Project no.:	**610658**
Project full title:	**eWALL for Active Long Living**
Project Acronym:	**eWALL**
Deliverable no.:	**D3.3.2**
Title of the deliverable:	**eWALL configurable metadata streams**

Contractual Date of Delivery to the CEC:	**30.04.2015**
Actual Date of Delivery to the CEC:	**30.04.2015**
Lead contractor for deliverable:	**AIT (P07)**
Author(s):	**Aristodemos Pnevmatikakis (AIT)**
Participants(s):	**AIT (P07), UKIM (P09), AAU (P01), UPB (P08), ENT (P03), TUS (P11)**
Contributing work package:	**WP3**
Nature:	**P**
Version:	**1.0**
Total number of pages:	**45**
Start date of project:	**01.11.2013**
Duration:	**36 months – 31.10.2016**

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 610658

Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Abstract:

This deliverable is a report on the JSON representation of the metadata generated by the sensing environment (either physical or simulated) of the eWALL caring home and their local storage in a home database. It also details the user simulator generating the simulated metadata.

Keyword list: Metadata, streaming metadata, JSON, databases, SQL, no-SQL

Document History

Version	Date	Author (Unit)	Description
0.1	23/03/15	A. Pnevmatikakis (AIT)	Proposed modifications to the previous version
0.2	03/04/15	A. Pnevmatikakis (AIT)	Section 3.2 and chapter 5
0.3	15/04/15	A. Pnevmatikakis (AIT)	Final edits, ready for internal review
1.0	29/04/15	S. Kyriazakos (CTIF/AAU)	Final review and approval

Table of Contents

1 Executive Summary

This deliverable describes the metadata streams generated by the sensing environment of the eWALL caring homes, their local storage in a home database system and the means of generating simulated metadata for testing the eWALL higher-level systems. The possible metadata are summarised in a dictionary, while those that are already generated by the equivalent perceptual components are formally described using JSON messages. The metadata are stored in CouchDB databases. The selection of CouchDB is backed by an in-depth analysis of three major types of database classes, namely SQL, non-SQL, and RDF. Finally, a simulator generates metadata of a fully-equipped home, to facilitate development of the higher-level components of eWALL.

1.1 Targeted audience

This report is addressed to the members of the consortium wishing to build higher-level components that need to know the context of the user in the caring home, since it explains the available metadata.

It is also addressed to the Commission Services, to facilitate the evaluation of the choices regarding the metadata formatting and in-home storage.

Finally, it is addressed to the general public wishing to be informed of what can be measured in an eWALL caring home.

2 Introduction

The sensing environment of any eWALL caring home generates metadata describing the environment and the user. The different categories of metadata have been identified and catalogued in a metadata dictionary for eWALL. Some of them are also implemented, meaning that there are perceptual components that operate on sensors' outputs and generate the metadata. These are also formally described using JSON messages, originating from the perceptual components and subsequently stored within the eWALL system.

Storage in the context of eWALL is implemented in two places – one local in the home and another in the eWALL cloud. This document analyses potential databases for in-home storage, taking into account the requirements and constraints of the project. The reason behind the local storage is twofold:

- To provide a temporary buffer in situations where home - cloud connection fails.
- To provide the mechanism for certain time-critical cervices or notifications about life-threatening situations to be handled locally by the caring home.

All the higher-level components of eWALL manipulate, reason about and present information mainly based on the metadata generated by the home sensing environment. To take advantage of the functionality of all these components, the system needs to have a wealth of metadata both in timespan, number of users and type and number of sensors. This requires fully functional deployments of the sensing infrastructure, which is not yet possible. The needed metadata are provided by a simulator.

The deliverable is structured as follows:

- Chapter 3 discusses the eWALL configurable metadata streams, resulting to their formal descriptions using JSON messages.
- Chapter 4 summarises the reasons behind the choice of CouchDB for storing the metadata at home.
- Chapter 5 details the user simulator, the mechanism for generating artificial but plausible metadata from a fully equipped caring home.
- Chapter 6 concludes the deliverable.

2.1 Comparison to the previous version

This is the second version of the deliverable. The text in this version is not incremental over the first version. Instead a self-contained document is built, reusing most of the information from the first version and adding new material. To facilitate reading the deliverable in an incremental fashion, the following list outlines the changes between the first and the current (second) versions:

- Chapter 3 has been revised with the latest JSON messages including the new environmental sensing message, the unchanged wearable message and the new 2D face tracking and bed sensing messages.
- Only the database selection material (chapter 4 in D3.3.1) remains unchanged in this new version. Due to the length of the comparison tables offering an in-depth analysis of the options for the storage of the metadata at home and thus supporting our CouchDB selection, they have been moved into Appendix A. Chapter 4 now contains only the conclusions summarising the tables.

- Chapter 5 is altogether new.
- The conclusions are updated and can now be found in chapter 6.

3 eWALL configurable metadata streams

All the eWALL metadata are generated by sensing the users and their environments in the caring homes. To describe this wealth of information, we maintain a metadata dictionary, shown in Section 3.1. As we build perceptual components that interface to the necessary sensors and produce the metadata, we structure formal JSON messages that convey this information to the home database. The messages currently in use are discussed in Section 3.2.

3.1 Metadata dictionary

The metadata dictionary at the time of writing is given in .the following table. The metadata are structured in RDF triples: The subject, predicate and object are given in the first three columns of the table. The implementation status is given in the final column.

Subject	Predicate	Object	Status
person	trackID	Integer	Implemented Metadata is provided to the database
	x, y, width, height	Integer (pixel coordinates of the face bounding box for 2D face tracking)	
	x, y, z	Integer (cm in space for 3D person tracking)	
	positionConf	Double	
	gender	Integer	
	genderConf	Double	
	age	Integer	
	ageConf	Double	
	emotion	Integer	
	emotionConf	Double	
	timestamp	Time string	
activity	ISA	Double	Implemented Metadata is provided to the database
	IMA	Double	
	steps	Integer	
	physicalActivity	String	
	fall	Boolean	
environment	movement	Logical	Implemented Metadata is provided to the database.
	temperature	Double	
	humidity	Double	
	illuminance	Double	
	LPG	Integer	
	NG	Integer	
	CO	Integer	
	door_open	Boolean	
	timestamp	Time string	
sound	micID	Integer	Not implemented

	level	Double	
	soundType	String (voice, snore, vacuum cleaner, etc.)	
	period	Double	
	speakerID	Integer	
	angleOfArrival	Double	
	timestamp	Time string	
bed	pressure	Boolean	Implemented Metadata is provided to the database
	ISA	Double	
	IMA	Double	
	timestamp	Time string	
vitals	SPO	Integer	Implemented Metadata is provided to the database
	BPM	Integer	
	timestamp	Time string	
voiceComm	carrier	Array (phone, skype or similar voice service)	Not implemented
	startTime	Time string	
	endTime	Time string	
	otherParty	String (phone number) or array of string (Skype IDs)	
	incoming	Boolean	
socialStatus	recipient	String	Not implemented
	carrier	String (facebook, twitter, IM service)	
	message	String	
	timestamp	Time string	
entertainment	startTime	Time string	Not implemented
	endTime	Time string	
	ID	String (game or media title or ID)	
	gameScore	Integer	
domotics	socketID	Integer	Not implemented
	powerConsumption	Double	
	tapID	Integer	
	waterFlow	Integer	
	timestamp	Time string	

3.2 Metadata JSON specification

This section contains the detailed specification in JSON format and examples for all the implemented metadata.

3.2.1 Environmental sensing

The environmental sensing message collects information from a single room, grouping together readings from environmental and domotics sensors. The elements of the JSON message are as follows:

- movement: The boolean value from the PIR sensor
- illuminance: The double value of light intensity in lux
- temperature: The double value of the temperature in degrees Celcius
- humidity: The double value of the relative humidity as a percentage
- NG, LPG, CO: An integer value between 0 to 10 indicating the level of the three gases. The levels 1-10 are approximately logarithmic in terms of gas concentration. Level 0 indicates a value below the lower measurable concentration for every sensor.
- door_open: The boolean value from the door dry contact. True corresponds to the door being open
- timestamp: The string with the time the message is sent, including time zone information

Such a message is sent asynchronously, every time there is a significant change in the metadata.

Open issues:

How do we handle more dry contacts in the room? With more elements, an altogether different domotics JSON message, or an array of elements (comprising name of contact and boolean status)? The array can be the implementation in the independent message as well.

An example of the message is as follows:

```
{
    "_id" : "2014-09-01T00:00:00.000Z",
    "_rev" : "1-b34306f2f0344672d653f5b5c7df711c",
    "movement" : false,
    "illuminance" : 2.0464407112347436,
    "temperature" : 19.40782989444393,
    "humidity" : 52.07199253060395,
    "NG" : 1,
    "CO" : 2,
    "LPG" : 1,
    "door_open" : true,
    "timestamp" : "2014-09-01T00:00:00.000Z"
}
```

3.2.2 Wearable sensing

Currently the only wearable sensor is the accelerometer, yielding only activity-related metadata, grouped in an activity element. The elements of the JSON message are as follows:

- activity: Grouping element
- IMA, ISA: The double values of the integral modulus of acceleration and integral sum of acceleration, both computed over a 10 seconds interval
- steps: The integer (long actually) number of steps since the installation of the system
- physicalActivity: The string describing the activity type (RESTING, WALKING, RUNNING, EXERCISING, NODATA)
- fall: The boolean result of analysing the accelerometer data for possible fall.
- timestamp: The string with the time the message is sent, including time zone information

Such a message is sent synchronously, every 10 seconds.

