Oula Paltto

INTEGRATING A SMART CITY DATA WAREHOUSE EFFICIENTLY WITH A CLOUD INFRASTRUCTURE

UNIVERSITY OF JYVÄSKYLÄ
DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS
2015

TIIVISTELMÄ

Paltto, Oula
Fiksun kaupungin tietovaraston integroiminen tehokkaasti pilvi-infrastruktuuriin
Jyväskylä: Jyväskylän yliopisto, 2015, 114 s.
Tietojärjestelmätiede, pro gradu -tutkielma
Ohjaajat: Tyrväinen, Pasi & Mazhelis, Oleksiy

Kankaan hanke on Jyväskylän kaupungin seuraavien vuosikymmenten tärkein aluekehityshanke. Kankaan alue muodostaa tulevaisuudessa fiksun kaupungin, mikä edellyttää muun muassa alueen tietovaraston toteuttamista. Ennen tietovaraston toteuttamista on kuitenkin tarpeen selvittää, miten fiksun kaupungin tietovarasto voidaan integroida tehokkaasti pilvi-infrastruktuuriin ylipäänsä, mikä oli tämän tutkimuksen päätutkimuskysymys. Tätä varten luotiin yleistettävä, teoreettinen viitekehys, jonka avulla voidaan vastata esimerkiksi tähän kysymykseen. Viitekehyksen avulla voidaan tulkita, että fiksu kaupunki vaatii pilvi-infrastruktuurilta ainakin saatavuutta, autonomisuutta, skaalattavuutta, suorituskykyä, yhteentoimivuutta, vikasietoisuutta, yksityisyyttä ja turvallisuutta sekä käyttäjien osallistamista ja kestävää kehitystä. Viitekehyksen käyttöä demonstroitiin valitsemalla Kankaan alueen tietovaraston tärkeimmät vaatimukset: suorituskyky ja skaalattavuus. Näistä vaatimuksista suorituskyky operationalisoitiin, minkä jälkeen kahden tietovaraston ohjelmistokandidaatin, Stardogin ja Neo4j:n, suorituskyky testattiin. Ne asennettiin Eucalyptus-pilveen ja luotiin suorituskykytesti, joka lisäsi ja kyseli tietoa niistä. Neo4j suoriutui suorituskykytestistä paremmin kuin Stardog. Stardogia ja Neo4j:tä vertailtiin myös subjektiivisesti, mikä toi esille muun muassa, että Neo4j on kypsempi tuote kuin Stardog mutta että molempia tietokantoja voidaan potentiaalisesti hyödyntää Kankaan hankkeessa. Lopuksi viitekehystä itseään arvioitiin, mikä kertoi, että se toimii ohjenuorana melko hyvin, joskin sillä on myös joitakin heikkouksia. Se ei esimerkiksi tarjoa teknisiä tietoja. Tutkimus toteutettiin suunnittelutieteellisesti.

Asiasanat: pilvilaskenta, fiksu kaupunki, Eucalyptus, NoSQL, graafitietokanta, Stardog, Neo4j.

ABSTRACT

Paltto, Oula
Integrating a smart city data warehouse efficiently with a cloud infrastructure
Jyväskylä: University of Jyväskylä, 2015, 114 pp.
Information systems science, master's thesis
Ohjaajat: Tyrväinen, Pasi & Mazhelis, Oleksiy

The Kangas project is the main urban development project of the City of Jyväskylä for the next several decades. The Kangas area will form a smart city in the future, which requires implementing, among others, the data warehouse of the area. Before implementing the data warehouse, however, there is a need to know how a smart city data warehouse can be efficiently integrated with a cloud infrastructure in general, which was the main research question of this study. To this end, a generalizable, theoretical framework was created that can be used to answer e.g., to this question. With the help of the framework, it can be interpreted that a smart city requires of a cloud infrastructure at least availability, autonomicity, scalability, performance, interoperability, fault tolerance, privacy, and security, as well as user involvement and sustainability. The use of the framework was demonstrated by choosing the most important requirements for the data warehouse of the Kangas area: performance and scalability. Of these requirements, performance was operationalized, after which two candidates for the software of the data warehouse, Stardog and Neo4j, were tested for it. They were installed on a Eucalyptus cloud and a benchmark was created that inserted data into and queried it from them. Neo4j performed better than Stardog in the benchmark. Stardog and Neo4j were compared subjectively as well, which brought out, among others, that Neo4j is a more mature product than Stardog, but that both databases can potentially be utilized in the Kangas project. Finally, the framework itself was evaluated, which revealed that it functions as a guiding principle quite well, although it has also some weaknesses. E.g., it offers no specifications. The study was conducted as design science.

