other pilots the integration might require:

- Reading the data from a single or multiple data bases (as in SWKA).
- File parsing.
- Calling services (as with ACA). Here, format and protocol can differ considerably.

- Screen scraping or embedding legacy code into the integration module.
- A mix of the access mechanisms above.

Consequently, the general design of the integration module must enable the data-access layer to be very flexible. Nevertheless, there is a possibility that the implementation for one pilot can be used for other pilots as well, in case that the pilots use the same infrastructure to make their observation results available.

Data Mapping:

In data mapping, there are many possible strategies to select suitable pilot's data in order to build a WaterML2 document:

- The data that is required to populate the WaterML2 document might already be available via a database or a service call. However, this data has to be transformed.
- If not all required data is available in the pilot infrastructure, this data has to be provided by the integration module.

In the concrete WatERP pilot use cases, for ACA, the main information for the data mapping is provided by the existing data infrastructure whereas for SWKA, the PIM must provide most of the mapping information. The implementation depends on the information the data access supplies. In consequence, for the general PIM design the same level of flexibility is required for the data mapping and for the data access.

Import Data:

Here the mapping information and the time series have to be transformed into WaterML2 and this has to be sent to the SOS server as the external interface to the WDW for observations registration and storage.

For both, ACA and SWKA, the result of data access and mapping consists of the same set of information. Therefore in both scenarios the generation of WaterML2 is identical as it is based on the same mapped data. Also, the interactions with the SOS server are identical.

In consequence for the WatERP project context, the data import is considered to not require any pilot-specific implementation. To enable the interaction of data import with data access and data mapping, which are highly pilot dependent, Java classes have been implemented that contain a set of information about mapping time series that is required for the WaterML2 generation. The combination of data access and data mapping has to provide these transfer objects and pass them to the data import layer.

If in future integration activities with other pilots the set of information to be added to WaterML2 changes or becomes client specific, this decision will have to be reconsidered.

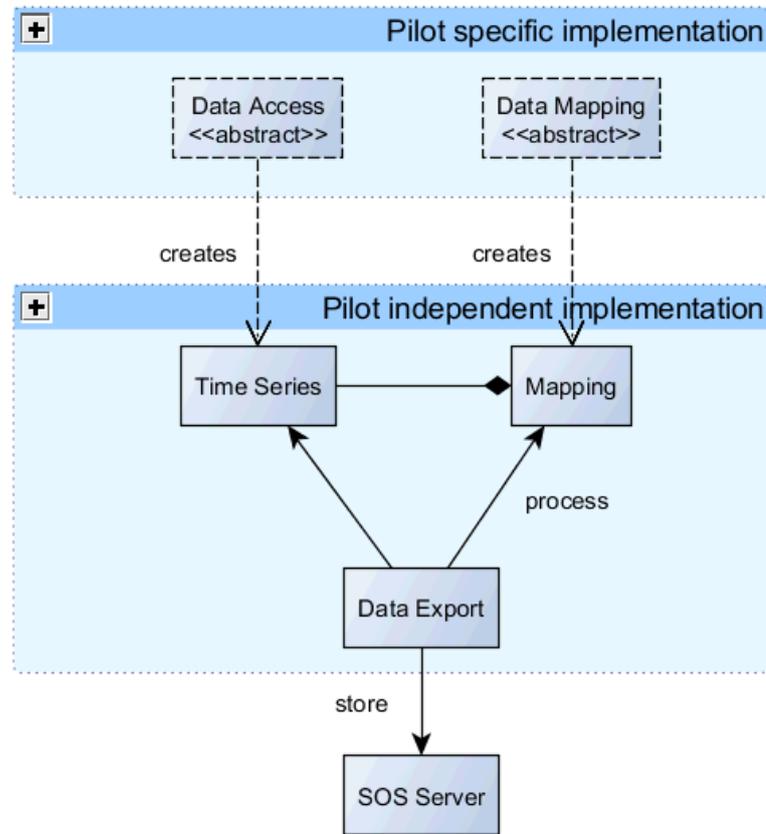


Figure 32: General module design of PIM

Figure 32 shows the general module design. Data access and data mapping are abstract but have to populate the time series and mapping information using a predefined structure. The data export which consumes the data, is independent from the actual pilot context as the pilot specifics have no influence on the export processing.

In the following two subsections, we will describe how the general considerations are instantiated within the PIMs for the SWKA and the ACA pilots.

4.2 SWKA Integration

SWKA runs a data warehouse which can be considered as a black box. All the sensor-network results are fed into this system. To make the data available for WatERP, the observation results have to be exported. In this export, the time interval and the step size of the time series have to be specified. Furthermore, the exported data has to some preparatory steps in order to clean and avoid anomalies in the information stored.

The work to analyze and process the data from the SWKA pilot site has been based on an export provided in the format of an Oracle database dump file⁴⁴ which had been created by SWKA containing a snapshot of the observation results. The file was imported in an Oracle⁴⁵ database on the development site to implement the data access. From the exemplary table depicted in Figure 33, it can be noted that the sensor data, with other data fields such as quality and rate, is a series of time and value pairs. In this table PP_ID is the sensor ID and each sensor has different readings at different times. Other sensor related information such as Feature of Interest, Procedure, Phenomenon, Observation, Location along with PP_ID are provided via an Excel file also shown in Figure 33. This data has been inserted into a PostgreSQL database table. The PostgreSQL comes with the PostGIS⁴⁶ add-on that consists of a geospatial database which includes three key features: (i) spatial types, (ii) spatial indexes, and (iii) special functions.

	PP_ID	AR_DATE	DS	AR_FLAG	PR_DATE	PR_VALUE	PR_QUALITY	RATE	COMMENT_ID
1	15866	01-JUN-12	0	0000	01-JUN-12	4.245	0009	0	(null)
2	13966	01-JUN-12		0000	01-JUN-12	2160	0005	0	(null)
3	16229	01-JUN-12		0000	01-JUN-12	2.055	0002	0	(null)
4	14637	01-JUN-12		0000	01-JUN-12		0009	0	(null)
5	16635	01-JUN-12		0000	01-JUN-12	0.0375	0009	0	(null)

FOI	Phenomenon	Procedure	TimeSeries	Observation	PPID
					Variable Id
MainReservoir	Flow Discharge	Sensor lecture		TankInflow_obs	14753
MainReservoir	Flow Discharge	Sensor lecture		TankOutflowMain_obs	14788
MainReservoir	Flow Discharge	Sensor lecture		TankOutflowSub1_obs	14776
MainReservoir	Flow Discharge	Sensor lecture		TankOutflowSub2_obs	14777
MainReservoir	Flow Discharge	Calculated		TankOutflowSub_obs	-
MainReservoir	Water Table	Sensor lecture		TankLevelSouth_obs	14763
MainReservoir	Water Table	Sensor lecture		TankLevelNorth_obs	14762
MainReservoir	Water Table	Calculated		TenkLevelAver_obs	-
MainReservoir	Tank Volume	Sensor lecture		TankVolume_obs	14768
HwP1-Pump	Flow Discharge	Sensor lecture		Flow Discharge_obs	
HwP1-Pump	CurveFlowHead	DesignedCurve		Flow Discharge_obs	
HwP1-Pump	CurveFlowEfficiency	DesignedCurve		CurveFlowEfficiency_obs	

Figure 33: Database table from pilot site SWKA

The first step towards integration is based on converting the accessed data into WaterML2 such that the accessed data can be stored into an SOS server. To accomplish this task, the designed and populated ontology (WatERP WP1), with the SWKA pilot, is consulted. According to this ontology, the time series are described as “Results” entities that are aligned with the “MeasurementTimeSeries” concept in the OGC WaterML2 schema inside the “TimeSeriesObservationMetadata.xsd” subschema (Figure 34). A client side application as part of PIM is built in Eclipse⁴⁷ using Maven⁴⁸, which will be used by pilots to perform this task.

⁴⁴ Data Pump Export http://docs.oracle.com/cd/B28359_01/server.111/b28319/dp_export.htm#i1006388

⁴⁵ Oracle Database (<http://www.oracle.com/technetwork/database/enterprise-edition/downloads/index.html>)

⁴⁶ Introduction to PostGIS <http://revenant.ca/www/postgis/workshop/introduction.html>

⁴⁷ Eclipse: <http://www.eclipse.org/>

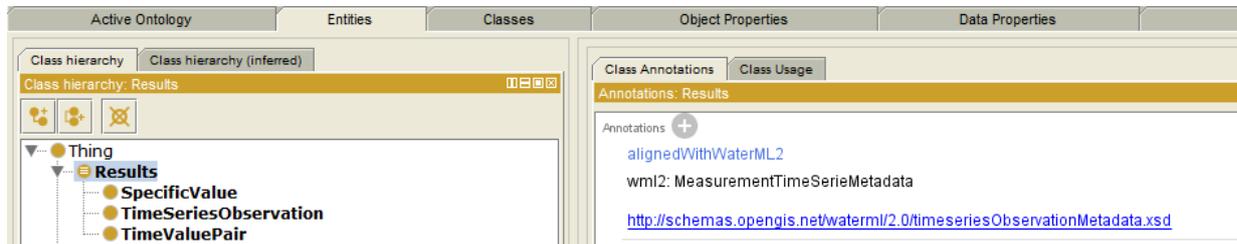


Figure 34: Pilot site SWKA ontology

The application consists of a wrapper module to perform each step at this level. A flow chart diagram is shown in Figure 35 to describe the implementation steps. A Database wrapper first reads sensor information and location Excel files, puts this data into the PostgreSQL data table and then gets sensor information from the PostgreSQL table and matches PP_ID in the Oracle table to get time series for the sensor.

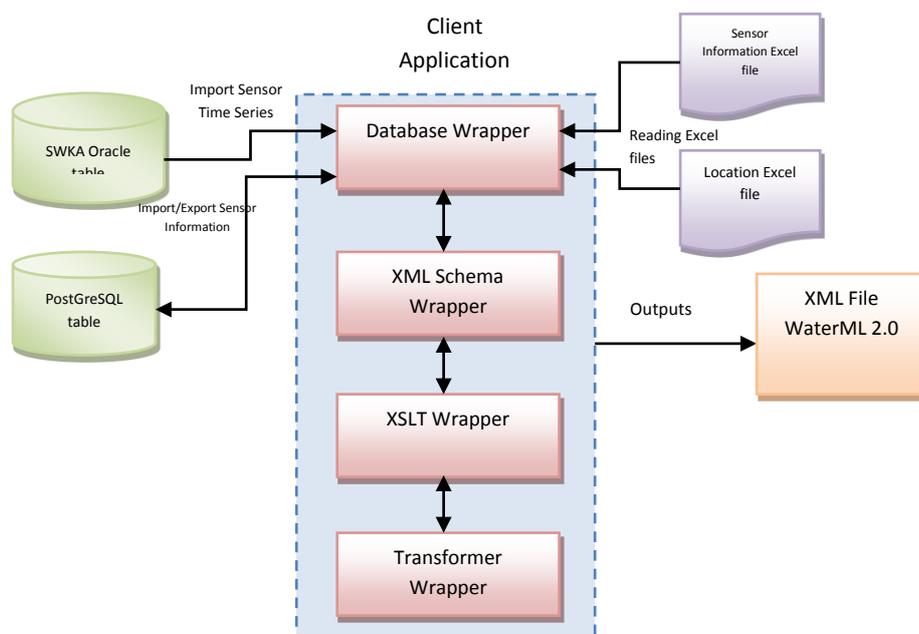


Figure 35: Flow chart of client application

An XML schema defines the look of an XML document, and the transformer wrapper transforms data from a database into a WaterML2 document with the help of an XSLT wrapper which provides style sheets. The output of this application is an XML document in WaterML2 format. It contains a root element *MeasurementTimeSeries* with several time-series observations inside a *Point* element as shown in Listing 21. To store the observations, the WaterML2 data is sent to the SOS Server.

⁴⁸ Apache Maven Project: <http://maven.apache.org/index.html>

```

<?xml version="1.0" encoding="UTF-8"?>
- <wml2:MeasurementTimeSeries xmlns:wml2="http://www.opengis.net/waterml/2.0"
  - <wml2:point>
    - <wml2:MeasurementTVP>
      <wml2:time>2012-06-01T00:01:00.000+01:00</wml2:time>
      <wml2:value>109.9161</wml2:value>
    </wml2:MeasurementTVP>
  </wml2:point>
  - <wml2:point>
    - <wml2:MeasurementTVP>
      <wml2:time>2012-06-01T00:02:00.000+01:00</wml2:time>
      <wml2:value>109.9161</wml2:value>
    </wml2:MeasurementTVP>
  </wml2:point>
</wml2:MeasurementTimeSeries>

```

Listing 21: Output in the form of WML 2.0

The final flowchart diagram of the PIM client application for SWKA is shown in Figure 36.

