Notes on Data Integration in FLEXe

This document was authored by Markus Stocker in January 2016 and reflects some of his thoughts on potential practices for data integration in FLEXe.

Modelling for flexible energy systems relies on heterogeneous data and information resources. These are typically distributed and curated by different institutions. The data and information are accessed via heterogeneous interfaces, ranging from text files to Web services, and are presented in heterogeneous formats. Integrating such resources is thus crucial in order to provide modelling, and systems more generally, with data that can be easily accessed, understood, and processed. Integration aims at reducing entry barries to data utilization by abstracting from the heterogeneity of interfaces, data structures, and schemas. The goal is important because considerable time is spent alone on preparing data required for modelling.

We begin with a general and brief discussion about data integration. Following this, we will introduce to the data sources relevant to these notes. They are also relevant to FLEXe. Finally, we will discuss the presented data integration approaches in a use case that utilizes the data sources.

Data Integration

In this section we discuss data integration. There exist, of course, various approaches to data integration. We discuss basic, syntactic and semantic data integration. Basic is *ad hoc* and can quickly serve a purpose while semantic integration is more sophisticated, better engineered, but also more complex.

Basic

The most basic approach is arguably to get the data from services and then access the data from a computing environment, such as Matlab, in whatever format they are delivered by the service. Matlab supports reading files in a range of formats. For instance, the `shaperead` Matlab function allows reading ArcGIS shapefiles. The same is true for other environments, e.g. R, or programming languages, e.g. Python or Java.

In the basic approach, data integration is thus performed in the computing environment. The script determines which data is relevant, extracts the relevant data from the corresponding source files, and integrates the relevant data into data structures supported by the computing environment. At this point, data can be manipulated to meet a particular purpose.

The approach is *ad hoc* and designed to quickly serve the particular purpose. There are considerable downsides of this approach. Specifically, users continue to be exposed to the heterogeneity of data access interfaces and technologies as well as data formats. Each script needs to implement program logic required to abstract such heterogeneity and translate source data into data structures supported by the computing environment. Furthermore, the program logic required to access source data and to extract relevant data from a source depends on the particular file

format. The more heterogeneous data sources are used in scripts that serve different purposes the more costly this approach becomes. Furthermore, the approach is prone to errors as it requires users to write more code.

Syntactic

Syntactic integration abstracts from the heterogeneity of data format (encoding) and schemas of multiple data sources, as well as technologies needed to access data. For instance, given a source with data formatted as CSV text files with schema determined by column labels on the first row and another source with data formatted as XML text files according to OGC Web Feature Service schema, the goal with syntactic integration is to provide an overarching schema for the data and format the data according to a shared encoding.

It is advisable to manage integrated data using a database, in particular a relational database management system. Doing so comes with the benefit of a unified declarative query interface. This significantly reduces the amount of code required to retrieve data relevant for a particular purpose. Indeed, simple questions can be formulated without implementing any program code.

The classical approach is that of Extract-Transform-Load (ETL), whereby a component standing between the data sources and the database extracts data from data sources, transforms the extracted data to meet the chosen format and schema, and loads the transformed data into the database. Applications can thus query the database using a high-level declarative query language, such as SQL, and retrieve data relevant to the particular task to be implemented. An important limitation of this approach is synchronization: the ETL process needs to be executed in order to synchronize integrated data with changed source data.

This limitation can be addressed by using mappers between the integrated schema and each data source, rather than an ETL component. With this approach, queries on the integrated schema are translated to specialized queries for relevant data sources. Users are thus exposed to the state of each data source, rather than the state of an intermediate component. Synchronization is not required.

Semantic

Syntactic integration does not address semantic interoperability of data. Perhaps the primary goal in semantic integration is to enrich data with machine interpretable semantic annotations that describe the meaning of data. Furthermore, semantic integration attempts to resolve conflicts in the semantics of concepts shared between data sources. A classical problem is that the same labels used in the schema of two or more data sources may mean different things. Semantic integration attempts to resolve such conflicts.

Ontologies are a tool for semantic integration. Beyond specifying how the data is structured, i.e. its syntax, ontologies support the specification of the meaning of terms, their semantics. The *de facto* standard set of technologies in this area are those of the Semantic Web, in particular the Resource Description Framework (RDF), the Web Ontology Language (OWL), and the SPARQL query language.

Semantic integration is arguably more challenging than both the basic and syntactic integrations discussed here. In fact, today semantic integration is probably practiced by only the most advanced systems. The task is complex, as it requires teams to agree on definitions of terms and their formalization in a language. Furthermore, the technologies are complex and require specifically trained people. Systems that practice semantic integration are thus still relatively rare exceptions. It is probably so that semantic integration is beyond the means of FLEXe. Nevertheless, we discuss the approach in these notes for the sake of completeness and to highlight what could potentially be done in future work.