Keywords: cloud computing, smart city, Eucalyptus, NoSQL, graph database, Stardog, Neo4j.

ACKNOWLEDGEMENTS

I would like to thank my supervisors, Pasi Tyrväinen and Oleksiy Mazhelis, for their expertise, ideas, suggestions, and patience that helped me greatly in writing this thesis. It provided a great learning experience for a content management systems enthusiastic who was almost destined to learn more about cloud computing, databases, Linux distributions, etc.

I would also like to acknowledge Tapani Tarvainen who helped me to get Eucalyptus working, as well as Matias Oksa who helped me to get Neo4j SPARQL Plugin working.

Thanks go also to Neo Technology's Davig Montag and Karl Sjöborg for providing me valuable information about Neo4j.

Many other people also helped me in one way or the other. I am much obliged to them for their help.

Oula Paltto
Jyväskylä, Finland
February 18, 2015

FIGURES

TABLES

CONTENTS

1 INTRODUCTION

The Kangas project is the main urban development project of the City of Jyväskylä for the next several decades. The Kangas area is introduced later on, but in brief, it will form a smart city in the future, being a home to 5000 inhabitants and offering 2000 new jobs. (Jyväskylän kaupunki, 2011.) This project requires implementing, but first, planning for many things. One of them is the data warehouse of the area. It was decided at the University of Jyväskylä that the data warehouse will be built on the cloud with the help of the university's hardware, network, and other resources, e.g., Eucalyptus cloud software. Hence, it can be said that many concepts and technologies are combined in the Kangas project including cloud computing, cloud data management, and smart cities. These are briefly characterized below, being discussed in more detail later on.

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models. (Mell & Grance, 2011.) These are elaborated further later on, but in a nutshell, cloud computing can be seen as a broad umbrella definition encompassing many kinds of technologies and services. This can also be said of *cloud data management* that is a somewhat vague concept, but in this thesis, it refers to the many ways of saving and managing data on the cloud. Exemplars of these are so-called NoSQL databases.

Smart city is a fuzzy concept as well. It can be conceptualized in many different ways, e.g., as Caragliu, Del Bo & Nijkamp (2009) according to which a city is smart when investments in human and social capital and traditional (transport) and modern (ICT) communication infrastructure fuel sustainable economic growth and a high quality of life, with a wise management of natural resources, through participatory governance.

In practice, smart cities produce enormous amounts of data that needs to be saved and managed somehow. As cloud computing provides at least in theory infinite amount of resources, it is a good candidate for such a task. *Elastic*

Utility Computing Architecture for Linking Your Programs to Useful Systems (Eucalyptus) (Wolski et al., 2008) will be utilized in the Kangas project. Eucalyptus is open source software for building AWS-compatible (Amazon Web Services) private and hybrid clouds (Eucalyptus Systems, 2014b).

Cloud software such as Eucalyptus is naturally only a platform onto which something can be build, e.g., the data warehouse of the Kangas area. A *data warehouse* refers to a system capable of supporting decision-making, receiving data from multiple operational data sources (Connolly & Begg, 2005). In this thesis, two candidates for the software of the data warehouse, Stardog and Neo4j, are introduced, benchmarked against each other, and compared subjectively as well.

This thesis represents *design science* that is fundamentally a problem-solving paradigm that creates and evaluates IT artifacts intended to solve identified organizational problems (Hevner, March, Park & Ram, 2004). Design science consists of two basic activities, building and evaluating. *Building* is the process of constructing an artifact for a specific purpose. *Evaluation* is the process of determining how well the artifact performs. (March & Smith, 1995.)

Before implementing the data warehouse of the Kangas area, there is a need to know how a smart city data warehouse can be efficiently integrated with a cloud infrastructure in general. This requires knowledge of the requirements for smart cities, especially their data management, and the requirements for cloud computing systems, especially their data management. In the research literature exist many such requirements, but there appears to be no generalizable framework that would integrate them with each other. It was thus realized that this kind of framework could be useful e.g., to researchers and decision-makers. Hence, the main objective of this study is to build such an *artifact* and answer with the help of it to the main research question:

- How a smart city data warehouse can be efficiently integrated with a cloud infrastructure?

