ALMANAC

RELIABLE SMART SECURE
INTERNET OF THINGS FOR SMART CITIES

(FP7 609081)

D6.1 A scalable data management architecture for Smart City environments

Date 2014-09-15 – Version 1.0

Published by the Almanac Consortium

Dissemination Level: Public

Project co-funded by the European Commission within the 7th Framework Programme
Objective ICT-2013.1.4: A reliable, smart and secure Internet of Things for Smart Cities

Document control page

Document file: D6.1 A Scalable Data Management Architecture for Smart City Environments.docx
Document version: 1.0
Document owner: Matts Ahlsén (CNet)

Work package: WP6 – Scalable Data Management
Task: T6.1 – Data Context Management Framework Architecture
T6.2 – Framework for Information Aggregation, Storage and Mining
T6.3 – Event Detection and Management

Deliverable type: R

Document status: ☒ approved by the document owner for internal review
☒ approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Matts Ahlsen (CNET)	2014-05-28	Initial structure and ToC
0.2	Matts Ahlsen (CNET)	2014-06-24	Revised structure for plenary meeting
0.3	Matts Ahlsen, Mathias Axling (CNET)	2014-08-05	Revised assignments, Data model section update
0.35	Matts Ahlsen, Mathias Axling, Peter Rosengren (CNET), Jaroslav Pullman, Angel Carvajal Soto (FIT)	2014-08-13	Structure update after telco
0.5	Matts Ahlsen, Mathias Axling, Peter Rosengren (CNET), Angel Carvajal Soto (FIT)	2014-08-25	Update data fusion and data model and data storage section
0.7	Matts Ahlsen, Mathias Axling, Peter Rosengren (CNET)	2014-09-04	Resource and services model, Query and data model update
0.9	Matts Ahlsen, Mathias Axling, Peter Rosengren (CNET)	2014-09-09	Version for internal review
1.0	Matts Ahlsen, Mathias Axling, Peter Rosengren (CNET), Angel Carvajal Soto (FIT)	2014-09-15	Final version for submission to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Dario Bonino	2014-09-09	Approved with major comments
Alexandre Alapetite	2014-09-09	Approved with minor comments

Index:

List of Figures

1. Executive summary

The Data Management Framework (DMF) in the ALMANAC architecture is characterized by the following features,

- IoT-oriented data models for physical world, entity and resource management
- Distributed and scalable storage system based on big data principles in cloud environments
- REST-based interfaces for data retrieval and actuation
- Data Fusion Engine and language integrated with support for Complex Event Processing (CEP)

The purpose of the DMF is to provide flexible and robust data and event management to Smart City applications that need to access a variety of (IoT) resources, devices and services possibly in distributed cloud environments. The initial test beds are being defined for the waste and water management domains in the project.

The design of the Data Management Framework is done within the context of the overall ALMANAC platform architecture design, reported in its initial version in deliverable D3.1.1. The adjacent architecture tiers, i.e., the Virtualization Layer (VL) and the Smart City Resource Adaptation Layer (SCRAL), interact with the DMF through a generic ALMANAC REST interface and cloud services.

This document describes the first version of the DMF and discusses the main design assumptions and technical solutions adopted so far.

A first set of prototype implementations of the main DMF components are introduced in this deliverable: the IoTWorld Gateway, IoT Resource Catalogue, Storage Manager, the Data Fusion Engine and the Event Manager.

2. Introduction

2.1 Purpose, context and scope of this deliverable

The purpose of this deliverable is to provide and updated view on the current design of the Data Management Framework in ALMANAC. This is done in the context of the WP6 ("Scalable Data Management") and contributes to the development of the overall ALMANAC platform architecture (carried out in WP3.).

The objective of WP6 is to provide an integrated approach to data and event management in which message-based information exchange is combined with highly efficient event processing and configurable, and scalable, data management. The main outcome of the work package will be a Smart City Data Management Framework integrated into the overall ALMANAC Smart City Platform, and built upon the LinkSmart middleware.

The DMF envisioned role, in the overall architecture, is to provide flexible and robust data and event management to Smart City applications that need to access a variety of (IoT) devices and services possibly in cloud environments. Initial test beds, in this project, will be focused on the waste and water management domains.

The Data Management Framework in the ALMANAC architecture is characterized by the following features,

- IoT-oriented data models for physical world, entity and resource management
- Distributed and scalable storage system based on big data principles in cloud environments
- REST-based interfaces for data retrieval and actuation
- Data Fusion Engine and language integrated with support for Complex Event Processing (CEP)

This document describes, in particular, the first design cycle of core functionalities of the DMF, encompassing:

- the IoTWorld Gateway design and implementation,
- the Storage Manager design and implementation
- the Query interfaces and data fusion language design,
- the Event and Data model, and,
- the REST interfaces for event and data management.

This document is an extension of the documentation accompanying the internal deliverable ID6.2, and describing the initial prototypes for data fusion, storage management and event processing.

2.2 ALMANAC Architecture

The ALMANAC system is a platform for Smart City applications exploiting, and advancing, IoT, M2M and Cloud technologies.

A first version of the overall architecture is documented in the deliverable D3.1.1, and a functional view of this architecture is reported in Figure 1 for the sake of clarity, high-lighting the main components and interfaces.

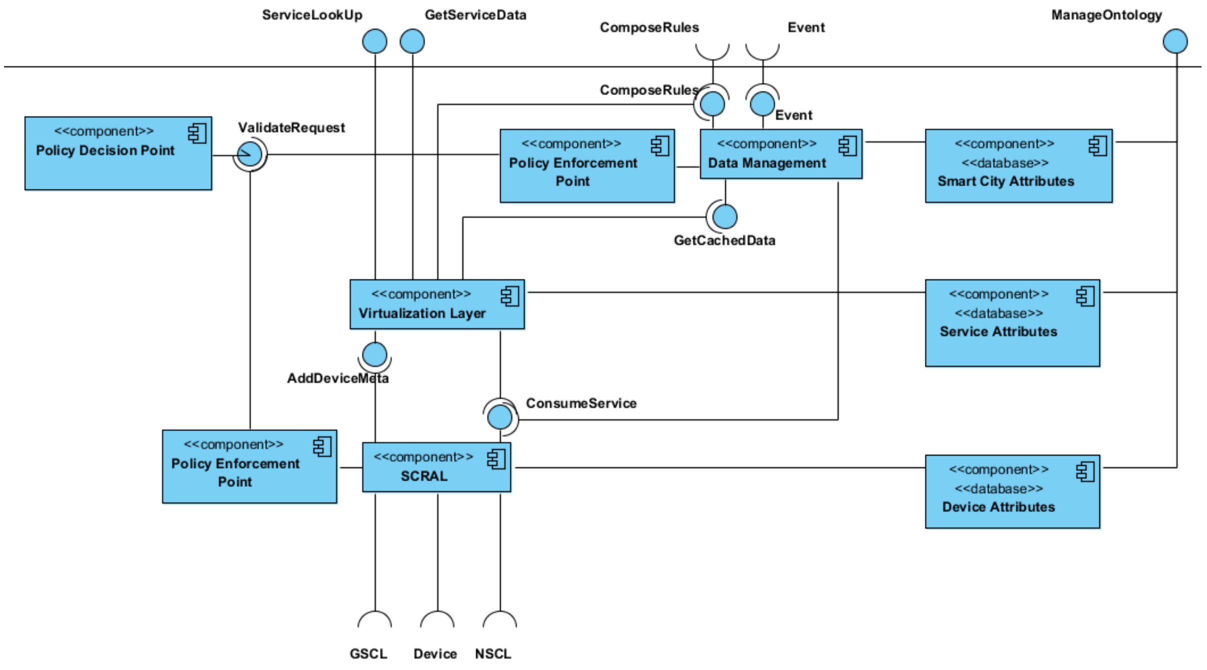

ServiceLookUp
GetServiceData
ComposeRules
Event
ManageOntology
ComposeRules
Event
<<component>> Policy Decision Point
ValidateRequest
<<component>> Policy Enforcement Point
<<component>> Data Management
<<component>> <<database>> Smart City Attributes
GetCachedData
<<component>> Virtualization Layer
<<component>> <<database>> Service Attributes
AddDeviceMeta
ConsumeService
<<component>> Policy Enforcement Point
<<component>> SCRAL
<<component>> <<database>> Device Attributes
GSCL
Device
NSCL

Figure 1: Functional View of ALMANAC platform components (from D3.1.1)

As can be noticed, the platform provides to third party applications 4 main services:

- *Interoperability over devices*. Enabled by the Smart City Resources Adaptation Layer (SCRAL), applications can access any kind of devices, whichever proprietary protocol they may adopt, over a uniform web-service based interface.
- *Service virtualization*. Enabled by the Virtualization Layer, the applications relying on the middleware do not have to know where the services, or devices, they consume / interface are placed, or whether they actually exist. The Virtualization Layer provides a service look-up mechanisms that bridges physical network boundaries, and can even wrap arbitrary data-sets – like historic measurements or cached values – as consumable services.
- *Composition of rules and caching of data*. There are multiple scenarios, where applications are not interested in current device values, but would rather like to be informed if specific conditions are met, or if specific patterns occur in particular intervals: this capability is provided by the Data Management Framework. The component directly grabs data coming from devices, and by parsing and indexing it, enables later complex querying. The rules or mechanisms that the Data Management Framework should execute are either specified by applications either directly or through the Virtualization Layer.
- *Privacy policies of individual providers*. While mainly required by providers of services and data sets, applications can also benefit from knowing that they cannot run into the threat of invading privacy of individual services they consume. Service and device providers individually define policies regarding the data they provide. These policies are then enforced by multiple Policy Enforcement Points (PEP) distributed throughout the platform.

3. Big Data Management

In Smart City applications the amount of data gathered from the city sensors and processes is clearly huge and therefore the ALMANAC platform needs solutions face big data challenges. This section, reports the results of a survey of existing approaches and solutions to Big Data management and analysis, to be used as reference in the DMF design.

Big data is usually collected through various data sources: data warehouses, the Web, networked machines, virtual machines, sensors over the network, legacy applications and clusters, etc. Roughly the challenges in BigData can be divided into how to store the data and the computational framework for processing and extracting knowledge from the data.

3.1 Storage Frameworks

The way data is stored in a database management system (DBMS) determines the allowed kind of queries and their computational properties (e.g., time efficiency). Both the storage and the access methods define the possible optimizations we can think of, and implement, over an existing architecture. Two major storage layouts for DBMSs are currently adopted in storage systems: the *row-oriented* and the *column-oriented* architecture. In a row-oriented database data is stored and processed one tuple at a time, i.e., by elaborating one row of a table at a time. Row-oriented technology has been the basis for all major commercial DBMSs and prevailed in the past years for many good reasons related to enterprise data handling processes. For example, organizing data in form of tuples allows easily loading, updating, and processing, relevant information given a single database entry point. This kind of processing is typical of databases used for online transaction processing (OLTP).

Over the years, the application needs have changed. In addition to OLTP, applications now critically need the ability to handle analytical queries for OnLine Analytical Processing (OLAP). This kind of queries do not always need to process full tuples. Instead, they focus on analyzing a subset of a table's attributes, e.g., by running various aggregations to understand and analyze the data about a given enterprise process. For this kind of applications column-oriented architectures seem more natural, and this lead to the design of a number of systems, e.g., MonetDB and C-Store, based on such assumption. These systems were originally inspired by the Decomposition Storage model (DSM) [2]. Column-oriented DBMSs store data one column at a time, as opposed to one tuple at a time. This means the system only needs to load those attributes (i.e., columns) that are relevant to our query, rather than always loading an entire tuple. Column-oriented storage systems go far beyond simply storing data by row, by offering optimizations such as column-specific compression techniques.

As data collections become larger and larger, response times evolve to one of the major bottlenecks. Applications in, e.g., scientific data analysis and social networks, avoid using database systems due to the complexity and the high response time. For these applications data collections keep growing fast, generating high data volumes to move, store, let alone analyze.

Driven by scalability requirements of Big Data, *NoSQL* data stores have recently emerged as alternatives to classical relational DBMSs. Namely, NoSQL data stores rely on horizontal scaling, which, in turn, relies on partitioning a data store across several machines to cope with massive amounts of data. A specific architecture for horizontal scaling is the shared-nothing architecture in which each machine is independent and self-sufficient and none of the machines being part of a storage system share memory or disk. In a shared-nothing architecture, each portion of a partitioned data store residing on a given machine is called a *shard* and the data store partitioning process is called *sharding*. Horizontal scaling and sharding are in sharp contrast to the vertical scaling approach, traditionally employed in commercial DBMSs, which relies on enhancing the hardware characteristics (e.g., CPU/memory) of a single machine in order to provide scalability.

