A Demonstration of 4D Digital Avatar Infrastructure for Access of Complete Patient Information

Project acronym: MyHealthAvatar

Deliverable No. 6.2
Design for data and RDF repositories

Grant agreement no: 600929

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)	

COVER AND CONTROL PAGE OF DOCUMENT	
Project Acronym:	MyHealthAvatar
Project Full Name:	A Demonstration of 4D Digital Avatar Infrastructure for Access of Complete Patient Information
Deliverable No.:	D6.2
Document name:	Design for data and RDF repositories
Nature (R, P, D, O)[1]	R
Dissemination Level (PU, PP, RE, CO)[2]	PU
Version:	1
Actual Submission Date:	25/02/2014
Editor: Institution: E-Mail:	Xia Zhao University of Bedfordshire xia.zhao@beds.ac.uk

ABSTRACT:
This deliverable focuses the design of data and RDF repositories that will guide the data access and management for the MyHealthAvatar project. The report initially reviews the literatures and technologies for data and RDF repositories, then analyse the data required for the uses cases identified in the MyHealthAvatar peoject and design the data models as well as the communication between data and RDF repositories.

KEYWORD LIST:
Data repository, RDF repository, Repository design, Cassandra.

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 600929.

[1] **R**=Report, **P**=Prototype, **D**=Demonstrator, **O**=Other

[2] **PU**=Public, **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)

The author is solely responsible for its content, it does not represent the opinion of the European Community and the Community is not responsible for any use that might be made of data appearing therein.

MODIFICATION CONTROL			
Version	**Date**	**Status**	**Author**
0.1	13/01/2014	Draft	Xia Zhao
0.2	31/01/2014	Draft	Xia Zhao Hong Qing Yu
0.3	07/02/2014	Draft	Xia Zhao Hong Qing Yu
0.4	21/02/2014	Draft	Xia Zhao Hong Qing Yu
1	25/02/2014	Final	Xia Zhao Hong Qing Yu

List of contributors

- Xia Zhao (BED)
- Hong Qing Yu (BED)
- Feng Dong (BED)
- Enjie Liu (BED)

Contents

1 Executive Summary

MyHealthAvatar follows recommendations from relevant VPH activities on "Digital Patient". MyHealthAvatar architectural platform will be designed as an integrated facility that allows multiple functionalities rather than just a data storage facility as in the previous attempts. Its distinctive features include:

- ICT utilities to support data collection with minimal user input, including web information extraction, mobile apps, etc.
- ICT toolbox to support clinical decisions by using simulation models and by using visual analytics.
- Data and model repositories to provide rich resources of data and models
- Ontology and RDF repositories to support data search and reasoning.
- A cloud based ICT architecture that allows the access of data from a range of different sources, and integration of the repositories, the toolbox and the ICT utilities.
- A local cloud solution to support the computing requirement for the avatars without remote data transfer.
- A proof of market on open sources for MyHealthAvatar APIs.

The 4D Avatar proposed within the MyHealthAvatar project is intended to be a citizen's lifelong companion a complete digital representation of his/her health-related data, including both medical records and social or lifestyle information, to create a comprehensive picture of the person in the context of their individualised healthcare. All of the data will be useful at different levels in assisting health decisions to be made.

This deliverable presents the design for data and RDF repositories which will guide the next stage development of both repositories. This document covers the activities for Tasks T6.3 and T6.4 and Milestone MS062 in respect to the selection of implementation technologies, overall design of the two-layered repositories and the detailed schema design for each of them. We highlight the reviews of data and RDF repositories performed within *D3.1 User Requirements* and select NoSQL Apache Cassandra for data repository and OWLIM for RDF repositories.

2 Introduction

Within MyHealthAvatar, collecting, accessing, managing and possibly sharing healthcare related data are not only important to individuals who can manage their own health, but also important for clinicians and other healthcare workers for patient monitoring and providing suitable in-time care.

The design of repositories takes into account the consideration that the data will be large and likely to fall into the category of Big Data, bearing in mind the number of users and the quantity and types of the information that will be stored on daily basis for the duration of a person's entire life.

2.1 Structure of this document

This document reviews the literatures and technologies in the implementation of data and RDF repositories and reports the design for data and RDF repositories for MyHealthAvatar.

The structure of this document is organised as follows. Section 3 and 4 discuss the state-of-art literature review of data repository and RDF repository, respectively. Section 5 describes the overall design structure of repositories for MyHealthAvatar. After that, Section 6 and 7 discuss the detail design of data and RDF repositories, respectively. Section 8 concludes the report and outlines the future work.

3 Review of the State-of-the-art Data Repositories

To support the design and implementation of the MyHealthAvatar data repository, this sections reviews two types of current available data repositories: RDBMS and NoSQL.