Answering to the main research question requires answering to the sub-questions of this study as well. They form its sub-objectives:

- What is cloud computing?
- What is cloud data management?
- What are the requirements for cloud data management?
- What are smart cities?
- What are the requirements for smart city data management?

This part of the study is conducted as a literature review. The data, consisting of scholarly papers, books, websites, etc., was found with the help of Google, Google Scholar, Nelli portal, and the JYKDOK service of the Jyväskylä University Library.

The use of the framework is demonstrated by choosing the most important requirements for the data warehouse of the Kangas area: performance and scalability. Of these requirements, performance is operationalized, after which Stardog and Neo4j are tested for it. They are installed on Eucalyptus and a benchmark is built that inserts data into and queries it from the databases. The benchmark compares the performance of Stardog's public SPARQL endpoint (Clark & Parcia, 2014c) to Neo4j's Transactional Cypher HTTP endpoint (Neo Technology, 2014f). Then, Stardog and Neo4j are compared subjectively as well, and finally, based on all these experiences, the framework itself is evaluated.

This thesis is organized as follows. Chapter 2 is an introduction to cloud computing. It defines cloud computing and discusses its essential characteristics, service models, deployment models, and technologies. Chapter 3 covers cloud data management. It defines cloud data management, compares relational databases to NoSQL databases, and presents requirements for cloud data management. The chapter ends with the framework of requirements for cloud data management. Chapter 4 deals with smart cities and their data management. It discusses what smart cities and the Internet of Things (IoT) are, deals with enabling technologies of the IoT, and presents requirements for smart city data management. The chapter is crowned by the framework of requirements for integrating a smart city with a cloud infrastructure. Chapter 5 presents the research method of this study. It briefly introduces design science and then goes through the research process of the study, the central concepts of the study, and the benchmark for comparing the performance of Stardog and Neo4j. Chapter 6 presents the results of this benchmark and their analysis, the subjective comparison of Stardog and Neo4j, and the evaluation of the framework of requirements for integrating a smart city with a cloud infrastructure. Finally, chapter 7 summarizes the results and conclusions of the study, discussing subjects for further study as well.

2 CLOUD COMPUTING

This chapter is organized as follows. First, cloud computing is defined. Then, essential characteristics of cloud computing are presented. Next, cloud computing service and deployment models are dealt with. Finally, cloud computing technologies are discussed.

2.1 Definition of cloud computing

With the rapid development of processing and storage technologies and the success of the Internet, computing resources have become cheaper, more powerful, and more ubiquitously available than ever before. This technological trend has enabled the realization of a new computing model called *cloud computing* in which resources (e.g., CPU and storage) are provided as general utilities that can be leased and released by users through the Internet in an on-demand fashion. (Zhang, Cheng & Boutaba, 2010.)

The main idea behind cloud computing is not a new one (Zhang et al., 2010). According to Parkhill (1966, as cited in Zhang et al., 2010), John McCarthy envisioned already in the 1960s that computing facilities will be provided to the general public like a utility. The term *cloud* has also been used in various contexts, e.g., describing large asynchronous transfer mode (ATM) networks in the 1990s. However, after Google's CEO Eric Schmidt used the word to describe the business model of providing services across the Internet in 2006, the term really started to gain popularity. Since then, the term 'cloud computing' has been used mainly as a marketing term in a variety of contexts to represent many different ideas. (Zhang et al., 2010.)

The lack of a standard definition of cloud computing has generated not only market hypes, but also a fair amount of skepticism and confusion. For this reason, there has been work on standardizing the definition of cloud computing during the past years. (Zhang et al., 2010.) According to Vaquero, Rodero-Merino, Caceres, and Lindner (2009), cloud computing is associated with a new paradigm for the provision of computing infrastructure. This paradigm shifts the location of this infrastructure to the network to reduce the costs associated with the management of hardware and software resources (Vaquero et al., 2009; see also Hayes, 2008). However, the variety of technologies in the cloud makes the overall picture confusing (Hwang, 2008, as cited in Vaquero et al., 2009), and the hype around cloud computing further muddles the message (Geelan, 2008, as cited in Vaquero et al., 2009; Milojicic, 2008, as cited in Vaquero et al., 2009). According to Vaquero et al. (2009), clouds did not have a clear and complete definition in the literature at the time when they published their paper. Hence, they propose their definition of clouds: Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms, and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the infrastructure provider by means of customized service-level agreements (SLAs). (Vaquero et al., 2009.)