In contrast to SQL databases, NoSQL data stores typically have a simpler API interface (e.g., a key-value store) and are able to effectively support dynamic addition of new attributes to data records. One way to classify NoSQL data stores is with respect to their different data models. We distinguish three main categories of NoSQL data store models.

- *Key-Value data Stores* (KVS). These data stores handle key-value couplesindexed over the key content. KVS systems typically provide replication, versioning, locking, transactions, sorting, and/or other features. The client API offers simple operations including puts, gets, deletes, and key lookups. Examples include: Amazon Dynamo, Project Voldemort and RIAK.
- *Document Data Stores* (DDS). DDS typically store more complex data than KVS, allowing for nested values and dynamic attribute definition at runtime. Unlike KVS, DDS generally support secondary indexes and multiple types of documents (objects) per database, as well as nested documents or lists. Examples include Amazon SimpleDB, CouchDB, Membase/Couchbase and MongoDB.
- *Extensible Record Data Stores* (ERDS). ERDS store extensible records, where default attributes (and their families) can be defined in a schema, but new attributes can be added per record. ERDS can partition extensible records both horizontally (per-row) or vertically (per-column) across a data store, and can simultaneously adopt both partitioning approaches. Examples include Google BigTable, HBase and Cassandra.

Among other recent approaches to data storage architectures, specifically addressing data loading performance, is the proposed *NoDB* [3]. This design does not require traditional data loading while still maintaining the whole feature set of a modern database system. NoDB uses raw data files fully integrated with the query engine. It is an API with inputs from internal and external data and it is ultimately responsible for serving the information on the basis of a trailing confidence score on the item provided as query. In situ query processing, however, creates new bottlenecks, namely the repeated parsing and tokenizing overhead and the expensive data type conversion costs, which must be addressed by a NoDB prototype system. The techniques used to address these issues involve adaptive indexing mechanisms that maintain positional information to provide efficient access to raw data files, flexible caching structure, etc.

3.2 Computational Frameworks

There are basically two ways of analyzing big data to create knowledge – *batch engines* and *dynamic, interactive querying*. Batch processes are eminently meant to process large amounts of raw data, in order to produce results that can be afterwards analyzed efficiently by an interactive query engine. MapReduce is currently the most widely used programming paradigm for this kind of applications.

The MapReduce paradigm has been developed originally at Google. The approach takes inspiration from functional programming, and it is based on a map phase where local data is processed and transformed in a series of (key; value) pairs. Those pairs are then shuffled across the data center to reducers, each responsible of a set of keys, in order to produce aggregated results. By virtue of having no centralized bottleneck, software that adopts the MapReduce solution can scale to thousands of machines; this allows scaling horizontally a cluster, i.e. adding more machines. The success of this approach can be largely attributed to the fact that the alternative of scaling vertically, by using more powerful machines, is generally much more expensive and in several cases unfeasible.

To support interactive querying of Big Data there are several frameworks such as: Dremel originally from Google, Apache Drill and Impala (which builds on the principles from Dremel). The data model is based on strongly-typed nested records. Dremel moreover uses a columnar data layout: all the values of a given field are stored consecutively, to improve the retrieval efficiency and implements its own SQL-like query syntax.

3.2.1 Hadoop

The most widely used tool for MapReduce approaches is Apache Hadoop which is an open-source framework written in Java that implements the MapReduce programming model discussed above. Rather than relying on hardware to deliver high-availability, Hadoop itself detects and handles failures at the application layer, delivering a highly-available service on top of a cluster of commodity machines.

In short Hadoop is:

- An open source software platform managed by the Apache Software Foundation.
- A way of storing enormous data sets across distributed clusters of servers and then running "distributed" analysis applications in each cluster.
- Robust and safe due to replicas among servers. The NameNode (which keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept) is a single point of failure though.

Hadoop consists of two main parts:

- A data processing framework
- A distributed file system for data storage (HDFS)

In HDFS, data is replicated across multiple nodes which provide data protection and computational performance (Figure 2).

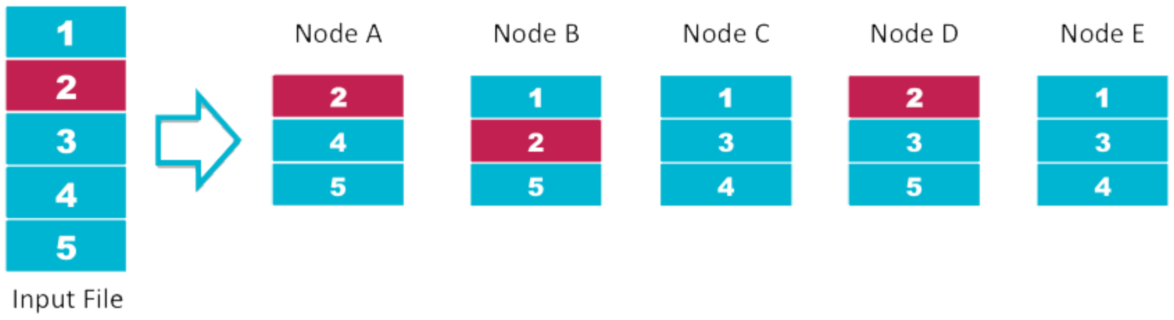


Figure 2: HDFS Data Distribution

Hadoop is not a database but more of a data warehousing system. There are no queries like SQL involved, MapReduce is the mechanism that actually gets data processed. It runs as a series of jobs, with each job being essentially a separate Java application that goes out into the data and starts pulling out information as needed.

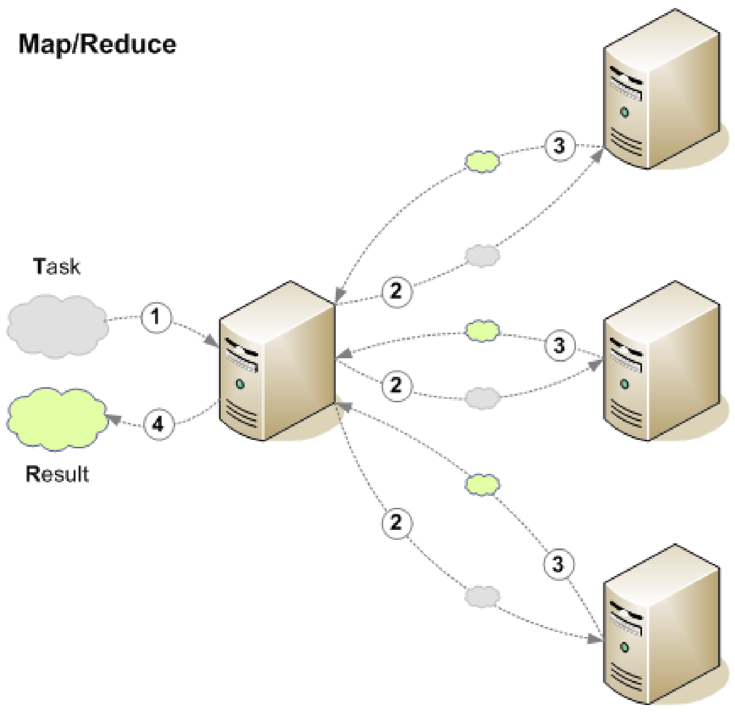


Figure 3: MapReduce task distribution

The task is done in parallel on each node (*mapped*) and then the results from these nodes are *reduced* into one final result.

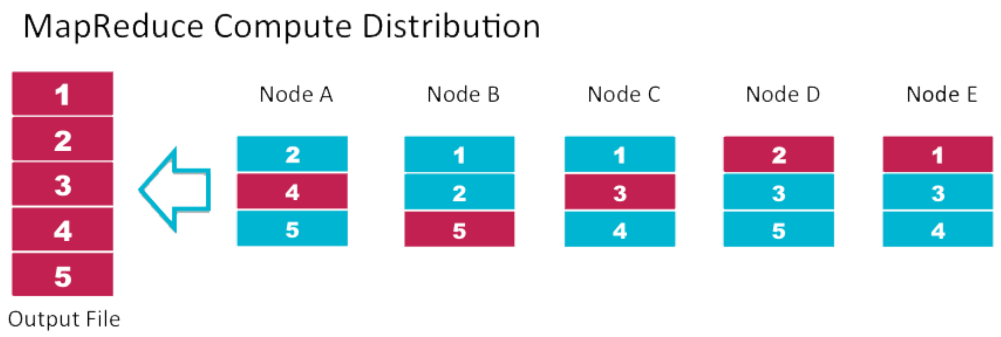


Figure 4: MapReduce results distribution

Using MapReduce instead of a query-language like SQL adds a lot of complexity and there are therefore tools to convert query-language into MapReduce jobs. Examples of these are Hive and Pig. But MapReduce's complexity and its limitation to one-job-at-a-time batch processing tends to result in Hadoop getting used more often as a data warehousing than as a data analysis tool.

3.2.2 Hadoop based systems: HBase and Cassandra

To provide a higher level of database functionality there are products that add a database model/structure on top of Hadoop. The two most well-known are HBase and Cassandra.

HBase is a column-oriented database management system that runs on top of HDFS. HBase is a set of tables with rows and columns. It does not support structured query language like SQL, but provides interfaces for Avro, REST, and Thrift. A master node manages cluster and region servers, which store portions of the tables and perform the work on the data. Each table must have an element defined as a Row (Primary) Key, and HBase is indexed by row key, column and timestamp. Row key design is the single most important aspect of HBase table designs.

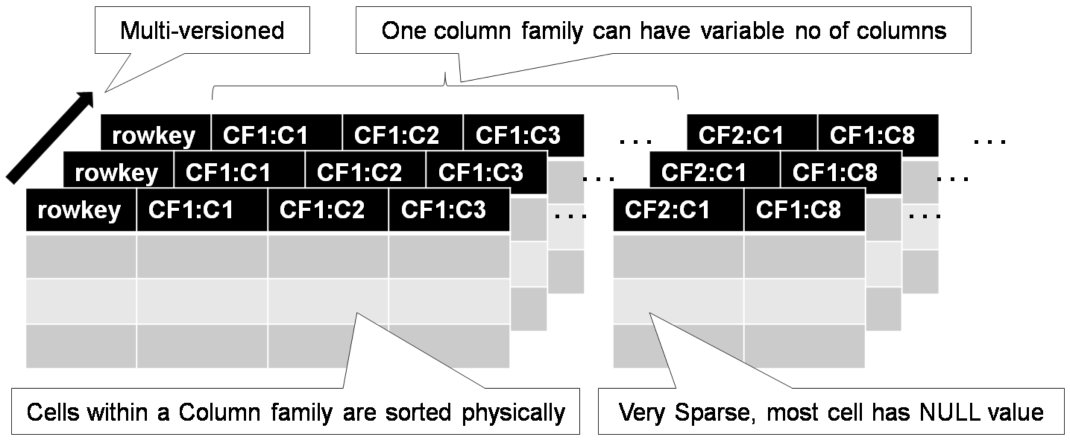
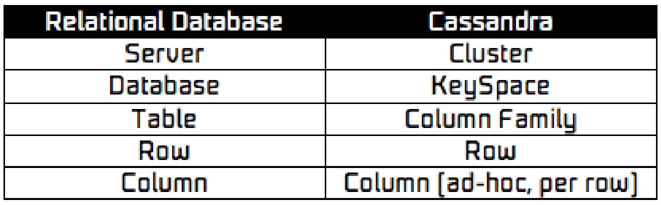

Multi-versioned
One column family can have variable no of columns
rowkey
CF1:C1
CF1:C2
CF1:C3
CF2:C1
CF1:C8
. . .
Cells within a Column family are sorted physically
Very Sparse, most cell has NULL value

Figure 5: HBase columns

Cassandra is a column-oriented database management system with a decentralized architecture where, every node in the cluster has the same role. This provides high availability with no single point of failure. There is also CQL (Cassandra Query Language), a SQL-like alternative query-language.



Relational Database	Cassandra
Server	Cluster
Database	KeySpace
Table	Column Family
Row	Row
Column	Column (ad-hoc, per row)

Figure 6: Cassandra concepts

The following tables summarizes some of the differences between these two Hadoop based systems.