Open issues:

- Do we want to flatten the activity element?
- Do we need ISA? Some months ago the decision was to keep both IMA and ISA

- Do we want the steps to be counted since installation, since the beginning of the day, or only in the 10 second interval? The latter is easier since it does not require a mechanism to read the steps from the CouchDB in case of restart.

An example of the message is as follows:

```
{
    "_id": "2014-09-01T12:29:10.000Z",
    "_rev": "1-156f011ef2ecd6643a089eb61bc0b24e",
    "activity": {
        "IMA": 0.1112141600593139,
        "ISA": 0.1112141600593139,
        "steps": 5473,
        "physicalActivity": "WALKING"
    },
    "fall": false,
    "timestamp": "2014-09-01T12:29:10.000Z"
}
```

3.2.3 Visual sensing

This message is about the face analytics from the 2D face tracker. The elements of the JSON message are as follows:

- people: An array element holding the metadata from each face
- trackID: An integer uniquely identifying each face track
- x, y, width, height: The pixel, i.e. integer, values for the centre point (x,y) and the width, height of the face
- gender: The string representing the gender of the person (MALE, FEMALE)
- age: The integer value of the age of the person
- emotion: The string representing the emotion of the person (NEUTRAL, ANGER, CONTEMPT, DISGUST, FEAR, HAPPINESS, SADNESS, SURPRISE)
- positionConf, genderConf, ageConf, emotionConf: Doubles between 0 and 1 representing the confidence of the face position and size, gender, age and emotion estimates. If the respective algorithms do not return confidence, use unity.
- timestamp: The string with the time the message is sent, including time zone information

Such a message is sent asynchronously, every time there is a significant change in the metadata.

Open issues:

How will we handle 3D body tracking? In a different message? In the same, associating bodies with faces?

An example of the message is as follows:

```
{
    "_id": "2014-09-01T12:29:10.000Z",
    "_rev": "1-156f011ef2ecd6643a089eb61bc0b24e",
    "people": [
        {
            "trackID": 0,
            "x": 400,
            "y": 340,
            "width": 40,
            "height": 40,
            "positionConf": 0.9,
```

```
            "gender": "MALE",
            "genderConf": 0.9,
            "age": 70,
            "ageConf": 0.8,
            "emotion": "NEUTRAL",
            "emotionConf": 0.7
        },
        {
            "trackID": 2,
            "x": 230,
            "y": 310,
            "width": 43,
            "height": 42,
            "positionConf": 0.9,
            "gender": "MALE",
            "genderConf": 0.9,
            "age": 68,
            "ageConf": 0.7,
            "emotion": "NEUTRAL",
            "emotionConf": 0.7
        }
    ],
   "timestamp": "2014-09-01T12:29:10.000Z"
}
```


3.2.4 Resting furniture sensing

This message is about sensing at any furniture where the user can rest, with the bed and sofa being prime examples. The sensors are an accelerometer and a pressure one. The elements of the JSON message are as follows:

- pressure: The boolean value indicating if someone is sitting or lying on the furniture
- IMA: The double value of the integral modulus of acceleration, computed over a 10 seconds interval
- timestamp: The string with the time the message is sent, including time zone information

Such a message is sent synchronously, every 10 seconds.

Open issues:

The measurement period might be excessive, since in this case the IMA value is not for activity intensity estimation, but to know if the user is sleeping. What could be a more suitable measurement period?

An example of the message is as follows:


```
{
    "_id": "2014-09-01T00:00:10.000Z",
    "_rev": "1-80ede81818d8a45211212921ae6749a7",
    "pressure": true,
    "IMA": 0.08626862285226562,
    "timestamp": "2014-09-01T00:00:10.000Z"
}
```


3.2.5 Medical sensing

This message is about the values from the medical sensors, containing the vital signs readings of the care recipient. The elements of the JSON message are as follows:

- SPO: The integer percentage of the SPO2 value
- BPM: The integer number of heart beats per minute
- timestamp: The string with the time the message is sent, including time zone information
- Such a message is sent asynchronously, every time the user takes a new measurement.

Open issues:

There are other medical sensors to be included in the future. Will they be in the same message, grouped under different container elements? Or will there be different messages, i.e. different databases?

An example of the message is as follows:

```
{
    "_id": "2014-09-01T00:00:10.000Z",
    "_rev": "1-80ede81818d8a45211212921ae6749a7",
    "SPO": 15,
    "BPM": 60,
    "timestamp": "2014-09-01T00:00:10.000Z"
}
```

4 Database analysis and selection

Based on the requirements in the project and considering all the properties and features of the databases reviewed in the previous section, a CouchDB is selected as a storage database for the in-house environment. Specifically, the easy multi-master replication is attractive regarding the context of the project. The following list represents the advantages of CouchDB:

- ***JSON & JavaScript*** – CouchDB stores and serves JSON documents, and utilizes JavaScript to manipulate them during querying/validation via HTTP. This function is primary advantage since eWall platform utilizes Java as a main implementation language;
- ***Notification mechanism*** – trough "`_changes`" API a streaming mechanism can be provided which is excellent feature for implementing notifications to eWall cloud without the need for polling which is connection demanding;
- ***Schema free*** – each document can define its own validation function which introduces flexibility;
- ***Scalability*** – CouchDB is efficient on one machine and through replication can be scaled out to many machines;
- ***Multi-master asynchronous replication*** – documents can be bi-directionally replicated to many instances and every instance can simultaneously modify all of them;
- ***Optimistic locking*** – in some scenario this feature can be very useful. CouchDB stores "_rev" (revision-version) field in every document and this way provides the optimistic locking;
- ***Replication*** - CouchDB can be driven extremely easy by "_replicator" special-db running as a separate process and replicating data between two instances.

Its internal architecture is fault-tolerant, and failures occur in a controlled environment and are dealt with gracefully. Single problems do not cascade through an entire server system but stay isolated in single requests. It was also intended for accumulating, occasionally changing data, on which pre-defined queries are to be run (such as the signals from the particular sensors).

Every database in CouchDB is a master. If a big part of our application is sync between databases that may go offline and online at any time and may require conflict resolution and merging, CouchDB can handle a lot of the low level work. It is also excellent at replicating between multiple nodes, either on-demand or continuously. Thanks to its replication abilities and RESTful API a horizontally can be handled quite easy using mature tools. (Nginx or Apache for reverse proxying, HTTP load balancers, etc.). A map/reduce functions can be developed in JavaScript to precompute queries. The results are built up incrementally on disk which means they only need to be computed once per signal. In other words, queries can be really fast because it only has to do calculations on the signal data recorded since the last time we ran the query. CouchDB trades disk space for performance or in other word the queries can be comparably fast while conserving disk space.

5 User simulator

We introduce the ‘shadow users’ to facilitate the evolution of the higher-level components without having to wait for the perceptual ones to reach a level of high maturity and the installations to be many and deployed for a long time. These are repositories of caring home metadata that are simulated rather than stemming from actual sensors.

5.1 Introduction

The higher-level components of eWALL utilise metadata from the caring homes in order to reason about the context of the care recipients and to provide personalised services to them. To do so, these components need a large volume of metadata, in terms of:

- Timespan covered
- Diversity of sensors and processing algorithms
- Diversity of users

Currently the caring home sensing functionality is still evolving. Moreover, its deployments are few and partial in terms of what each site supports. Also, the timespan the deployments are functional is sparse: the sensing environments are relatively new and they are not always operational. Finally, the environments are still in our labs; the people sensed are not care recipients, they are researchers working on the systems. As a result, the needed volume of metadata is not available from the actual sensor deployments.

A shadow user is a simulation, driven by models for the home, the weather, the sensing environment and the behaviour of the user in it. The home model is fixed at initialisation time, the weather is obtained from an external service, and the sensing environment model comprises conditional probabilities of the measurements given the user state. Finally, the user behaviour model is described by state transition probabilities that vary based on four user driving forces, as discussed at the user state section.

The simulator returns all the metadata expected from an actual home, obtained by processing the measurements of the sensors. The metadata are organised into the same CouchDB databases, in the same JSON format found in the actual deployments. In addition, it returns the user state that resulted to these metadata. The goal of the higher-level components is to infer this state from the given metadata.

5.2 Simulator code structure and installation and execution

The simulator code is a Maven project that can be found at the WP3 project GitLab repository:

http://serv2.radio.pub.ro/gitlab/wp3group/wp3project/tree/master/UserSim

There are two files in the resources directory:

- *init.json*: Contains all initialisation information for the software and the models used.
- *logback.xml*: Contains the initialisation information for the data logger used.

The code itself is organised in the following packages:

- eu.ewall.sensing.db: Hosts the *CouchDbConnection* class.
- eu.ewall.sensing.simulator: Hosts the *StartingPoint* class with the *main()* method.

- eu.ewall.sensing.simulator.init: Hosts the *InitInfo*, *Posture*, *Room*, *State* and *StateTransition* classes, each handling reading a part of the *init.json* file.
- eu.ewall.sensing.simulator.weather: Hosts the classes for importing weather information from the weather service. The *WeatherHistoryResponse* and *WeatherNowResponse* classes read the weather history or the current weather status respectively. The *WeatherMain*, *WeatherList*, *WeatherClouds* and *WeatherWeather* classes handle the respective parts of the JSON response of the weather service.
- eu.ewall.sensing.simulator.models: Hosts the *BodySensing*, *RoomEnvironment*, *User* and *WeatherModel* classes that implement the simulator functionality.

To use the simulator one needs to build the Maven project and install CouchDB. The CouchDB installation needs to have a user set up, who will be the owner of the databases containing the metadata.