Data Sources

In this section we describe the data sources relevant to these notes, in particular the following use case. These data sources are also relevant to FLEXe. We describe in particular two sources: FMI Open Data and building information of the Population Register Center. Further data sources may be included.

The fundamental principles are arguably independent from the number of integrated data sources. Much of what is said here for these two data sources is applicable to other sources. The main exception to this are the integrated schema in the syntactic and semantic approaches. Obviously, with additional data sources the the integrated schema may change.

FMI Open Data

This section presents some notes about FMI Open Data.

The service description can be found here

`https://en.ilmatieteenlaitos.fi/open-data`

The service is excellent for people with a basic background in programming. There exist libraries to retrieve data, in particular for JavaScript but clients exist also for other languages, e.g. R.

Without a library or a tool (see FMIdownloader below), users are required to (programmatically) interact with the service by formulating HTTP requests. Much of the returned data is encoded in XML. It is thus advisable to programmatically post-process the returned XML to extract the required data. Data may be encoded in formats other than XML, such as NetCDF or HDF.

The programmatic interaction with the service obviously comes with the greatest flexibility and best exposure to the data available for retrieval. FMI provides a fairly good documentation online how the service needs to be used. Additional skills in XML data processing are, however, left to the user.

For users without these skills, there is at least the following tool, called FMIdownloader, which supports basic retrieval of daily and real-time (historical) observation data. The tool makes retrieval of such data straightforward, but it should be remembered that it is limited in the type of data it can retrieve.

FMI Weather Data Downloader

There exists a free (and open source) Windows tool that supports downloading FMI open (weather) data to Excel files.

The tool has some advantages:

- It makes data download easy, and thus it makes the FMI Open Data interface accessible to a wider audience. Also users with programming skills can potentially find the tool useful, e.g. to quickly draw some sample data.
- It gives a list of available locations and tells from what time point there is available data. This is basic but useful information.

However, there are some important limitations:

- The tool only supports download of historical daily observation and real-time data. It does not support other types of data download, e.g. model (forecast) data or radar data.
- The tool is also limited in the type of parameters for which data is downloaded.
- The user needs to manually configure the download request. This is fine to quickly get a small data sample but to draw larger amounts of data (e.g. for various locations) automation is preferable.

The year from which data is available is location dependent and is hinted by the tool after location selection. Longterm time series (multiple years) can be downloaded in one request. The time intervals for which data is available can vary between daily observations and real time observations. Missing data should be expected. It is possible to enter a time interval for which no data is available (the tool will return an error).

Daily Observations

The tool supports download of daily (24-h average) observation data for a (point) location (e.g. Kuopio Savilahti) and a time interval, and the following parameters:

- Precipitation amount, 24-hour accumulated (`rrday`) [mm]
- Air temperature, 24-hour avg (`tday`) [degC]
- Snow depth, 1-min instant (`snow`) [cm]
- Minimum temperature, 24-hour min (`tmin`) [degC]
- Maximum temperature, 24-hour max (`tmax`) [degC]

The years from which daily observation data are available vary considerably among the locations. For instance, for Helsinki Kaisaniemi the year is 1844 while for Porvoo Kilpilahti satama it is 2014.

Real Time Observations

Real time observations are 10-min averages. The following parameters are returned:

- Air temperature, 1-min avg (`t2m`) [degC]
- Wind speed, 10-min avg (`ws_10min`) [m/s]

- Gust speed, 10-min max (`wg_10min`) [m/s]
- Wind direction, 10-min avg (`wd_10min`) [deg]
- Relative humidity, 1-min avg (`rh`) [%]
- Dew-point temperature, 1-min avg (`td`) [degC]
- Precipitation amount, 1-hour accumulated (`r_1h`) [mm]
- Precipitation intensity, 10-min avg (`ri_10min`) [mm/h]
- Snow depth, 1-min instant (`snow_aws`) [cm]
- Pressure (msl), 1-min avg (`p_sea`) [hPa]
- Horizontal visibility, 1-min avg (`vis`) [m]
- Cloud amount, 1-min instant (`n_man`) [1/8]
- Present weather (auto), 1-min rank (`wawa`)

In contrast to daily observations, real time observation data typically starts on January 1, 2010, with some locations starting only more recently. There seems to be no real time observation data prior to 2010 for any location.

Download

The tool can be found at the following Web page

`http://tumetsu.github.io/FMI-weather-downloader/`

Follow the instructions. Most importantly, you will need an FMI `apikey' which has to be created via https://ilmatieteenlaitos.fi/rekisteroityminen-avoimen-datan-kayttajaksi.