Point	**HBase**	**Cassandra**
CAP Theorem Focus	Consistency, Availability	Availability, Partition-Tolerance
Optimized For	Reads	Writes
RowKey Slices	Yes	No
Explicit Row Locks	Yes	No
Cluster Admin Nodes	Zookeeper, NameNode, HMaster	All Nodes are Equal

Table 1: HBase vs Cassandra

When to use	
Cassandra	**HBase**
Very high velocity random writes & reads	Optimized for reads
Flexible sparse/wide column requirements	Well suited for doing range based scans
No multiple secondary index needs	Applications with strict consistency requirements
	Applications with fast reads and writes with scalability

When not to use	
Cassandra	**HBase**
Dynamic queries on columns	Applications that need: full table scans data to be aggregated across rows
Searching column data	
Low latency	

Table 2: Use of Hbase vs Cassandra

3.3 Management of time series data

Most data generated from Internet of Things resources, and managed in ALMANAC, are time series and querying and analyzing them often requires aggregating observations over specified hours, days, weeks, months etc. Traditional row-oriented SQL databases and query languages are not optimized for handling time series data at large scale. Although systems like Cassandra can be exploited to effectively handle time series, approaches have been suggested for adding time series functionality on top of both HBase and Cassandra. Among these OpenTSDB, a solution for HBase, and, KairosDB on top of Cassandra.

OpenTSDB

OpenTSDB (Open Time Series Database[1]) is, as name suggests, a time series database running on Hadoop and HBase. The HBase schema is highly optimized for fast aggregations of similar time series and to minimize storage space. Users of the TSDB never need to access HBase directly. Applications can communicate with the TSD via a simple telnet-style protocol, an HTTP API or a simple built in GUI.

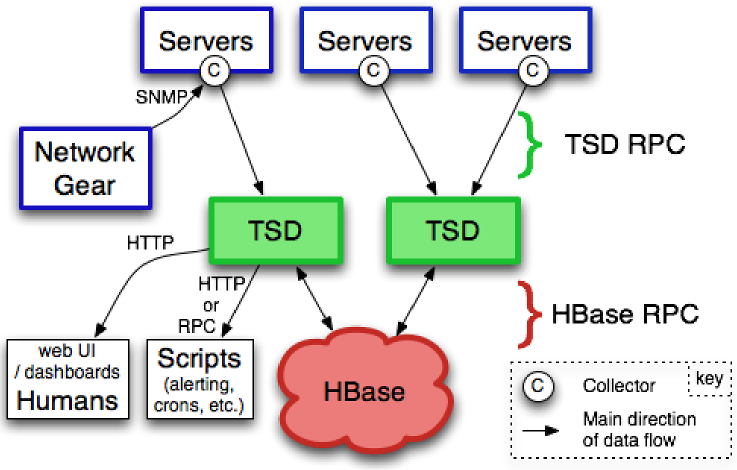


Figure 7: OpenTSDB and HBase

In OpenTSDB, a time series data point consists of:

- a metric name.
- a UNIX timestamp (seconds or milliseconds since Epoch).
- a value (64 bit integer or single-precision floating point value).
- a set of tags (key-value pairs) that describe the time series the point belongs to.

An example of a time series data point in OpenTSDB:

Metric	Timestamp	Value	Tag1	Tag2
<temperature>	<1386132221000>	<27>	<name=S22>	<type=AIR_TEMPERATURE>

[1] http://opentsdb.net/

OpenTSDB provides a batch import-function to fill the database with data from another source. To import data from MSSQL one is first forced to export this data to a text file. For OpenTSDB to be able to import the data, the format should be:

```
<metric> <unixTimestamp> <value> <tagk>=<tagv> [<tagkN>=<tagvN>]
```

KairosDB

KairosDB is a rewrite of OpenTSDB and is written primarily for Cassandra, but works with HBase. KairosDB is much like OpenTSDB in both design, API, and purpose. Yet there are some differences like:

Rows

- OpenTSDB: One row is 1h of metrics (default)
- KairosDB: One row is 3 weeks of metrics (default)

Aggregations

- KairosDB: Aggregations are optional (enables easy access to raw data)
- KairosDB: Aggregators can aggregate data for a specified period

Grouping

- KairosDB: Offers grouping by tags, time range, or value

KairosDB has, like OpenTSDB, a built in batch import-function that enables easy import from other existing databases. However, the format of the import file is not supposed to be in csv but in JSON.

```
{
 "name":"absolutePressure",
 "tags":
       {
         "name":"S3",
         "type":"ABSOLUTE_PRESSURE"
        },
                              "datapoints":[
              [1386076339000,102493],
              [1386076399000,102505],
              …
              [1402491075000,101948]
            ]
}
```

Figure 8: Example of KairosDB-importable JSON-metric

3.4 Design considerations

Based on this survey and our test of some the big data solutions our conclusion is that the “divide and conquer” principles to data management used in MapReduce techniques lend themselves very well to smart city applications, since these applications are by nature often hierarchical, e.g., multiple levels of data aggregation are needed (apartment, building, block, district, city).

The use case analysis shows that the need for managing time series data is important and requires optimized solutions, whereas other data types will require their specific solutions. Therefore, the

ALMANAC data management framework, specifically the data storage, needs to be open and extensible.

In addition to efficient data processing on individual gateways/nodes, the MapReduce approaches can also be applied on a macro level to data management in smart city applications taking full advantage of the network and cloud environments. The approach supports decentralization of processing and data, and thus fits well in a federated networking environment as foreseen in ALMANAC.

4. Data Management Architecture

4.1 Overview

The Data Management Architecture in ALMANAC is based on three core concepts: *IoTWorld, IoTEntity* and *IoTResource.*

- IoTWorld, represents a delimited subset of physical entities which we want to access as a domain (e.g., a house, block, and district).
- IoTEntity, represents a physical entity. A set of IoTEntities forms an IoTWorld
- IoTResource represents a software component that provides services for observation of or actuation on IoTEntities.

An *IoTWorld Gateway* is a software component that provides a logical interface towards an IoTWorld. The IoTWorld gateway exposes a number of IoTEntities and provides a high-level API for communicating with this part of the physical world. For example the IoTWorld gateway "Peters House" can expose a number of rooms and a set of appliances per room. Two or more "House" IoTWorld gateways can report to a "Block" IoTWorld gateway which does various aggregations and also includes data other types of IoTEntities such as Waste Bins and provides a high level API towards the Block. Blocks can be combined into "Districts" which can be combined into "Cities". As an example, when a "District" needs to calculate its water/energy consumption it asks each individual "Block" about its water/energy consumption which in turn will ask each of its underlying "House".

In this way IoTWorld gateways can be combined in many different ways depending on the application, providing different ways of scalable data processing in smart city applications. It should be noted that an IoTWorld gateway does not necessarily correspond to a physical hardware gateway, it just provides a logical interface to part of the physical world.

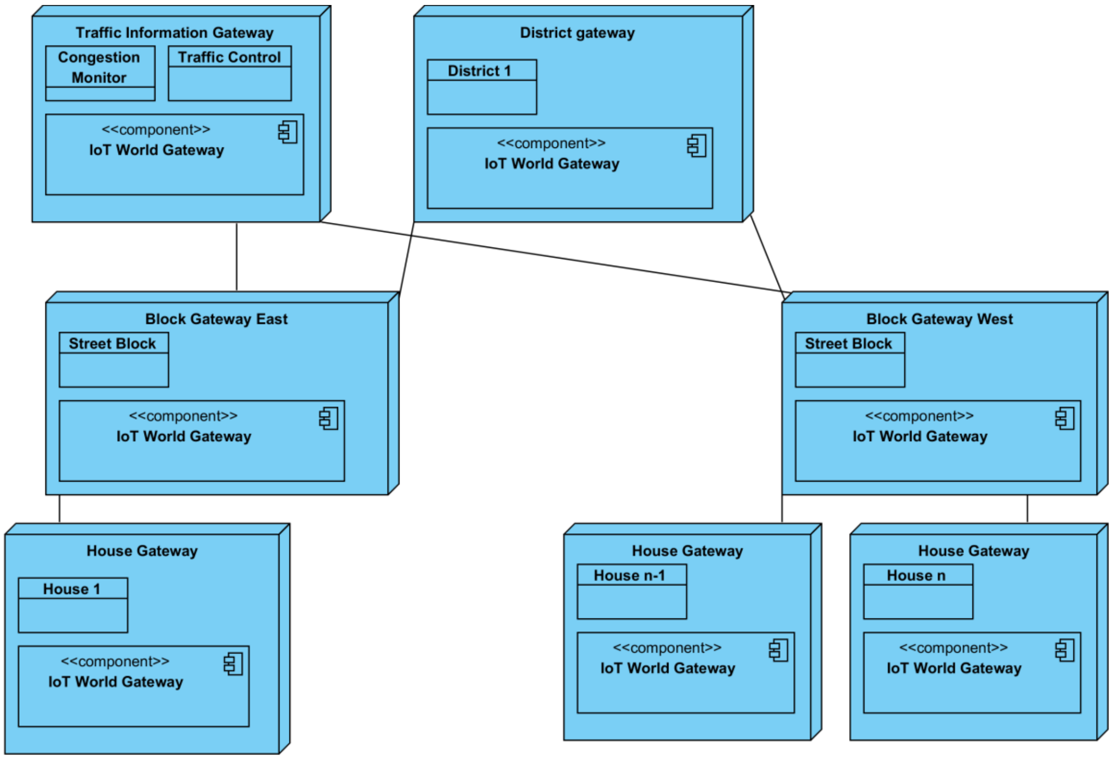


Figure 9. IoTWorld gateways provide a way to distribute and aggregate smart city related data.

Each IoTWorld gateway contains a set of IoTResources providing several services for accessing the IoTEntities of the corresponding IoTWorld. The gateway also contains an *Application Resource Catalogue*, which provides means of searching and locating IoTResources and a *Storage Manager* for management of the data generated from the IoTWorld, see Figure 10 below.

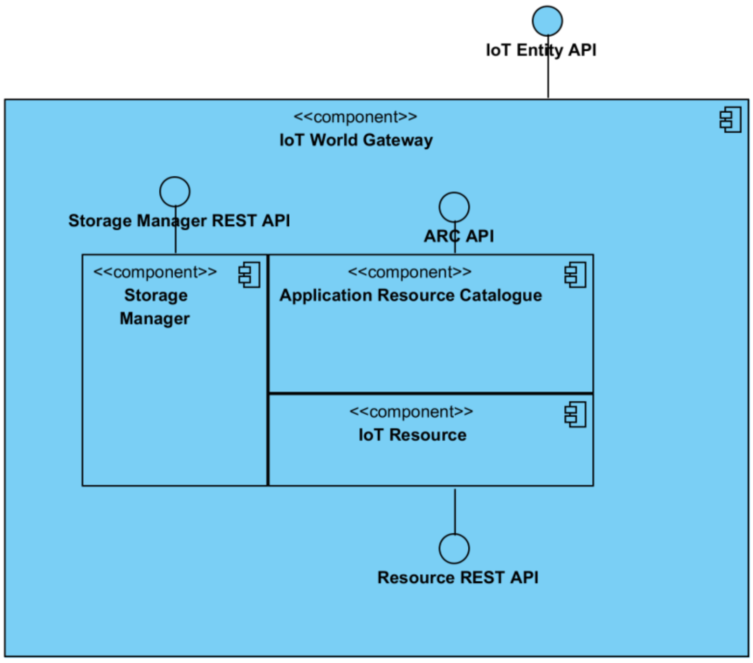


Figure 10: IoTWorld gateway details

The Data Management Framework provides functionality for

- Publishing and subscribing to data streams for components within the ALMANAC platform as well as for end user applications outside the ALMANAC platform.
- Requesting values from IoTEntities and IoTResoruces in the system (sensors and aggregates of sensor data).
- Creating aggregates of data streams to provide higher level objects.

The ALMANAC data model defines the syntax and semantics of the structures needed to retrieve, distribute and store, any data objects, messages and events generated in the ALMANAC platform. As have been explained above IoT applications that needs to interface with real world objects do this through IoTEntities and the underlying IoTResources (providing IoTServices). This allows transparent and simple monitoring of objects state, and easier actuation, independently from the actual object technology or location.

The Data Management Framework in ALMANAC Specifically tackles these 2 aspects, i.e.,(a) Physical World Data Management;(b) IoT Resource and Service Data Management.

4.2 Storage Manager

The StorageManager handles data persistence for all ALMANAC sources, sensors or results of data fusion queries in the Data Fusion Manager. Such data may be distributed by the StorageManager over different IoTWorld Gateways (nodes) in the ALMANAC system. These IoTWorld Gateways can be implemented on local hardware gateways or servers, or by services in the cloud, and optimized for the scenarios in ALAMANAC where short-term storage and long term storage are used. Storage will be transparent to devices and resources producing data, and to resources and smart city applications.

Data is accessed through REST-based interfaces with URI structure and semantics like the other interfaces in the Data Management Framework, making call forwarding and caching simple. A common event and data model is used in communication between other components and the Storage Manager API. The actual implementation of the Storage Manager may differ depending on the type of data to be stored (e.g. metadata or time series data) and the capabilities of the node where storage is deployed (e.g. resource constrained device or high-end server).