According to Armbrust et al. (2010), cloud computing is a popular topic for blogging and white papers and has been featured in the title of workshops, conferences, and even magazines. However, confusion remains about exactly what it is and when it is useful (Armbrust et al., 2010). According to Armbrust et al. (2010), cloud computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services. According to Armbrust et al. (2010), the services themselves have long been referred to as Software as a Service (SaaS). The data center hardware and software is what they call a 'cloud.' They mention that some vendors also use the terms IaaS (Infrastructure as a Service) and PaaS (Platform as a Service) to describe their products, but Armbrust et al. (2010) eschew them, noting that accepted definitions for them still vary widely (2010).

There are, indeed, many definitions of cloud computing, aforementioned being, in the author's opinion, some of the best. In this thesis, cloud computing is defined according to National Institute of Standards and Technology's (NIST) 16th and final working definition of cloud computing that has been, according to Brown (2011), the de facto definition of cloud computing a long time. According to NIST (Mell & Grance, 2011), cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models (Mell & Grance, 2011). These are discussed next.

2.2 Essential characteristics of cloud computing

According to NIST (Mell & Grance, 2011), the cloud model is composed of five essential characteristics:

On-demand self-service. A consumer can unilaterally provision computing capabilities, e.g., server time and network storage, as needed automatically without requiring human interaction with each service provider. (Mell & Grance, 2011.)

Broad network access. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations). (Mell & Grance, 2011.)

Resource pooling. The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources, but may be able to specify location at a higher level of abstraction (e.g., country, state, or data center). Examples of resources include storage, processing, memory, and network bandwidth. (Mell & Grance, 2011.)

Rapid elasticity. Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time. (Mell & Grance, 2011.)

Measured service. Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service. (Mell & Grance, 2011.)

Zhang et al. (2010) list similar characteristics. According to them (2010), cloud computing provides several salient features that are different from traditional service computing:

Multi-tenancy (see e.g., 'resource pooling' above). In a cloud environment, services owned by multiple providers are co-located in a single data center. The performance and management issues of these services are shared among service providers and the infrastructure provider. The layered architecture of cloud computing provides a natural division of responsibilities: the owner of each layer only needs to focus on the specific objectives associated with this layer. However, multi-tenancy also introduces difficulties in understanding and managing the interactions among various stakeholders. (Zhang et al., 2010.)

Shared resource pooling (see e.g., 'resource pooling' above). The infrastructure provider offers a pool of computing resources that can be dynamically assigned to multiple resource consumers. Such dynamic resource assignment capability provides much flexibility to infrastructure providers for managing their own resource usage and operating costs. (Zhang et al., 2010.)

Geo-distribution and ubiquitous network access (see e.g., 'broad network access' and 'resource pooling' above). Clouds are generally accessible through the Internet and use the Internet as a service delivery network. Hence, any device with Internet connectivity, be it a mobile phone, a personal digital assistant (PDA), or a laptop, is able to access cloud services. Additionally, to achieve high network performance and localization, many of today's clouds consist of data centers located at many locations around the world. A service provider can easily leverage geo-diversity to achieve maximum service utility. (Zhang et al., 2010.)

Service oriented (see e.g., 'on-demand self-service' and 'measured service' above). Cloud computing adopts a service-driven operating model. Hence, it places a strong emphasis on service management. In a cloud, each IaaS, PaaS, and SaaS provider offers its service according to the SLA negotiated with its customers. (Zhang et al., 2010.)

Dynamic resource provisioning (see e.g., 'rapid elasticity' above). One of the key features of cloud computing is that computing resources can be obtained and released on the fly. Compared to the traditional model that provisions resources according to peak demand, dynamic resource provisioning allows service providers to acquire resources based on the current demand, which can considerably lower the operating cost. (Zhang et al., 2010.)

Self-organizing (see e.g., 'rapid elasticity' above). Since resources can be allocated or deallocated on-demand, service providers are empowered to manage their resource consumption according to their own needs. In addition, the automated resource management feature yields high agility that enables service providers to respond quickly to rapid changes in service demand, e.g., the flash crowd effect. (Zhang et al., 2010.)