Building Information

Building information from the Population Register Center comes as an Excel file and an additional textual description (PDF document) of the data. The PDF includes useful metadata and information about the data. Specifically, it includes an explanation for the attribute labels used in the Excel dataset as well as an explanation for the codes. For instance, we can learn that for the attribute *main facade material* the code `1` means *concrete*. The PDF also gives some information about the datatype of each attribute. I am not going to detail more about the available information, not least because it is originally in Finnish and a translation effort is beyond the scope of this document.

The dataset is an interesting case because the metadata is, essentially, not processable by machines (at least not without a major text parsing effort). Without manual intervention, it is thus impossible to join the codes used in the Excel file data with their labels specified in the PDF. For instance, if I like to search for buildings with *wood* as main facade material, I need to lookup the code in the PDF (`5`) and then filter the data for this code. This manual lookup is both error-prone as well as cumbersome.

An important task for the building information dataset is thus to first and foremost integrate data and metadata so that (1) data and metadata can be joined

programmatically and (2) metadata can be processed automatically. This integration can be done in the basic, syntactic, and semantic approach. Once this internal integration is completed, we can proceed with integrating building information with other data and information, weather data in particular.

Ideally, of course, the Population Register Center would provide the metadata in machine processable format. This would not just enable automated processing of metadata but it would also guarantee that metadata used in applications is synchronized with the source. Currently, the manual step required to translate metadata increases chances that data processed automatically is not synchronized with metadata, and thus errors in processing occur which may go unnoticed. In fact, the metadata description in the PDF may change and such changes can go unnoticed by developers. This side note may be useful and important feedback to the Population Register Center. Clearly, if they aim at making their data more useful to consumers, providing the metadata in a machine processable format is arguably a key requirement.

Case Study

Integration between weather data and building information is relevant for flexible energy systems and modelling because weather at a particular location and the characteristics of buildings at that location can impact on energy use. This case study uses the data sources presented above and explores basic, syntactic, and semantic integration of data serviced by the individual sources. Given its popularity as a computing environment, the case study is developed primarily using Matlab.

The Basic Approach

The *ad hoc* approach is to use the FMIdownloader tool to retrieve required data from the FMI Open Data service for a particular purpose. The data will be conveniently translated to CSV and thus straightforward to read into computing environments such as Matlab. For instance, let's assume we need daily observations for Kuopio Savilahti for the year 2015. We obtain a CSV file with 365 rows of data and a header row with the labels `rrday`, `tday`, `snow`, `tmin`, `tmax`. This file can be imported into a Matlab data structure, which can then be manipulated according to the purpose specified by the corresponding Matlab script.

However, already this import is not entirely trivial. First, the CSV file is actually not comma-separated, but semicolon-separated. The Matlab function `csvread` fails. Furthermore, the file returned by FMIdownloader has mixed data types, datetime strings, numbers, and text. The Matlab matrix data structure works well for numbers but does not support other data types.

An example one-liner to import the data is

```
[num char raw] = xlsread('weather_data.csv');
```

The three data structures `num`, `char`, and `raw` include the subsets that are numeric, text, or the complete dataset as a `cell` array. This is relatively

convenient, as all data is imported and can now be manipulated. However, surprisingly, the function seems to drop the time information in the datetime string. This seems to be the default behaviour and is the case only if the time is **00:00**. Time other than **00:00** are retained. We can now convert the datetime strings to Matlab `datenum`. This can be achieved as follows

```
date = datenum(char(2:366,1), 'dd.mm.yyyy');
```

and join the resulting vector with the `num` matrix. The date format string is required here, as the default expects the time component. Fortunately, the implementation is flexible on leading zeros, and will treat the date string `1.1.2015` correctly, even though it does strictly speaking not meet the format `dd.mm.yyyy`.

At this stage, we have a Matlab numeric matrix that can be further processed. As the example shows, alone the import of data into the computing environment is not entirely without pitfalls. The required instructions are dependent on the source data encoding, details about the format of data types, as well as the peculiarities of implementations of used functions.

While this *ad hoc* approach does not require upfront data integration the obvious downside of the approach is that each script against the data sources needs to again deal with the same issues. Surely, one can save intermediate results to disk. However, if the source data changes the approach needs additional synchronization. The *ad hoc* approach may thus be suitable to quickly address a particular task but looses appeal the more users and applications require integrated data for various purposes.