The credentials of this user and the port the CouchDB listens to need to be known to the simulator. For this the *init.json* file contains the "couchdburl" property that concatenates all this information into a URL:

"couchdburl" : "http://user:pa$$word@localhost:5984"

The username in this particular example is “user”, the password is “pa$$word” and the port is 5984 (the default at the CouchDB installation). For hints on how setting up the user in the CouchDB (with admin role), please refer to:

http://www.staticshin.com/programming/easy-user-accounts-management-with-couchdb/

If you decide to run the simulator again, go to /home/ewall and execute:

java -jar simulator-0.0.5-jar-with-dependencies.jar

Note that the needs Java 8. Also note that by running, the databases will be deleted.

Finally, if you are running the simulator at the home-dev environment of the eWALL cloud and want to have a web-based interface to the CouchDB of home-dev, then you should setup a SSH tunnel. At the connection settings you have for the home-dev, select a port and assign it to localhost:5984 from the Putty settings. Add it and get something like:

L10010 localhost:5984

Then at your browser go to:

http://localhost:10010/_utils/index.html

5.3 Databases in CouchDB

The names of the databases for the different metadata categories are given in the init.json file, using the properties "userdbname" for the user state, "bodydbname" for the metadata from the wearable sensors and "dbname" in every “rooms” instance for the metadata from the environmental sensors in each room.

For the default *init.json* file there are:

- The four *_environment databases have the environmental sensors.
- The wearable database has the acceleration IMA and steps.

- The userstate is the database that normally does not exist: In contains the state of the user that generated the measurements, i.e. what we will try to predict given the measurements in all the other databases.

After running the simulator for the default duration (or looking at the results in the home-dev environment, the status of the CouchDB is depicted in Figure 5.1.

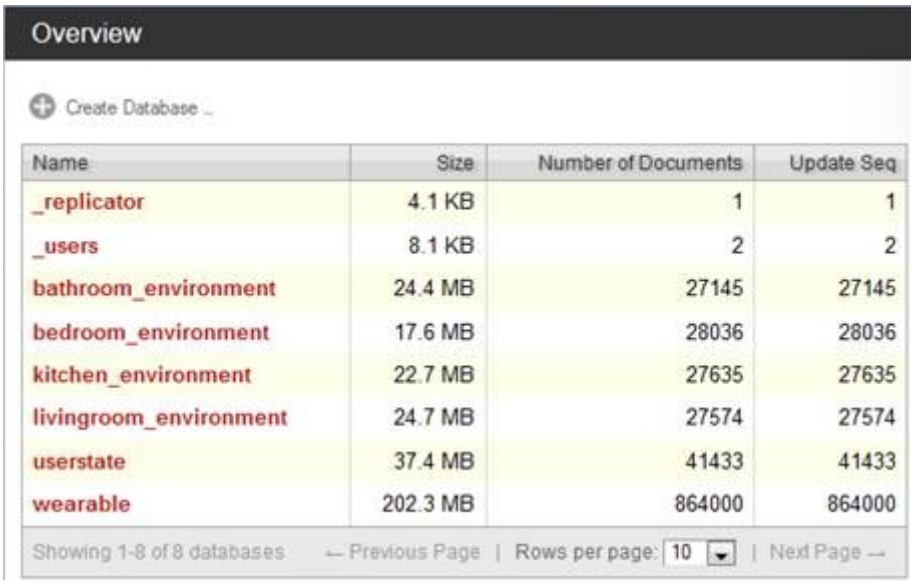

Figure 5.1: Status of the CouchDB with the six default databases holding the metadata of 100 days of simulation.

5.4 StartingPoint

The *StartingPoint* class hosts the *main()* method. The following steps are followed:

- Initialisation.
- Run the simulator in increments of 10 seconds.
  - Every hour the weather status is obtained.
  - The rooms' environment is simulated based on the weather and on the user state.
  - The body measurements are obtained.
  - The user life is advanced by 10 seconds.

5.4.1 Initialisation

At initialisation, first the *weather* object of the *WeatherModel* class is constructed. This will hold the weather data.

Then the *init.json* file is read into the *info* object of class *InitInfo*, to get the simulator parameters, the supported body postures (*Posture* class), the home structure in rooms (*Room* class), the possible user states (*State* class) and within the states, the state transition parameters (*StateTransition* class).

The *rooms*, an *ArrayList* of objects of the *RoomEnvironment* class is initialised next. Each hold the environmental measurements of a room and the name of the database to write them into as state variables.

The timestamp is set at the starting time of the simulation and the *user* object of the *User* class is initialised. It holds the various aspects of the user state and the name of the relevant database as state variables.

Finally, the *bodySensors* object of the *BodySensing* class is initialised to hold the measurements of the wearables and the name of the relevant database as member variables.

Note that during initialisation, as part of the *rooms, user* and *bodySensors* objects' constructors, the corresponding databases are erased from CouchDB if they exist, and are created anew.

5.4.2 Simulation steps

After initialisation the simulator is run for the timespan defined in the "starttime" and "duration" properties of the *init.json* file using a for loop in 10 second increments.

First the simulator decides if the simulated time is in the past, or the future. In the latter case, it waits of actual time to catch up. This means that any timespan belonging to the past is simulated at the maximum possible speed, while when the simulator catches up with the actual time, then it progresses in real time.

If the time is a multiple of an hour, then the weather data are read into the *weather* object. The weather service used sometimes fails. In this case the failure is logged, and getting the weather is retried at the next simulation step.

Next the environment in each of the rooms in the *rooms ArrayList* is set in two steps:

- The effect of the weather is simulated utilising its *setEnvironment()* method with the *weather* object and the insulation of that room as parameters. The latter is obtained from the *info* object by getting the rooms, accessing the relevant list member and getting its insulation, in total calling *info.getRooms().get(i).getInsulation()*.
- The effect of the user action is simulated utilising the *setEnvironment()* method of the *user* object with the *rooms ArrayList* and states *Map<Integer,State>*. The latter are obtained from the *info* object calling its *getStates()* method.

After setting the environmental conditions, the sensor measurements from all the rooms are written to their respective databases by invoking the *put2db()* method.

Then the measurements of the body are updated by invoking the *measureBody()* method of the *user* object with the *bodySensors* object as argument. The method *put2db()* of the *bodySensors* object writes the measurements in the database.

Finally time is advanced by 10 seconds and the user state is updated by invoking the *stepLife()* method of the *user* object with the time and the map of the states as arguments. The method *put2db()* writes the state in the database.

5.5 Initialising the home: The Room class

The home model contains the rooms their names, their insulation factors and their doors, indicating which room transitions are possible. The outside space is also modelled as a room, to allow the user exiting the home. Hence a home with four rooms is modelled by 5 possible spaces where activities can happen.

As an example consider a home where the access from outside is to the living room. The kitchen, the bedroom and the bathroom are accessed from the living room. The bathroom is also accessed

from the bedroom. Actually this is done as there is a corridor connecting the living room, the bathroom and the bedroom, but we chose not to model this as there would be no sensors there in an actual deployment. The presence of the user in any of these rooms is part of the user state. The possible user states of our example are shown in Figure 5.2. The four rooms and the outside space are shown in thick borders and light blue fill. The possible room transitions described above are also shown with arrows.

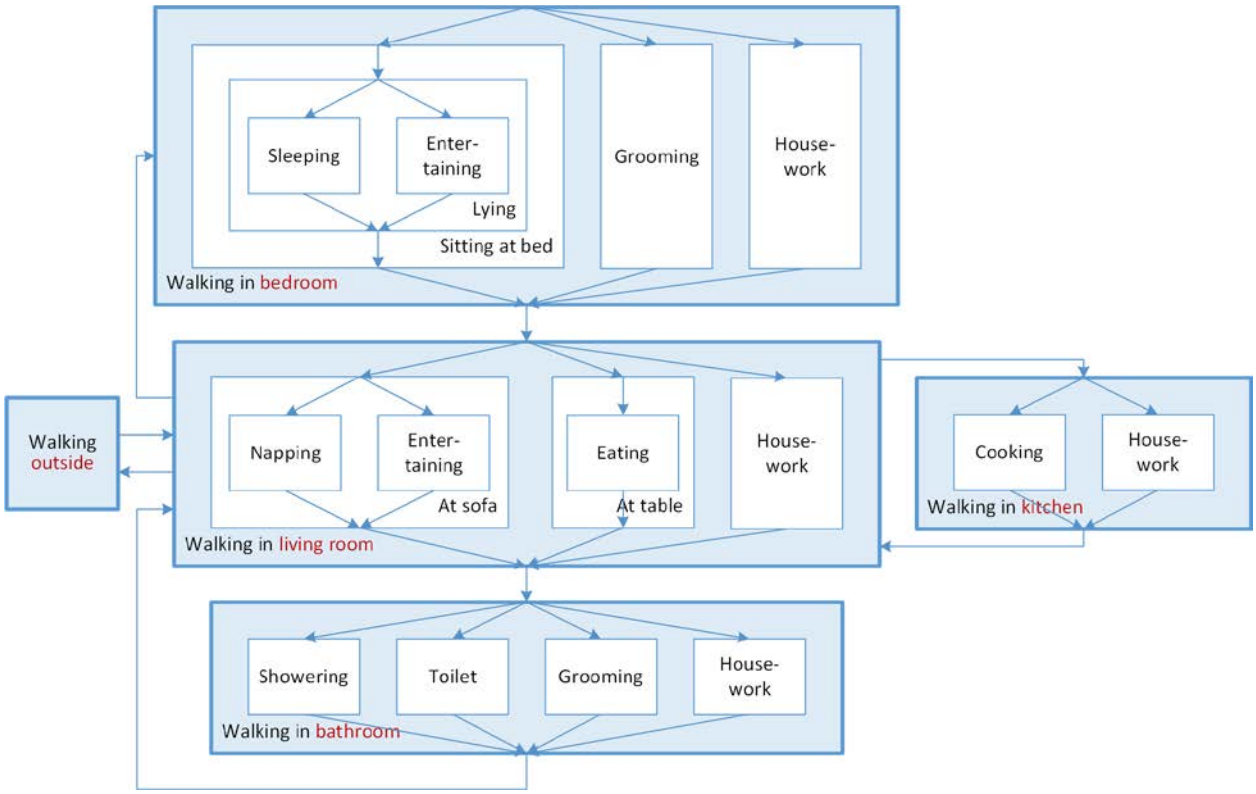

Figure 5.2: States of the care recipient. At a first level (light blue fill) the different rooms and the transitions between them are given. At a higher detail, all states are shown, as described in the *states* section of the *init.json* file.