Utility-based pricing (see e.g., 'measured service' above). Cloud computing employs a pay-per-use pricing model. The exact pricing scheme may vary from service to service. Utility-based pricing lowers service operating cost as it charges customers on a per-use basis. However, it also introduces complexities in controlling the operating cost. (Zhang et al., 2010.)

2.3 Cloud computing service models

According to NIST (Mell & Grance, 2011), the cloud model is composed of three service models:

Software as a Service (SaaS). The capability provided to the consumer is to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based e-mail) or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings. (Mell & Grance, 2011.)

Platform as a Service (PaaS). The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment. (Mell & Grance, 2011.)

Infrastructure as a Service (IaaS). The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure, but has control over operating systems, storage, and deployed applications, and possibly limited control of select networking components, e.g., host firewalls. (Mell & Grance, 2011.)

Names of these three service models vary. E.g., Vaquero et al. (2009) discuss 'types of cloud systems' or 'scenarios where clouds are used' and their actors. According to them, many activities use software services as their business basis. These service providers (SPs) make services accessible to the service users through Internet-based interfaces. Clouds aim to outsource the provision of the computing infrastructure required to host services. This infrastructure is offered 'as a service' by infrastructure providers (IPs), moving computing resources from the SPs to the IPs, so the SPs can gain in flexibility and reduce costs. (Vaquero et al., 2009.)

In IaaS, IPs manage a large set of computing resources, e.g., storing and processing capacity. Through virtualization, they are able to split, assign, and dynamically resize these resources to build ad-hoc systems as demanded by customers, the SPs. They deploy the software stacks that run their services. PaaS denotes that cloud systems can offer an additional abstraction level. Instead of supplying a virtualized infrastructure, they can provide the software platform in which systems run on. The sizing of the hardware resources demanded by the execution of the services is made in a transparent manner. A well-known example is the Google App Engine. As for SaaS, there are services of potential interest to a wide variety of users hosted in cloud systems. This is an alternative to locally run applications. Examples of this are the online alternatives of typical office applications, e.g., word processors. (Vaquero et al., 2009.)

Zhang et al. (2010) define IaaS, PaaS, and SaaS as 'business models.' According to them, cloud computing employs a service-driven business model. Hardware- and platform-level resources are provided as services on an on-demand basis. Conceptually, every layer of the architecture can be implemented as a service to the layer above, and every layer can be perceived as a customer of the layer below, which is depicted in the figure 1. It is entirely possible that a PaaS provider runs its cloud on top of an IaaS provider's cloud, but in the current practice, IaaS and PaaS providers are often parts of the same organization, e.g., Google and Salesforce. In a cloud computing environment, the traditional role of a service provider is divided into two: infrastructure providers who manage cloud platforms and lease resources according to a usage-based pricing model, and service providers who rent resources from one or many infrastructure providers to serve the end-users. (Zhang et al., 2010.)

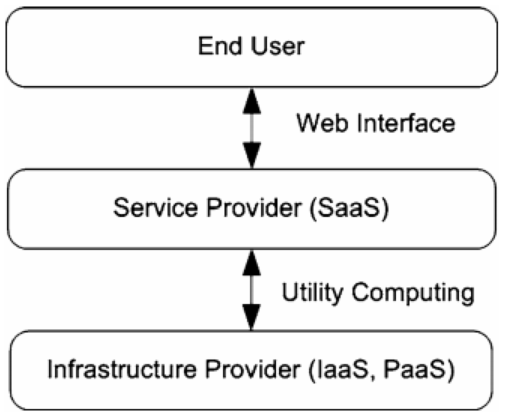

FIGURE 1 Business models of cloud computing (Zhang et al., 2010, 10)

2.4 Cloud computing deployment models

According to NIST (Mell & Grance, 2011), the cloud model is composed of four deployment models:

Private cloud. The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers, e.g., business units. It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on- or off-premises. (Mell & Grance, 2011.)

Community cloud. The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on- or off-premises. (Mell & Grance, 2011.)

Public cloud. The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider. (Mell & Grance, 2011.)

Hybrid cloud. The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability, e.g., cloud bursting for load balancing between clouds. (Mell & Grance, 2011.)