In the Architecture specification D3.1.1 System Architecture Analysis & Design Specification 1), the Storage Manager is request based only - it does not push data, nor is it possible to subscribe to data or schema changes in the Storage Manager. In the initial design, all data from sensors is published by the IoTWorld gateway to both the Storage Manager and the Virtualization Layer. The Virtualization Layer is responsible for forwarding data to external applications as well as for responding to requests for data and routing these to either the Storage Manager or SCRAL. Data that is derived from event processing is also published both to the Virtualization Layer and the Storage Manager.

ALMANAC query facilities can be used to retrieve data in two ways depending on applications and end-user context. Observations on properties of physical world entities are stored in the Storage manager which allows retrieval, processing and analysis of historical data, typically represented as time series. Other applications might, instead, require a direct communication with IoTResources, for instance if requirements on real time information is important. The distributed IoT Storage architecture of ALMANAC supports a flexible way of configuring how to store data over a set of IoT Resources, IoT gateways and IoT Cloud services.

4.3 Data Fusion Manager

The Data Fusion Manager provides the means to compose new data from existing data streams using a data fusion language that will be developed in the ALMANAC project. This new data describes properties of real world entities or new aspects of the data coming from a sensor. By combining these in layers, a set of representations of real world entities in the smart city can be built up that fits the need of a specific ALMANAC end user application (e.g. waste management, water supply, traffic management).

4.4 Scalability

A main design objective for the DMF in ALMANAC is scalability in order to gracefully adapt to increasing number of data sources and event rules generating new data. This will primarily be achieved through horizontal scaling (scaling out), i.e., by adding new IoTWorld Gateways (nodes) and distributing data and computations on them. The mechanisms for scaling out will be designed into the ALMANAC architecture, but at the individual node (IoTWorld Gateway) level we will also make use of the optimal database technology and standards for the particular data types handled.

Data will be distributed over the IoTWorld Gateways in the ALMANAC system by the Storage Manager optimizing storage for ALAMANAC applications with different short-term storage and long term storage requirements.

Storage will be transparent to applications, devices and resources producing data and to consumers such as other resources and smart city client applications. REST-based interfaces for data retrieval, with URI structure and semantics like the other interfaces in the Data Management Framework, is making call forwarding and caching simple. A common event and data model is used in communication between components and the Storage Manager REST API.

4.5 Caching

The Data Management Framework provides caching of data on different levels in the architecture in order to support high availability and resilience. This multi-level caching exploits the layered and modular architecture of the ALMANAC platform. This means that storage capacity is utilized wherever it is available, on devices, gateways, data stores etc.

5. Data Retrieval and Query

In Almanac IoT data is retrieved in two ways depending on applications and end-user context. The first data retrieval channel is through the Storage Manager query interface. Observations on properties of physical world entities are, in fact, stored in the Storage managers on the different IoTWorld Gateways which allows retrieval, processing and analysis of historical data most often time series related data.

The second data delivery means involves, instead, real-time communication with IoT Resources connected to ALMANAC, thus supporting applications needing real-time information, e.g., to update dashboards. As already mentioned in previous chapters, the distributed IoT Storage architecture of ALMANAC supports a flexible way of configuring how to store data over a set of IoTWorld gateways.

In this chapter we will focus on how to retrieve data.

5.1 Query API

The forces acting on the design of the Data Management Framework Query API are

- the desired expressiveness of Query language and API,
- the requirements on the Query Language from the Data Fusion Language (DFL) and,
- the restrictions (some of them purely practical, e.g. available resources) imposed by using different database engines.

For performance reasons, the DFL engine will use queries for long time series against static data stores in the Data Management Framework.

It has been a design choice in the Data Management Framework to use different database technologies to better address different application needs, and to be able to substitute the actual storage used by ALMANAC instances depending on features, licensing, costs and future technical developments. E.g., we would like to use NoSQL, column stores or similar databases for time series data that has huge amounts of data, frequent inserts, no updates and heavy analysis and querying.

For metadata on time series (data streams), document, XML or RDF stores are a better choice. For storing and querying for relationships (containment and other relations) between entities and resources, we may use graph databases such as Neo4J.

To allow for a uniform way of expressing this data across the whole system, and to build uniform interfaces to different kind of databases, we have chosen to limit the schema and data querying capabilities of the DMF. A common schema for exchanging IoT information on entities, data streams and data points has therefore been defined and querying has been designed for using a REST paradigm, only and to provide data encoded in a subset of the OData[2] primitives (or similar) that will be suitable for most clients, and the possibility of constructing and storing more complex parameterized queries in some other language.

In addition to queries and access to data streams there may be a need for statistical analysis of data. It will be investigated whether it may be sufficient for analysis purposes if data can be provided in a simple format such as comma separated values (csv) and supplied to services like Azure Machine Learning[3] or another R engine[4].

5.2 Data format

An XML-based content model provides a representation of an IoTEntity (a Virtual Entity in IoT-A vocabulary [1]) and its properties, modelled by IoTProperty, a serialization that can be used as

[2] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=odata
[3] http://azure.microsoft.com/en-us/services/machine-learning/
[4] http://www.r-project.org/

payload in an Event and as result and body of the RESTful API's GET and POST operations. The IoTEntity corresponds to physical world entity upon which we can observe properties (and perform actions on. The content model is formally defined by an XML Schema. There may be JSON representations of this schema.

The format as it is now will be further developed with possibilities to express relations between entities (e.g. containment) and data streams (e.g. "data stream A is the integral of data stream B" or data "stream X is an estimate of the missing data in data stream Y where the sensor was off-line"). Different representations of the schema for resource-constrained environments and different clients will be implemented.

5.3 Location and position

The current Data Storage prototype supports location data (both as metadata and as stream for any entity or resource) and locations-based search. This is important functionality for a Smart City IoT platform, and the data management activities will have to further analyze both which formats will be supported and how querying will be expressed in the Query API. Currently, simple point coordinates are used internally, while GeoJson[5] support is available.

```
{
  "type":"Feature",
  "geometry":{
     "type":"Point","coordinates":[10.189983,56.170933]
  },
  "properties":{
     "name":"My iPhone | Location of phone",
     "description":"iPhone Device | Location of the iPhone, based on
GPS/WIFI/CELL"
    }
}
```

Figure 11: GeoJson for phone device

5.4 Standards

5.4.1 Query API

Querying the Storage Manager is currently done via an HTTP/REST-based API where resource collections can be filtered by using query string parameters for id, name, type and location. This proprietary API will be smoothly migrated to standard OData query filters to improve interoperability.

The Open Data Protocol (OData), enables the creation of REST-based data services, that allow resources identified using Uniform Resource Locators (URLs) and defined in an Entity Data Model (EDM), to be published and edited by Web clients using simple HTTP messages. The OData JSON Format extends the core specification by defining representations for OData requests and responses using a JSON format. The EDM for the Storage Manager is the schema described in section 7.

The IoTWorld Gateways and its Application Resource Catalogue locally support XPath-based querying for IoTResources, a feature that will be extended to the network level. By using XPath expressions in the Application Device Catalogue REST API, the collection of local gateway IoTResources may be filtered on any part of their XML representation.

5.4.2 Query Language

More expressive query capabilities, especially for metadata and relationships for entities and data streams, may be supported using persistent parameterized queries. These can be stored and later

[5] http://geojson.org/

executed with different parameters using HTTP POST and GET, or an RPC-style API. Depending on the storage implementation, SQL or SPARQL may be used. E.g. querying MongoDB using SPARQL is possible via AllegroGraph MongoGraph[6].

As the storage of time-series data is likely to use specialized data stores like KairosDB or OpenTSDB (discussed in section 3), the query language used for this data is likely to be different from the one used for metadata and relations.

5.4.3 Messaging

The Linksmart platform will be used throughout the project, with the Linksmart Event Manager as a publish/subscribe hub. For interoperability and performance reasons, MQTT[7] is considered for local gateways and PubSubHubbub[8] for end-user clients like web applications, mobile apps and external web services.

5.4.4 Data Formats

The primary serialization format (representation) for device and measurement data adopted by the Data Management Framework is XML-based, enriched with RDF-A annotations. However, additional formats may be supported including plain JSON, JSON-LD, RDF and CSV, depending on specific application requirements

[6] http://franz.com/agraph/allegrograph/
[7] http://mqtt.org/
[8] https://code.google.com/p/pubsubhubbub/

6. Data Fusion

6.1 Data Fusion in ALAMANAC

Data Fusion is the process of integration of multiple data and knowledge facts representing the same real-world object into a consistent, accurate, and useful representation. This general definition is typically specialized in a domain-specific manner and the corresponding tools and implementations might vary accordingly. In the ALMANAC platform the role of Data Fusion (DF) is to fuse events created across the network; doing so should be possible with a simple and uniform method. With this interpretation, for ALMANAC the functionality Data Fusion and Complex Event Processing (CEP) is tightly coupled.

CEP has already been extensively investigated and several tools and platforms are available, including both as research and commercial solutions. There are not only several engines, there are also multiple DF Languages (DFL) available – like PQL for the CEP engine Odysseus from the University of Oldenburg, or EPL of Esper CEP engine from EsperTech Inc. The different engines and languages offer different features and hold different advantages and disadvantages. Therefore the focus of the innovation provided in ALMANAC is on the integration of these engines, providing components for the ALMANAC platform through which this functionality can be reached.

The DFL in ALMANAC will be a language that provides the functionality else offered through several engines, taking into account that not all engines allow all possible operations. Therefore the interpret of this language shall be aware of the semantics of the operations described, doing so the DF engine from ALMANAC is able to "know" if the operation is possible with the engine or engines deployed in the platform. Additionally, a semantically defined language allows the management of the queries, the same way as queries manage data.

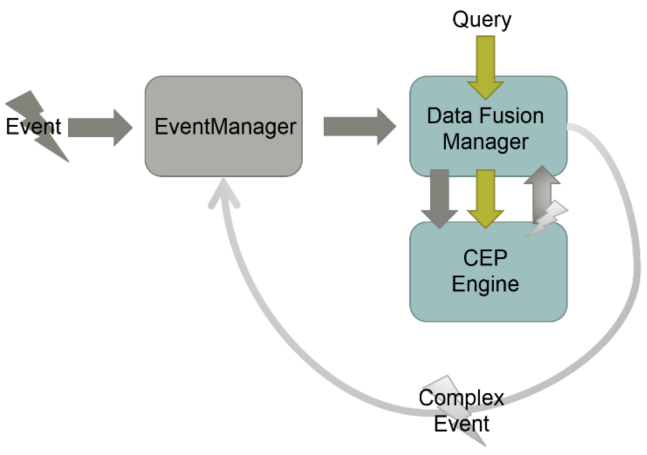


Figure 12: Data Fusion building blocks

The data to be fused by the DF infrastructure is delivered as payload of events pushed by the ALMANAC Event Manager, which works as an event broker. Such data is collected and possibly fused by the DF Manager, applying the stored DF-queries. As the fusion happens, new events can be generated from the query results, and can participate in further fusions to generate complex or aggregate data streams.-This process is depicted in Figure 12**.**

The complete fusion process is managed by available CEP engines, and might be different for specific ALMANAC instances. Therefore the elements needed for a working infrastructure are:

- A component which feeds events into the CEP engines,
- A handler which reacts to the newly created events by the queries,
- An interpreter which holds the queries as processable objects,
- A manager which controls these components,

- Finally a common abstraction of available CEP engines is needed.

The resulting logic architecture is reported Figure 13.