Since the room presence is only part of the state of the user, our actual state transition is more complicated.

The home model is fixed at initialisation of the simulator, utilising the relevant section of the *init.json* file. The four rooms of the above example are described in the following lines of the initialisation file:

```
"rooms": [{
            "id": 0,
            "name": "bedroom",
            "insulation": 0.95,
            "dbname": "bedroom_environment"
},{
            "id": 1,
            "name": "livingroom",
            "insulation": 0.9,
            "dbname": "livingroom_environment"
},{
            "id": 2,
```

```
            "name": "bathroom",
            "insulation": 0.85,
            "dbname": "bathroom_environment"
},{
            "id": 3,
            "name": "kitchen",
            "insulation": 0.9,
            "dbname": "kitchen_environment"
}]
```

The “id” and the “name” properties describe the room. The “insulation” property controls how the outside weather affects the room environment, as described at the environmental sensing section. The "dbname" property holds the name of the database holding the environmental metadata for that room.

All these are imported from the *init.json* file using the *Room* class.

5.6 User model: The User class

The user model comprises the state, the state transition probabilities, the driving forces and the time since the latest state change. They are all stored in the *User* class.

The user is characterised by the following parameters:

- Activity: sleeping, grooming, showering, eating, doing housework, being entertained, having visit, going out
- Room presence: where in the home he/she is. In part dictated by the activity
- Body posture: walking, standing (moving a little bit around the home), sitting, lying. In part dictated by the activity

The three components are connected, since the one influences the other: The activity fully determines the body posture and partly the room presence.

The driving forces are

- Tiredness: As it increases towards 1, the user in increasing compelled to sleep, either in the bed or the sofa. Some actions are not even attempted if tiredness is above a level.
- Hunger: As it increases towards 1, the user in increasing compelled to eat.
- Boredom: As it increases towards 1, the user in increasing compelled to do have entertainment of go for a walk.
- Toilet need: As it increases towards 1, the user in increasing compelled to visit the toilet.

The user state is stored in the *state* class variable. The possible states and the transitions between them are shown in Figure 2 and are described in the states section of the *init.json* file.

An example of the definition of a state in the *init.json* file is as follows:

```
"110": {
            "id": 110,
            "name": "In bedroom, at bed",
            "room": 0,
            "body": 2,
            "hunger": 7e-4,
            "tireness": 1e-4,
            "boredom": 7e-4,
            "transitions": [{
                        "toState": 100,
```

```
                    "_name": "In bedroom, walking",
                    "probability": 0.2,
                    "tirenessfactor": -0.1,
                    "tirenessless": 0.5,
                    "tirenessmore": 0
          },{
                    "toState": 110,
                    "_name": "In bedroom, at bed",
                    "probability": 0.5,
                    "tirenessfactor": -0.4,
                    "tirenessless": 1,
                    "tirenessmore": 0
          },{
                    "toState": 111,
                    "_name": "At bed, lying",
                    "probability": 0.3,
                    "tirenessfactor": 0.5,
                    "tirenessless": 1,
                    "tirenessmore": 0
          }]
},
```

The states are described as a map of an integer (the state identifier) and an object of the *State* class. The reason for this is not to have the state identifiers as increasing indices in an array of states. The identifier numbers are 3-digit codes. The first digit denotes the room, so all 1xx identifiers correspond to states in the bedroom. The second number is a second-level grouping. E.g. all 11x identifiers correspond to states in bed.

The properties in the *init.json* file are:

- name: It provides a human-readable state name, but is also used by the simulator to identify some activities that affect the environment, as explained later in this document.
- room: The room number, indexing the array of rooms in the *info.getRooms()* list of *Room* objects.
- body: The body posture number, indexing the array of postures in the *info.getPostures()* list of *Posture* objects.
- hunger, tiredness, boredom: The number to add to that particular user driving force after 10 seconds in this state, either increasing or decreasing its value. The driving forces affect the user behaviour as discussed in the state update section.
- transitions: A list of transitions from this state.

The list of state transitions also has the following properties:

- toState: The state to transition to (including the current state if no change happens).
- _name: The name of the state to jump to. It is not read by the simulator, it is there just as a comment to help reading the *init.json* file.
- probability: The base probability (in the absence of the driving forces) for the particular transition.
- tirenessfactor, tirenessless, tirenessmore: The way the tiredness driving force affects the state transition probability, as discussed in the state update section.

Note that the rest of the driving forces are currently not supported by the simulator.

5.7 Getting the weather: The WeatherModel class

We get the weather for the current simulated date from a weather service. The weather fixes the conditions outside the home, which somewhat affect the conditions in the rooms. We get:

- Temperature,
- Humidity,
- Pressure,
- Cloud coverage percentage,
- Whether it is day or not, and
- A string describing the weather condition

These, together with the time, are stored as member variables in the *WeatherModel* class. There are two methods, the *readWeatherHistory(long timestamp)* for getting historical weather data for the time instance in *timestamp* and the *readWeatherNow()* for getting the current weather situation. The weather service's history service is utilised for the past, while it switches to the current service when real time is reached.

5.8 Sensing models

These models describe the sensors and the perceptual components in the home. These are grouped per location and function, each group providing metadata in a database in the CouchDB. Currently the simulator supports:

Environmental sensing in every room

- Body acceleration
- These models are detailed in the following subsections.

5.8.1 Environmental sensing: The *RoomEnvironment* class

The environmental mote measures and reports temperature, presence of moving objects (heat source), illuminance, three gasses, humidity and door status for every room. These measurements are stored in an object of the *RoomEnvironment* class, one per room in the home.

Presence and gasses are solely connected to the user state. Temperature, humidity and illuminance are partly affected by the user state (activity) but also by the weather model and the time of day. The part dependent on the environment is considered in the following subsections. The way the user affects these measurements is considered in the next section.

The class has two methods (apart from the constructor, getters and setters):

- *put2db(String dateStr)*: Writes the environmental measurements to the database with the timestamp given in *dateStr*.
- *setEnvironment(WeatherModel weather, double insulation, double alpha)*: Sets the temperature, humidity and illuminance based on the weather in the *weather* object, the insulation and the memory factors, as explained next.

5.8.1.1 Temperature

The steady-state room temperature is modelled at an ideal temperature of $T_{ideal} = 21$ Celsius, pulled downwards or upwards by the outside temperature:

$$T_{ss} = \alpha_{ins} T_{ideal} + (1 - \alpha_{ins}) T_{out}$$

where α_{ins} is a room parameter encapsulating factors like room insulation. It is given as the "insulation" parameter of every room described in "rooms" in the *init.json* file. A value of unity renders outside conditions irrelevant, while a value of zero equates the room conditions to the outside ones.

The weather state is read into the simulator once per hour. Any resulting temperature change is not applied instantly: instead the expected room temperature is a linear combination of the previous and the steady-state one:

$$T_{room} = \alpha_{mem} T_{room} + (1 - \alpha_{mem}) T_{ss}$$

where α_{mem} is a memory factor. Since the simulator updates the room conditions once every 10 sec, a value of $\alpha_{mem} = 0.99$ will result to the expected conditions being fully applied in the room after about 17 minutes.

The measured room temperature deviates from the expected by some additive white measurement noise of standard deviation of 0.2 degrees.

5.8.1.2 Humidity

We can simulate the humidity in a similar manner. This time the ideal (comfortable) level is 55%. The measurement noise standard deviation is 1.

5.8.1.3 Illuminance

The illuminance is determined by the cloud coverage percentage and the time of day. During the night, the steady-state value ranges from 0 lux (full clouds) to 1 lux (no clouds) plus a Gaussian component or standard deviation 1 (that is not allowed to result to negative values). During the day, the steady-state value ranges from 10 lux (full clouds) to 1010 lux (no clouds) plus a Gaussian component or standard deviation 20. The measured illuminance is updated based on the steady-state one, with the usual memory factor.

5.8.1.4 Gases

Three gases are reported: CO, NG and LPG. Only the first two are simulated though. Their values are simulated in ppm, but are reported as integers between 0 and 10 in logarithmic scale. Zero corresponds to a reading at the minimum value the sensor can detect.

5.8.1.5 Door status

This is a Boolean value indicating the state of the door. For the living room, this door is the main one towards outside. Only the living room and the bathroom doors are simulated based on the activity of the user.

5.8.2 User affecting the home environment

As already discussed, the user state affects the environment in which the user lives in, and this is reflected to the measurements obtained in each room. This functionality is implemented in the *setEnvironment()* method of the *User* class.

The changes the user imposes to the home environment are as follows:

- When the user is present in a room, the probability the PIR sensor gives a positive reading increases as the user activity level increases.
- When the user is showering, then the bathroom humidity and temperature increase.
- When the user is cooking, then the kitchen temperature and humidity increase.

- When the user is not lying down in a dark room it is assumed that the lights are turned on.