Accessing FMI data from Matlab using the Open Data interface is perhaps somewhat less trivial, due the Web Service request and XML data processing. However, the FMI Open Data interface provides access to more diverse data than the FMIdownloader tool and the data is real-time, directly from the source (while the FMIdownloader results into an intermediate persistence of data). The following instructions retrieve the sample data directly from the FMI Open Data Web Service.

```
apikey = ''; % Specify your key here
base = 'http://data.fmi.fi/fmi-apikey/';
path = '/wfs';
request = 'getFeature';
storedQueryId = 'fmi::observations::weather::daily::timevaluepair';
parameters = 'tday';
place = 'kuopio';
starttime = '2015-01-01T00:00:00Z';
endtime = '2015-12-31T00:00:00Z';
query = strcat('?request=', request, '&storedquery_id=', \\
  storedQueryId, '&parameters=', parameters, '&place=', place, \\
  '&starttime=', starttime, '&endtime=', endtime);
url = strcat(base, apikey, path, query);
doc = xmlread(url);
```

```
tvp = doc.getElementsByTagName('wml2:MeasurementTVP');

data = zeros(365,2);

for i=0:tvp.getLength-1
  item = tvp.item(i);
  child = item.getFirstChild;

  while ~isempty(child)
    if child.getNodeType == child.ELEMENT_NODE
      text = char(child.getFirstChild.getData);
      switch char(child.getTagName)
        case 'wml2:time' ; \\
          data(i+1,1) = datenum(text,'yyyy-mm-ddTHH:MM:SS');
        case 'wml2:value' ; \\
          data(i+1,2) = str2double(text);
      end
    end
    child = child.getNextSibling;
  end
end
```

Note that the blackslashes are used here to break the line in order to fit it on A4 documents. Remove the `\\` and breakline in code to execute.

This however only retrieves and processes the temperature parameter `tday`. In order to get other parameters, the `parameters` variable can be modified by listing the desired parameters (see FMI Open Data documentation). However, doing so requires adapting the `for`-loop that processes the XML document.

Clearly, alone the access and processing of data from the FMI Open Data Web Service in order to import data into Matlab data structures requires a considerable amount of program logic. To ease some of the burden, the required programming logic can be encapsulated in a wrapper (e.g. a function) to avoid replicating the code in client scripts. However, developing a function that can be parametrized and retains the full flexibility may not be entirely trivial. Furthermore, each data source will likely have its own wrapper. In order to integrate data from multiple sources, client applications are required to utilize corresponding wrappers. The actual integration remains task of the client application. Perhaps the greatest advantage of the basic approach is that there is no upfront integration work required: it is left to individual client applications.

Given that building information is available as Excel file, loading the data into Matlab in the basic approach is similar to how it is done for FMI weather data retrieved using the FMIdownloader, using `xlsread`. So far so good. As we determined earlier, a first task for the building information is to get the data and metadata integrated. As we noted, the metadata is available as PDF. This raises now the question how to integrate the PDF metadata in Matlab. Two basic

approaches come to mind: (1) the metadata is added to the documentation in Matlab script(s) or (2) it is translated to a suitable Matlab data structure.

Let's take facade material as an example. We want our scripts to use a variable for facade material. The variable takes a number `1, 2, ..., 7`.

```
mainFacadeMaterial = 1;
```

Having the number is of course not very useful as users will have to either learn the mapping between numbers and labels by heart or look it up in the PDF. In order to make script parametrization easier we just add the metadata to the documentation in our Matlab script:

```
% 1=concrete
% 2=brick
% 3=metal
% 4=stone
% 5=wood
% 6=glass
% 7=other
mainFacadeMaterial = 1
```

This is slightly better, as users will not have to lookup the key-value pairs in a separate file. However, documentation gets generally quickly outdated and, unless taken good care of, it is thus likely that at some point our documentation will be wrong with respect to the processed data.

An alternative, and arguably more elegant approach, is to organize the metadata in a suitable data structure. One possibility is Matlab Map Containers. The following is an example:

```
keys = [1, 2, 3, 4, 5, 6, 7];
values = {'concrete', 'brick', 'metal', ...
  'stone', 'wood', 'glass', 'other'};
code2MaterialMap = containers.Map(keys,values)

keys = {'concrete', 'brick', 'metal', ...
  'stone', 'wood', 'glass', 'other'};
values = [1, 2, 3, 4, 5, 6, 7];
material2CodeMap = containers.Map(keys,values)
```

We now have two data structures that we can interrogate for the material of a given code

```
code2MaterialMap(1) % = concrete
```

or the code for a given material

```
material2CodeMap('concrete') % = 1
```

These data structures can also be persisted to disk using Matlab and thus serve as processable metadata objects in scripts that can now simply load these objects. Applications can use these metadata data structures and join metadata and data to resolve the codes.

As there is no intermediate integrative system standing between the data sources and individual applications, in the basic approach integration of two or more resources is left to applications. In fact, also the integration between building information data and metadata is left to individual applications. Of particular interest between weather data and building information is also the integration over space. Lacking an intermediate integrative system, spatial integration is also left to individual applications.