3.1 Relational Database Management Systems (RDBMS)

RDBMS defines the SQL language by using a data manipulation method that can model structured and organised data using relations and tables. Data and their relationships are stored in separate tables. RDBMS typically support transactions with the ACID attribute which are defined as follows [1]:

- **Atomicity***:* Everything in a transaction must happen successfully or none of the changes are committed. This avoids a transaction that changes multiple pieces of data from failing halfway and only making a few changes.
- **Consistency**: The data will only be committed if it passes all the rules in place in the database (i.e. data types, triggers, constraints, etc).
- **Isolation:** Transactions won't affect other transactions by changing data that another operation is counting on; and other users won't see partial results of a transaction in progress (depending on isolation mode).
- **Durability:** Once data is committed, it is durably stored and safe against errors, crashes or any other (software) malfunctions within the database.

RDBMS have tight consistency and strong support for structured queries. However, it is difficult to satisfy ACID (e.g. atomicity and isolation) while keep the database distributed for large scale storage and data processing. For example, if the relational database replications have to be implemented on distributed systems, all related locks must be applied to each system while the distribution transaction protocol is being processed; the higher the service load of the system, the heavier the lock competition becomes.

3.2 NoSQL Database Systems

NoSQL (Not Only SQL) database implementations put high performance, high availability and scalability as their priorities for large volume data manipulations. Rather than ACID properties, most of NoSQL databases seek eventual consistency and apply Brewer's CAP theorem [2]. Figure 1 shows the three aspects defined as CAP, and the CAP-Theorem suggests that only two of the three different aspects of scaling out are can be achieved fully at the same time.

- **Consistency***:* All the servers in the system will have the same data so anyone using the system will get the latest data regardless of any updates happening.
- **Availability**: All of the servers will always return data they have (even if it's not the latest data or consistent across the system).
- **Partition Tolerance**: The system continues to operate as a whole even if individual servers fail or can't be reached.

NoSQL database generally supports dynamic schema and automatic sharding. Thus, no predefined schemas are required before implementation, and the system is good at scaling in distributed environment for handling large volume data storage and processing.

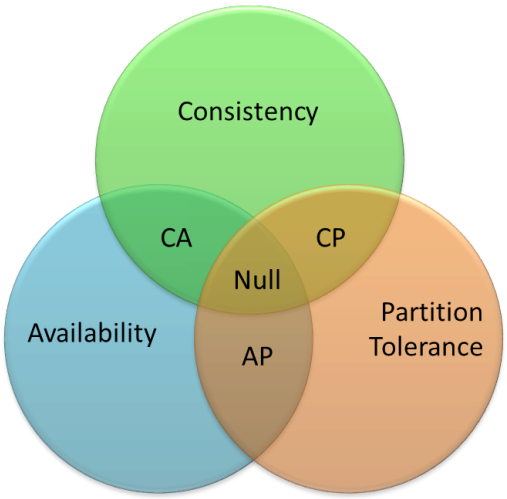

Figure 1: The CAP theorem.

Four categories of data models for NoSQL implementations are available: Key-value, Column, Document and Graph.

Key-value stores use a hash table where a unique key is used to refer to a particular item of data. The key is normally stored as a string, and the data itself is usually some kind of primitive data types (e.g. a string, an integer, and an array) or an object. Although the model structure is simple, querying a single value by the key is straightforward and faster than relational databases. However, as the data stored in the key-value stores has no schema at all, it is inefficient when querying or updating part of a value.

Both, keys and values can be complex, compound objects as well consisting of lists and maps. Compared to relational databases, the simple data structure and API of a key-value store does not provide complex querying capabilities: joins have to be implemented in client applications while constraints on foreign-keys are impossible; besides, no triggers and views may be set up.

Column stores are created to store and process large amount of data stored on distributed file systems. Keys are still used but refer to multiple columns which are arranged by column families. In contrast to relational database systems, data are still stored in tables, but are organised by columns rather than rows, which make key-based data aggregation much easier. However, there are no joins in column stores, relations have to be managed when developing applications. Column stores usually offer good concurrent read and write performance, fast key-based queries and scalable distributed store over the network. Column stores are typical used for data analytics and business intelligence in distributed environment.

Document stores also use keys (or indexes) to refer to a collection of documents which are typically in semi-structure and stored in formats such as JSON and XML. Document stores usually focus on big data storage and query performance rather than high concurrent read and write performance.

Graph stores are designed to store data with their relations can be represented in graphs. Social networks are typical examples can be represented by graph stores. Graph stores can take advantage of graph algorithm for application functionalities; however, it usually requires traverse the entire graph to achieve a definitive answer, so the query performance gets lower when the size of the graph increases.

3.3 Overall Technical Analysis on Data Repositories

3.3.1 RDBMS vs. NoSQL Databases

Table 1: RDBMS vs. NoSQL databases.

	RDBMS	NoSQL Databases
Schemas	Non-dynamic.	Typically dynamic.
Data Storage Model	Rows/columns in tables	Key value, column, document, graph
Scaling	Vertical	Horizontal
Sharding	Non-automatic	Automatic
Data Replication	Depends on provider.	Yes
Data Query	Good; SQL	Not powerful; unstructured query via object-oriented APIs.
Development Model	Mixed open-source and proprietary.	Open-source.
Consistency	Yes	Varies based on databases. Some provide strong consistency (e.g. HBase) and others offer eventual consistency (e.g. Apache Cassandra).
Transaction Supports	Yes	No
Properties	ACID	CAP theorem

Table 1 provides a summary on RDBMS and NoSQL databases based on the following aspects.