The PIR of any room can be active, based on the following probabilities conditioned to the user state:

- 0.01% if the user is not present
- 1% if the user is lying
- 10% if the user is sitting
- 50% if the user is standing
- 90% is the user is walking

If the user is not lying down in a room that has illuminance lower than 100, then it is assumed that the lights are turned on, leading to an illuminance of 100 plus a Gaussian component of standard deviation 2 lux.

If the user is having a shower, then the humidity and temperature in that room increase. The increase towards the target humidity of 90% is progressive, with a memory factor of 0.9. This factor is smaller than the one used for the change towards the environmental conditions, leading to faster humidity adaptation. Similarly the temperature target in the room is set to 30 degrees. For the simulator to know that any state is associated with a shower, the state "name" property needs to include the characters "shower". In our example, the state is described as:

```
{
            "id": 410,
            "name": "In bathroom, showering",
            "room": 3,
            "body": 1,
…
}
```

If the user is cooking, then the humidity and temperature in that room increase. The increase towards the target humidity of 75% is progressive, with a memory factor of 0.9. This factor is smaller than the one used for the change towards the environmental conditions, leading to faster humidity adaptation. Similarly the temperature target in the room is set to 25 degrees. For the simulator to know that any state is associated with cooking, the state "name" property needs to include the characters "cook".

The concentration of gases increases when the user is in the room. It increases ten times faster if the user is cooking. It decreases when the user is performing houseworks in the room, as the simulator assumes the windows are then open to air the room. For the simulator to know that any state is associated with houseworks, the state "name" property needs to include the characters "houseworks".

5.8.3 Wearable sensors: The *BodySensing* class

Currently the simulator supports only an accelerometer wearable sensor. The metadata obtained from it are the integral modulus of acceleration (IMA) and number of steps. These are stored in the *BodyModel* class, that beyond the constructor, the getters and setters, only has the *put2db(String dateStr)* method to write the wearable measurements to the database with the timestamp given in *dateStr*.

The acceleration metadata depend only to the user state and more specifically the body posture. As such, they are set by the *measureBody(BodySensing bodySensors)* method of the *User* class.

The number of steps in the simulated interval of 10 seconds and the IMA values for the four body postures are given in Table 1. There $u(a,b)$ is a uniformly distributed random integer in the range of $[a,b]$ and $g(\mu,\sigma)$ is a Gaussian distributed random number with mean μ and standard deviation σ.

Table 1: Number of steps and IMA values for the given body posture in an interval of 10 seconds.

Body posture	Number of steps	IMA
Walking	$20 + u(-5,5)$	$g(2.3,0.23)$
Standing	$5 + u(-2,2)$	$g(0.6,0.06)$
Sitting	$u(0,1)$	$g(0.3,0.03)$
Lying	0	$g(0.1,0.01)$

5.8.4 Visual person analysis: The *Person* class

We have face tracks for the people present in the living room.

When the user is in the living room and being entertained, then it is very probable to have a single face track. When the user is in the living room but is walking or standing, then there is a small chance of having a face track.

When the user has visits, then we have multiple face tracks. Visits are not implemented yet.

These are stored in the *Person* class.

5.9 Life goes on: The stepLife() method of the User class

Every 10 seconds, the life of the user moves on by recalculating the driving forces and considering a state transition. This is done by invoking the *stepLife(long timestamp, Map<Integer,State> states)* method of the *User* class.

First the driving forces are updated by adding the numbers designated in the current state. Then a uniform random number between zero and one is drawn and for every state into which a transition is probable, the probability of transition is accumulated. In essence the cumulative probability for state i is incremented by:

$\Delta p_i = 0$ if <driving force> is not between the <driving force>more and <driving force>less limits

or if α_f is the <driving force>factor for driving force f and $p_{t,i}$ is the transition probability into state i, it is incremented by:

$$\Delta p_i = p_{t,i} + \sum_f \alpha_f \cdot f \text{ if it is within these limits}$$

The transition into state i happens if the cumulative probability exceeds the drawn random number.

6 Conclusion and planning

This report discussed the metadata generated in the caring home: their format and local storage in a database. Regarding the format, we first presented the metadata dictionary, as it is currently used or envisioned and then we presented the actual JSON messages generated by the perceptual components and sent to the database. The next version of this deliverable will include the final metadata dictionary and all their JSON representation.

The variety of readily available databases imposes challenges in the selection of suitable one for eWall. This requires careful analysis of each DB in terms of performance, available interfaces, licensing, advantages and disadvantages. Such analysis is presented in this document allowing for clear identification of relevant as well as irrelevant to the project criteria for selection. Taking into account eWall requirements, CouchDB as in-house storage was selected. This selection is based on list of clearly identified advantages with one of the main being the readily available interface for the metadata format of the sensors. This format is in RDF triplets and the description of the metadata is in JSON. The metadata for in-house sensors, available up to the current development of the project is presented in a table. Examples of two types of sensors are presented in JSON. Based on this selection of database and metadata format currently is implemented initial prototype consisting of sensors including sensors for presence, accelerometer, gas sensors, etc., feeding the database with data in JSON format. Furthermore CouchDB streams this data to the Cloud platform which runs on on-line server.

Finally, the user simulator is introduced and documented. This allows creating simulated home environments, rich with metadata to test our applications. The current version of the simulator is 0.0.6. The final version of this deliverable will document the advances of this software, both in user modeling quality and in multiple user support.

Also, the final version of this deliverable will contain information on the management of the local metadata in CouchDB. Currently metadata are written continuously there, but the space is finite, so we will focus on a mechanism to clean up the database from old metadata that are already transferred to the cloud database.

7 Appendix A: Database selection

7.1 Relational Databases – SQL

7.1.1 Advantages

- Structured data, less storage required
- Easily accessible, SQL language
- Very commonly used, easier to learn
- Provide advanced functionalities for related data, such as cascade deletions and for multi-step transactions

7.1.2 Disadvantages

- Need to design a schema for the database beforehand, not easy to update and adapt to constantly changing data
- Overhead when handling bursts of data insertions/retrievals, as the data needs to be converted
- Difficulty in working with huge amounts of data

7.1.3 Relational Databases Description

The table below briefly describes some of the most popular non-proprietary (GPL-based or similar) and proprietary relational DBs with current (or recent) support: **MySQL, Oracle, Apache Derby, CUBRID, Drizzle, HSQLDB, Ingres, LucidDB, MariaDB, PostgreSQL, SmallSQL, SQLAnywhere, SQLite**

Database	Supported OS	License	Maintainer, Latest version	Features and Properties
MySQL [1]	Windows OSX Linux BSD UNIX AmigaOS Symbian z/OS Android	GPL v2 or Proprietary	Sun Microsystems (Oracle Corp.) 5.6.21 (2014-09-23)	– ACID, Transactions, Unicode, SQL and GUI interface – Unlimited DB size, 256 TB max table size, 64 kB max row size, 4096 max columns per row, 4 GB as max blob/clob size, 64 kB max CHAR size, 64 bit max NUMBER size, 1000 min DATE value, 9999 max DATE value, 64 max column name size – Temporary table – Indices – B-/B+ (some additional exclusions) – Capabilities – Union, Inner joins, Outer joins, Inner selects, Blobs and Clobs – Other objects – Cursor, Trigger, Function, Procedure, External routine – Partitioning – Range, Hash, Composite, List – Access control – Native network encryption, Enterprise directory compatibility, Patch access (partial), Run unprivileged

Oracle [2]	Windows OSX Linux UNIX z/OS	Proprietary	Oracle Corp. 12c Release 1 (Patchset as of July 2014)	– ACID, Referential integrity, Transactions, Unicode, SQL, GUI and API interface – Unlimited DB size, 4 GB max table size, 8 kB max row size, 1000 max columns per row, 128 TB as max blob/clob size, 32 767 B max CHAR size, 126 bit max NUMBER size, -4712 min DATE value, 9999 max DATE value, 30 max column name size – Temporary table, Materalized view – Indices – B-/B+, R-/R+ tree, Hash (some exceptions), Expression, Partial, Reverse, Bitmap, Full-text, Spatial – Capabilities – Union, Intersect, Except, Inner joins (via MINUS), Outer joins, Inner selects, Merge joins, Blobs and Clobs, Common Table Expressions, Windowing Functions, Parallel Query – Other objects – Data domain, Cursor, Trigger, Function, Procedure, External routine – Partitioning – Range, Hash, Composite, List – Access control – Native network encryption, Brute-force protection, Enterprise directory compatibility, Password complexity rules, Run unprivileged, Audit, Resource limit, Separation of duties, Security certification
Apache Derby [3]	Windows OSX Linux BSD UNIX z/OS	Apache	Apache 10.11.1.1 (2014-08-27)	– ACID, Referential integrity, Transactions, Unicode, SQL interface – Unlimited DB size, Unlimited max table size, Unlimited max row size, 1012 max columns per row, 2147483647 chars as max blob/clob size, 254 max CHAR size, 64 bit max NUMBER size, 0001-01-01 min DATE value, 9999-12-31 max DATE value, 128 max column name size – Temporary table – Indices – B-/B+ – Capabilities – Union, Intersect, Except, Inner joins, Outer joins, Blobs and Clobs, Common Table Expressions, Windowing Functions – Other objects – Cursor, Trigger, Function, Procedure, External routine
CUBRID [4]	Windows Linux	GPL v2	NHN Corporation 9.3 (2014-05-23)	– ACID, Referential integrity, Transactions, Unicode, SQL and GUI interface – 2 EB DB size, 2EB max table size, Unlimited max row size, 6400 max columns per row, Unlimited as max blob/clob size, 1 GB max CHAR size, 64 bit max NUMBER size, 0001-01-01 min DATE value, 9999-12-31 max DATE value, 254 max column name size – Indices – B-/B+, Expression, Partial, Reverse – Capabilities – Union, Intersect, Except, Inner joins, Outer joins, Inner selects, Merge joins, Blobs and Clobs, Windowing Functions – Other objects – Data domain, Cursor, Trigger,