The Syntactic Approach

We discuss the Extract-Transform-Load (ETL) based syntactic integration of data. Specifically, we first specify a common schema for the required data obtained from the distributed data sources. The relational model for databases is used for the schema. After schema specification, we implement the schema in a database and load required data. With this, we obtain a database that can be queried using SQL as a high-level declarative query language to retrieve data required by individual client applications.

Database Installation

As a first step, we need to setup a relational database management system. We use MySQL. Specifically, we install and configure for remote access a MySQL instance on the `enviapps.uef.fi` server. MySQL can be installed on other operating systems, including Windows. The installation and configuration is relatively trivial. For remote access, we will need to change the `bind-address` in `my.cnf` to the IP address of the server and create a new user with localhost and remote access rights as follows

```
CREATE USER 'myuser'@'localhost' IDENTIFIED BY 'mypass';
CREATE USER 'myuser'@'%' IDENTIFIED BY 'mypass';
GRANT ALL ON *.* TO 'myuser'@'localhost';
GRANT ALL ON *.* TO 'myuser'@'%';
```

Let's also create a database `test_db` with table `test_table` and populate the table with some data

```
CREATE DATABASE test_db;
USE test_db;
CREATE TABLE test_table(id int(6));
INSERT INTO test_table VALUES(1);
INSERT INTO test_table VALUES(2);
```

Matlab Database Interaction

In order to use MySQL from Matlab, you will also need the JDBC connector. Load the JAR file and set the user and password variables in Matlab as follows

```
% Fix the path '...'
javaaddpath('...\mysql-connector-java-5.1.38-bin.jar')
user = 'myuser';
password = 'mypass';
```

The following instruction allows you to connect to the database `test_db` on remote host `enviapps.uef.fi`

```
conn = database('test_db',user,password,'com.mysql.jdbc.Driver',\\
       'jdbc:mysql://enviapps.uef.fi:3306/test_db');
```

Test if the connection is active, query the table `test_table`, print the retrieved data, and close the connection

```
isconnection(conn);
curs = exec(conn,'select * from test_table');
res = fetch(curs);
res.Data % Prints the results
close(curs);
close(conn);
```

Now that we have covered the basics of interaction between Matlab and a remote instance of MySQL, we can turn to our data integration.

Extract-Transform-Load (ETL)

First we need to specify the relational schema that integrates data extracted from our data sources. Depending on the data, this can be more or less challenging. In our case, we start creating a database for the use case and a table for wheater data

```
CREATE DATABASE use_case;
USE use_case;
CREATE TABLE weather_data(
  date TIMESTAMP,
  rrday DOUBLE,
  tday DOUBLE,
  snow DOUBLE,
  tmin DOUBLE,
  tmax DOUBLE);
```

We can now use Matlab to ETL weather data into the table, either using the FMIdownloader tool as an intermediate step or directly from the FMI Open Data

Web Service (as described above). For brevity, the following example uses CSV data downloaded using the FMIdownloader tool.

```
[num char raw] = xlsread('weather_data.csv');

conn = database('use_case',user,password,'com.mysql.jdbc.Driver',\\
  'jdbc:mysql://enviapps.uef.fi:3306/use_case');

for i=2:length(raw)
  date = raw(i,1);

  if (isnan(raw{i,2}))
    rrday = 'NULL';
  else
    rrday = num2str(raw{i,2});
  end

  if (isnan(raw{i,3}))
    tday = 'NULL';
  else
    tday = num2str(raw{i,3});
  end

  if (isnan(raw{i,4}))
    snow = 'NULL';
  else
    snow = num2str(raw{i,4});
  end

  if (isnan(raw{i,5}))
    tmin = 'NULL';
  else
    tmin = num2str(raw{i,5});
  end

  if (isnan(raw{i,6}))
    tmax = 'NULL';
  else
    tmax = num2str(raw{i,6});
  end

  fetch(exec(conn,strcat('INSERT INTO \\
    weather_data(date,rrday,tday,snow,tmin,tmax) \\
    VALUES(STR_TO_DATE("', date, '", "%d.%m.%Y"),', \\
    rrday, ',', tday, ',', snow, ',', tmin, ',', tmax, ')')));
end

close(conn);
```

This code will populate the `weather_data` table with data extracted from the `weather_data.csv` file.