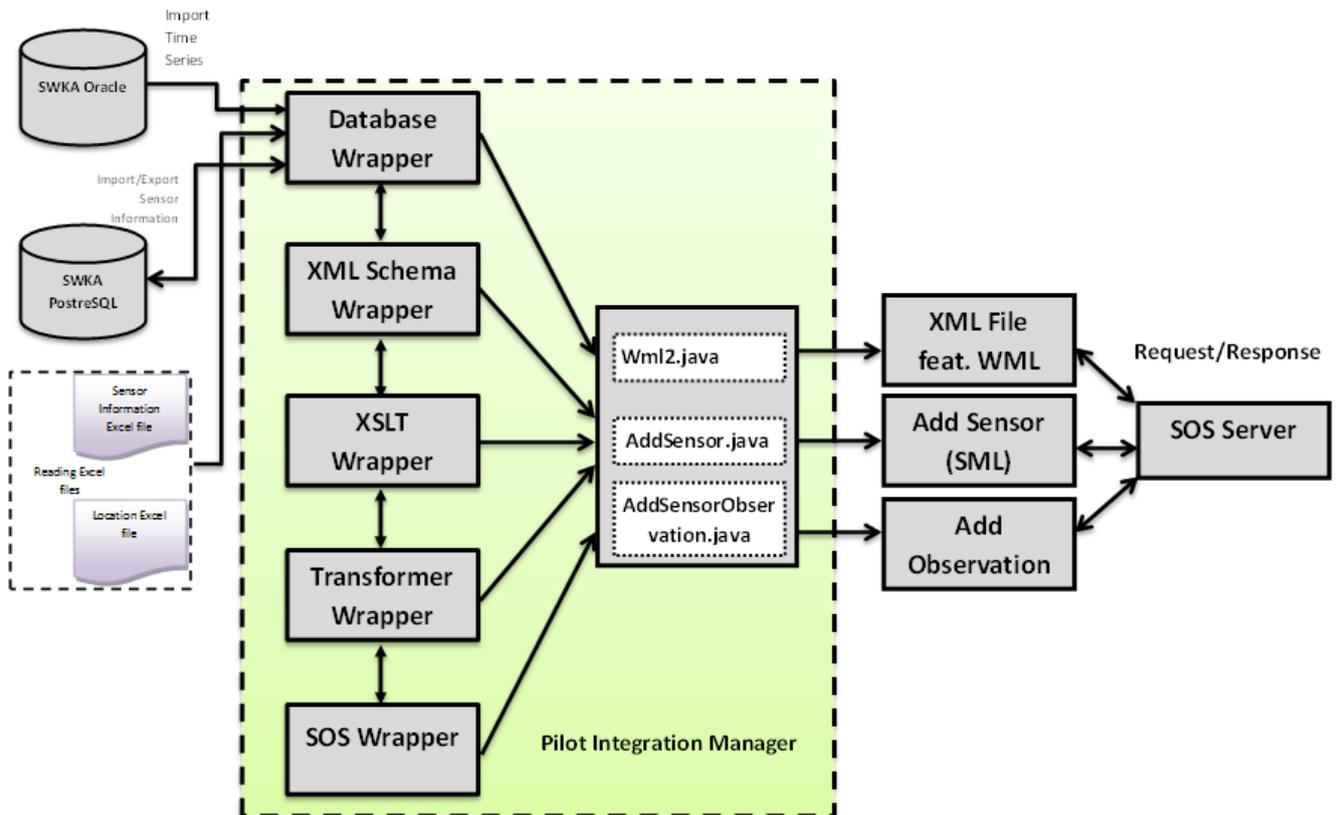


Figure 36: Flow chart diagram of final client application for the SWKA instantiation

4.2.1 Load Sensor Metadata

The sensor data is populated from Excel files into PostgreSQL database. The database connection settings are stored in configuration files. In order to perform this task, the Hibernate⁴⁹ library is applied.

4.2.2 Configure New Sensor

There are two types of sensors stored in the SWKA database: in-situ and dynamic. SOS provides an API for managing deployed sensors and retrieving sensor data and specifically observation data. This API has the three main operations *GetObservation*, *DescribeSensor* and *GetCapabilities*. The final task of the pilot integration manager is to add the sensor to WDW server. To do this, the sensor needs to be inserted into SOS first.

The next task of PIM is to register the sensor and to insert sensor observation from the database which is possible due to the transactional operations *RegisterSensor* and *InsertObservation* provided by the SOS. For this reason the client application is fitted with two more SOS wrapper modules, one module with the purpose of registering a sensor and the other module for inserting sensor observations. The standard used for registering a sensor is Sensor Model Language (SML) and inserting an observation is encoded in XML by following the Observation and Management Specification (O&M). Each *RegisterSensor* operation has two mandatory attributes of service and version. The request for an operation contains the *SensorDescription* element. The response to a *RegisterSensor* request contains an *AssignedSensorID* which is the identifier assigned by the SOS to designate the new sensor. The last part of this identifier is the sensorID which in this case is PP_ID. The identifier is of type *anyURI* and must be either a URN or a URL that resolves to a procedure which is advertised in an offering in the SOS capabilities response. The request for registering a sensor and the response from SOS is shown in Listing 22 and Listing 23, respectively.

```
<terminated> AddSosSensor [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (23 Sep 2013 13:07:02)
Sending request to http://waterp.env.disy.net:4711/52nSOSv3.5.0_EE/sos

Response status code: 200
Response body:
<?xml version="1.0" encoding="UTF-8"?>
<sos:RegisterSensorResponse xmlns:sos="http://www.opengis.net/sos/1.0" xmlns:xsi="http://www.w3
  <sos:AssignedSensorId>urn:ogc:object:feature:Sensor:DEMO:sensor-14642</sos:AssignedSensorId>
</sos:RegisterSensorResponse>
```

Listing 22: Response from SOS for sensor registration

⁴⁹ Hibernat <http://hibernate.org/ogm/>

```

<?xml version="1.0" encoding="UTF-8"?>
<RegisterSensor service="SOS" version="1.0.0" xmlns="http://www.opengis.net/sos/1.0"
xmlns:swe="http://www.opengis.net/swe/1.0.1" xmlns:ows="http://www.opengis.net/ows"
xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:gml="http://www.opengis.net/gml"
xmlns:ogc="http://www.opengis.net/ogc" xmlns:om="http://www.opengis.net/om/1.0" xmlns:sml="http://www.opengis.net/sensorML/1.0.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.opengis.net/sos/1.0
http://schemas.opengis.net/sos/1.0.0/sosRegisterSensor.xsd
http://www.opengis.net/om/1.0 http://schemas.opengis.net/om/1.0.0/extensions/observationSpecialization_override.xsd" >
  <SensorDescription>
    <sml:SensorML version="1.0.1">
      <sml:member>
        <sml:System xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" >
          <sml:identification>
            <sml:IdentifierList>
              <sml:identifier>
                <sml:Term definition="urn:ogc:def:identifier:OGC:uniqueID">
                  <sml:value>urn:ogc:object:feature:Sensor:DEMO:sensor-14642</sml:value>
                </sml:Term>
              </sml:identifier>
            </sml:IdentifierList>
          </sml:identification>
          <sml:capabilities>
            <swe:SimpleDataRecord>
              <swe:field name="FeatureOfInterestID">
                <swe:Text definition="FeatureOfInterest identifier">
                  <swe:value>Demo-FOI-1</swe:value>
                </swe:Text>
              </swe:field>
            </swe:SimpleDataRecord>
          </sml:capabilities>
        </sml:System>
      </sml:member>
    </sml:SensorML>
  </SensorDescription>
</RegisterSensor>

```

Listing 23: Request to register a sensor

The SWKA PIM generates an XML file in the format of SML which is sent to SOS to register the sensor. If sensor was registered before, then re-registration request receives exception message from SOS in response stating “The sensor is already registered at this SOS”.

4.2.3 Trigger Import / Insert Observation

Once the sensor is registered with SOS, the PIM can begin inserting observations for that sensor. The *AddSosObservation* wrapper in the client application (i) gets sensor information from the combination of database tables as described above, (ii) creates an XML document using XML schema, and (iii) transforms it into observation and measurement standard using XSLT. Each *InsertObservation* request includes the *AssignedSensorID* returned from the *RegisterSensor* operation. The request of *InsertObservation* and response from SOS is shown in Listing 24 and Listing 25.

```
<?xml version="1.0" encoding="UTF-8"?>
<sos:InsertObservation xmlns:ogc="http://www.opengis.net/ogc" xmlns:sa="http://www.opengis.net/sampling/1.0" xmlns:gml="http://www.opengis.net/gml"
xmlns:ows="http://www.opengis.net/ows/1.1" xmlns:om="http://www.opengis.net/om/1.0" xmlns:InsertObservation="http://Integration.WatERP.eu/InsertObservation"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xlink="http://www.w3.org/1999/xlink" xmlns:swe="http://www.opengis.net/swe/1.0.1"
xmlns:sos="http://www.opengis.net/sos/1.0" version="1.0.0" service="SOS">
  <sos:AssignedSensorId>urn:ogc:object:feature:Sensor:DEMO:sensor-15866</sos:AssignedSensorId>
  <om:Measurement>
    <om:samplingTime>
      <gml:TimeInstant>
        <gml:timePosition>2013-09-23T14:24:57.239+01:00</gml:timePosition>
      </gml:TimeInstant>
    </om:samplingTime>
    <om:procedure xlink:href="urn:ogc:object:feature:Sensor:DEMO:sensor-15866"/>
    <om:observedProperty xlink:href="urn:ogc:def:phenomenon:waterpressure"/>
    <om:featureOfInterest>
      <sa:SamplingPoint gml:id="demo-id-15866">
        <gml:name>demo-id-15866</gml:name>
        <sa:sampledFeature xlink:href="urn:ogc:def:nil:OGC:unknown"/>
        <sa:position>
          <gml:Point>
            <gml:pos srsName="urn:ogc:def:crs:EPSG:4326">0 0</gml:pos>
          </gml:Point>
        </sa:position>
      </sa:SamplingPoint>
    </om:featureOfInterest>
    <om:result uom="bar">4.1832</om:result>
  </om:Measurement>
</sos:InsertObservation>
```

Listing 24: Request to SOS for InsertObservation

It is important to note that the response from SOS contains *AssignedObservationID* for the new observation entered. If the same observation is entered twice, an exception message is received in the response body stating that the observation is already present in the SOS database.



```
<terminated> AddSosObservation [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (23 Sep 2013 14:24:49)
<?xml version="1.0" encoding="UTF-8"?><sos:InsertObservationResponse service="SOS" version="1.0.0" xmlns:sos="http://www.opengis.net/sos/1.0" xmlns:swe="http://www.opengis.net/swe/1.0.1" version="1.0.0" service="SOS">
Sending request to http://waterp.env.disy.net:4711/52nSOSv3.5.0_EE/sos
Response status code: 200
Response body:
<?xml version="1.0" encoding="UTF-8"?>
<sos:InsertObservationResponse xmlns:sos="http://www.opengis.net/sos/1.0">
  <sos:AssignedObservationId>o_6537</sos:AssignedObservationId>
</sos:InsertObservationResponse>
2013-09-23 14:24:57,632 [main] DEBUG util.JaxbTransformer (marshalAndTransform:53) - Output message:
```

Listing 25: Response from SOS

4.3 ACA Integration

This subsection describes the considerations and activities to integrate the ACA sensor data into the WDW. It first depicts the current infrastructure that the PIM has to integrate. Next, it analyses how ACA data can be mapped to WaterML2. More general information about the ACA case can be found in WatERP deliverable D7.3.1-“Implementation of WDW”.

4.3.1 WaterOneFlow

In autumn 2013, ACA installed a WaterOneFlow server as a central instance to obtain sensor-observation results – WaterOneFlow was developed by the Consortium of the Advanced of Hydrological Sciences Inc. (CUAHSI) for use in the US as a standard mechanism for the transfer of hydrologic data. The WaterOneFlow web service uses WaterML1 for encoding hydrological observations.

The structure of the data transferred over WaterOneFlow is very similar to the Observation Data Model (ODM) of the CUAHSI Hydrologic Information System (HIS)⁵⁰. A full diagram of the ODM is shown in Appendix I: Observation Data Model (ODM) of the CUAHSI HIS. Figure 37 shows the simplified structure of WaterOneFlow data.

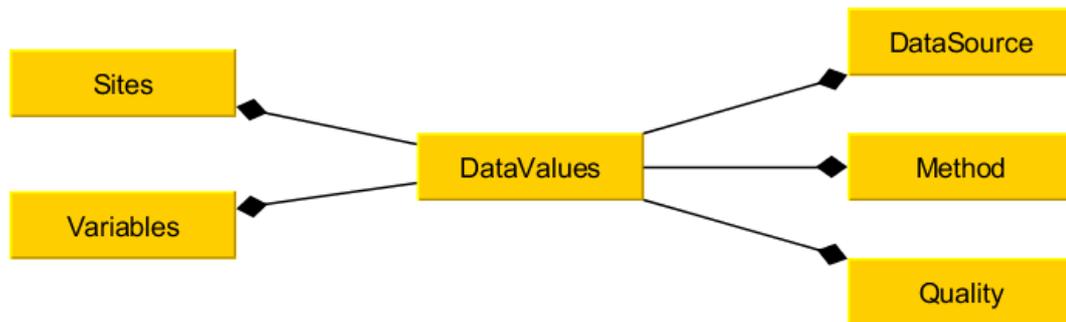


Figure 37: Simplified WaterOneFlow data structure

4.3.1.1 Web Service Interface

A summarizing diagram of all WaterOneFlow web service methods is shown in Appendix II: WaterOneFlow Service. In this subsection, all methods will be shortly explained. Table 12 gives a brief overview of the business methods, not the additional authorization calls.

Method	Description
getSites	The <code>getSites</code> method provides a description of the sites. An optional list of <code>sitelds</code> can be passed as a parameter.
getSitesXML	The method <code>getSitesXML</code> works very similar to <code>getSites</code> , but it separates the SOAP document from the site information by embedding the sites XML as a string in the SOAP document.
getSiteInfo	Given a site number, this method returns the site's metadata. It also lists all variables that exist for the specified site, Like with <code>getSitesXML</code> this method provides the information within an embedded XML document.
getSiteInfoObject	The method <code>getSiteInfoObject</code> corresponds to <code>getSiteInfo</code> except that the site information is part of the SOAP document.
getValues	The <code>getValues</code> request queries the time series for a specific site and variable. Besides the time series the <code>getValues</code> response contains a description of the site and the variable which have been queried. For each value it also provides

⁵⁰ <http://his.cuahsi.org/odmdatabases.html>

	<p>the identifier for source, quality control and procedure. The source, quality control and feature entities referenced by the values are also embedded inside the response but are linked with the value tags to avoid redundant information, and thereby reduce the document size.</p> <p>Like with <code>getSitesXML</code> and <code>getSiteInfo</code> the <code>timeSeriesResponse</code> is separated from the SOAP body and as a string embedded in the <code>GetValuesReturn</code> tag.</p>
<code>getValuesObject</code>	<p>The <code>getValuesObject</code> request is similar to the <code>getValues</code> request except that the time series are directly embedded into the SOAP body. The document structures differ but regarding the content the description of <code>getValues</code> applies also to <code>getValuesObject</code>.</p>
<code>getVariableInfo</code>	<p>The method <code>getVariableInfo</code> provides a description of one specific variable or all existing variables, depending on the parameters passed. The main information which is returned is the name of the variable and the unit of the values provided. Like with <code>getSitesXML</code>, <code>getSiteInfo</code> and <code>getValues</code>, this method strictly separates the <code>variablesResponse</code> from the SOAP body by embedding it as a string.</p>
<code>getVariableInfoObject</code>	<p>Except for some differences in the response structure, the <code>getVariableInfoObject</code> works exactly in the same way as the <code>getVariableInfo</code> method. It can also return the information of one specific variable or all variables. The only difference with respect to <code>getVariableInfo</code> is that the response is directly embedded inside the SOAP body.</p>

Table 12: WaterOneFlow method description

4.3.2 General Processing Steps

This Subsection describes the general steps to integrate the WaterML1 information provided by ACA through the WaterOneFlow service.