				Function, Procedure, External routine – Partitioning – Range, Hash, List
Drizzle [5]	OSX Linux BSD UNIX	GPL v2, v3 Some BSD Components	Brian Aker 7.2.4 (2012-09-23)	– ACID, Referential integrity, Transactions, Unicode, SQL interface – Unlimited DB size, 64 TB max table size, 8kB max row size, 1000 max columns per row, 4GB max blob/clob size, 64 kB max CHAR size, 64 bit max NUMBER size, 0001 min DATE value, 9999 max DATE value, 64 max column name size – Temporary table – Indices – B-/B+ – Capabilities – Union, Inner joins, Outer joins, Inner selects, Blobs and Clobs – Other objects – Data domain, Cursor, Trigger, Function, Procedure, External routine
HSQLDB [6]	Windows OSX Linux BSD UNIX z/OS	BSD	HSQL Development Group 2.3.1 (2013-10-08)	– ACID, Referential integrity, Transactions, Unicode, SQL interface – 64 TB DB size, Unlimited max table size, Unlimited max row size, Unlimited columns per row, 64 TB max blob/clob size, Unlimited max CHAR size, Unlimited max NUMBER size, 0001-01-01 min DATE value, 9999-12-31 max DATE value, 128 max column name size – Temporary table – Indices – B-/B+ – Capabilities – Union, Intersect, Except, Inner joins, Outer joins, Inner selects, Merge joins, Blobs and Clobs, Common Table Expressions, Parallel Query – Other objects – Data domain, Trigger, Function, Procedure, External routine – Access control – Native network encryption, Enterprise directory compatibility, Password complexity rules, Patch access, Run unprivileged, Separation of duties
Ingres [7]	Windows OSX Linux BSD UNIX	GPL and Proprietary	Ingres Corp. Ingres Database 10 (2010-10-12)	– ACID, Referential integrity, Transactions, Unicode, SQL and QUEL interface – Unlimited DB size, Unlimited max table size, 256 kB max row size, 1024 max columns per row, 2 GB as max blob/clob size, 32 000 B max CHAR size, 64 bit max NUMBER size, 0001 min DATE value, 9999 max DATE value, 256 max column name size – Temporary table – Indices – B-/B+, R-/R+ tree, Hash, Expression, Bitmap – Capabilities – Union, Inner joins, Outer joins, Inner selects, Merge joins, Blobs and Clobs – Other objects – Data domain, Cursor, Trigger, Function, Procedure, External routine

				– Partitioning – Range, Hash, Composite, List
LucidDB [8]	Windows OSX Linux	GPL v2	The Eigenbase Project 0.9.3 (2010-06-16)	– ACID, Unicode, SQL interface – Indices – B-/B+, Bitmap – Capabilities – Union, Intersect, Except, Inner joins, Outer joins, Inner selects, Merge joins – Other objects – Cursor, Function, Procedure, External routine
MariaDB [9]	Windows OSX Linux BSD UNIX	GPL v2 (LGPL for client-libraries)	MariaDB Community 10.0.14 (2014-09-26)	– ACID, Transactions, Unicode, SQL interface – Indices – B-/B+ – Access control – Native network encryption (SSL), Enterprise directory compatibility (not on Windows), Patch access (partial), Run unprivileged
Postgre SQL [10]	Windows OSX Linux BSD UNIX Android	PostgreSQL (liberal Open Source license)	PostgreSQL Global Development Group 9.3.5 (2014-07-24)	– ACID, Referential integrity, Transactions, Unicode, SQL, GUI and API interface – Unlimited DB size, 32 TB max table size, 1.6 TB max row size, 250-1600 max columns per row, 1 GB as max blob/clob size, 1 GB max CHAR size, Unlimited max NUMBER size, -4713 min DATE value, 5874897 max DATE value, 63 max column name size – Temporary table, Materalized view – Indices – B-/B+, R-/R+ tree, Hash, Expression, Partial, Reverse, Bitmap, GiST, GIN, Full-text, Spatial (some exceptions) – Capabilities – Union, Intersect, Except, Inner joins, Outer joins, Inner selects, Merge joins, Blobs and Clobs, Common Table Expressions, Windowing Functions – Other objects – Data domain, Cursor, Trigger, Function, Procedure, External routine – Partitioning – Range, Hash, Composite, List – Access control – Native network encryption, Brute-force protection, Enterprise directory compatibility, Password complexity rules, Patch access, Run unprivileged, Resource limit, Separation of duties, Security certification
SmallSQL [11]	Windows OSX Linux BSD UNIX z/OS	LGPL	SmallSQL 0.21 (2011-06-22)	– Indices – B-/B+
SQL	Windows	Proprieta	Sybase	– ACID, Referential integrity, Transactions,

Anywhere [12]	OSX Linux UNIX Android	ry	16.0 (2013-04-18)	Unicode, SQL interface – 128 TB DB size, Limited by file size - max table size, Limited by file size - max row size, 32767 max columns per row, 2 GB as max blob/clob size, 2 GB max CHAR size, 64 bit max NUMBER size, No DATE type, Unlimited max column name size – Temporary table, Materialized view – Indices – B-/B+, Full-text – Capabilities – Union, Intersect, Except, Inner joins, Outer joins, Inner selects, Merge joins, Blobs and Clobs, Common Table Expressions, Windowing Functions, Parallel Query – Other objects – Data domain, Cursor, Trigger, Function, Procedure, External routine – Access control – Native network encryption, Enterprise directory compatibility, Password complexity rules, Run unprivileged, Audit, Separation of duties, Security certification
SQLite [13]	Windows OSX Linux BSD UNIX AmigaOS Symbian iOS	Public Domain	D. Richard Hipp 3.8.6 (2014-08-15)	– ACID, Referential integrity, Transactions, Unicode (Optional), SQL and API interface – 104 TB DB size, Limited by file size - max table size, Limited by file size - max row size, 45000 max columns per row, 2 GB as max blob/clob size, 2 GB max CHAR size, 64 bit max NUMBER size, 0001-01-01 min DATE value, 9999-12-31 max DATE value, Uknown max column name size – Temporary table – Indices – B-/B+, R-/R+ tree, Partial, Reverse, Full-text, Spatial (some exceptions) – Capabilities – Union, Intersect, Except, Inner joins, Outer joins (LEFT only), Inner selects, Blobs and Clobs – Other objects – Trigger, External routine – Access control – Patch access (partial), Run unprivileged, Audit, Resource limit

Remark: Some features and properties may be missing

7.1.4 Relational Database Selection Criteria

The criteria for selection a particular database management system (DBMS) for a given purpose could be clustered in various groups. Here two main factors are considered: support organization and capabilities – both of which are considered to be of equal importance for a final select decision.

Support organization factors:

– **Software license** – in its base lay some of the most motivating reasons for selection of a DBMS: paid vs. non-paid; usage time restrictions; number of clients/machines to run onto; commercial vs. academic (personal) usage restrictions; separate modules / libraries additional restrictions; incorporating into larger projects (systems) as a complete new product or supporting sub-product; re-licensing admissions; etc.

- **Supported operating system (OS)** – depending on the OS as a base platform for running the higher-level applications which in the most cases is of more fundamental role within a whole project planning, the support of that OS by the DBMS' maintainer may well be crucial factor for selection.
- **Latest version release date (latest stable version)** – reveals how current the development process of the maintainer is.
- **Lifespan of development (first stable version release date)** – indicates the release activity consistency over time – how many generations of the product has been produces taking into account all major technologies' changes during the years.
- **Number of stable versions being released during lifespan and releases' frequency** – shows how often a given set of currently used technologies are updated, how regular (active) is the production process and to what degree it correspond in number of enhancement stages to other maintainers' (developers') products.
- **Maintainer (Developer)** – although not a direct objective factor for selection of a DBMS, the maintainer itself given its current status as for the market rating (financial status), stated future plans for development (support), undertaken incorporating initiatives (merging the product in larger systems, frameworks, etc. including with other companies, forming enterprises, etc.) and other activities may well indicate indirectly important clues for proper choice.

  Technical features:

The technical features may be divided into two sub-groups. The first one concerns the fundamental features or native capabilities of the DBMS itself. Based on these features the basic functionalities for interconnection with the "outer" world (the whole framework to be developed at hand) are defined which must cover fully the expected system's functions. The second sub-group consists of all limiting factors inside the DBMS – mainly the minimal and maximal size of a particular database property.