Concerning the aforementioned deployment models, *cloud bursting* is a technique used by hybrid clouds to provide additional resources to private clouds on an as-needed basis. If the private cloud has the processing power to handle its workloads, the hybrid cloud is not used. When workloads exceed the private cloud's capacity, the hybrid cloud automatically allocates additional resources to the private cloud. Hence, hybrid clouds offer e.g., more flexibility than both public and private clouds. (Sakr, Liu, Batista & Alomari, 2011.)

Zhang et al. (2010) discuss different 'types of clouds', i.e., the deployment models, in a similar fashion, each type of cloud having its own benefits and drawbacks:

Private clouds, also known as *internal clouds*, offer the highest degree of control over performance, reliability, and security. However, they are often criticized for being similar to traditional proprietary server farms and do not provide benefits, e.g., no up-front capital costs. (Zhang et al., 2010.)

Public clouds offer several key benefits to service providers including no initial capital investment on an infrastructure and shifting of risks to infrastructure providers. However, public clouds lack fine-grained control over data, as well as network and security settings, which hampers their effectiveness in many business scenarios. (Zhang et al., 2010.)

Hybrid clouds offer more flexibility than both public and private clouds. Specifically, they provide tighter control and security over application data compared to public clouds, while still facilitating on-demand service expansion and contraction. On the downside, designing a hybrid cloud requires carefully determining the best split between public and private cloud components. (Zhang et al., 2010.)

Zhang et al. (2010) do not mention NIST's (Mell & Grance, 2011) community cloud, but they do present a type of cloud that NIST's definition does not comprise, a *virtual private cloud (VPC)* that is an alternative solution to addressing the limitations of both public and private clouds. A VPC is essentially a platform running on top of public clouds. The main difference is that a VPC leverages virtual private network (VPN) technology that allows service providers to design their own topology and security settings, e.g., firewall rules. VPC is essentially a more holistic design, since it virtualizes servers, applications, and the underlying communication network as well. Additionally, for most companies, VPC provides seamless transition from a proprietary service infrastructure to a cloud-based infrastructure, owing to the virtualized network layer. (Zhang et al., 2010.)

In addition, there exists at least the concept of *federated cloud* that refers to an infrastructure in which competing clouds are able to cooperate to maximize their benefits (Ranjan, Buyya & Parashar, 2012). Rouvinen (2013) has compared the terms 'community cloud' and 'federated cloud' in an explicit way in his master's thesis. According to him, a community cloud is essentially a private cloud, in any case more or less closed by its nature, while a federated cloud can comprise both public and private clouds.

2.5 Cloud computing technologies

According to Zhang et al. (2010), cloud computing is often compared to the following technologies, each of which shares certain aspects with cloud computing:

Grid computing. Grid computing is a distributed computing paradigm that coordinates networked resources to achieve a common computational objective. The development of grid computing was originally driven by scientific applications that are usually computation-intensive. Cloud computing is similar to grid computing in that it also employs distributed resources to achieve application-level objectives. However, cloud computing takes one step further by leveraging virtualization technologies at multiple levels (hardware and application platform) to realize resource sharing and dynamic resource provisioning. (Zhang et al., 2010.)

Utility computing. Utility computing represents the model of providing resources on-demand and charging customers based on usage rather than a flat rate. Cloud computing can be perceived as a realization of utility computing. It adopts a utility-based pricing scheme entirely for economic reasons. With on-demand resource provisioning and utility-based pricing, service providers can truly maximize resource utilization and minimize their operating costs. (Zhang et al., 2010.)

Virtualization. Virtualization is a technology that abstracts away the details of physical hardware and provides virtualized resources for high-level applications. A virtualized server is commonly called a virtual machine (VM). Virtualization forms the foundation of cloud computing, as it provides the capability of pooling computing resources from clusters of servers and dynamically assigning or reassigning virtual resources to applications on-demand. (Zhang et al., 2010.)

Autonomic computing. Originally coined by IBM in 2001, autonomic computing aims at building computing systems capable of self-management, i.e., reacting to internal and external observations without human intervention. The goal of autonomic computing is to overcome the management complexity of today's computer systems. Although cloud computing exhibits certain autonomic features, e.g., automatic resource provisioning, its objective is to lower resources' cost rather than to reduce system complexity. (Zhang et al., 2010.)