The application is implemented as a web application which polls the ACA server in configured intervals. The sensors' meaning pairs of site identifier and variable identifier are stored inside a database table. The full content of the table is listed in Table 13.

Column	Description
ID	Identifier
SITE_CODE	Site identifier
VARIABLE_CODE	Variable identifier
START_POLLING	Start date from which on the time series should be converted to SOS/WaterML2
HIGHEST_TIMESTAMP	Highest timestamp of the observation results that are already converted to SOS/WaterML2
DEFINED_IN_SOS	'false' indicates that the sensor isn't yet defined in the SOS server. Once the application creates the SOS sensor the column will be changed to true.

Table 13: Columns of sensor table in ACA pilot integration manager

As Hibernate is used to access the data and the database configuration is extracted into a properties file the application isn't bound to one database implementation. Table 5 lists all database dialects that are supported by Hibernate.

Figure 38 shows the processing steps required to integrate ACA into the water data warehouse.

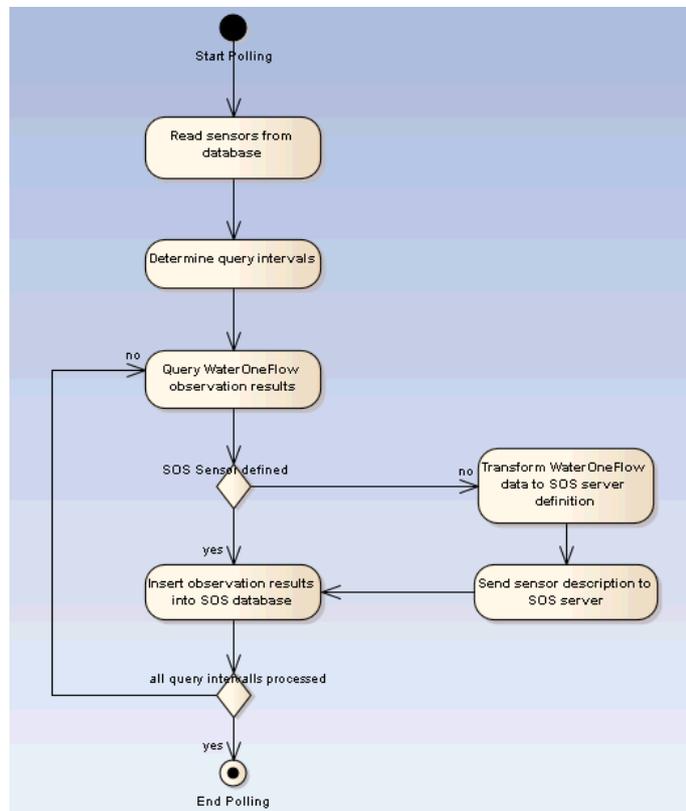


Figure 38: General processing steps for ACA integration

The application creates the ACA sensor before the first observation result is to be inserted. In order to avoid polling all the historical data when adding a new sensor, the timestamp where the polling starts can be configured.

To reduce the risk of destabilising the WaterOneFlow server, the ACA server has a maximum time interval that can be queried by one request. If this limit is exceeded, the server responds with an error code to avoid running out of memory. For this reason (and also not to cause instabilities in the PIM itself), a maximum query interval is defined in the configuration of the PIM. First, for each polled sensor the available time range available is determined by calling `getSiteInfoObject`. This data is merged with the additional information as `START_POLLING` and `HIGHEST_TIMESTAMP` stored in the sensor table. With this data the PIM determines the time interval, it has to be queried on the ACA server. After that process, the time interval is separated into subintervals as to meet the restrictions of the server about maximum time intervals allowed for querying. The observation results are obtained by calling `getValues`. Next, the response is analysed and converted to WaterML2. The current implementation realises a very simple conversion:

- The `sensor identifier` is generated by combining site and variable identifier of WaterOneFlow. The pattern is `<siteCodeNetwork>-<siteCode>/<variableCodeVocabulary>-<variableCode>`. For example for site S:430141-002 and variable S:3965408 the sensor identifier would be S-430141-002/S-3965408.
- The `observed properties` are created by applying the pattern `urn:x-ogc:def:phenomenon:OGC:<variableName>`. As for phenomena, there exists already a standard list of predefined URNs⁵¹. When mapping ACA data these predefined URNs should be used if possible. Therefore the mapping should map the variable identifier or the variable names to the phenomenon URNs to be used for WaterML2 generation.
- For `offering` and `offeringName` the variable name from WaterOneFlow is used. This mapping is to be reconsidered as it might be altered according to the changed mapping of the observed properties. Like with the observed property a mapping from the ACA variable name or identifier to the offering can be added to solve the problem.
- The `feature of interest` is created based on the pattern `<siteCodeNetwork>-<siteCode>`. The name of the feature of interest is taken from the site name in WaterOneFlow.
- `Unit` and `location` are transferred from their counterparts in WaterOneFlow. For location SRS the prefix `urn:ogc:def:crs:EPSG::` is added to meet the restrictions of the SOS protocol.

51

https://www.seegrid.csiro.au/subversion/xml/OGC/branches/SWE_gml2/sweCommon/current/examples/phenomena.xml

4.3.3 Implementation Details for the ACA PIM

This Subsection describes in detail the mechanisms on which the integration of ACA is based. First, it will describe the polling mechanism and afterwards how the WaterOneFlow web service is being processed.

Figure 39 shows the polling of the ACA sensor data. A poller timer task is triggered in configured intervals. It first loads all sensors from the database and afterwards starts the polling for all sensors. To be able to process many sensors in parallel the polling of a sensor is moved into a Runnable which is the passed to a TaskExecutor which controls a thread pool. The number of threads to process the polling can be adjusted in the configuration without changing the code. By default, the pool has an initial size of 5 and a maximum size of 10 parallel threads. Due to the asynchronous processing there is no guarantee that all threads from the previous polling cycle are finished before the next polling cycle starts. Therefore a *SensorPollingSynchronizer* has been added to control the polling on a single sensor.

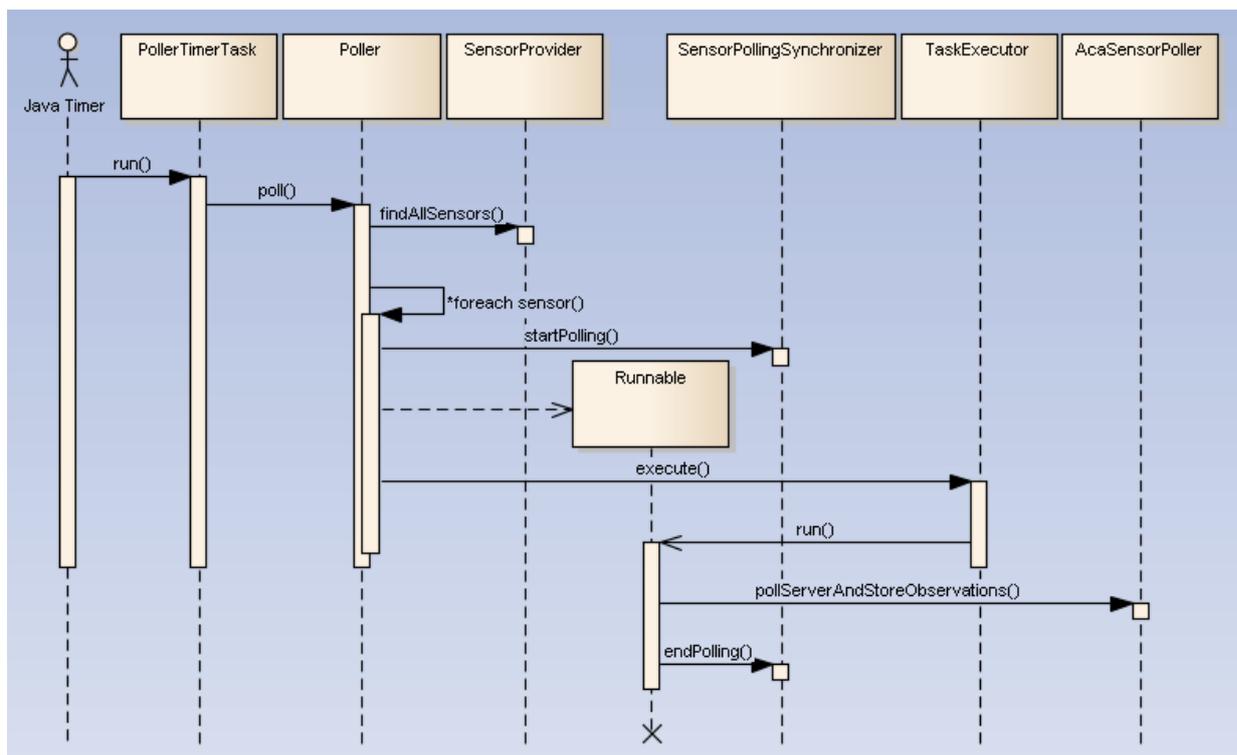


Figure 39: ACA polling

The polling for a single sensor is processed in the *AcaSensorPoller*. Figure 40 shows the sequence when processing a single sensor. First, the sensor information has to be obtained from the WaterOneFlow server. Afterwards, the poll intervals have to be computed before each poll interval is being processed. For each interval, the sensor results are queried and the content is being extracted. The sensor metadata is added to the SOS server in case it hasn't been defined yet. Then, the sensor results are inserted into the SOS server. Finally, for each interval the sensor data in the PIM sensor

table is updated. Thereby, the already polled interval wouldn't be processed again in case the polling process would abnormally terminate before all intervals have been processed.

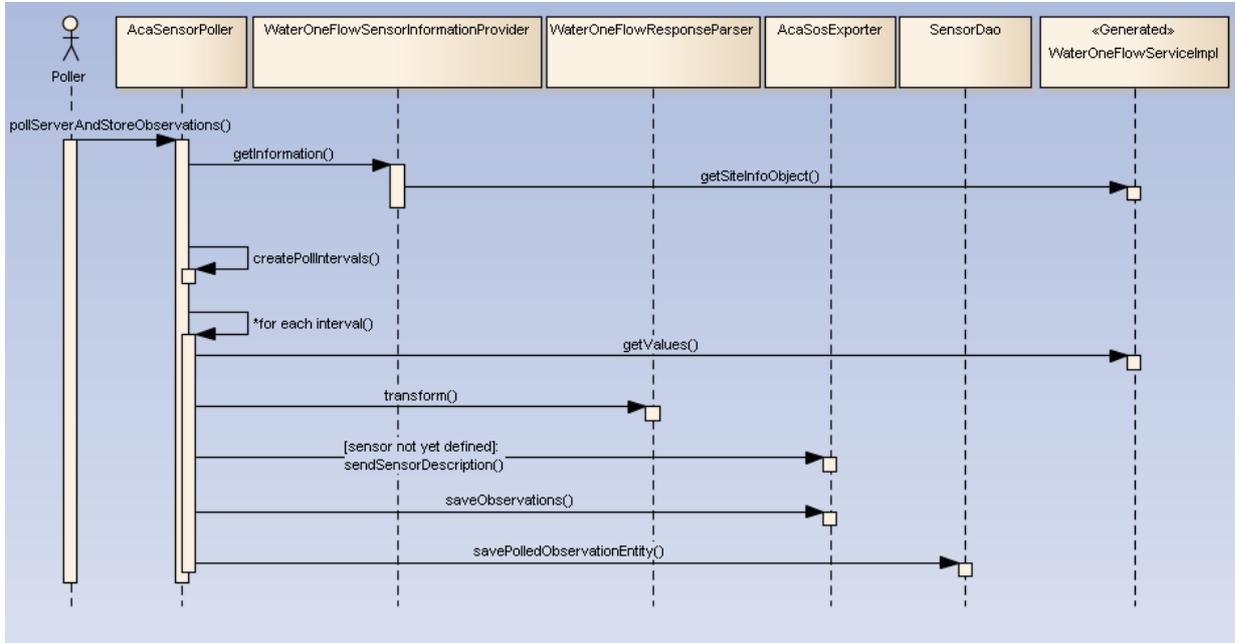


Figure 40: Polling of a single sensor

5. Installation

The Water Data Warehouse consists of a number of separate services that have to be installed. The deployment diagram in Figure 41 gives an overview over all the components and which communication channels exist.