The ETL process for our building information data and metadata is in principle similar to that of our weather data. There is, however, one aspect worth highlighting. For building information data and metadata the process requires a more complex schema. Specifically, we need to create individual tables for the code tables provided in the PDF metadata file. For the main facade material code table we thus create the following database table

```
CREATE TABLE main_facade_materials(
  id INT,
  label VARCHAR(32));
```

and then populate the table with the corresponding metadata provided in the PDF as follows

```
INSERT INTO main_facade_materials VALUES(1,'concrete');
...
INSERT INTO main_facade_materials VALUES(7,'other');
```

We thus obtain the following populated table

```
mysql> SELECT * FROM main_facade_materials;
+------+----------+
| id   | label    |
+------+----------+
|    1 | concrete |
|    2 | brick    |
|    3 | metal    |
|    4 | stone    |
|    5 | wood     |
|    6 | glass    |
|    7 | other    |
+------+----------+
7 rows in set (0.00 sec)
```

Naturally, we need to ETL also the remaining data and metadata. For the purpose here, we only discuss a small subset of the building information dataset. Specifically, we ETL the building information for coordinates, type of heating, and the fuel. The necessary steps are already discussed above in details. Hence, we skip the details and only briefly discuss the kind of joined SQL queries we can now execute over the integrated schema of the database.

Working with Syntactically Integrated Data

Now that we have data from our sources syntactically integrated in our database, we can use the database to extract and process data for our purposes. The database clearly abstracts from the heterogeneity of the original data sources. It is now

possible to interact with the data using a single technology, that is SQL. Furthermore, the data is consistent with a well defined set of datatypes. Also, SQL allows us to flexibly declare the data retrieval task and natively supports basic data processing, such as aggregation and joins. Rather than implementing those basic processing tasks in Matlab, we can now formulate them in SQL queries.

With the following query, we can retrieve all data in the `weather_data` table. Note that with tables containing a lot of data you need to be careful with such "fetch all" queries.

```
curs = exec(conn,'select * from weather_data');
res = fetch(curs);
data = res.Data;
close(curs);

data{1,1} % = 2015-01-01 00:00:00.0
cell2mat(data(1,2:6)) % = 0.6000 2.7000 20.0000 0.9000 3.5000
```

As we have the full power of SQL at our fingertips, we can formulate also more interesting queries, such as

```
SELECT date FROM weather_data WHERE tday < -15
```

with results

```
'2015-01-05 00:00:00.0'
'2015-01-06 00:00:00.0'
'2015-01-11 00:00:00.0'
'2015-01-12 00:00:00.0'
'2015-01-20 00:00:00.0'
'2015-01-21 00:00:00.0'
'2015-01-22 00:00:00.0'
'2015-01-23 00:00:00.0'
```

or

```
 SELECT AVG(snow) FROM weather_data WHERE tday < -15
```

with result `35.6250`. As we can see, the code required to retrieve data for a particular purpose is reduced compared to the basic approach described above. Simple processing logic can be formulated directly in the request. But more importantly, syntactic integration abstracts from the heterogeneity of data access interface and technology as well as data encoding and formats, typically encountered when we require data from multiple distributed sources.

With syntactic integration, we have data and metadata of our building information dataset in a database. Rather than manually joining the codes for types of heating and fuel of buildings, we can formulate retrieval tasks that delegate such joins to

the database management system. The following query is an example:

```
SELECT COUNT(*)
FROM building b, type_of_heating h, fuel f
WHERE b.type_of_heating_id=h.id
AND b.fuel_id=f.id
AND h.label='central water heating'
AND f.label='district heating';
```

The query returns the number of buildings with heating type *central water heating* and fuel *district heating*. The answer is `1462`. We can also formulate the request of this number to be as percentage of the total number of buildings in a single query

```
SELECT COUNT(*)/T.total*100 AS percent
FROM building b, type_of_heating h, fuel f,
(SELECT count(*) AS total FROM building) AS T
WHERE b.type_of_heating_id=h.id
AND b.fuel_id=f.id
AND f.label='district heating'
AND h.label='central water heating';
```

with result `3.6303`. Notably, we do not have to resolve the codes for fuel and heating type manually in order to interact with the building data. Furthermore, thanks to the integrated technology and uniform query interface, we can formulate retrieval tasks using a declarative query language that frees us from having to implement program code to answer such questions.

So far we have only touched on the integration of building information *data and metadata*. Given that we also have weather data, it is of course interesting to integrate building information and weather data.

First, let's explore some of the spatial querying features provided in modern database management systems. The following query returns the buildings with fuel *wood* that are located in a particular region of interest (delimited by a polygon) that overlaps with the municipality of Iisalmi. The result of the query is `7` buildings.

```
SELECT COUNT(*)
FROM building b, fuel f
WHERE b.fuel_id=f.id
AND f.label = 'wood'
AND MBRContains(GeomFromText('Polygon(
(
  7044000 511000,
  7044000 512000,
  7043000 512000,
  7043000 511000,
  7044000 511000
))'), location);
```