Zhang et al. (2010) summarize that cloud computing leverages virtualization technology to achieve the goal of providing computing resources as a utility. It shares certain aspects with grid computing and autonomic computing, but differs from them in other aspects. Therefore, it offers unique benefits and imposes distinctive challenges to meet its requirements. (Zhang et al., 2010.)

Wang, Tao, Kunze, Castellanos, Kramer, and Karl (2008), and later on, Wang et al. (2010) list a number of enabling technologies contributing to cloud computing. Next, some technologies that have not been discussed so far are briefly presented:

Web services and SOA. Computing cloud services are normally exposed as web services that follow the industry standards, e.g., Web Service Description Language (WSDL), Simple Object Access Protocol (SOAP), and Universal Description Discovery and Integration (UDDI). The services organization and orchestration inside clouds could be managed in a service-oriented architecture (SOA). Furthermore, a set of cloud services could be used in a SOA application environment, thus making them available on various distributed platforms. They could be further accessed across the Internet. (Wang et al., 2010.)

Web 2.0. According to Wikipedia (2008, as cited in Wang et al., 2010), Web 2.0 is an emerging technology describing the innovative trends of using World Wide Web (WWW) technology and web design that aims to enhance creativity, information sharing, collaboration, and functionality of the web. The essential idea behind Web 2.0 is to improve the interconnectivity and interactivity of web applications. The new paradigm to develop and access web applications enables users to access the web more easily and efficiently. Cloud computing services are in nature web applications that render desirable computing services on-demand. (Wang et al., 2010.)

World-wide distributed storage system. A cloud storage model should foresee a network storage system that is backed by distributed storage providers, e.g., data centers, offering storage capacity for users to lease. The data storage could be migrated, merged, and managed transparently to end-users for whatever data formats. A cloud storage model should also foresee a distributed data system that provides data sources accessed in a semantic way. Users could locate data sources in a large distributed environment by the logical name instead of physical locations. (Wang et al., 2010.)

Programming model. Users drive into the computing cloud with data and applications. Some cloud programming models should be proposed for users to adapt to the cloud infrastructure. For the simplicity and easy access of cloud services, the cloud programming model should not, however, be too complex or too innovative for end-users. (Wang et al., 2010.) The MapReduce is a programming model and an associated implementation for processing and generating large data sets across the Google's worldwide infrastructures (Dean, 2007, as cited in Wang et al., 2010; Dean & Ghemawat, 2008, as cited in Wang et al., 2010). Hadoop is a framework for running applications on large clusters built of commodity hardware (Hadoop, 2008, as cited in Wang et al., 2010). It implements the MapReduce paradigm and provides a distributed file system, the Hadoop Distributed File System (Wang et al., 2010).

Related to these technologies, Zhang et al. (2010) present a layered model of cloud computing, i.e., the architecture of a cloud computing environment. It can be divided into four layers: the hardware / data center layer, the infrastructure layer, the platform layer, and the application layer. These are depicted in the figure 2:

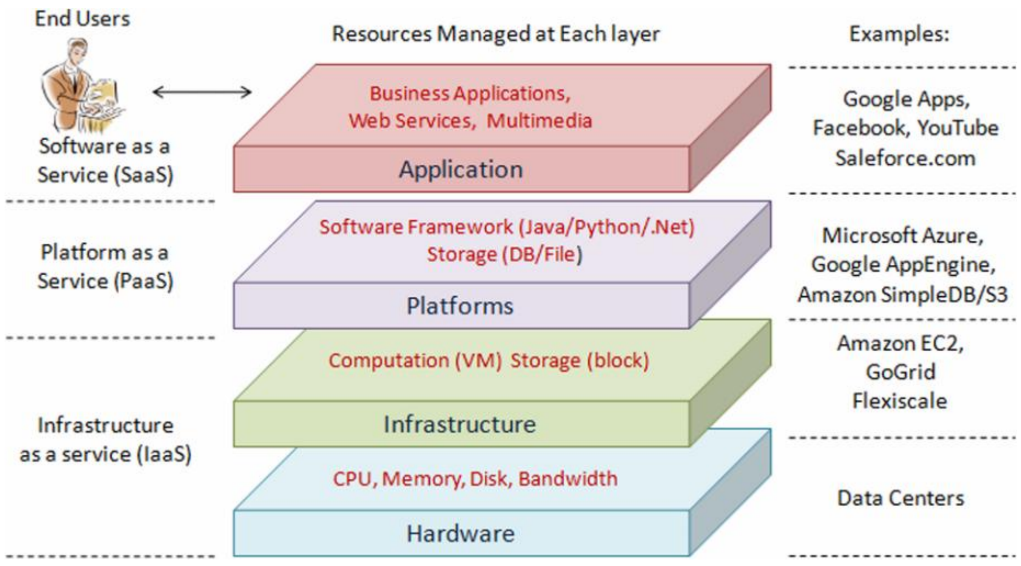
End Users
Resources Managed at Each layer
Examples:
Software as a Service (SaaS)
Business Applications, Web Services, Multimedia
Application
Google Apps, Facebook, YouTube Saleforce.com
Platform as a Service (PaaS)
Software Framework (Java/Python/.Net) Storage (DB/File)
Platforms
Microsoft Azure, Google AppEngine, Amazon SimpleDB/S3
Computation (VM) Storage (block)
Infrastructure
Amazon EC2, GoGrid Flexiscale
Infrastructure as a service (IaaS)
CPU, Memory, Disk, Bandwidth
Hardware
Data Centers