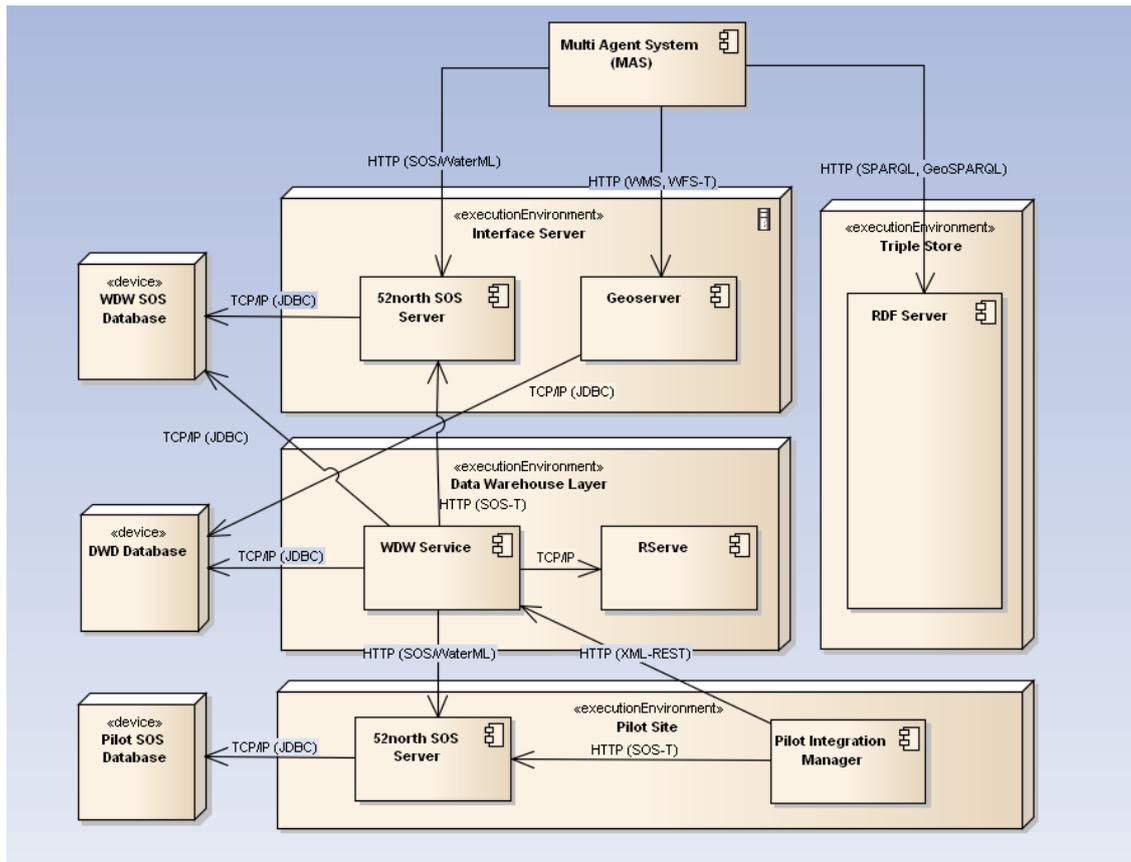
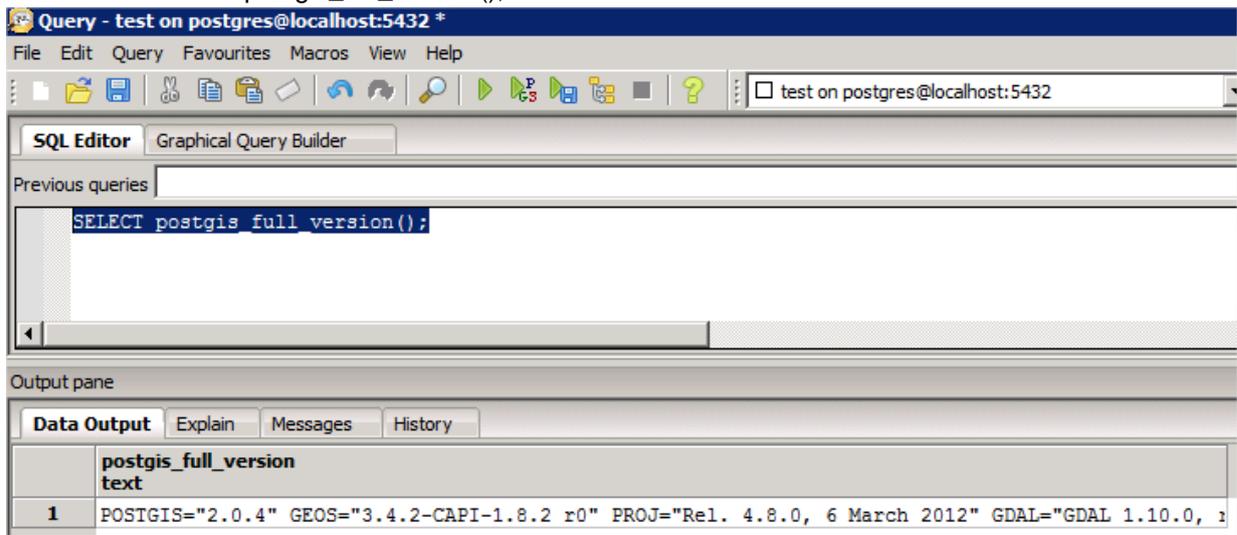


Figure 41: WDW deployment diagram

5.1 Installation of the triple store

In this section we provide the installation guide for the final prototype of the triple store. Further details about installation and usage of the tool can be found in WatERP deliverable D3.2. In the following procedure we assume that the operating system and the RDBMS (PostgreSQL 9.X) are already installed. The installation guide of PostgreSQL can be found online at <http://www.postgresql.org/docs/9.3/interactive/admin.html>.

- 1) Install PostGIS
 - a. Following the installation guide found on http://www.bostongis.com/PrinterFriendly.aspx?content_name=postgis_tut01 install postGIS
 - b. Add the spatial extensions to your database instance (e.g., “useekm”)
 - i. Using the tool PGAdmin III of PostgreSQL create a database with template postgis20_template
 - ii. In order to verify PostGIS on your database execute the query “SELECT postgis_full_version();”



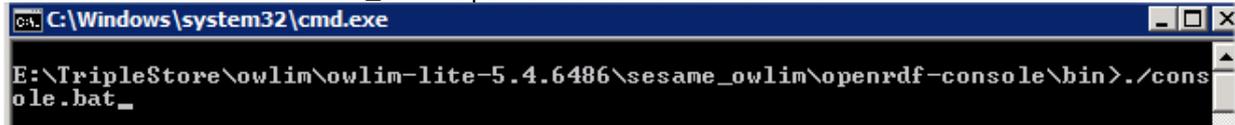
- 2) Update the configuration file of your uSeekM repository (e.g., “useekm-config.xml”) in order to populate the database connection settings i.e. if the database name is “useekm”, username is “postgres” and the password is “Po5t6r3s” the relevant section of the repository configuration file should look like the following extract:

```
<bean id="pgDatasource" lazy-init="true"
class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
  <property name="driverClassName" value="org.postgresql.Driver"/>
  <property name="url" value="jdbc:postgresql://localhost:5432/useekm"/>
  <property name="username" value="postgres"/>
  <property name="password" value="Po5t6r3s"/>
</bean>
```

- 3) Install Apache Tomcat 6 following the instructions found online at <http://tomcat.apache.org/tomcat-6.0-doc/setup.html>
- 4) Install OWLIM and uSeekM

- a. Install uSeekM HTTP Server by following the instructions at <https://dev.opensahara.com/projects/useekm/wiki/HttpServer>
- b. Download OWLIM from <http://download.ontotext.com>
- c. Unzip the OWLIM file (e.g. E:\TripleStore\owlim\owlim-lite-5.4.648\owlim-lite-5.4.6486.zip)
- d. Run the Sesame console once to create the data/configuration directory. If "OWLIMDIR" is the directory where you unzipped OWLIM run the command :

- i. OWLIMDIR\sesame_owlim\openrdf-console\bin\console.bat



```
C:\Windows\system32\cmd.exe
E:\TripleStore\owlim\owlim-lite-5.4.6486\sesame_owlim\openrdf-console\bin>./console.bat
```

- ii. Enter "quit ." to exit the console

- e. Create the Sesame console's templates directory:
 - i. mkdir "%APPDATA%\Aduna\OpenRDF Sesame console\templates"
 - f. Copy the OWLIM-Lite repository template file to the Sesame console templates directory:
 - i. E.g., copy owl-lite-5.4.6486\templates*.ttl "%APPDATA%\Aduna\OpenRDF Sesame console\templates"
 - g. Copy the OWLIM-Lite jar file to the Sesame console and Sesame server lib directories:
 - i. Copy E:\TripleStore\owlim\owlim-lite-5.4.648\owlim-lite-5.4.6486\lib\owlim-lite*.jar "C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\openrdf-sesame\WEB-INF\lib"
 - ii. Copy E:\TripleStore\owlim\owlim-lite-5.4.648\owlim-lite-5.4.6486\lib\owlim-lite*.jar "C:\Program Files\Apache Software Foundation\Tomcat 6.0\webapps\openrdf-workbench\WEB-INF\lib"
 - h. Run the Sesame console to create a new repository:
 - i. openrdf-sesame-2.5.0\bin\console.bat
 - ii. Connect to the Sesame server: > **connect http://localhost:8080/openrdf-sesame** . (from apache/useekm installation)
 - iii. Create a new repository: > **create owl-lite .**
 - iv. At this point values for a number of parameters are asked for, but all the defaults can be accepted pressing <return> for each question. When finished, the message 'Repository created' should be displayed, so exit the console: > **quit .**
 - i. Copy the OWLIM Java Beans wrapper class (provided in the file **iccs-spring-owlim-sail.v1.0.jar**) under the WEB-INF/lib directories of useekm Apache Tomcat webapps (openrdf-workbench and openrdf-sesame)

5) Create a new useekm repository

- a. Update a useekm configuration file (<https://dev.opensahara.com/projects/useekm/wiki/HttpServer#Configuration>) like the following :

```
<!-- This example uses the OWLIM-Lite v5.4 SpringOwlSail sail wrapper as
the underlying sail (npapag@mail.ntua.gr 2013)-->
    <bean id="sail" class="org.iccs.waterp.triplestore.SpringOwlSail" >
        <property name="properties">
            <map>
                <entry key="storage-folder"
value="E:\TripleStore\owlim-storage"/>
                <entry key="new-triples-file" value="new-triples-
file.nt"/>
            </map>
        </property>
    </bean>
```

```

        <entry key="ruleset" value="owl-max"/>
    </map>
</property>
<property name="dataDir">
    <bean class="java.io.File">
        <constructor-arg value="E:\TripleStore\datadir"/>
    </bean>
</property>
</bean>

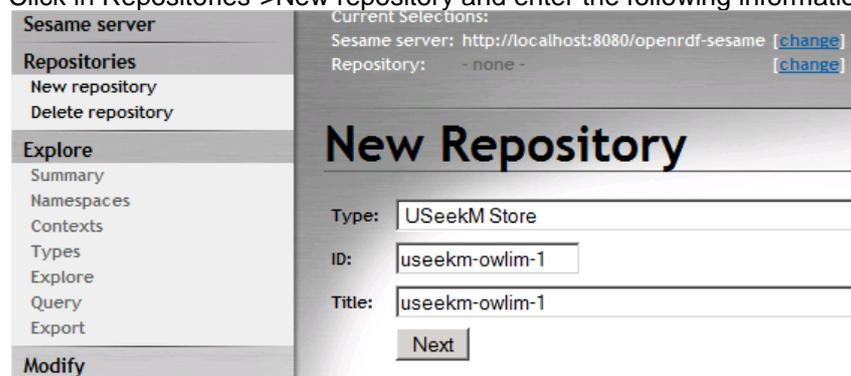
<bean id="pgDatasource" lazy-init="true"
class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="org.postgresql.Driver"/>
    <property name="url"
value="jdbc:postgresql://localhost:5432/useekm"/> <!-- CUSTOMIZE! -->
    <property name="username" value="postgres"/>
<!-- CUSTOMIZE! -->
    <property name="password" value="Po5t6r3s"/>
<!-- CUSTOMIZE! -->
</bean>

```

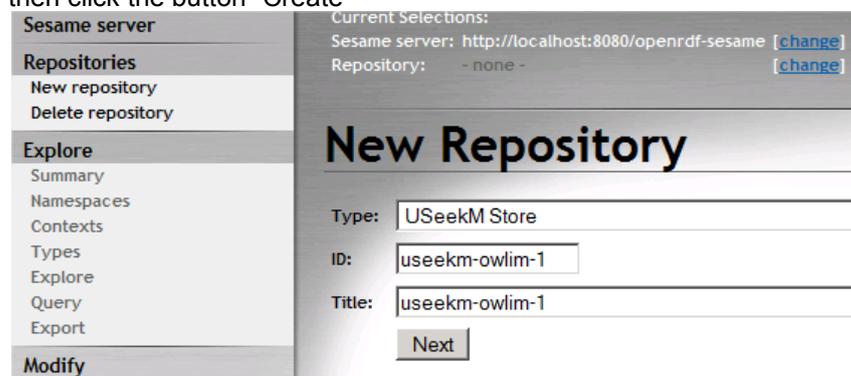
- b. Use the uSeekm/Sesame Workbench to create a new repository
 - i. Navigate with a web-browser to

<http://localhost:8083/openrdf-workbench/repositories/useekm-owlim-1/summary>

- ii. Click in Repositories->New repository and enter the following information



- iii. Click the button "Next", enter the location of your useekm configuration file and then click the button "Create"



- 6) Upload your ontologies to the repository
 - a. Select the repository that you created and navigate to Modify->Add

b. Click Explore->Explore to view your data

Subject	Predicate	Object	Context
georesource:Geom_points_1989682935	rdf:type	sf:Point	<file://cyprus-points.rdf>
georesource:points_295658932	rdf:type	georesource:points	<file://cyprus-points.rdf>
georesource:points_295658932	rdf:type	geontologv:Feature	<file://cyprus-points.rdf>
georesource:points_295658932	rdf:type	georesource:hospital	<file://cyprus-points.rdf>
georesource:Geom_points_2101780217	rdf:type	sf:Point	<file://cyprus-points.rdf>
georesource:Geom_points_952464662	rdf:type	sf:Point	<file://cyprus-points.rdf>
georesource:points_2541583480	rdf:type	georesource:points	<file://cyprus-points.rdf>
georesource:points_2541583480	rdf:type	geontologv:Feature	<file://cyprus-points.rdf>
georesource:points_2541583480	rdf:type	georesource:bench	<file://cyprus-points.rdf>

5.2 SOS Server Installation

The SOS Server has to be installed twice, on the pilot site and on the interface layer. In this document only the most important steps are described. For more detailed instructions read [how2install_SOS.pdf](#).