- **Schemas**: Whether the database uses dynamic schemas, which means whether the structure and data type are required to be defined in advance; to store a new data item, whether the database must be taken offline and the entire database must be altered or the new data items can be added on the fly.
- **Data storage model**: whether the database supports various data storage models, such as hierarchical data structure.
- **Scaling**: whether the database is vertical or horizontal scaling. Vertical scaling means when the storage demand increases, the system increases the server power to handle it. In contrast, horizontal scaling adds more servers or use cloud instances to handle increased demands.

- **Sharding**: whether the database supports automatic sharding. If not, complex configurations are typically required across the distributed servers.
- **Data replication**: whether the database supports automatic data replication.
- **Data query**: whether the database is good at handling complex queries and what query languages does it support.
- **Development model**: whether the database is developed in the open-source or proprietary.
- **Consistency***:* whether the database can be configured for strong consistency.
- **Transaction support***:* whether the database is good at supporting complex and high load transaction.
- **Properties***:* what properties does the database support: ACID or CAP.

We consider the following aspects when choosing between SQL and NoSQL databases for the MyHealthAvatar.

- **Natural of the data to be stored**: To store and access the lifelong information of each citizen in MyHealthAvatar, there will be large volume of data in unstructured, semi-structured and structured formats. For example, data collected from wearable devices and social networks may come with different formats (e.g. XML and JSON) and involves multiple data types (e.g. objects, lists and customised data types), which are difficult to be incorporated into the rows/columns table style in SQL. Furthermore, data from different resources with unknown structure are likely to be added in future. It is not feasible to define the data type and structured beforehand as required by SQL databases.
- **Data analytics**: It is important to perform data analytics and possibly near real-time analytics in MyHealthAvatar in order to support risk detection.
- **System scalability and performance**: Although the MyHealthAvatar is a proof-of-concept project, we foresee the future of such systems will manage large amount of user data with high demands on updating and reading. It is vital the system can scale and adapt the distributed computation paradigms (e.g. cloud computing) when the volume of the data grows.
- **Agility of the application development**: MyHealthAvatar aims to collect and access the lifelong health data for citizens. The uncertainty of the data types from different source centres requires that the development is done in iterations and be flexible and dynamic when new data resources are added.

3.3.2 Summary of Current NoSQL Databases

There are a number of existing literatures on the surveys of these implementations [3-6]. This section summarise the current NoSQL databases reviewed in deliverable D3.1 *user requirements* based on these existing literatures and benchmarks. Table 2 provides the overview based on the following aspects. We leave the table cell as empty if no clear and specific content is found in the regarding aspects.

- **Developer:** the company who supports the development of the database.
- **License:** the software license of the database.
- **Language:** the language used to develop the database.

- **Protocols:** communication protocols used for clients to access the database servers.
- **Data Model:** the type of the data model storage the database use, i.e. key value, column, document or graph.
- **File system:** the file system the database uses when storing the data on the disk.
- **Consistency:** the consistency model (e.g. eventual, strong) supported by the database. Eventual consistency means if no new updates are made to the given data item, eventually all accesses to that data item will return the last updated value. Strong consistency guarantees that if a write is successful, any read that happen after the write would return the latest value. Strong consistency can be achieved by controlling the reads and writes, such as consistency by write, consistency by read and consistency by quorum. In the case of consistency by write, all copies of data are written, then any one of these copies can be read to guarantee the strong consistency. In the case of consistency by read, only one copy of the data is written, and all copies are read then reconciled to get the latest version. For consistency by quorum, both writes and reads are performed to majority of copies; the workload between write and read is even.
- **Partitioning methods:** the partitioning methods the database use to save the file on the disk. Examples are consistent hashing and range based.
- **Transaction concept:** whether the database supports strong transaction control, such as ACID properties.
- **Concurrency control:** the mechanism the database uses to control concurrency; examples are multiversion concurrency control (MVCC), locking and transaction, and optimistic locking.
- **Replication:** the mechanisms are used when copying data among servers/clusters. Three methods are typically used in our surveyed database implementations: equal nodes, master/slave, multi-master. In equal nodes environment, all server nodes act in the same way and they do not hold information about others. Data can be accepted by any node and then are replicated to a number of other nodes according to the replication factor setting. In mater/slave nodes environment, nodes are grouped into the master cluster and the slave clusters; the master cluster normally receives the data and the copy them to slaves. Masters also hold information about slaves, such as tracking the data being replicated. In multi-master scenarios, data are stored by a group of computers and updated by any member of the group.
- **Query:** the query methods the database support, such as Java APIs, REST APIs, and query languages.
- **Secondary index:** whether the database supports secondary index.
- **MapReduce:** whether the database implementation supports the MapReduce parallel programming model [9], which is used in cluster and cloud environments.
- **Support language binding:** the programming language bindings especially useful for high volume applications.