Now let's integrate the building information with the weather data and ask if buildings that utilize *wood* as fuel in our region of interest have been exposed to temperatures lower than -20 degrees Celsius. The following query returns their location

```
SET @poly = 'Polygon(
(
  7044000 511000,
  7044000 512000,
  7043000 512000,
  7043000 511000,
  7044000 511000
))';

SELECT astext(b.location) as location
FROM building b, fuel f, weather_data w
WHERE b.fuel_id=f.id
AND f.label = 'wood'
AND w.tday < -20
AND MBRContains(GeomFromText(@poly), b.location)
AND MBRContains(GeomFromText(@poly), w.location);
```

with result

```
+------------------------+
| location               |
+------------------------+
| POINT(7043390 511850)  |
| POINT(7043377 511713)  |
| POINT(7043640 511560)  |
| POINT(7043240 511920)  |
| POINT(7043110 511340)  |
| POINT(7043131 511241)  |
| POINT(7043977 511074)  |
+------------------------+
7 rows in set (0.00 sec)
```

The examples should underscore how database management systems provide a flexible interface to formulate complex data retrieval tasks on integrated data originally obtained from heterogeneous sources. Such retrieval tasks can include spatial features. The database acts as an integrative system, intermediate to the heterogeneous sources and the applications and systems built on top of the database. Naturally, it depends on the goals whether or not it is worth investing the required time and resources to build such an integrative system. However, the more applications and systems are developed on top of heterogeneous data sources the more it may arguably make sense to build a suitable integrative system.

The Semantic Approach

With the syntactic approach we achieved to abstract from heterogeneous data access, format, encoding. However, the data are not expressive. Often we only know a value and perhaps its type. For instance, in the example above we know that `2015-01-05 00:00:00.0` is of type `TIMESTAMP`. For the query requesting the average snow level during days with temperature below **-15** we obtain the value `35.6250`. The system is not explicit about anything else about this value.

Specifically, what is `35.6250`**?** Implicitly, we know it is the average snow level matching the query. Implicitly, we can also assume that the value is in `cm`. The problem is that we human agents can draw these conclusions. Software cannot, unless programmers include it explicitly in program code or data self-describes its semantics. Otherwise data semantics are unavailable to computers systems.

With the semantic approach data semantics are available and machine interpretable. For instance, the value `35.6250` can be described as a `QuantityValue` having `35.6250` as its numeric value and `cm` as its unit.

Quantities and units are ubiquitous concepts. It is thus plausible that people have designed ontologies for these concepts. In fact, there exist several. One example is the suite of Quantities, Units, Dimensions and Data Types Ontologies (QUDT). QUDT includes a concept `QuantityValue` with relations `unit` and `numericValue` to a `Unit` and a `double` numeric value, respectively. There are further relations of interest, but this basic pattern is sufficient to make the point here. Rather than to merely be given the value `35.6250` as in the syntactic integration approach, the semantic approach annotates the value with machine processable semantic annotations, as follows

```
QuantityValue(qv)
numericValue(qv, "35.6250"^^xsd:double)
unit(qv, u)
Unit(u)
```

These statements go beyond a mere numeric value, the result of our query in the syntactic approach. They also make explicit the type of the numeric value (`xsd:double`) and specify that the numeric value is the representation of a particular kind of value, namely a `QuantityValue`. Interesting is also the explit relation between the quantity value (`qv`) and the unit (`u`). Note that the identifier `u` is explicitly typed: systems know it is of kind `Unit`.

It becomes even more interesting if we replace `u` with an actual unit, specifically `cm`. Not surprisingly, QUDT also includes identifiers for frequently used units, in particular also for `cm`. The identifier for this unit is `http://www.qudt.org/qudt/owl/1.0.0/unit/Instances.html#Centimeter` or, accordingly with prefix, `unit:Centimeter`. We can thus update our semantic annotations as follows

```
QuantityValue(qv)
numericValue(qv, "35.6250"^^xsd:double)
unit(qv, unit:Centimeter)
```

Note that we dropped the statement `Unit(unit:Centimeter)`. This is on purpose because it is redundant. QUDT already specifies this for us. We simply inherit this additional statement from QUDT. QUDT states further interesting relations about the unit `unit:Centimeter`. Among other things, it states that its `abbrevation` and `symbol` is `cm` and that it is of `quantityKind Length`. These statements are also inherited from QUDT: We obtain them for free, thanks to ontology engineering efforts made by the community.

Implementing the Semantic Approach

The technologies required to implement the semantic approach are drastically different from the technologies discussed above for the syntactic approach. In principle, one can of course map semantic statements to a relational database management system. Indeed, some systems do precisely this. However, most systems depart from the relational data model and adopt RDF (graph) databases, such as Stardog. The languages used for the representation of data and schemas, as well as query languages used to retrieve data, are different, too. Furthermore, support for RDF, OWL, and SPARQL in computing environment such as Matlab and R is arguably experimental and at early stages.