1. Make sure that the Java Development Kit 1.7.* is installed on the server.
2. Install PostgreSQL and PostGIS
3. Create the database. Use 'template_postgis' as template for the new database.
Afterwards execute `datamodel_postgres9x_postgis2.sql` to create the SOS table structure.
4. Install a Tomcat. For details about the installation see <http://tomcat.apache.org/tomcat-7.0-doc/setup.html>.
5. Copy `52nSOSv3.5.0_EE.war` into the `/webapps` folder of the tomcat installation.
6. Start the Tomcat Server. `52nSOSv3.5.0_EE.war` will now be unpacked into `<tomcat-root>/webapps/`
7. Adjust the server configuration by editing `<tomcat-root>/webapps/52nSOSv3.5.0_EE/WEB-INF/conf/sos.config`: change `SOS_URL`.

```
156  ### capabilitiesCacheUpdateInterval (0 = no automatic update!)
157  CAPABILITIESCACHEUPDATEINTERVAL=5
158
159  ### URL of this SOS.
160  SOS_URL=http://waterp.env.disy.net:8081/52nSOSv3.5.0_EE
161
162  # support for dynamic locations as spatial values (default=false)
```

Also edit `<tomcat-root>/webapps/52nSOSv3.5.0_EE/testClient-SOSv1.html` and `<tomcat-root>/webapps/52nSOSv3.5.0_EE/testClient-SOSv2.html` and adjust the URL `http://localhost:8080/52nSOSv3.5.0_EE/sos` if necessary.

8. Adjust the database configuration by editing `<tomcat-root>/webapps/52nSOSv3.5.0_EE/WEB-INF/conf/dssos.config`:
change `CONNECTIONSTRING`, `user` and `password`.

```
33  # the connectionstring to the DB (CHANGE!)
34  # example for an postgresql server: jdbc:postgresql://HOST:PORT/DBNAME
35  CONNECTIONSTRING=jdbc:postgresql://localhost:5432/sos-pilot
36
37  # classname of the JDBC Driver (CHANGE!)
38  # example for postgres: org.postgresql.Driver
39  DRIVER=org.postgresql.Driver
40
41  #your DB username (CHANGE!)
42  user=sos
43
44  #your DB password (CHANGE!)
45  password=SOS
46
```

9. Restart the Tomcat server.

To test installation open the SOS application in a browser, a welcome page as shown in Figure 42 should be accessible.

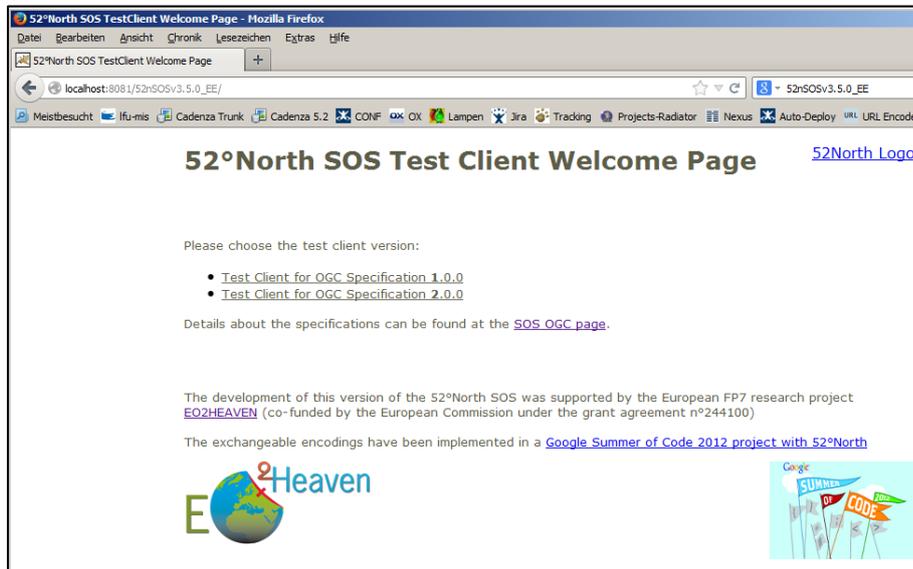


Figure 42: Test SOS server after Installation - step 1

Click onto 'Test Client for OGC Specification 2.0.0', a screen like the one depicted in Figure 43 should load.

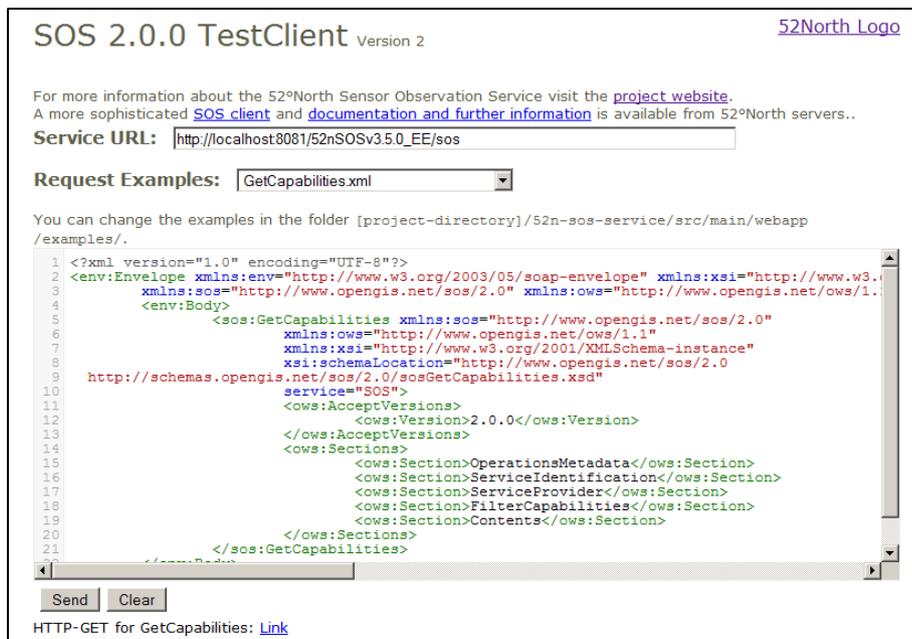


Figure 43: Test SOS server after Installation - step 2

As a request example choose GetCapabilities.xml and press the 'Send' Button. The result should be an XML document showing the capabilities of the SOS Server (see Listing 26).

```
-<env:Envelope xsi:schemaLocation="http://www.w3.org/2003/05/soap-envelope http://www.w3.org/2003/05/soap-envelope http://www.opengis.net/sos/2.0
http://schemas.opengis.net/sos/2.0/sos.xsd">
  <env:Header/>
  <env:Body>
    <sos:Capabilities version="2.0.0">
      <ows:ServiceIdentification>
        <ows:Title>IFGI SOS</ows:Title>
      </ows:ServiceIdentification>
      <ows:Abstract>
        52n SOS at IFGI, Muenster, Germany (SVN: 0 @ 2013-06-24 11:15:07)
      </ows:Abstract>
      <ows:Keywords>
        <ows:Keyword>water level</ows:Keyword>
        <ows:Keyword>gauge height</ows:Keyword>
        <ows:Keyword>waterspeed</ows:Keyword>
      </ows:Keywords>
      <ows:ServiceType codeSpace="http://opengeospatial.net">OGC:SOS</ows:ServiceType>
      <ows:ServiceTypeVersion>2.0.0</ows:ServiceTypeVersion>
      <ows:ServiceTypeVersion>1.0.0</ows:ServiceTypeVersion>
      <ows:Profile>
        http://www.opengis.net/spec/OMXML/2.0/conf/samplingPoint
      </ows:Profile>
      <ows:Profile>http://www.opengis.net/spec/SOS/2.0/conf/soap</ows:Profile>
      <ows:Profile>
        http://www.opengis.net/spec/OMXML/2.0/req/SWEArrayObservation
      </ows:Profile>
      <ows:Fees>NONE</ows:Fees>
    </sos:Capabilities>
  </env:Body>
</env:Envelope>
```

Listing 26: Test SOS server after installation - step 3

5.3 Geoserver Installation

1. Download and run the latest stable Geoserver installer from <http://geoserver.org/display/GEOS/Stable> to start the installation wizard and follow the installation steps.
2. Start Geoserver and open the Geoserver main page in the browser.
3. Login as administrator with the username and password you defined during the installation process.
4. Select 'Stores