Fundamental features:

- Atomicity, Consistency, Isolation, Durability (ACID)
- Referential integrity
- Transactions
- Unicode support
- Supported Interfaces Type
- Temporary table support
- Materialized view
- Supported indices type – most popular B-/B+ tree, R-/R+ tree, Hash, Expression, Partial, Reverse, Bitmap, GiST, GIN, Full-text, Spatial, FOT
- Supported data types – apart from standard (most common) types, e.g. Integer, Floating point, Decimal, String, Binary, Date/Time, Boolean, interest may represent special types of data supported by some DBMS such as PICTURE, GEOMETRY, SEQUENCE, etc. when processing image/audio/video data in a more complete form (context based).
- Database capabilities – Union, Intersect, Except, Inner joins, Outer joins, Inner selects, Merge joins, Blobs and Clobs, Common Table Expressions, Windowing Functions, Parallel Queries
- Other supported objects - Data Domain, Cursor, Trigger, Function, Procedure, External routine
- Supported partitioning methods – Range, Hash, Composite (Range+Hash), List, Expression

- Access control functionalities - Native network encryption, Brute-force protection, Enterprise directory compatibility, Password complexity rules, Patch access, Run unprivileged, Audit, Resource limit, Separation of duties (RBAC), Security Certification, Label Based Access Control (LBAC)

  Limiting factors:
- Maximal DB size
- Maximal table size
- Maximal row size
- Maximum number columns per row
- Maximal Blob/Clob size
- Maximal CHAR size
- Maximal NUMBER size
- Minimal DATE value
- Maximal DATE value
- Maximal column name size

Given the table comparison of the most popular DBMS a precise balanced decision should be made for selecting the proper one for the current project. Some factors to be considered are:

- Type, speed, supported transfer protocols by client's connection
- Centralized (server/cloud) OS/environment
- Number of simultaneously served clients
- Average growth of users in time
- Type of data to be stored (to be compared with the limiting factors of the candidate DBMS) – numbers, text, images/frames, audiosamples, medical signals (packets) – separation of short- and long-term storage data for temporal analysis/decision of immediate action and prolonged analysis of clients' state (possibly for diagnosis, etc.)
- Clients' and operator's user interfaces – search and visualization capabilities, (re-) ordering of data, cross-connectivity among various users' data (personal/public distinction), etc.
- Network and local security level (type)
- Initial lifespan prediction of the developed system
- Compatibility and interoperability demands (changes) in time (possible OS change, ceased/transferred DBMS support by the maintainer, etc.)

7.2 Non-relational Databases – NoSQL

7.2.1 Advantages

- Can accept varying data in JSON format, no need for predefining a database schema
- Can handle high rates for data insertions
- Scale up to handle huge amounts of data
- CouchDB provides direct REST interface and easy replication
- MongoDB : Consistency and Partition Tolerance
- CouchDB : Availability and Partition Tolerance

7.2.2 Disadvantages

- No common standard for access

- Can be complex to program the required functions for data retrieval
- User management in CouchDB is quite difficult

7.2.3 Non-relational Databases Description

In the table below are briefly described some of the most commonly used Non-relational Databases: CouchDB, MongoDB, Redis, Cassandra, Riak, Accumulo, HBase, Hypertable, Neo4j, ElasticSearch, Couchbase (ex-Membase), Scalaris, VoltDB

Database	Supported OS / Impl. language	License	Maintainer, Latest version	Protocol , Features and Properties, Usage
CouchDB [14]	Android BSD Linux OS X Solaris Windows / Erlang	Apache	Apache Software Foundation 1.6.1 (2014-09-03)	Protocol – HTTP/REST Features and Properties – DB consistency, ease of use – Bi-directional replication, – continuous or ad-hoc, – with conflict detection, – thus, master-master replication. – MVCC - write operations do not block reads – Previous versions of documents are available – Crash-only (reliable) design – Needs compacting from time to time – Views: embedded map/reduce – Formatting views: lists & shows – Server-side document validation possible – Authentication possible – Real-time updates via changes – Attachment handling – thus, CouchApps (standalone js apps) Usage – For accumulating, occasionally changing data, on which pre-defined queries are to be run. Places where versioning is important. – For example: CRM, CMS systems. Master-master replication is an especially interesting feature, allowing easy multi-site deployments.
MongoDB [15]	Linux OS X Solaris Windows / C++	AGPL Drivers Apache	MongoDB, Inc 2.6.4 (2014-08-11)	Protocol – Custom, binary (BSON) Features and Properties – Retains some friendly properties of SQL. (Query, index) – Master/slave replication (auto failover with replica sets) – Sharding built-in – Queries are javascript expressions – Run arbitrary javascript functions server-side – Better update-in-place than CouchDB

				– Uses memory mapped files for data storage – Performance over features – Journaling (with --journal) is best turned on – On 32bit systems, limited to ~2.5Gb – An empty database takes up 192Mb – GridFS to store big data + metadata (not actually an FS) – Has geospatial indexing – Data center aware Usage – If you need dynamic queries. If you prefer to define indexes, not map/reduce functions. If you need good performance on a big DB. If you wanted CouchDB, but your data changes too much, filling up disks. – For example: For most things that you would do with MySQL or PostgreSQL, but having predefined columns really holds you back.
Redis [16]	BSD Linux OS X Windows / C	BSD	Salvatore Sanfilippo 2.8.16 (2014-09-16)	Protocol – Telnet-like, binary safe Features and Properties – Blazing fast – Disk-backed in-memory database, – Dataset size limited to computer RAM (but can span multiple machines' RAM with clustering) – Master-slave replication, automatic failover – Simple values or data structures by keys but complex operations like ZREVRANGEBYSCORE. – INCR & co (good for rate limiting or statistics) – Bit operations (for example to implement bloom filters) – Has sets (also union/diff/inter) – Has lists (also a queue; blocking pop) – Has hashes (objects of multiple fields) – Sorted sets (high score table, good for range queries) – Lua scripting capabilities – Has transactions – Values can be set to expire (as in a cache) – Pub/Sub lets one implement messaging Usage – For rapidly changing data with a foreseeable database size (should fit mostly in memory). – For example: To store real-time stock prices. Real-time analytics. Leaderboards. Real-time communication. And wherever you used memcached before.
Cassandra [17]	BSD Linux	Apache	Apache Software	Protocol – CQL3 & Thrift

	OS X Windows / Java		Foundation 2.1.0 (2014-09-11)	Features and Properties – Store huge datasets in "almost" SQL – CQL3 is very similar SQL, but with some limitations that come from the scalability (most notably: no JOINs, no aggregate functions.) – CQL3 is now the official interface. Don't look at Thrift, unless you're working on a legacy app. This way, you can live without understanding ColumnFamilies, SuperColumns, etc. – Querying by key, or key range (secondary indices are also available) – Tunable trade-offs for distribution and replication (N, R, W) – Data can have expiration (set on INSERT). – Writes can be much faster than reads (when reads are disk-bound) – Map/reduce possible with Apache Hadoop – All nodes are similar, as opposed to Hadoop/HBase – Very good and reliable cross-datacenter replication – Distributed counter datatype. – You can write triggers in Java. Usage – When you need to store data so huge that it doesn't fit on server, but still want a friendly familiar interface to it. – For example: Web analytics, to count hits by hour, by browser, by IP, etc. Transaction logging. Data collection from huge sensor arrays.
Riak [18]	Linux BSD Mac OS X Solaris / Erlang & C, some Java Script	Apache	Basho Technologies 2.0.0 (2014-09-02)	Protocol – HTTP/REST or custom binary Features and Properties – Fault tolerance – Stores blobs – Tunable trade-offs for distribution and replication – Pre- and post-commit hooks in JavaScript or Erlang, for validation and security. – Map/reduce in JavaScript or Erlang – Links & link walking: use it as a graph database – Secondary indices: but only one at once – Large object support (Luwak) – Comes in "open source" and "enterprise" editions – Full-text search, indexing, querying with Riak Search – In the process of migrating the storing backend from "Bitcask" to Google's "LevelDB" – Masterless multi-site replication replication and SNMP monitoring are commercially licensed Usage – If you want something Dynamo-like data storage, but no way you're going to deal with the bloat and complexity. If you need very good single-site scalability, availability and fault-tolerance, but

				you're ready to pay for multi-site replication. – For example: Point-of-sales data collection. Factory control systems. Places where even seconds of downtime hurt. Could be used as a well-update-able web server.
Accumulo [19]	Linux OS X Unix Windows / Java and C++	Apache	Apache Software Foundation 1.6.0 (2014-05-02)	Protocol – Thrift Features and Properties – A BigTable with Cell-level security – Another BigTable clone, also runs of top of Hadoop – Originally from the NSA – Cell-level security – Bigger rows than memory are allowed – Keeps a memory map outside Java, in C++ STL – Map/reduce using Hadoop's facitlities (ZooKeeper & co) – Some server-side programming Usage – If you need to restrict access on the cell level. – For example: Same as HBase, since it's basically a replacement: Search engines. Analysing log data. Any place where scanning huge, two-dimensional join-less tables are a requirement.
HBase [20]	Linux OS X Unix Windows / Java	Apache	Apache Software Foundation 0.98.4 (2014-07-21)	Protocol – HTTP/REST (also Thrift) Features and Properties – Billions of rows X millions of columns – Modelled after Google's BigTable – Uses Hadoop's HDFS as storage – Map/reduce with Hadoop – Query predicate push down via server side scan and get filters – Optimizations for real time queries – A high performance Thrift gateway – HTTP supports XML, Protobuf, and binary – Jruby-based (JIRB) shell – Rolling restart for configuration changes and minor upgrades – Random access performance is like MySQL – A cluster consists of several different types of nodes Usage – Hadoop is probably still the best way to run Map/Reduce jobs on huge datasets. Best if you use the Hadoop/HDFS stack already. – For example: Search engines. Analysing log data. Any place where scanning huge, two-dimensional join-less tables are a requirement.
Hypertable	Linux	GPL	Hypertable	Protocol – Thrift, C++ library, or HQL shell