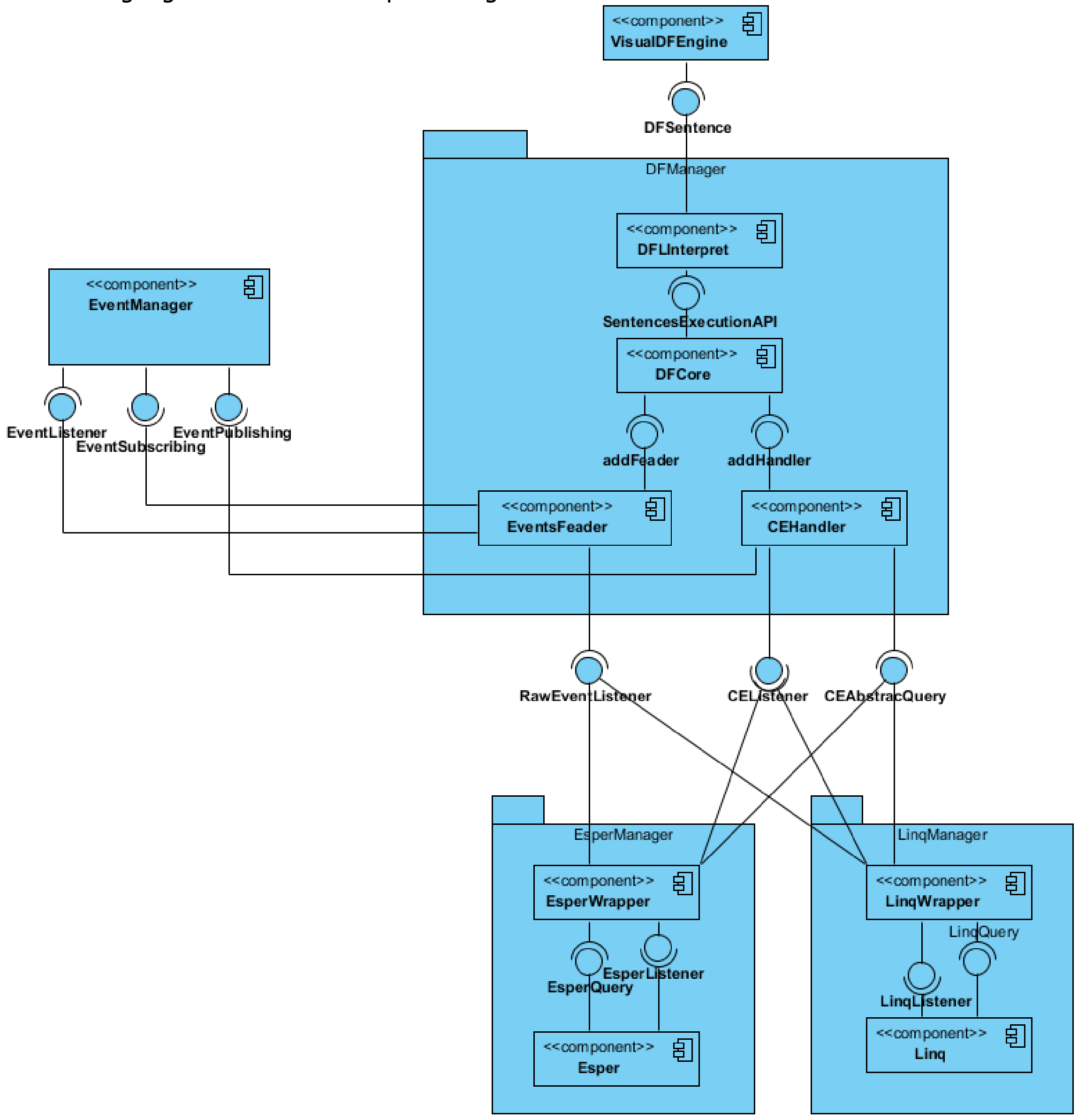


Figure 13: Architecture of first Data Fusion prototype integrating to sample CEP engines.

6.2 Data Fusion Language review and design choices

In order to define a Data Fusion Language able to tackle both fusion and CEP, in a technology independent manner, a first survey of existing languages has been performed and the corresponding features have been categorized and compared to find the greatest common subset of operations supported by them. In general CEP languages can roughly be divided in 2 families, one exploiting SQL-like syntax and the other using no-SQL constructs. In the SQL-like family we have for example: the EPL language from Esper, CQL from Oracle, and Siddhi WSO2; Conversely,non-SQL languages encompass: PQL, a procedural language from Oldenburg Universität, Drools Fusion, a rule oriented language from JBoss, the druid community has also a query language for Data Management which is JSON based.

The ALMANAC DFL does not want to attach to one single technology; therefore a study was made to extract the maximum amount of functionality possible through available CEP engines. With this study it was possible to abstract the functionality and to develop a new language which is independent form the CEP engine implementation running. This language is semantically defined through RDF and its AST (Abstract Syntax Tree). Thanks to the syntactical comprehension of the DFL, the interpretation can identify the functionality and transform the DFL statements to the format of the specific CEP engine deployed. Additionally it is possible to provide three variations of the same language: XML based, JSON based and a Domain Specific Language (DSL) for ALMANAC; in this way the XML and JSON variants can be used for M2M transactions and the DF-DSL would be used by humans.

Due to the heterogeneity requirements of the ALMANAC platform and the high amount of functionality provided by various languages, we decided to concentrate our efforts on EPL at first. We then identified how the query features could be mapped to other languages. After comparing the features of these languages, we tried to work out possible translations between languages and to identify which features are covered by one language and which are not. As an example the attached tables (see appendix) shows the early comparison between some CEP Languages.

Until now our study concentrated on the functionality and semantics of several languages, but there are other functionalities needed in ALMANAC platform for a DFL. Looking at SQL-like syntaxes three sub-languages can easily be identified:

- A Data Definition Language (DDL): CREATE, DROP, ALTER, RENAME
- A Data Manipulation Language (DML):SELECT, INSERT, UPDATE, DELETE
- A Data Control Language (DCL): GRANT, REVOKE

Similarly, the proposed DFL has several sub-languages of which the query language is just one. A possible division of the ALMANAC DFL could be as following:

- Statement Definition Language (SDL): This part of the language manages queries. This can be implemented through SQL and/or SPARQL. The possible statements are:
  - CREATE: The query will be stored and continuously querying the events of the system.
  - RUN: Execute the query and return immediately an answer with the events that match. After the execution the system returns to the state it was before.
  - DROP: A previously created query is removed.
  - PAUSE: A stored query is deactivated but not dropped.
  - CONTINUE: An inactive query is reactivated.
  - QUERY: For querying sored queries.
- Fusion Data Language (FDL): this is the same functionality as provided by e.g. ESPER, Odysseus, etc.
  - SELECT: For query statements
- Data Cloud Language (DCL): this part indicates which data is published outside the local broker:
  - SCOPE: Indicates if the result is sent to another ALMANAC instance or stays in the local scope
- Manipulation Language (DML): As in SQL, this part indicates what should happen with the result:
  - INSERT: This instruction determines whether the event is stored or republished and how.
- Event Generation Language (EGL): Allows the definition of event generation rules

7. Data Management Framework Data Model

7.1 Overview

The ALMANAC data model defines the syntax and semantics of the structures needed to retrieve, distribute, and post to storage, any data objects, messages and events on the ALMANAC platform. IoT applications built upon the ALMANAC platform typically interface real world objects through their abstract counterparts named IoTServices. This allows transparent and simple monitoring of objects state, and easier actuation, independently from the actual object technology or location To support this abstraction, the Data Management Framework in Almanac therefore cover both these aspects, i.e.:

- Physical World Data Management
- IoT Resource and Service Data Management

The ALMANAC project builds on the earlier work on the Hydra/LinkSmart middleware[9]. Within the LinkSmart community, there have been efforts towards a representation to exchange measurement and device data inside the LinkSmart middleware. This has been further developed within the ALMANAC project. It builds on the Linksmart Device model and the UPnP[10] SCPD[11] representation used by the LinkSmart components. Desirable qualities of the representation were that it should:

- Be able to represent the state of sensor and actuator state variables of the LinkSmart resources and devices at different points in time as well as updates (revisions) of these state observations.
- Be able to represent the devices and real-world entities in a way that matched the LinkSmart Device model and IoT-A Virtual Entities[12].
- Be extensible by the LinkSmart community and under the control of the LinkSmart community.
- Leave the choice of XML, JSON, binary or other representations up to the LinkSmart community.

There are of course a number of existing ontologies and formats that have been developed before and after LinkSmart, which have served as input to the design. However, the choice was to design a custom model for LinkSmart intra-component communication. The UPnP SCPD format already featured in the middleware was not chosen since it did not cover the state observation (time series data) aspect. However, SCPD is used as the discovery protocol at the gateway level, where IoT Resources announces themselves either through the UPnP protocol or through a direct registration with the IoT Application Resource Catalogue.

The extensive process-oriented model in SensorML[13] was not a good fit to the LinkSmart device model, which puts more emphasis on the physical devices and real-world entities. Also, the SensorML standard itself does not have a representation for encoding state observations. However, the Observations and Measurements[14] (O&M) standard, which is one of the schemas used to represent measurements together with SensorML, was used as a basis for the state observation elements in the schema. Even though the elements in the observation use the same names as in O&M, the XML representation for O&M was not used. This was in order to have the choice within the LinkSmart community to construct the representation in a way optimal for the data exchange between components inside the LinkSmart middleware, e.g. with regards to the choice between XML

[9] http://sourceforge.net/projects/linksmart/
[10] http://www.upnp.org
[11] http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v2.0.pdf
[12] http://www.iot-a.eu/public/public-documents
[13] http://www.opengeospatial.org/standards/sensorml
[14] http://www.opengis.net/doc/is/om/2.0

or JSON and the compactness of the representation. SenML[15], used to represent measurements in some HyperCat[16] catalogues, had the desired compactness and choice of JSON or XML representation, but lacked some of the features incorporated from O&M (e.g. ResultTime) and did not tie into the rest of the data format naturally.

The conclusion for ALMANAC was to build on the existing LinkSmart models, incorporating elements from other standards, while keeping the design of the data exchange format up to the LinkSmart community. If the need arises to integrate with other middleware that uses any of the aforementioned standards, transformations to and from subsets of these are certainly possible.

7.2 Physical World Data Management

The IoTEntity XML content model is a representation of an IoTEntity (corresponding to the Virtual Entity in IoT-A [1]) and its properties, a serialization that can be used as payload in an Event and as result and body of the RESTful API's GET and POST operations. An IoTEntity corresponds to a physical world entity upon which we can observe properties and perform actions on. The content model is formally defined by an XML Schema. There may be JSON representations of this schema.

7.2.1 Element: IoTEntity

When used as part of the RESTful API, five types of resources may be accessed:

- IoTEntity - Represents a physical entity as a whole. This level is also used as content in Event messages.
- IoTProperty - An individual property of an entity identified by the URI in the request.
- ArrayOfIoTEntity - A list, or collection, of IoTEntity elements.
- ArrayOfIoTProperty - A list, or collection, of IoTProperty elements of an entity identified by the URI in the request.
- ArrayOfStateObservations - A list, or collection, of IoTStateObservation elements of a property of an entity, identified by the URI in the request

Each level is represented as a root element in the corresponding XML document and is described in detail below. Note that IoTProperty will also be present as a child element to IoTEntity.

Entity objects are represented by IoTEntity elements with the following content model:

[15] http://tools.ietf.org/html/draft-jennings-senml-10
[16] http://www.hypercat.io/

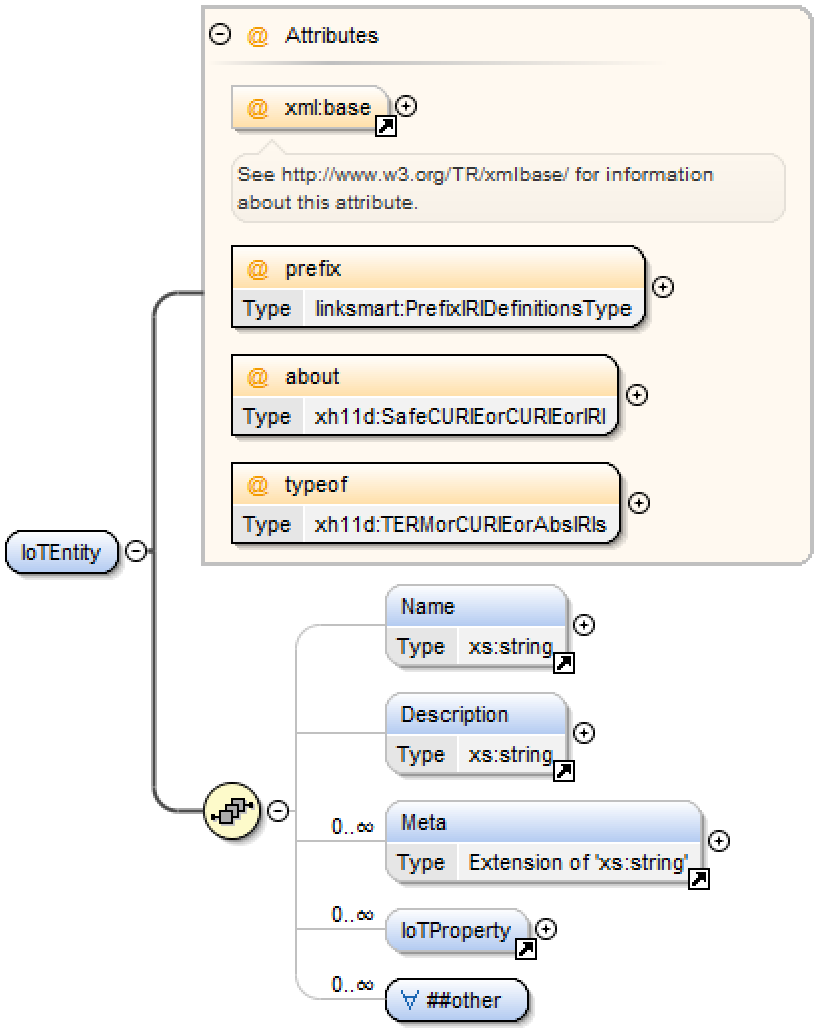

Attributes
xml:base
See http://www.w3.org/TR/xmlbase/ for information about this attribute.
prefix
Type linksmart:PrefixIRIDefinitionsType
about
Type xh11d:SafeCURIEorCURIEorIRI
typeof
Type xh11d:TERMorCURIEorAbsIRIs
IoTEntity
Name
Type xs:string
Description
Type xs:string
0..∞
Meta
Type Extension of 'xs:string'
0..∞
IoTProperty
0..∞
##other

Figure 14: The IoTEntity Content Model

The IoTEntity is using a set of RDFa[17] attributes @about, @prefix, @xml:base and @typeof where all attributes except @xml:base are required. The @about attribute must be used for providing a globally unique identifier of the entity.