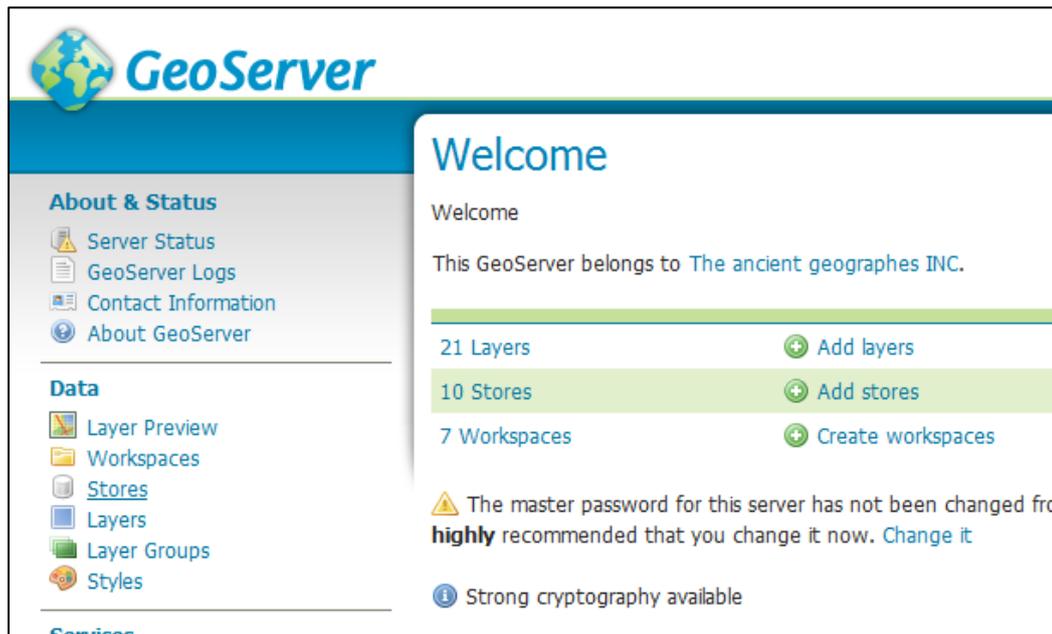


Figure 44: Geoserver stores

5. Select 'Add new Store'

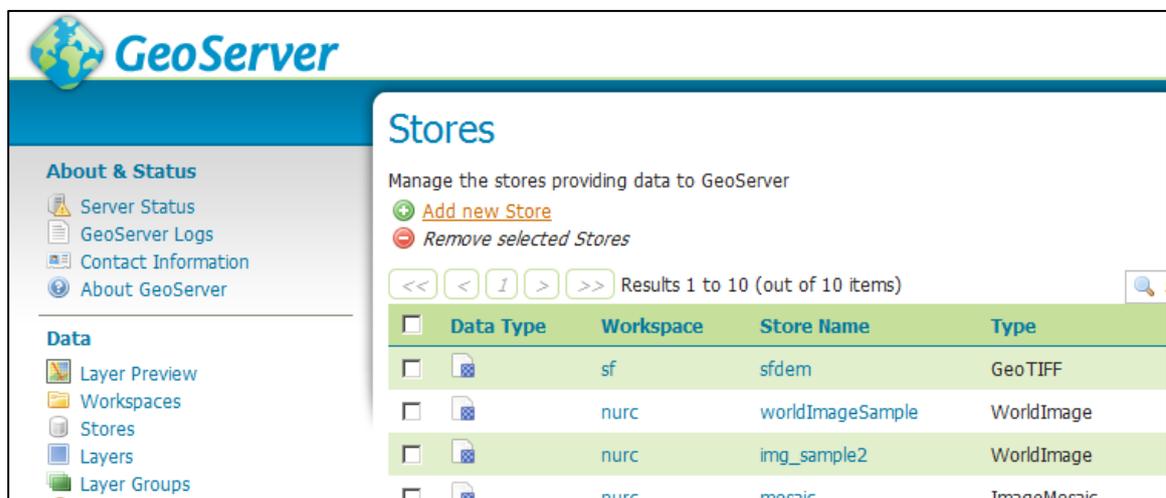


Figure 45: Geoserver stores list

6. Select database PostGIS

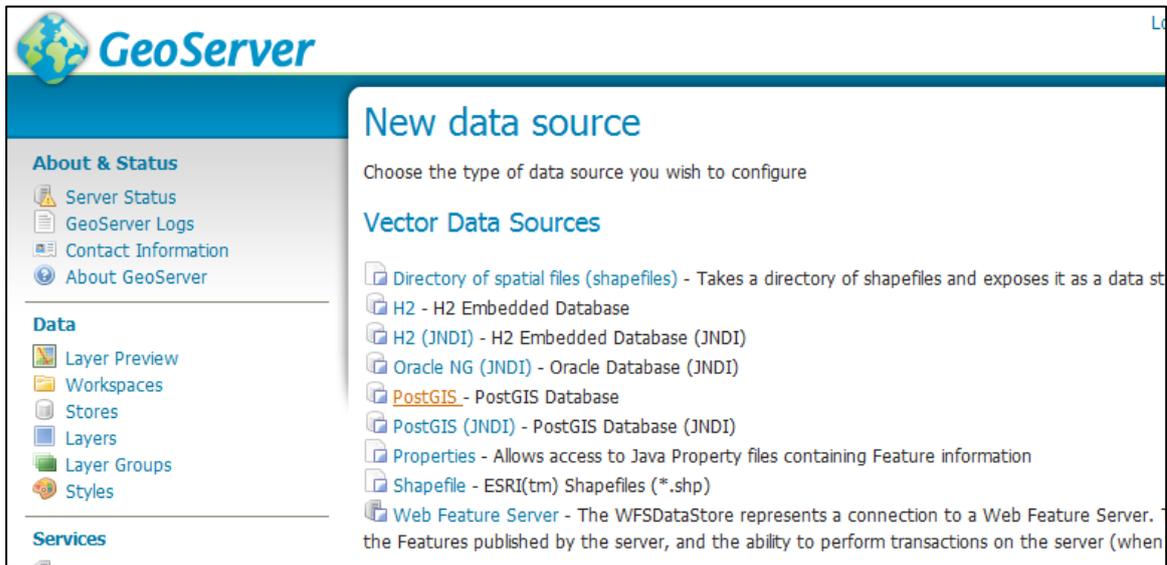


Figure 46: Geoserver new data source wizard

7. Edit database connection to WDW Service scheme:

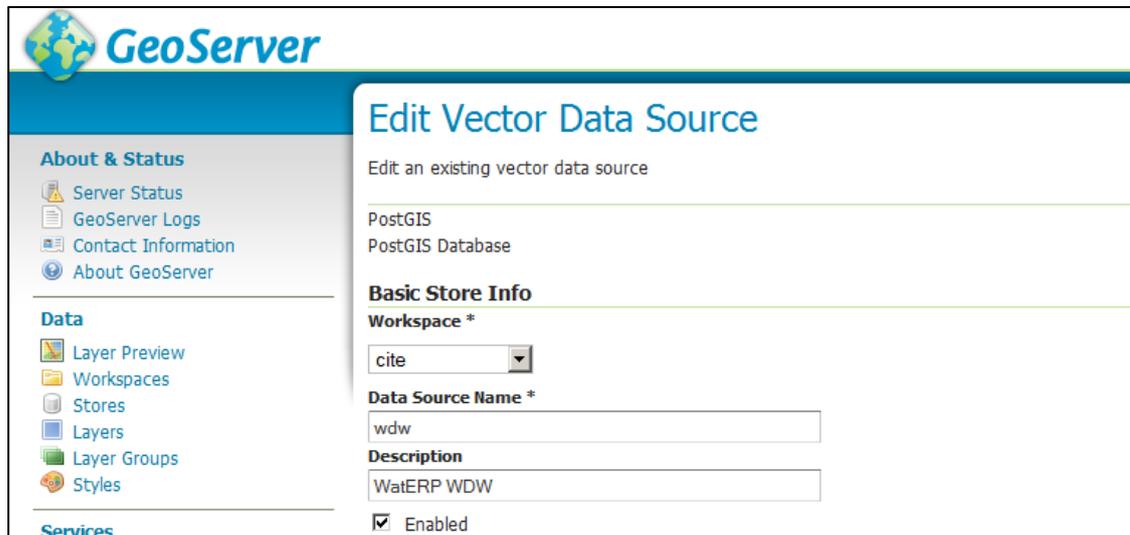


Figure 47: Geoserver database connection configuration

5.4 Pilot Integration Manager Installation

As a prerequisite for the installation of the PIM for ACA, an Apache Tomcat and a SOS server must be installed. Further a PIM database must be created to store the sensor data. There is no need to install the sensor table itself as the ACA-PIM will create it as soon as it's launched. The aca.war must be copied into the webapps folder of the tomcat installation. After the war file has been extracted into an ACA folder the configuration.properties file in the WEB-INF\classes subdirectory must be adjusted.

Parameter	Description
<i>JDBC Connection parameters to access the Pilot Site SOS Server.</i>	
sos.database.driverClassName	Class name of the JDBC driver
sos.database.hibernateDialect	Hibernate dialect for the database implementation.
sos.database.url	Database URL
sos.database.username	Database user
sos.database.password	Database password
<i>JDBC Connection parameters to access the sensor table of the PIM module.</i>	
pim.database.driverClassName	Class name of the JDBC driver
pim.database.hibernateDialect	Hibernate dialect for the database implementation.
pim.database.url	Database URL
pim.database.username	Database user
pim.database.password	Database password
<i>Connection parameters for the running servers.</i>	
sos.url	Connection URL for the Pilot Site SOS Server
wdw.management.url	Connection URL of the WDW Service
<i>WaterOneFlow parameters.</i>	
wof.url	URL of the ACA WaterOneFlow server
wof.PollerInterval	Poller interval in milliseconds
wof.MaxPollerIntervallnDays	Maximum interval size in days that are requested on the WaterOneFlow server.

Table 14: Configuration parameters

Example:

```

configuration.properties
1 sos.database.driverClassName=org.postgresql.Driver
2 sos.database.hibernateDialect=org.hibernate.dialect.PostgreSQLDialect
3 sos.database.url=jdbc:postgresql://localhost/sos
4 sos.database.username=postgres
5 sos.database.password=admin
6
7 pim.database.driverClassName=org.postgresql.Driver
8 pim.database.hibernateDialect=org.hibernate.dialect.PostgreSQLDialect
9 pim.database.url=jdbc:postgresql://localhost/aca
10 pim.database.username=postgres
11 pim.database.password=admin
12
13 sos.url=http://localhost:4711/52nSOSv3.5.0_EE/sos
14
15 wof.url=http://aca-web.gencat.cat/ws/public/WaterOneFlowService_v01_00/services/WaterOneFlowServiceImpl/
16 wof.PollerInterval=60000
17 wof.MaxPollingIntervalInDays=2
18
19 wdw.management.url=http://localhost/wdw
  
```

Listing 27: Sample ACA-PIM configuration

5.5 WDW Service Installation

Here are the steps to install the WDW Service:

1. Make sure that the Java Development Kit 1.7.* is installed on the server.
2. Install PostgreSQL and PostGIS
3. Create the database. Use 'template_postgis' as template for the new database.
Afterwards execute createWdwScheme.sql to create the WDW Service table structure.
4. Install a Tomcat. For details about the installation see <http://tomcat.apache.org/tomcat-7.0-doc/setup.html>. Copy wdw_service.war into the /webapps folder of the tomcat installation.
5. Edit the configuration file <tomcat-root>/webapps/wdw-service/WEB-INF/classes/configuration.properties and restart the Tomcat.

Configuration parameters:

Name	Description
wdw.PollerInterval	Poller interval in milliseconds
JDBC connection parameters for the WDW database scheme.	
wdw.database.driverClassName	JDBC driver name
wdw.database.hibernateDialect	Database dialect
wdw.database.url	Database connection url

wdw.database.username	User name
wdw.database.password	Password
JDBC connection parameters for the SOS database scheme.	
sos.database.driverClassName	JDBC driver name
sos.database.hibernateDialect	Database dialect
sos.database.url	Database connection URL
sos.database.username	User name
sos.database.password	Password
HTTP connection parameters for the Geoserver.	
geoserver. datastore.url	Full qualified REST URL in the format: <base-url>/rest/workspaces/<ws>/datastores/<ds> (ws = workspace, ds = datastore)
geoserver. datastore.user	Geoserver administration username
geoserver. datastore.password	Geoserver administration password

Table 15: WDW Service configuration parameters

Example:

```

configuration.properties
1 wdw.database.driverClassName=org.postgresql.Driver
2 wdw.database.hibernateDialect=org.hibernate.dialect.PostgreSQLDialect
3 wdw.database.url=jdbc:postgresql://localhost/wdw
4 wdw.database.username=postgres
5 wdw.database.password=admin
6
7 sos.database.driverClassName=org.postgresql.Driver
8 sos.database.hibernateDialect=org.hibernate.dialect.PostgreSQLDialect
9 sos.database.url=jdbc:postgresql://localhost/sos
10 sos.database.username=postgres
11 sos.database.password=admin
12 sos.url=http://localhost:4711/52nSOSv3.5.0_EE/sos
13
14 geoserver.datastore.url=http://localhost:8090/geoserver/rest/workspaces/cite/datastores/wdw
15 geoserver.datastore.user=admin
16 geoserver.datastore.password=admin
17
18 wdw.PollerInterval=60000
19 sos.MaxPollingIntervalInDays=30

```

Listing 28: WDW Service configuration example

5.6 WPS4R Installation

WPS4R requires R, RServe and WPS. The complete installation process can be found online at <https://wiki.52north.org/bin/view/Geostatistics/WPS4RDebian>. More information about the configuration of WPS4R can be found at <https://wiki.52north.org/bin/view/Geostatistics/WPS4RDocumentation>.

The Waterp R scripts require the installation of additional R packages. As root user, from within an R console execute the commands:

```
install.packages('jsonlite')
```

```
install.packages("forecast",dependencies=TRUE)
```

In order to upload new annotated R scripts in wps4r :

1. Open Web Admin Console in your browser (e.g. <http://172.20.10.201:8080/wps/webAdmin/index.jsp>)
2. Click "Upload R Script"
3. Choose an annotated R script (Process id will be org.n52.wps.server.r.[id]). The process id is derived from the filename, recent versions derive it from the wps.des annotation inside the script
4. Click "submit", process will be submitted and added to the [LocalRAlgorithmRepository](#)

In order to activate wps4r processes:

1. Open Web Admin Console in Browser
2. Go to "Algorithm Repositories", scroll down name "LocalRAlgorithmRepository"
3. Among the properties are the registered processes. They got the name "Algorithm" and a value which is the process identifier
4. Activate them with the checkbox
5. Click "Save and Activate configuration" to make changes take effect

6. Summary and Conclusions

In this document, the major developments made in the WatERP Workpackage 3 “Water Data Warehouse” have been presented. The WatERP Water Data Warehouse (WDW) copies data from operational systems within the WatERP network of actors in the water-supply chain and offers them in specialized or optimized views, aggregations and formats to the other WatERP modules, for analysis or decision-support purposes. Of course, the WatERP WDW can also act as an intermediary between different WatERP modules, for instance for transferring inputs, outputs or intermediate results between different software tools.

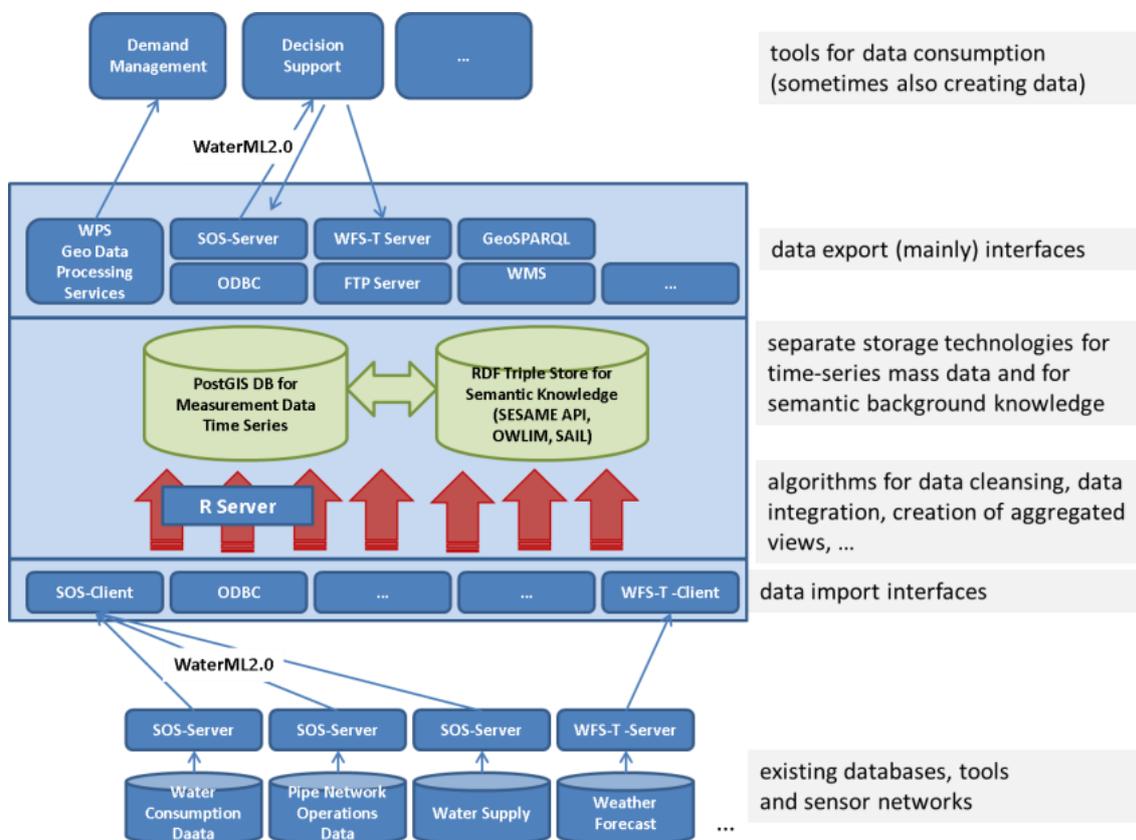


Figure 48: Illustration of WDW architecture

Figure 48 is another illustration highlighting the WDW functions in context and the main WDW design decisions. All WDW developments are based to the most reasonable extent on the open standards of the Open Geospatial Consortium (OGC) as the most important data and software interoperability organization in the geo data area. For most of the data transmissions between different tools, the WDW employs the OGC SOS protocol together with WaterML2 as data format. To implement these interfaces, the 52North SOS Server has been used and further developed. Of course, time-series data is not only imported but also exported through SOS/WaterML2.0. Other interfaces for data access and

querying comprise the typical OGC geodata interfaces (WMS, WFS/WFS-T), direct database access (ODBC) and a semantic query interface through SPARQL/GeoSPARQL.