[21]	Mac OS X / C++	2.0	Inc. 0.9.8.1 (2014-09-14)	Features and Properties – A faster, smaller HBase – Implements Google's BigTable design – Run on Hadoop's HDFS – Uses its own, "SQL-like" language, HQL – Can search by key, by cell, or for values in column families. – Search can be limited to key/column ranges. – Sponsored by Baidu – Retains the last N historical values – Tables are in namespaces – Map/reduce with Hadoop Usage – If you need a better HBase. – For example: Same as HBase, since it's basically a replacement: Search engines. Analysing log data. Any place where scanning huge, two-dimensional join-less tables are a requirement.
Neo4j [22]	Linux OS X Unix Windows / Java	GPL, some features AGPL/ commercial	Neo Technology 2.1.5 (2014-09-04)	Protocol – HTTP/REST (or embedding in Java) Features and Properties – Graph database - connected data – Standalone, or embeddable into Java applications – Full ACID conformity (including durable data) – Both nodes and relationships can have metadata – Integrated pattern-matching-based query language ("Cypher") – Also the "Gremlin" graph traversal language can be used – Indexing of nodes and relationships – Nice self-contained web admin – Advanced path-finding with multiple algorithms – Indexing of keys and relationships – Optimized for reads – Has transactions (in the Java API) – Scriptable in Groovy – Online backup, advanced monitoring and High Availability is AGPL/commercial licensed Usage – For graph-style, rich or complex, interconnected data. – For example: For searching routes in social relations, public transport links, road maps, or network topologies.
ElasticSearch [23]	Linux OS X Unix Windows / Java	Apache	Elasticsearch 1.3.3 (2014-09-29)	Protocol – JSON over HTTP (Plugins: Thrift, memcached) Features and Properties – Advanced Search – Stores JSON documents – Has versioning – Parent and children documents – Documents can time out

				– Very versatile and sophisticated querying, scriptable – Write consistency: one, quorum or all – Sorting by score – Geo distance sorting – Fuzzy searches (approximate date, etc.) – Asynchronous replication – Atomic, scripted updates (good for counters, etc.) – Can maintain automatic "stats groups" (good for debugging) – Still depends very much on only one developer (kimchy). Usage – When you have objects with (flexible) fields and you need "advanced search" functionality. – For example: A dating service that handles age difference, geographic location, tastes and dislikes, etc. Or a leaderboard system that depends on many variables.
Couchbase (ex-Membase) [24]	Linux OS X Unix Windows / Erlang & C++	Apache	Couchbase Inc. 2.5.1 (2014-03-31)	Protocol – memcached + extensions Features and Properties – Memcache compatible, but with persistence and clustering – Very fast (200k+/sec) access of data by key – Persistence to disk – All nodes are identical (master-master replication) – Provides memcached-style in-memory caching buckets, too – Write de-duplication to reduce IO – Friendly cluster-management web GUI – Connection proxy for connection pooling and multiplexing (Moxi) – Incremental map/reduce – Cross-datacenter replication Usage – Any application where low-latency data access, high concurrency support and high availability is a requirement. – For example: Low-latency use-cases like ad targeting or highly-concurrent web apps like online gaming (e.g. Zynga).
Scalaris [25]	Windows Linux/ Erlang	Apache	Scalaris 0.7.1 (2014-09-30)	Protocol – Proprietary & JSON-RPC Features and Properties – Distributed P2P key-value store – In-memory (disk when using Tokyo Cabinet as a backend) – Uses YAWS as a web server – Has transactions (an adapted Paxos commit) – Consistent, distributed write operations – From CAP, values Consistency over Availability

				(in case of network partitioning, only the bigger partition works) Usage – If you like Erlang and wanted to use Mnesia or DETS or ETS, but you need something that is accessible from more languages (and scales much better than ETS or DETS). – For example: In an Erlang-based system when you want to give access to the DB to Python, Ruby or Java programmers.
VoltDB [26]	Linux Mac OS X/ Java, C++	GPL 3	VoltDB Inc. 4.0 (2014-01-26)	Protocol – Proprietary Features and Properties – Fast transactions and rapidly changing data – In-memory relational database. – Can export data into Hadoop – Supports ANSI SQL – Stored procedures in Java – Cross-datacenter replication Usage – Where you need to act fast on massive amounts of incoming data. – For example: Point-of-sales data analysis. Factory control systems.

Remark: Some features and properties may be missing

7.2.4 Non-relational Database Selection Criteria

Here are described several important factors that need to be taken into account to make the right choice of NoSQL database:

Storage Type

- For instance, get, put and delete functions are best supported by Key Value systems.
- Aggregation becomes much easier while using Column oriented systems as against the conventional row oriented databases. They use tables but do not have joins.
- Mapping data becomes easy from object oriented software using a Document oriented NoSQL database such as XML or JSON as they use structure document formats.
- Tabular format is replaced and data is stored in graphical format.

Concurrency Control

Concurrency control is what defines how two users can simultaneously edit the same bit of information. It happens quite often that one of the users is locked out and is unable to edit or perform other actions till the active user has finished editing.

- **Locks** - prevent more than one active user to edit an entity such as a document, row or an object.
- **MVCC** (Multi-Version Concurrency Control) - guarantee a read consistent view of the database, but result in conflicting versions of an entity if multiple users modify it at once.

MVCC makes it possible for a transaction to seamlessly go through by maintaining many different versions of the object. That means transaction consistency is maintained even if that shows varying snapshots to different users at any given point in time. Any changes made to the database will be shown to others depending which snapshot are they referring to.

- **None –** Atomicity is missing in some systems thereby not providing the same view of the database to multiple users editing the database.
- **ACID –** For reliable database transactions, ACID or Atomicity, Consistency, Isolation, Durability is a safe bet. It allows for pre-screening transactions to avoid conflicts with no deadlocks.

Replication

Replication ensures that mirror copies are always in sync.

- **Synchronous Mode** – Though it is an expensive approach as there is a dependency on the second server to respond, but it always ensures consistency. After receiving response from the second server, the first server sends back the ACK to the client. This ensures data is placed in multiple nodes at the same time.
- **Asynchronous mode** – In this mode, one database gets updated without waiting for the answer from the other database. Two databases could be not consistent in the range of few milliseconds. This should explain why this cost-effective and synchronous replication method is also dubbed as 'Eventually Consistent.'
- **Implementation Language** – Implementation language helps to determine how fast a database will process. Typically NoSQL databases written in low level languages such as C/C++ and Erlang will be the fastest. On the other hand, those written in higher level languages such as Java make customizations easier.

7.3 RDF Databases

7.3.1 Advantages

- No need for database schema, storing data in RDF triplets
- Can perform reasoning on the data
- Standardised language for access (SPARQL)
- Federated queries, e.g. queries can retrieve data from multiple databases
- Sesame provides REST interface

7.3.2 Disadvantages

- Slow input of data
- Not easy to cope with huge amounts of data

7.3.3 RDF Databases Description

In the table below are given some details about three of the most popular RDF Databases: **Sesame, Jenna, Virtuoso**

Database	Supported	License	Maintainer,	Features and Properties

	OS / Impl. language		Latest version	
Jena [27]	Linux OS X Unix Windows/ Java	Apache	Apache Software Foundation 2.12.0 (2014-08-02)	– Appropriate for building semantic web applications. – Can be tied to an existing RDBMS such as MySQL or PostgreSQL. – Providing scalable storage and query of RDF datasets using conventional SQL databases (for use in standalone applications, J2EE and other application frameworks) – Provides access to a reasoned that configured to perform reasoning of various degrees. – Can be used to perform simple RDFS reasoning to the more memory intensive OWL-DL reasoning Java framework for building semantic web applications.
Sesame [28]	Linux OS X Unix Windows/ Java	BSD	Aduna 2.7.13 (2014-08-13)	– Supporting both memory-based and a disk-based storage. – Open source framework for storage, inferencing and querying of RDF data. – Matches the features of Jena with the availability of a connection API, inferencing support, availability of a web server and SPARQL endpoint. – Provides support for multiple backends like MySQL and Postsgre. – Sesame Native is the native triple store offering from Sesame. As compared to be Jena's native triple, TDB, it is less scabable.
Virtuoso [29]	AIX FreeBSD HP-UX Linux OS X Solaris Windows / C	GPL v2 and Proprietary	OpenLink Software 7.1 (2014-02)	– Provides command line loaders, a connection API, support for SPARQL and web server to perform SPARQL queries and uploading of data over HTTP. – Scalable to the region of 1B+ triples. – Provides bridges to be used with Jena and Sesame.

8 Bibliography

[1] http://www.mysql.com/

[2] https://www.oracle.com/database/index.html

[3] http://db.apache.org/derby/

[4] http://www.cubrid.org/

[5] http://www.drizzle.org/

[6] http://hsqldb.org/

[7] https://www.openhub.net/p/ingres

[8] http://luciddb.sourceforge.net/

[9] https://mariadb.org/

[10] http://www.postgresql.org/

[11] http://www.smallsql.de/

[12] http://www.sap.com/pc/tech/database/software/sybase-sql-anywhere/index.html

[13] http://www.sqlite.org/

[14] http://couchdb.apache.org/

[15] http://www.mongodb.org/

[16] http://redis.io/

[17] http://cassandra.apache.org/

[18] http://basho.com/riak/

[19] https://accumulo.apache.org/

[20] http://hbase.apache.org/

[21] http://hypertable.org/

[22] http://neo4j.com/

[23] http://www.elasticsearch.org/

[24] http://www.couchbase.com/

[25] https://code.google.com/p/scalaris/

[26] http://voltdb.com/

[27] https://jena.apache.org/documentation/sdb/

[28] http://www.sesamedatabase.com/

[29] http://virtuoso.openlinksw.com/

9 Abbreviations

AGPL - Affero General Public License
API - Application Programming Interface
BSD - Berkeley Software Distribution
DB - Database
DBMS - Database Management System
ECG - Electrocardiography
GPL - General Public License
GUI - Graphical User Interface
HTTP - Hypertext Transfer Protocol
IO - Input/Output
IMA - Integrated Modulus of Accelerometer output
ISA - Integrated Squared output of Accelerometer
JSON - JavaScript Object Notation
OS - Operating System
RDF - Resource Description Framework
REST - Representational state transfer
SQL - Structured Query Language
XML - Extensible Markup Language