The optional Name and Description elements can be used to provide human readable descriptions of the entity. They are not intended to hold any machine interpretable data.

The optional and repeatable Meta element can be used to record any additional metadata about an entity. The type of metadata MUST be recorded using the RDFa attribute @property.

The repeatable IoTProperty element represents all properties of the entity, its content model is described int the following subsections.

Finally, the content of an IoTEntity is open for extension by any other element provided that it is defined within a namespace that differs from the target namespace of the IoTEntity schema.

7.2.2 Element: IoTProperty

The IoTProperty element describes a particular property of an entity in an IoT system, including both metadata and possibly several observations of state conditions, data values or measurements.

[17] http://www.w3.org/TR/xhtml-rdfa-primer/

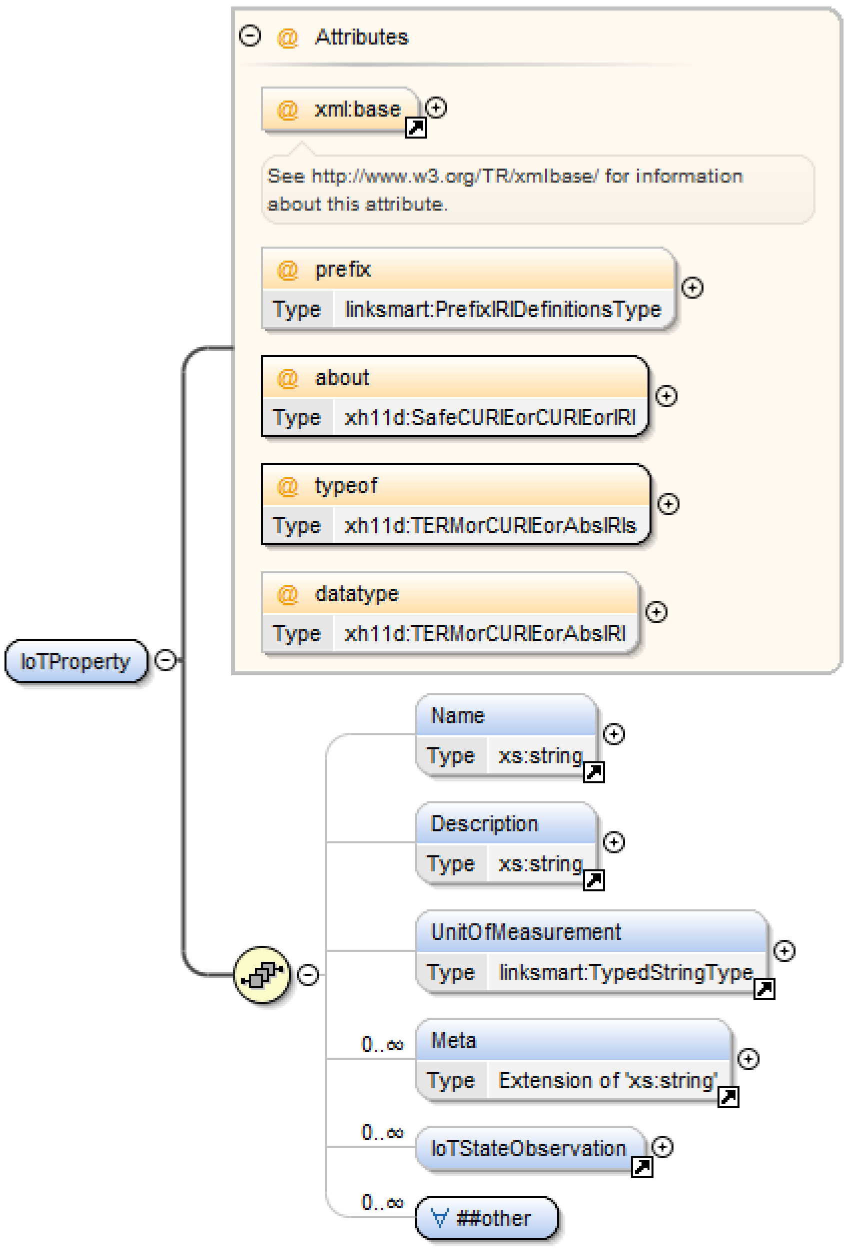

Attributes
xml:base
See http://www.w3.org/TR/xmlbase/ for information about this attribute.
prefix
Type linksmart:PrefixIRIDefinitionsType
about
Type xh11d:SafeCURIEorCURIEorIRI
typeof
Type xh11d:TERMorCURIEorAbsIRIs
datatype
Type xh11d:TERMorCURIEorAbsIRI
IoTProperty
Name
Type xs:string
Description
Type xs:string
UnitOfMeasurement
Type linksmart:TypedStringType
0..∞
Meta
Type Extension of 'xs:string'
0..∞
IoTStateObservation
0..∞
##other

Figure 15: The IoTProperty Content Model

Similar to the IoTEntity, also IoTProperty is using the same set of RDFa attributes @about, @prefix, @xml:base and @typeof.

The @about attribute must be used for providing a unique identifier of the property within the scope of its parent entity, i.e. the id space for properties is locally scoped in contrast to the IoTEntity id space that should provide globally unique identifiers. The concatenation of IoTEntity and IoTProperty identifiers forms a globally unique identifier of the property.

The @prefix attribute is optional but must be used to define any CURIE[18] prefix used by any child element of IoTProperty and not defined by any other @prefix attribute of the parent IoTEntity element.

[18] Compact URI expressions, http://www.w3.org/TR/curie/

The IoTProperty element can also include an optional @datatype attribute that should be used to record the type of the data value of the property recorded within the children IoTStateObservations/Values elements. It is recommended to use XML Schema datatypes such as 'xs:datetime' where the prefix 'xs' must be defined using the @prefix attribute.

The optional Name and Description elements can be used to provide human readable descriptions of the property. They are not intended to hold any machine interpretable data.

The optional UnitOfMeasure element can be used to record the unit of measurement for the data value provided in the child IoTStateObservation/Value element, such as "kW" or "ppm". The type of data MUST be recorded using the RDFa attribute @typeof.

The optional and repeatable Meta element can be used to record any additional metadata about an entity. The type of metadata must be recorded using the RDFa attribute @property.

The values of the property, a time-series based data set is represented by the optional and repeatable IoTStateObservation element, that includes the Value element recording the data as a string, and time stamps for both PhenomenonTime and ResultTime. The Phenomenon Time describes the time that the result applies to the property of the IoTEntity and the Result Time is the time when the procedure associated with the observation act was applied, equivalent to the concepts with the same names in the OGC Observations and Measurements Model[19]. Also IoTStateObservation includes an optional extension point that can hold additional complex data structures representing property values using elements from another namespace than used as target namespace by the IoTEntity schema.

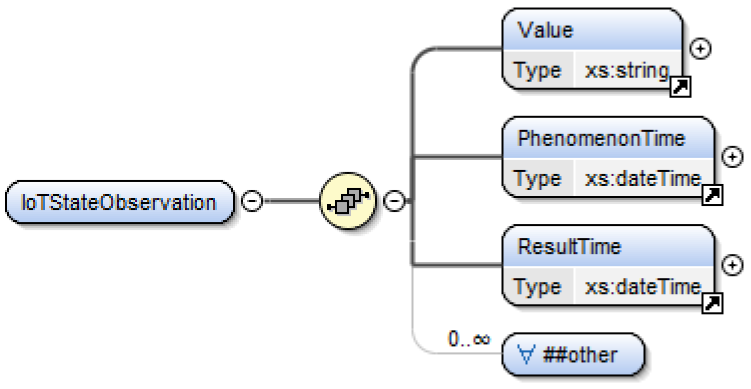
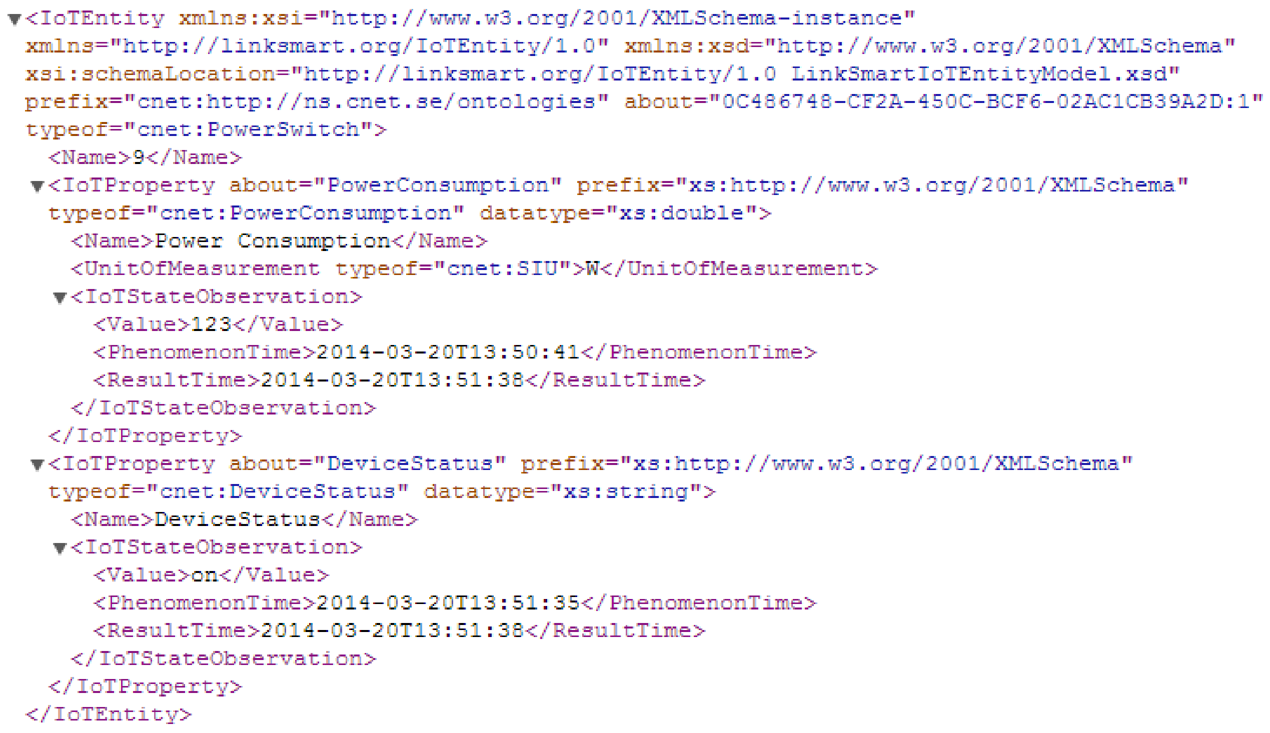


Figure 16: The IoTStateObservation content model

Finally, the content of an IoTProperty is open for extension by any other element provided that it is defined within a namespace that differs from the target namespace of the IoTEntity schema.

An example of a Power Switch entity is provided below (Figure 17). A power switch has two properties, current power consumption and a status that can be either on or off.