Typical functionalities of a Data Warehouse for pre-processing incoming data (red arrows in Figure 48) comprise syntactic or semantic data harmonization, assessment of measuring faults, data cleansing (like outlier analysis, null-value replacement, etc.). Further, the Data Warehouse may comprise software routines that support the more efficient consumption and processing of data by specific tools through mechanisms for purpose-specific selection, (spatial, temporal or topical) data projection and views, and purposeful aggregations. In the WDW, the R statistics software workbench is integrated for realizing such pre-processing algorithm.

An important design decision was to devise two separate storage areas within WDW (green database elements in Figure 48):

- Left-hand side: Mass data (time series describing sensor measurements about hydrology, network operations, meteorology, etc.) is highly repetitive and simply structured, but comes in large volumes. Huge amounts of this kind of data can easily and efficiently be managed with conventional object-relational database technologies like a PostGIS database. These data populate a data schema which is based on the structure of the OGC / ISO conceptual model „Observations and Measurements“and on OGC WaterML2.0, respectively.
- Right-hand side: In order to describe all decision-relevant aspects of a water-supply chain comprehensively, one also needs to describe aspects that fit not well in an (object-)relational database schema. For instance, irregularly structured knowledge, complex relationships between objects, definitions of technical terms or interrelationships of them, or generic relationships which abstract away from specific facts (like rule-based knowledge or arithmetic relationships). For expressing such aspects, computer science has developed Semantic Web technologies. In the Semantic Web area, RDF triple stores have been introduced for storing and processing complex knowledge. More recently, triple stores have been extended towards geospatial reasoning which allows drawing logical deductions that include also simple notions of spatial relationships and spatial deductions. So, the second main storage area of the WDW is an RDF triple store enabled to do geospatial reasoning.

The incorporation of a semantic knowledge base follows the current trend to empower modern software solutions by knowledge-based components, to increase interoperability through ontologies and to provide data according to the Linked Open Data paradigm. However, a complete “semantification” (representation, storage and processing of all the collected data and information with Semantic Web methods and tools) of all data in the WDW seemed not feasible and promising to us, especially regarding the measurement-data time series. Instead, the time-series data and the semantic knowledge complement each other and can together be exploited for powerful analyses and queries about the considered water supply system. For instance, the structured relationships in the triple store can be used to logically describe a water-distribution network and its geospatial aspects, as well as some

background knowledge, for instance, about measurement methods for assessing water quality. Then, specific water-quality measurements can continuously be fed as time series into the PostGIS database. The metadata for measurements have references into the semantic knowledge base. This allows for queries which combine time-series sensor data and semantic background knowledge.

For instance, one could ask for all measurements

- ... made in a certain geographic area
- ... and using an analytical sensor technology with certain characteristics;

or, about all sensor data

- ... from a certain point on in the water supply chain
- ... which make statements about a certain group of related chemical or biological pollutants.

Analysing the answers could help to find upstream dischargers responsible for a contamination and take appropriate counter-measures, or it could help to identify endangered spots further downstream and take suitable protection or purification measures.

Now, coming close to the end of the WatERP project duration, a few reflections and conclusions can be drawn:

- **Technological contribution.** We see a main scientific-technological contribution of WP3 in the engineering and the proof-of-concept implementation of the WDW architecture with its different kinds of data and knowledge and different kinds of possible reasoning. Piloting experience confirms the design decisions and underlines the stability and usefulness of the selected and integrated software elements (especially the 52°North SOS server, SOS protocol, WaterML2.0 data schema, R server, OWLIM, uSeekM). Also the way of integration (separate storage areas for measurement data and for semantic knowledge, defining R processing results as derived sensors, etc.) seems to be a reasonable balance between pragmatic decisions and innovative ambitions. The performance testing of time-series data management and of geospatial reasoning shows that the system is fast enough to meet the requirements of the pilot usage scenarios within the project.
- **Status and expected real-world impact.** At the time of preparing this deliverable, the first fully working prototypes of the WDW are up and running and have been filled with real-world test data of the two WatERP pilot users, the Catalan Water Agency (Agencia Catalana de l'Aigua / ACA, Barcelona, Spain) and the water utility of the city of Karlsruhe (Stadtwerke Karlsruhe GmbH, Germany). The WDW implementation is stable and efficient and – through the Pilot Integration Manager – can be easily linked to existing data infrastructures. One important remark must be made, however: In WatERP, we deliver and test only a *technical* solution. The concrete impact of applying such solutions highly depends on the exact usage situation and the exact way of usage, in particular the *non-technical* factors. In general, the success in many

specific use cases will, for instance, highly depend on factors like the regulatory framework, the incentive systems for data exchange, etc. It might also happen that for an operational roll-out of a WatERP-like solution, a very strong privacy and data-protection layer would have to be integrated because stakeholders might not want to send their data to a central system if it was not ensured that unauthorized access is impossible. Further, it turned out that – from the perspective of geodata and hydrological data exchange – settling upon OGC standards completely made sense. While this may foster the take-up of project results among public authorities, we also have to state that in the area of water utilities, OGC standards are still very seldom used or even known. In order to achieve more impact in this area, one would also have to check other data formats like industry standards from machine-control software providers, sensor-data exchange formats of commercial sensor suppliers, or even data-exchange formats for economic data. Also compliance with INSPIRE data models might be interesting in the mid-term future. Regarding, exploitation of project results, one can already see that bits and pieces of the WDW solution will be applicable within the DISY product landscape (e.g., the SOS infrastructure or the R coupling).

- **Next steps, known deficiencies, and future work.**

- The main emphasis of the last project semester Summer 2015 will be the piloting and performance testing in the two test beds ACA and SWKA within Workpackage WP7. Technically, we do not expect much feedback or change requests from that, but some code consolidation or bug fixing might be necessary.
- One aspect which could be reworked in the case of building a stand-alone tool from the WDW which should also be usable independently from the overall WatERP infrastructure, is the following: The current implementation allows querying the time-series data and the semantic knowledge independently; if one would need a *joint* query that requires to consider numeric (on the observation data) constraints, spatial restrictions and ontological reasoning together⁵², this would have to be realized by two independent queries⁵³ that would have to be combined outside the WDW by a WatERP agent. This comes from the overall architectural design decisions of the WDW and of WatERP. Of course, if needed, this kind of queries considering both the observation data themselves and the observation metadata, could also be integrated into the WDW implementation.

⁵² Like: “Find all observation data (1) above a certain threshold which (2) stem from sensors in a certain geographic region and (3) are measured with a certain sensor technology”.

⁵³ One through the SOS interface to the PostGIS database regarding the query aspect (1) and another one through the GeoSPARQL interface to the triple store for query aspects (2) and (3).

- In general, we expect that the technical solution of WatERP for multi-stakeholder data exchange can be applied far beyond the scope of water resource management, e.g., in the broader scope of *Smart Cities*. A technical direction of potential future work on WDW could be scaling-up towards a more real-time and more fine-grained data-collection mode – which would certainly create new challenges regarding performance and software architecture. Here, recent methods from *Big Data* processing could be employed. Such SOS approaches for *streaming data*, maybe combined with spatial complex event processing, could be a powerful extension for a data infrastructure like the one presented.

7.2 Appendix II: WaterOneFlow Service



Figure 50: WaterOneFlow Service diagram

Figure 50 lists all methods implemented by the WaterOneFlow web service.

7.3 Appendix III: Basic Data-Mining Scripts With WPS4R Annotations

This Appendix includes the scripts implementing the basic data-mining functions along with the WPS4R annotations, i.e. the lines starting with the string “# wps”, which are needed by wps4r to convert them into WPS processes. The lines between the annotations “# wps.off “ and “# wps.on” contain test/example code that is executed only in case the script is run within R but not when the scripts are run as WPS processes from WPS4R.

7.3.1 Script `bdm_auto_cf.v1.R`

```
# wps.des: bdm_auto_cf.v1.R.v1.R, Timeseries Auto Correlation process. Computes estimates of the
autocovariance or autocorrelation function;
##### dependencies #####
library(jsonlite);
##### helper functions #####
myLog <- function(...) {
  cat(paste0("[bdm_auto_cf.v1.R] ", ..., "\n"))
}
##### manual testing #####
# wps.off;
tsdata_json <- "[25.31,25.8,24.73,25.4,25.54,26.03,25.94,24.27,25.88,30.02,29.56,29.86,30.12,31.34,28.88,31.69,30.13,30.18,31.98,24.13,24.88,23.02,25.6,23.65,24.62,23.79,23.15,24.95,23.92,24.61,26.03,25.26.1,26.36,27.62,27.16,26.81,26.16,26.49,27.38,26.62,27.66,25.96,28.07,28.15,26.23,25.79,24.68,27.29,27.16]"
tsstart <- 1
tsend <- 50
tsfrequency <- 1
width <- 6
acftype <- "correlation"
# wps.on;
##### input definition #####
# wps.in: tsdata_json, type = string, title = a json list of time-series data,
# minOccurs = 1, maxOccurs = 1;
# wps.in: tsstart, type = integer, title = time series start,
# minOccurs = 1, maxOccurs = 1;
# wps.in: tsend, type = integer, title = time series end,
# minOccurs = 1, maxOccurs = 1;
# wps.in: tsfrequency, type = integer, title = time series frequency,
# minOccurs = 1, maxOccurs = 1;
# wps.in: acftype, type = string, title = "correlation" or "covariance",
# minOccurs = 1, maxOccurs = 1;

myLog(tsdata_json)

tsdata <- fromJSON(tsdata_json)
tsdata.ts = ts(tsdata,start=tsstart,end=tsend,frequency=tsfrequency)
out <- acf(tsdata.ts,type=acftype,plot=FALSE)
output <- toJSON(cbind(out$lag,out$acf))
# wps.out: output, string ;
# wps.off;
out
# wps.on;
```

7.3.2 Script bdm_categorical.v1.R

```
# wps.des: bdm_categorical.v1.R, Convert numerical to factor process;
##### dependencies #####
library(jsonlite);
##### helper functions #####
myLog <- function(...) {
  cat(paste0("[bdm_categorical.v1.R] ", ..., "\n"))
}
##### manual testing #####
# wps.off;
data_json <- "[-5,5,15,20,21,-10,1,11,31,23,32,42,60,4,123]"
breaks_json <- "[-20,10.5,30.5,200]"
labels_json <- "[\"low\", \"medium\", \"high\"]"
# wps.on;

##### input definition #####
# wps.in: data_json, type = string, title = a json object of vector data,
# minOccurs = 1, maxOccurs = 1;

# wps.in: breaks_json, type = string, title = a json object. either a numeric vector of two or more unique
cut points or a single number (greater than or equal to 2) giving the number of intervals into which data
is to be cut.,
# minOccurs = 1, maxOccurs = 1;

# wps.in: labels_json, type = string, title = a json object containing the labels for the levels of the
resulting category,
# minOccurs = 1, maxOccurs = 1;

data <- fromJSON(data_json)
breaks <- fromJSON(breaks_json)
labels <- fromJSON(labels_json)

out <- cut(data,breaks=breaks,labels=labels)

output <- toJSON(out)
# wps.out: output, string ;

# wps.off;
data_json
breaks_json
labels_json
output
# wps.on;
```

7.3.3 Script bdm_cross_cf.v1.R

```
# wps.des: bdm_cross_cf.v1.R, Timeseries Cross-correlation process. Computes the cross-correlation
or cross-covariance of two univariate series.
##### dependencies #####
library(jsonlite);

##### helper functions #####
myLog <- function(...) {
  cat(paste0("[bdm_cross_cf.v1.R] ", ..., "\n"))
}
```

```
}

##### manual testing #####
# wps.off;
ts1data_json <-
"[25.31,25.8,24.73,25.4,25.54,26.03,25.94,24.27,25.88,30.02,29.56,29.86,30.12,31.34,28.88,31.69,30.
13,30.18,31.98,24.13,24.88,23.02,25.6,23.65,24.62,23.79,23.15,24.95,23.92,24.61,26.03,25.26.1,26.36
,27.62,27.16,26.81,26.16,26.49,27.38,26.62,27.66,25.96,28.07,28.15,26.23,25.79,24.68,27.29,27.16]"
ts1start <- 1
ts1end <- 50
ts1frequency <- 1
ts2data_json <-
"[24.31,24.8,23.72,25.4,24.54,23.03,24.94,23.25,24.88,31.02,29.57,29.87,30.12,31.34,28.81,31.62,30.
14,30.18,21.98,14.13,14.88,13.02,25.6,33.65,34.62,33.79,43.15,44.95,23.92,25.62,25.03,24.26.1,25.36
,27.62,27.16,26.81,26.16,26.49,27.38,26.62,24.66,23.96,28.07,27.15,26.23,25.79,24.68,27.29,27.16]"
ts2start <- 1
ts2end <- 50
ts2frequency <- 1
ccftype <- "covariance"
# wps.on;

##### input definition #####
# wps.in: ts1data_json, type = string, title = a json list of time-series data,
# minOccurs = 1, maxOccurs = 1;

# wps.in: ts1start, type = integer, title = time series start,
# minOccurs = 1, maxOccurs = 1;

# wps.in: ts1end, type = integer, title = time series end,
# minOccurs = 1, maxOccurs = 1;

# wps.in: ts1frequency, type = integer, title = time series frequency,
# minOccurs = 1, maxOccurs = 1;

# wps.in: ts2data_json, type = string, title = a json list of time-series data,
# minOccurs = 1, maxOccurs = 1;

# wps.in: ts2start, type = integer, title = time series start,
# minOccurs = 1, maxOccurs = 1;

# wps.in: ts2end, type = integer, title = time series end,
# minOccurs = 1, maxOccurs = 1;

# wps.in: ts2frequency, type = integer, title = time series frequency,
# minOccurs = 1, maxOccurs = 1;

# wps.in: ccftype, type = string, title = "correlation" or "covariance",
# minOccurs = 1, maxOccurs = 1;

myLog(ts1data_json)
myLog(ts2data_json)
ts1data <- fromJSON(ts1data_json)
ts1data.ts = ts(ts1data,start=ts1start,end=ts1end,frequency=ts1frequency)

ts2data <- fromJSON(ts2data_json)
ts2data.ts = ts(ts2data,start=ts2start,end=ts2end,frequency=ts2frequency)
```

```
out <- ccf(ts1data.ts,ts2data.ts,type=ccftype,plot=FALSE)
```

```
output <- toJSON(cbind(out$lag,out$acf))
```

```
# wps.out: output, string ;
```

```
# wps.off;
```

```
output
```

```
# wps.on;
```