As a generic recipe, one first needs to create an ontology ("schema") for the data integrated from heterogeneous data sources. The ontology should provide an integrated description for the relevant terms (concepts and relations). The terms should at least be given identifiers, which in RDF are URIs (e.g. the one for the centimeter unit). Terms should also be organized into hierarchies. Good ontologies additionally make the semantics of terms explicit. For instance, the term "Mother" can be described as "Female having at least one Child". Existing terms are thus utilized to define new terms.

Following the ETL approach, we then need to extract required data from our sources and transform the data so that it meets our ontology and languages for the representation of the transformed data (RDF and OWL). With this, we can import the data into an RDF database and use a so-called SPARQL endpoint to serve RDF data for retrieval. In principle this is very similar to what we did for the syntactic approach.

Extract-Transform-Load (ETL)

The example here is limited to our weather data and for the sake of brevity we merely transform the date and air temperature. We utilize the OWL-Time ontology to represent the date and QUDT to represent air temperature quantity values. As an example, consider the following values for date and temperature:

```
01.01.2015 00:00
2.7
```

OWL-Time distinguishes instants and intervals. A day is arguably an interval but for the sake of simplicity it is considered to be an instant in this example. The data above for date and temperature is transformed to the following statements

```
time:Instant(i)
time:inXSDDateTime(i, "2015-01-01T00:00"^^xsd:dateTime)
qudt:QuantityValue(qv)
qudt:numericValue(qv, "2.7"^^xsd:double)
qudt:unit(qv, unit:DegreeCelsius)
```

In addition to semantically describing the two values, we also need to describe the pair. The pair `(01.01.2015 00:00, 2.7)` can be understood as a record (row or observation) of a dataset. Just as quantities and units, datasets are also ubiquitous and there exist ontologies for the concept of dataset and dataset observation, too. One example is the RDF Data Cube Vocabulary (QB). Following QB, the pair is a dataset observation, element of a dataset. The required statements can be summarized as follows

```
qb:DataSet(d)
qb:Observation(o)
qb:dataSet(o, d)
:date(o, i)
:airTemperature(o, qv)
```

We define a dataset `d` and observation `o` that relates to `d`. Moreover, we have two properties for `date` and `airTemperature` which relate the observation to the instant and quantity value, respectively.

The transformation here should of course be done programmatically. Due to mere experimental support for RDF in computing environments such as Matlab or R, it is arguably better to use a programming language for which there exist more advanced libraries. Java is a popular example.

Having the transformed data in RDF we can now import it into any database that supports RDF. For the purpose here, we use Apache Jena Fuseki but any other RDF database will do. Fuseki allows us to load RDF data into an in-memory database and acts as a SPARQL endpoint. After loading, we can query the semantically integrated data. The following SPARQL query retrieves the original weather data

```
PREFIX qb: <http://purl.org/linked-data/cube#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX time: <http://www.w3.org/2006/time#>
PREFIX qudt: <http://qudt.org/schema/qudt#>
PREFIX unit: <http://qudt.org/vocab/unit#>
PREFIX : <http://envi.uef.fi/flexe#>

SELECT ?date ?airTemperature ?unit
WHERE {
  [
```

```
    rdf:type qb:Observation ;
    :date [ time:inXSDDateTime ?date ] ;
    :airTemperature [
      qudt:numericValue ?airTemperature ;
      qudt:unit ?unit
    ] ;
  ]
}
ORDER BY ASC(?date)
```

which results in the table (first two rows are shown)

```
2015-01-01T00:00:00.000+02:00    2.7    unit:DegreeCelsius
2015-01-02T00:00:00.000+02:00    0.8    unit:DegreeCelsius
```

This may not look spectacular, given that our relational database system produced a similar result to a comparable query. Recall however that, in addition to having the unit of quantity values explicitly represented (here `unit:DegreeCelsius`), semantic data integration really has more explicit semantic information about the data. For instance, with the following query

```
SELECT DISTINCT ?dateType ?airTemperatureType
WHERE {
  [
    rdf:type qb:Observation ;
    :date [ rdf:type ?dateType ] ;
    :airTemperature [ rdf:type ?airTemperatureType ] ;
  ]
}
```

we can obtain the types (class) of the individuals describing the values of `date` and `airTemperature` component properties of our dataset observations. These are not merely datatypes for values, such as `double`, of a limited set of datatypes provided by a language. These types, or more accurately classes, are designed to meet the descriptive requirements of a particular domain: they form specialized vocabularies. Furthemore, their semantics can be described formally. As such, vocabulary semantics are machine processable and interpretable. Thanks to explicit semantics, machines can better assess whether or not exchanged data mean the same thing, and can thus be integrated, or if further steps are required to first resolve semantic conflicts.