[19] : http://www.opengis.net/doc/is/om/2.0



```
<IoTEntity xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns="http://linksmart.org/IoTEntity/1.0" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xsi:schemaLocation="http://linksmart.org/IoTEntity/1.0 LinkSmartIoTEntityModel.xsd"
 prefix="cnet:http://ns.cnet.se/ontologies" about="0C486748-CF2A-450C-BCF6-02AC1CB39A2D:1"
 typeof="cnet:PowerSwitch">
  <Name>9</Name>
  <IoTProperty about="PowerConsumption" prefix="xs:http://www.w3.org/2001/XMLSchema"
   typeof="cnet:PowerConsumption" datatype="xs:double">
    <Name>Power Consumption</Name>
    <UnitOfMeasurement typeof="cnet:SIU">W</UnitOfMeasurement>
    <IoTStateObservation>
      <Value>123</Value>
      <PhenomenonTime>2014-03-20T13:50:41</PhenomenonTime>
      <ResultTime>2014-03-20T13:51:38</ResultTime>
    </IoTStateObservation>
  </IoTProperty>
  <IoTProperty about="DeviceStatus" prefix="xs:http://www.w3.org/2001/XMLSchema"
   typeof="cnet:DeviceStatus" datatype="xs:string">
    <Name>DeviceStatus</Name>
    <IoTStateObservation>
      <Value>on</Value>
      <PhenomenonTime>2014-03-20T13:51:35</PhenomenonTime>
      <ResultTime>2014-03-20T13:51:38</ResultTime>
    </IoTStateObservation>
  </IoTProperty>
</IoTEntity>
```

Figure 17: A power switch entity example

The IoTEntity default namespace is declared on the root element to 'http://linksmart.org/IoTEntity/1.0'. The globally unique identifier for the entity is provided in the @about attribute as '0C486748-CF2A-450C-BCF6-02AC1CB39A2D:1'.

The entity is typed as a 'PowerSwitch' referencing an ontology via the prefix 'cnet' using the CURIE notation. The prefix 'cnet' is defined using the @prefix attribute to the URI 'http://ns.cnet.se/ontologies'.

The two properties have locally scoped identifiers recorded in the @about attributes, 'PowerConsumption' and "DeviceStatus". Both properties are also typed, referencing the previously defined 'cnet' prefix, using the @typeof attribute. The datatypes of the property values are recorded in the @datatype attributes referencing XML Schema datatypes via the 'xs' prefix.

The 'PowerConsumption' property use the UnitOfMeasurement element to record the unit of the value, 'W'. The @typeof attribute is used to reference a type system, again using a CURIE.

Finally, the data values and associated phenomenom and result times for the two properties are provided within IoTStateObservation elements.

7.2.3 Collections of objects - The arrays

Lists, or collections, of objects are provided within a wrapper element at the root level.

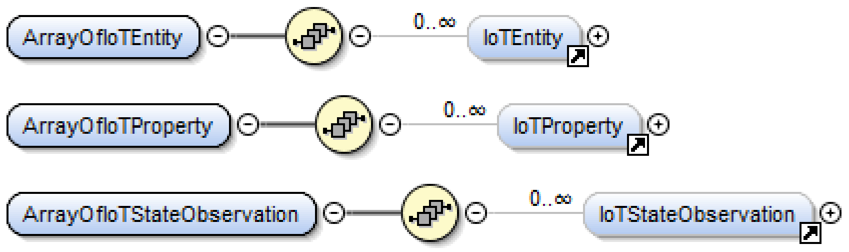


Figure 18: Collection wrappers for collections of IoTEntity, IoTProperty and IoTStateObservation elements

Each collection wrapper can have an arbitrary number of child elements of the specific type. The wrappers do not have any additional elements or attributes.

7.2.4 Events - Content Model

Event message header Content Model

Events are represented by exploiting an XML syntax. The content model for events consists of a generic metadata message header with a payload of data, such as IoTProperty state observations of IoTEntities. The content model is adapted to the semantic web standards from W3C, namely the RDFa Core 1.1 – Second Edition[20].

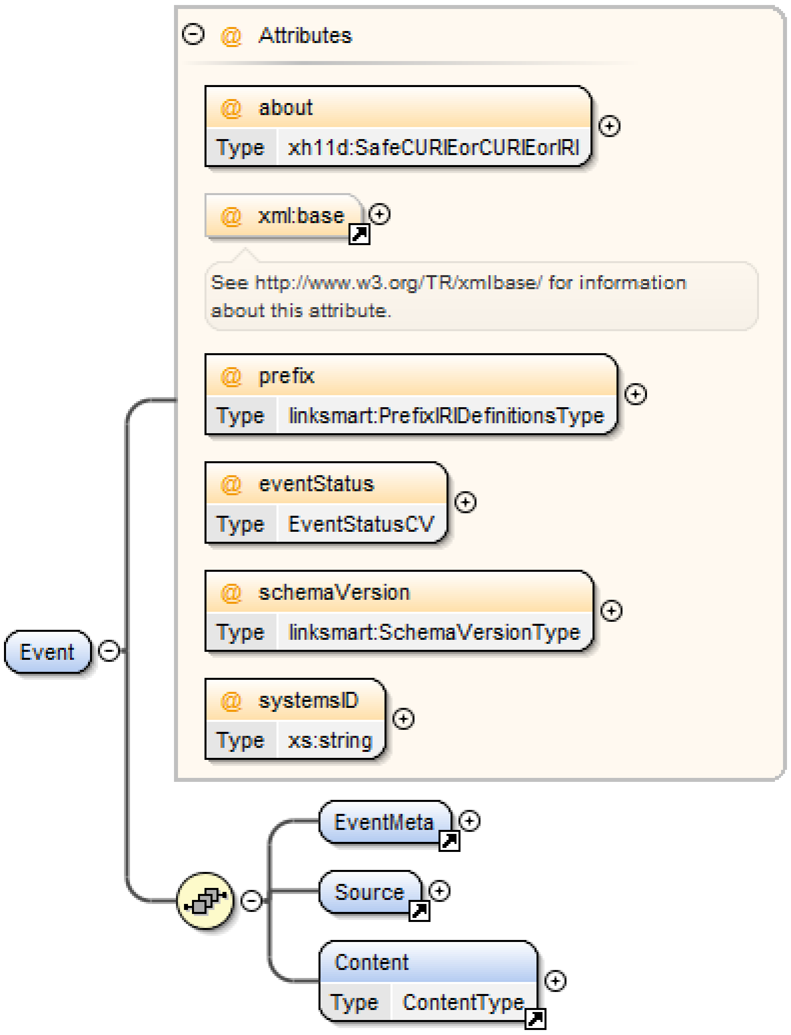

Attributes
about
Type xh11d:SafeCURIEorCURIEorIRI
xml:base
See http://www.w3.org/TR/xmlbase/ for information about this attribute.
prefix
Type linksmart:PrefixIRIDefinitionsType
eventStatus
Type EventStatusCV
schemaVersion
Type linksmart:SchemaVersionType
systemsID
Type xs:string
Event
EventMeta
Source
Content
Type ContentType

Figure 19: The top level of the generic Event Model

The Event model is defined in the Event XML Schema[21] and basically consists of three parts, a descriptive event metadata section (EventMeta), an event source description (Source) and a content body (Content) that includes the actual payload of the event, e.g. a temperature reading for a room entity.

[20] http://www.w3.org/TR/2013/REC-rdfa-core-20130822/
[21] https://fogbugz.cnet.se/default.asp?W90

The @about and @prefix attributes (see RDFa[22]) will be used to express a globally unique identifier for the event and ontology reference definitions for the event message.

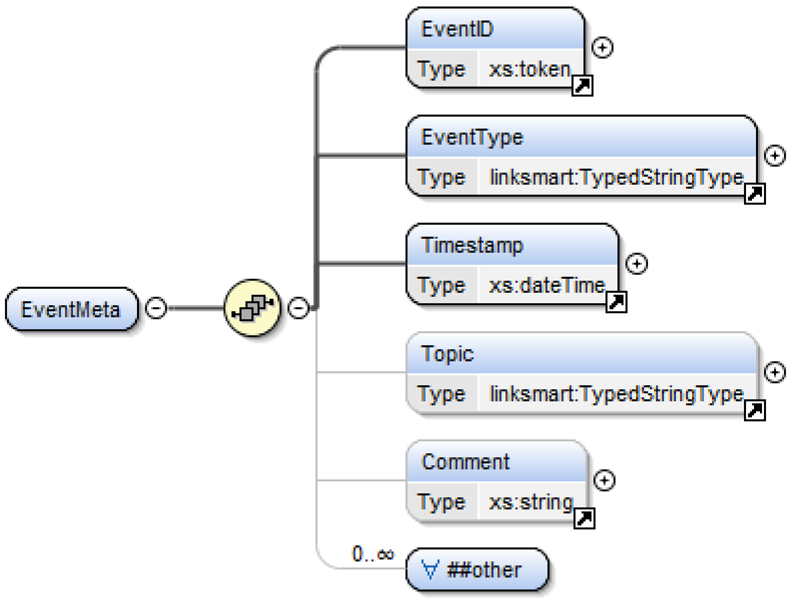


Figure 20: The EventMeta content model

The EventMeta element is described above. The EventID includes a globally unique identifier of the event. It is provided for compatibility reasons and must be the same value as the one expressed in the @about attribute of the Event element.

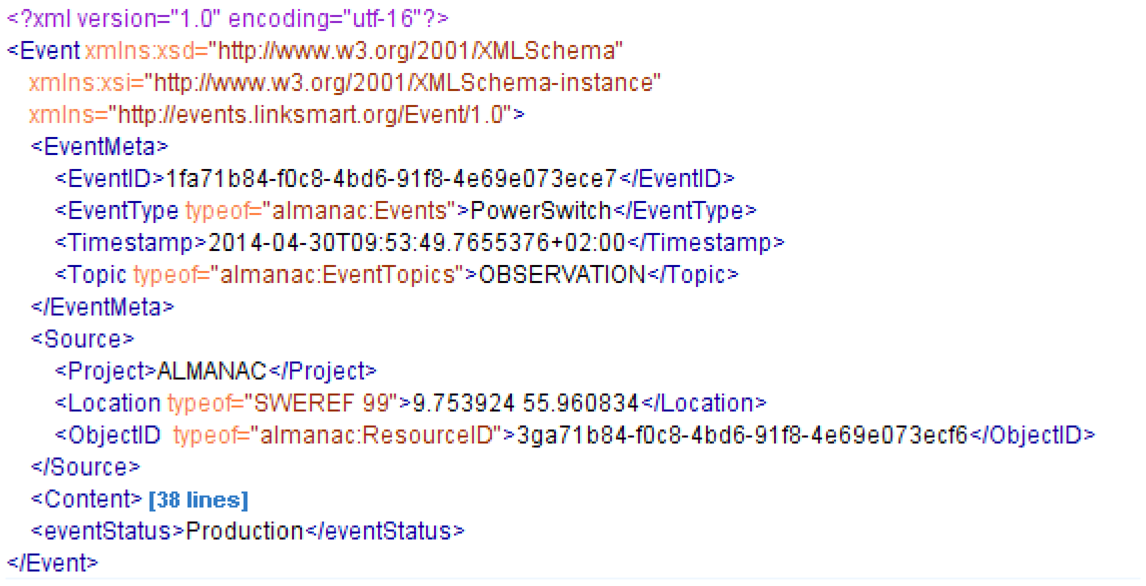
The TimeStamp element holds the xs:datetime of when the event was first generated.

The EventType and Topic elements provide means for tagging the event with descriptive metadata according to a predefined ontology referenced using the @typeof attribute (RDFa).

The Comment element can be used to provide human readable information about the event. The intention is that this element may be used during development and testing.

Finally, the content of an EventMeta is open for extension by any other element provided that it is defined within a namespace that differs from the target namespace of the Event model schema.

[22] http://www.w3.org/TR/2013/REC-rdfa-core-20130822/



```
<?xml version="1.0" encoding="utf-16"?>
<Event xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://events.linksmart.org/Event/1.0">
  <EventMeta>
    <EventID>1fa71b84-f0c8-4bd6-91f8-4e69e073ece7</EventID>
    <EventType typeof="almanac:Events">PowerSwitch</EventType>
    <Timestamp>2014-04-30T09:53:49.7655376+02:00</Timestamp>
    <Topic typeof="almanac:EventTopics">OBSERVATION</Topic>
  </EventMeta>
  <Source>
    <Project>ALMANAC</Project>
    <Location typeof="SWEREF 99">9.753924 55.960834</Location>
    <ObjectID  typeof="almanac:ResourceID">3ga71b84-f0c8-4bd6-91f8-4e69e073ecf6</ObjectID>
  </Source>
  <Content> [38 lines]
  <eventStatus>Production</eventStatus>
</Event>
```

Figure 21: An example of an Event message

Event data content - The message payload

The event payload as expressed within the Content element is normally expected to follow the IoTEntity XML Schema for IoT applications (please see the IoTEntity section for further information about this model). However it is defined opaquely and any XML content that does not violate the well-formed property of the XML document is accepted. The only constraint is that the Event XML Schema requires that the Content child elements are defined in another namespace than used as target namespace by the Event schema itself.