7.3.4 Script `bdm_mv_filter.v1.R`

```
# wps.des: bdm_mv_filter.v1.R, Filter missing values process;
```

```
##### dependencies #####
```

```
library(zoo);
```

```
library(jsonlite);
```

```
##### helper functions #####
```

```
myLog <- function(...) {
```

```
  cat(paste0("[bdm_mv_interpolate.v1.R] ", ..., "\n"))
```

```
}
```

```
##### manual testing #####
```

```
# wps.off;
```

```
tsdata_json <- "[{"Date":"2009-12-31","A":4.9,"B":18.4,"C":32.6,"D":77},{"Date":"2010-01-29","A":5.1,"B":17.7},{"Date":"2010-01-31","A":5,"C":32.8,"D":78.7},{"Date":"2010-02-26","A":4.8},{"Date":"2010-02-28","A":4.7,"B":18.3,"C":33.7,"D":79},{"Date":"2010-03-31","A":5.3,"B":19.4,"C":32.4,"D":77.8},{"Date":"2010-04-30","A":5.2,"B":19.7,"C":33.6,"D":79},{"Date":"2010-05-28","A":5.4,"D":81.7},{"Date":"2010-05-30","C":34.5},{"Date":"2010-05-31","A":4.6,"B":18.1}]"
```

```
# wps.on;
```

```
##### input definition #####
```

```
# wps.in: tsdata_json, type = string, title = a json object of time-series data,
```

```
# minOccurs = 1, maxOccurs = 1;
```

```
myLog(tsdata_json)
```

```
tsdata <- fromJSON(tsdata_json)
```

```
out.z <- na.omit(read.zoo(tsdata))
```

```
out.df <- as.data.frame(out.z)
```

```
output <- toJSON(out.df)
```

```
# wps.out: output, string ;
```

```
Script
```

```
# wps.des: bdm_mv_interpolate.v1.R, Interpolate missing values process;
```

```
##### dependencies #####
```

```
library(zoo);
```

```
library(jsonlite);
```

```
##### helper functions #####
```

```
myLog <- function(...) {
```

```

cat(paste0("[bdm_mv_interpolate.v1.R] ", ..., "\n"))
}

##### manual testing #####
# wps.off;
tsdata_json <- "[{"Date":"2009-12-31","A":4.9,"B":18.4,"C":32.6,"D":77},{"Date":"2010-01-29","A":5.1,"B":17.7},{"Date":"2010-01-31","A":5,"C":32.8,"D":78.7},{"Date":"2010-02-26","A":4.8},{"Date":"2010-02-28","A":4.7,"B":18.3,"C":33.7,"D":79},{"Date":"2010-03-31","A":5.3,"B":19.4,"C":32.4,"D":77.8},{"Date":"2010-04-30","A":5.2,"B":19.7,"C":33.6,"D":79},{"Date":"2010-05-28","A":5.4,"D":81.7},{"Date":"2010-05-30","C":34.5},{"Date":"2010-05-31","A":4.6,"B":18.1}]"
# wps.on;

##### input definition #####
# wps.in: tsdata_json, type = string, title = a json object of time-series data,
# minOccurs = 1, maxOccurs = 1;

myLog(tsdata_json)
tsdata <- fromJSON(tsdata_json)

out.z <- na.approx(read.zoo(tsdata))
out.df <- as.data.frame(out.z)
output <- toJSON(out.df)

# wps.out: output, string ;

```

7.3.5 Script bdm_outliers.v1.R

```

# wps.des: bdm_outliers, Outlier detection process;

##### dependencies #####
library(forecast);
library(jsonlite);

##### helper functions #####
myLog <- function(...) {
  cat(paste0("[bdm_outliers] ", ..., "\n"))
}

##### manual testing #####
# wps.off;
tsdata_json <- "[25.96,25.63,24.78,26.24,26.42,26.84,26.99,24.75,26.76,37.87,34.76,32.23,25.48,26.55,24.18,26.53,27.05,24.19,26.82,25.94,26.6,26.05,26.85,26.89,27.57,27.76,23.72,26.56,28.14,28.52,26.66,26.72,25.61,26.41,26.45,27.16,27.28,28.04,27.62,27.7,27.01,28.19,27.77,28.47,29.76,28.4,26.38,29.96,27.75,29.46]"
tsstart <- 1
tsend <- 50
tsfrequency <- 1
# wps.on;

myLog(tsdata_json)

##### input definition #####
# wps.in: tsdata_json, type = string, title = a json list of time-series data,

```

```
# minOccurs = 1, maxOccurs = 1;

# wps.in: tsstart, type = integer, title = time series start,
# minOccurs = 1, maxOccurs = 1;

# wps.in: tsend, type = integer, title = time series end,
# minOccurs = 1, maxOccurs = 1;

# wps.in: tsfrequency, type = integer, title = time series frequency,
# minOccurs = 1, maxOccurs = 1;

tsdata <- fromJSON(tsdata_json)

tsdata.ts = ts(tsdata,start=tsstart,end=tsend,frequency=tsfrequency)
res <- tsoutliers(tsdata.ts);

output <- toJSON(res)
# wps.out: output, string ;
#example output in json format : {index[10,11],replacements[28.58,30.41]}
myLog(res)
myLog(output)
```

7.3.6 Script `bdm_smooth.v1.R`

```
# wps.des: bdm_smooth.v1.R, Timeseries Smoothing process;

##### dependencies #####
library(robfilter);
library(jsonlite);

##### helper functions #####
myLog <- function(...) {
  cat(paste0("[bdm_smooth.v1.R] ", ..., "\n"))
}

##### manual testing #####
# wps.off;
tsdata_json <- "[25.31,25.8,24.73,25.4,25.54,26.03,25.94,24.27,25.88,30.02,29.56,29.86,30.12,31.34,28.88,31.69,30.13,30.18,31.98,24.13,24.88,23.02,25.6,23.65,24.62,23.79,23.15,24.95,23.92,24.61,26.03,25,26.1,26.36,27.62,27.16,26.81,26.16,26.49,27.38,26.62,27.66,25.96,28.07,28.15,26.23,25.79,24.68,27.29,27.16]"
tsstart <- 1
tsend <- 50
tsfrequency <- 1
width <- 6
method <- "RM,LMS,LTS,DR,LQD"
# wps.on;

##### input definition #####
# wps.in: tsdata_json, type = string, title = a json list of time-series data,
# minOccurs = 1, maxOccurs = 1;

# wps.in: tsstart, type = integer, title = time series start,
# minOccurs = 1, maxOccurs = 1;
```

```
# wps.in: tsend, type = integer, title = time series end,
# minOccurs = 1, maxOccurs = 1;

# wps.in: tsfrequency, type = integer, title = time series frequency,
# minOccurs = 1, maxOccurs = 1;

# wps.in: width, type = integer, title = a positive integer defining the window width used for fitting,
# minOccurs = 1, maxOccurs = 1;

# wps.in: method, type = string, title = a comma-separated list of methods. Any combination of the
values: "DR" Deepest Regression, "LMS" Least Median of Squares regression, "LQD" Least Quartile
Difference regression, "LTS" Least Trimmed Squares regression, "MED" Median, "RM" Repeated
Median regression,
# minOccurs = 1, maxOccurs = 1;

method_v <- strsplit(method, ",")
myLog(tsdata_json)

tsdata <- fromJSON(tsdata_json)
tsdata.ts = ts(tsdata, start=tsstart, end=tsend, frequency=tsfrequency)
tsdata.rr <- robreg.filter(tsdata.ts, width=width, method=method_v[[1]])

# wps.off;
# plot(tsdata.rr, main="Smoothing output vs input", xlab="Timestamp", ylab="Value")
# wps.on;

output <- toJSON(c(tsdata.rr$level, tsdata.rr$slope))
# wps.out: output, string ;

# wps.off;
output
# wps.on;
```

7.3.7 Script `bdm_spatial_interpolate.v1.R`

```
# wps.des: bdm_spatial_interpolate.v1.R, spatial interpolation (kriging) process;

##### dependencies #####
library(geoR)
library(jsonlite)

##### helper functions #####
myLog <- function(...) {
  cat(paste0("[bdm_categorical.v1.R] ", ..., "\n"))
}

##### manual testing #####
# wps.off;
locs_json <- "[[15,28],[36,28],[21,31],[39,40],[23,33],[40,41],[24,35],[29,28]]"
obs_json <- "[8.9,27.7,39.6,24.6,10,6,39.1,18.3]"

#Construct a prediction grid
x <- seq(13, 42, by=2)
y <- seq(20, 49, by=2)
```

```
xv <- rep(x,15)
yv <- rep(y,each=15)
test_grid_locs <- as.matrix(cbind(xv,yv))
grid_locs_json <- toJSON(test_grid_locs)
# wps.on;

##### input definition #####
# wps.in: locs_json, type = string, title = a json array of observation locations,
# minOccurs = 1, maxOccurs = 1;

# wps.in: obs_json, type = string, title = a json array of observation values,
# minOccurs = 1, maxOccurs = 1;

# wps.in: grid_locs_json, type = string, title = a json array of prediction locations,
# minOccurs = 1, maxOccurs = 1;

locs <- fromJSON(locs_json)
obs <- fromJSON(obs_json)
grid_locs <- fromJSON(grid_locs_json)

grid_predict <- ksline(
  coords=locs,data=obs,
  cov.model="exp",
  cov.pars=c(10,3.33),
  nugget=0,
  locations=grid_locs)

# wps.out: output, string ;
output <- toJSON(grid_predict$predict)

# wps.off;
output
#grid_predict
#grid_predict$summary
#grid_predict$predict
#grid_predict$kridge_var
#grid_predict$dif
#grid_predict$beta
#grid_predict$message
# wps.on;
```

8. References

- Esling, P., & Agon, C. (2012). Time-series data mining. *ACM Computing Surveys (CSUR)*, 45(1), 12.
- Fu, T.C. (2011). A review on time series data mining. *Engineering Applications of Artificial Intelligence*, 24(1), 164-181.
- Garbis, G., Kyzirakos, K., & Koubarakis, M. (2013). Geographica: A Benchmark for Geospatial RDF Stores. arXiv preprint arXiv:1305.5653.
- Geo.LinkedData.es Team, (2014), geometry2rdf Utility. Available at <http://mayor2.dia.fi.upm.es/oeg-upm/index.php/en/technologies/151-geometry2rdf>
- Kiryakov, A., Bishop, B., Ognyanoff, D., Peikov, I., Tashev, Z., & Velkov, R. (2010). The features of BigOWLIM that enabled the BBC's World Cup website. In *Workshop on Semantic Data Management SemData@ VLDB*.
- Kiryakov, A., Ognyanov, D., & Manov, D. (2005). OWLIM—a pragmatic semantic repository for OWL. In *Web Information Systems Engineering—WISE 2005 Workshops* (pp. 182-192). Springer Berlin Heidelberg.
- McLeod, A. I., Yu, H., & Mahdi, E. (2011). Time Series Analysis with R. *Handbook of Statistics*, 30.
- Ontotext (2014), OWLIM References, available online at <http://www.ontotext.com/owlim/references>
- Patrourmpas K., Alexakis M., Giannopoulos G., Athanasiou S., (2014), TripleGeo: an ETL Tool for Transforming Geospatial Data into RDF Triples. In *Proceedings of the 4th International Workshop on Linked Web Data Management (LWDM)*, Athens, Greece, March 2014.
- Perry M. and Herring J., (2012), OGC GeoSPARQL- A Geographic Query Language for RDF Data. In *OGC Implementation Standard*, ref: OGC 11-052r4
- R Development Core Team (2012). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.
- Ratanamahatana, C. A., Lin, J., Gunopulos, D., Keogh, E., Vlachos, M., & Das, G. (2010). Mining time series data. In *Data Mining and Knowledge Discovery Handbook* (pp. 1049-1077). Springer US.
- Razina E. and Janzen D., (2007), *Effects of Dependency Injection on Maintainability*, ISBN Hardcopy: 978-0-88986-705-5 / CD: 978-8-88986-706-2
- Reeves, G., Liu, J., Nath, S., & Zhao, F. (2009). Managing massive time series streams with multi-scale compressed trickles. *Proceedings of the VLDB Endowment*, 2(1), 97-108.

Semantrix (2014), OWLIM High-Performance Semantic Database, available online at <http://semantrix.com.au/pages/partners/ontotext/owlim-semantic-database>

Stiefmeier, T., Roggen, D., & Tröster, G. (2007, June). Gestures are strings: efficient online gesture spotting and classification using string matching. In Proceedings of the ICST 2nd international conference on Body area networks (p. 16). ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Thakker, D., Osman, T., Gohil, S., & Lakin, P. (2010). A pragmatic approach to semantic repositories benchmarking. In The Semantic Web: Research and Applications (pp. 379-393). Springer Berlin Heidelberg.

Urbanek, S. (2003). A fast way to provide R functionality to applications. In Proceedings of DSC (p. 2).

Voigt, M., Mitschick, A., & Schulz, J. (2012). Yet Another Triple Store Benchmark? Practical Experiences with Real-World Data. In SDA (pp. 85-94).

Yan, X., C. Zhang, and S. Zhang. 2003. Towards databases mining: Pre-processing collected data. Applied Artificial Intelligence 17(5-6):545-561.

Zhang, S., Zhang, C., & Yang, Q. (2003). Data preparation for data mining. Applied Artificial Intelligence, 17(5-6), 375-381.

Zhao, Y. (2012). R and Data Mining: Examples and Case Studies. Academic Press.