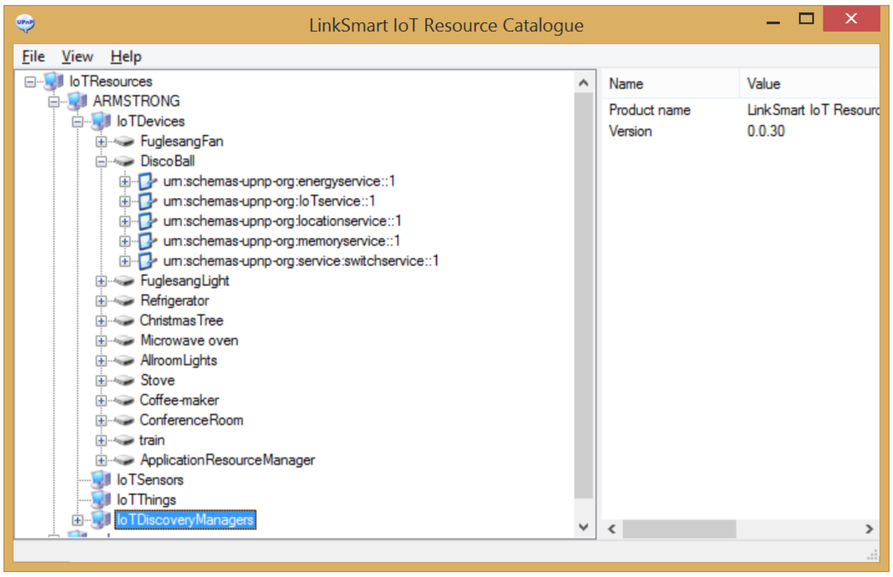
7.3 IoTResources and Services Data Management

IoTResources are software objects that provide IoT Services for applications and end-users, e.g., retrieving and analyzing data about the physical world, invoking actions and so on. Currently three types of IoTResources have been defined and implemented:

- IoTDevice
- IoTSensor
- IoTThing

IoTResources typically runs in IoTWorld gateways. The IoTResources provides the means to deliver the information about IoTEntities in the IoTWorld and to actuate on them.

As an example see figure below that shows which IoTResources have been discovered on the gateway ”ARMSTRONG” and which IoT Services they offer. IoTResources are discovered and managed by the IoT Application Resource Catalogue.


LinkSmart IoT Resource Catalogue
File
View
Help
IoTResources
ARMSTRONG
IoTDevices
FuglesangFan
DiscoBall
urn:schemas-upnp-org:energyservice::1
urn:schemas-upnp-org:IoTservice::1
urn:schemas-upnp-org:locationservice::1
urn:schemas-upnp-org:memoryservice::1
urn:schemas-upnp-org:service:switchservice::1
FuglesangLight
Refrigerator
ChristmasTree
Microwave oven
AllroomLights
Stove
Coffee-maker
ConferenceRoom
train
ApplicationResourceManager
IoTSensors
IoTThings
IoTDiscoveryManagers

Name	Value
Product name	LinkSmart IoT Resourc
Version	0.0.30

Figure 22: IoTResource Catalogue Browser

7.3.1 IoT Resource REST

EachIoTResource provides a REST interface to retrieve data about the resource but also the IoTEntity it is connected to.

http://192.68.1.99:61345/services	
http://192.68.1.99:61345/services/<name>	Retrieves all state variables associated with the service
http://192.68.1.99:61345/services/<name>/statevariables	Retrieves the current value of all state variables
http://192.68.1.99:61345/services/<name>/statevariables/<name>	Retrieves the value of a specified state variable
http://192.68.1.99:61345/services/<name>/actions	Retrieves all actions that can be performed on the IoTResource

7.3.2 IoT Resource Catalogue XPath

The IoT Resource Catalogue provides a REST-based interface to select and retrieve data about IoTResources and their services. The REST-interface is based on XPath-querying in the IoT Resource Description files which are based on the SCPD (Service Control Point Document) from the UPnP-standard. Some examples this interface are given below

http://192.68.1.97:40678/*	List all available resources
http://192.68.1.97:40678//IoT:gateway[.='KURSAAL']	List all resources at a

	gateway
http://192.68.1.97:40678//upnp:device[upnp:manufacturer='CNet'][IoT:gateway[.='KURSAAL']	List all resources from manufacturer CNet running at a gateway.
http://192.68.1.97:40678//upnp:device[upnp:manufacturer='CNet'][IoT:gateway[.='KURSAAL'][IoT:currentconsumption>100]	List all resources from manufacturer CNet running at a gateway KURSAAL and that is currentlyconsuming more than 100 W.

7.3.3 IoT Resource Catalogue Retrieval and Service Invocation

The IoT Resource Catalogue also provides an integrated way of querying and invoking services on matching IoT Resources. Two examples are given below:

http://192.68.1.97:40678/*/services/energy	List the state variables for all resources that provides an energy service
http://192.68.1.97:40678//IoT:gateway[.='KURSAAL']/services/switch/actions/TurnOff	Turn off all switches at gateway KURSAAL

8. Conclusions and Future Development

This report describes the first design and implementation choices made in the context of the Data Management Framework Workpackage (WP6) of the ALMANAC platform. We have shown how we will make use of established principles for big data management and implement them in a Smart City context using a Map Reduce technique at a high level and leverage on existing IoT Models like IoTEntity, IoTResources, and IoT Gateways derived from important reference models such as IoT-A. We have also explained how Almanac Data Management Framework makes use of important standards such as REST, XPath, Odata and SCPD.

The document includes the design and implementation of the Storage Manager, IoTWorld gateway, IoT Resource Catalogue, the Query interfaces and considerations about the Data Fusion Language, and the Event and Data model, including central concepts such as the IoTEntity and IoTResource. We also show examples of the REST based interfaces to the Data Management Framework, in particular to the IoT Resource Catalogue.

This document builds further on the results and documentation accompanying the internal deliverable ID6.2 of the initial prototype components for data fusion, storage management and event processing. In the upcoming project iterations, the implementations of the data management components will be refined, the REST APIs will be further evaluated and the functionality of the IoT Resources Catalogue will be extended with resource annotation capabilities. The work will coordinate with the review and revision of the overall ALMANAC architecture in WP3.

9. References

[1] A. Bassi, M. Bauer, M. Fiedler, T. Kramp, R. v. Kranenburg, S. Lange, and S. Meissner, Eds., *Enabling Things to Talk - Designing IoT solutions with the IoT Architectural Reference Model*. Springer, 2013.

[2] G. P. C. a. S. F. Khoshafian, "A Decomposition Storage Model," presented at the ACM SIGMOD International Conference on Management of Data, 1985.

[3] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki, "NoDB: efficient query execution on raw data files," presented at the SIGMOD '12 Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 2012.

10. Appendices

10.1 Data Fusion Language constructs comparisons

Model Window Comparison

	EPL Esper	CQL ORACLE	Siddhi WSO2	Drools Fusion Jboos
Time Window	.win:time(<period>)	'['range <period>']'	#window.time(<period>)	over window:time(<period>)
Time Batch Window	.win:time_batch(<period>)	'['range <period> slide <period>']'1	#window.time_batch(<period>)	
Time Batch Window /w accumulation	.win:time_batch(<period>, <miliseconds>) || .win:time(<size>).win:time_batch(<period>)	'['range <period> slide <period>']'		
Time-Accumulating Window	.win:time_accum(<period>)			
Time-Length Combination Batch Window	.win:time_length_batch(<period>, <size>)	'['rows <size> range <period>']'		
Time-Order Window?	.ext:time_order(<timestam_expresion>, <period>)	?		
Keep-All Window	.win:keepall()	'['range unbounded']'		declare <stream> @expires(<period>) end
Length Window	.win:length(<size>)	'['rows <size>']'	#window.length(<size>)	over window:length(<size>)
Length Window /w Output Slide	.win:length(<size>).win:length_batch(<size>)?	'['rows <size> slide <size>']'		
Length Batch Window	.win:length(<size>)	'['rows <size> slide <size>']'1	#window.length_batch(<size>)	
Sorted Window	.ext:sort(<size>, <column_name> <sort criteria>, <column_name> <sort criteria>, …)	?		
Ranked Window	.ext:rank(<column_name> [, <column_name>] , <size>, <column_name> <sort criteria>[, column_name> <sort criteria>])	?		
Unique Window	.std:unique(<column_name> [, <column_name>])	'['partition by <column_name> [, <column_name>] rows 1']'	#window.unique(<column_name>)	

Expiry Expression Window	.win:expr(expiry expression)	?		
Expiry Expression Batch Window	.win:expr_batch(expiry expression)	?		
Constant Value-Based Range	.win:expr(<expresion>)?	'['range <float> on <column_name>']'		
Constant Value-Timestap-Based Range	.win:expr(<expresion>)?	'['range INTERVAL <timestap> DAY TO SECOND on <column_name>']'		
First Length Window	.win:firstlength(size)	?		
First Time Window	.win:firsttime(time period)	?		
Last Event Window	.std:lastevent()	?		
First Event Window	.std:firstevent()	?		
First Unique Window	.std:firstunique(<column_name>)	?	#window.firstUnique(<column_name>)	
Grouped Data Window	.std:groupwin(<column_name> [, <column_name>])	'['partition by <column_name> [, <column_name>]']'?		
Grouped Data Window Length Window	.std:groupwin(<column_name> [, <column_name>]).win:length(<int>)	'['partition by <column_name> [, <column_name>] rows <size>']'		
Grouped Data Window Length Window & Time Window	.std:groupwin(<column_name> [, <column_name>]).win:length(<int>).win:time(<int>) ?	'['partition by <column_name> [, <column_name> rows <size> range <period>']'		
Grouped Data Window Length Window & Time Window /w Output Slide	.std:groupwin(<column_name> [, <column_name>]).win:length(<int>).win:time(<int>) . win:length_batch(size)	'['partition by A1,..., Ak rows N range T1 slide T2']'		

Externally-timed Window	`win:ext_timed(<timestamp expression>, <period>)`	?	`#window.externalTime(<timestamp expression>, <period>)`	
Externally-timed Batch Window	`win:ext_timed_batch(<timestamp expression>, time period[,optional reference point])`	?		

Expression Comparison

	EPL Esper	CQL ORACLE	Siddhi WSO2
Aritmetic: Add, Substract, Multiplication, Division	+, -, '*', /	+, -, '*', /	+, -, '*', /
Module	%	mod	%
Integer Division	<int>/<int>	div, idiv	
Logical Connectors	OR, AND	OR, AND, XOR	OR, AND
Logical Comparison	=, !=, '<', '>', '<=', '>='	=, !=, '<', '>', '<=', '>=', eq, ne, lt, le, gt, ge	==, !=, '<', '>', '<=', '>='
Logical Special	IS	IS	
Logical 1-aria	NOT	NOT	NOT
Navegability	object_identificator{ '.'(<function>|<property>)}	'.'	
Concatenate	'||'	'||'	
Binary	&, '|', ^		
Pattern: Alternation	'|'	'|'	
Pattern: Grouping	'(',')'	'(',')'	
Pattern: Single-character duplication	'*' '+' '?'	'*' '+' '?'	

Pattern: Concatenation	' '	' '	
Emptiness test	***none***	empty	
Existence	exists	exists	***none***
Cartridge		@	
Single Row Functions	<function>	<function>	
Array Definition	{ [<expression> [,<expression>]] }		
Dot Operator	<expression>.<method>([<parameter> [,<parameter>]])[.<method>(...)][...]	<expression>.<method>([<parameter> [,<parameter>]])[.<method>(...)][...]	
Duck Typing	<expression>? || <expression>.<method>? || <expression>?.<method>?		
Belongs to	<test_expression> [not] in (<expression> [,<expression>]*)		
In range	in ('['|'(')<int>:<int>(')'|']')		
Between	between <begin_expression> and <end_expression>		
Matching	<test_expression> [not] <like pattern_expression> [escape <string_literal>]		
Regular Expresion comarison	<test_expression> [not] regexp <pattern_expression>		
One or more True	<expression> <Logical_operator> (any|some) (<expression>		

	[,<expression>...])		
All true	<expression> <Logical_operator> all (<expression> [,<expression>...])		
Instantiation	new { <column_name> = [<assignment_expression>] [,<column_name>...] }		
Lamda function	<lambada>		
Alias	Create an alias for an expression	'AS' identifcator	