FIGURE 2 Cloud computing architecture (Zhang et al., 2010, 9)

The hardware layer. This layer is responsible for managing the physical resources of the cloud including physical servers, routers, switches, power, and cooling systems. The hardware layer is typically implemented in data centers. A data center usually contains thousands of servers that are organized in racks and interconnected through switches, routers, or other fabrics. Typical issues at hardware layer include hardware configuration, fault-tolerance, traffic management, power, and cooling resource management. (Zhang et al., 2010.)

The infrastructure layer. Also known as the virtualization layer, the infrastructure layer creates a pool of storage and computing resources by partitioning the physical resources using virtualization technologies, e.g., Xen, KVM, and VMware. The infrastructure layer is an essential component of cloud computing, since many key features, e.g., dynamic resource assignment, are only made available through virtualization technologies. (Zhang et al., 2010.)

The platform layer. Built on top of the infrastructure layer, the platform layer consists of operating systems and application frameworks. The purpose of the platform layer is to minimize the burden of deploying applications directly into VM containers. E.g., Google App Engine operates at the platform layer to provide application programming interface (API) support for implementing storage, database, and business logic of typical web applications. (Zhang et al., 2010.)

The application layer. At the highest level of the hierarchy, the application layer consists of the actual cloud applications. Different from traditional applications, cloud applications can leverage the automatic-scaling feature to achieve better performance, availability, and lower operating costs. (Zhang et al., 2010.)

According to Zhang et al. (2010), compared to traditional service hosting environments, e.g., dedicated server farms, the architecture of cloud computing is more modular. Each layer is loosely coupled with the layers above and below, allowing each layer to evolve separately. This is similar to the design of the

Open Systems Interconnection (OSI) model for network protocols. The architectural modularity allows cloud computing to support a wide range of application requirements, while reducing management and maintenance overhead. (Zhang et al., 2010.)

3 CLOUD DATA MANAGEMENT

This chapter is organized as follows. First, cloud data management and the central concepts related to it are defined. Then, relational databases are briefly compared to NoSQL databases. Next, requirements for cloud data management are discussed. Finally, a framework of requirements for cloud data management is presented.

3.1 Definition of cloud data management

As cloud computing is a broad umbrella definition encompassing many kinds of technologies and services, so is cloud data management as well. Before going into what cloud data management is, it is useful to define some general concepts of data management:

A *database* is a shared collection of logically related data, and a description of this data, designed to meet the information needs of an organization. A *database management system (DBMS)* is a software system that enables users to define, create, maintain, and control access to a database. A DBMS allows users to define the structure of a database, a *schema*, through its data definition language (DDL). A higher-level description of a schema is called a *data model*. A DBMS allows users also to insert, update, delete, and retrieve data from a database, usually through a data manipulation language (DML). A DML provides a general inquiry facility to the data of a database, called a *query language*. The most common query language is the *Structured Query Language (SQL)* that is both the formal and de facto standard language for *relational database management systems (RDBMSs)*. (Connolly & Begg, 2005.) As SQL and RDBMs go hand in hand, relational databases are also called *SQL* or *MySQL databases*. Relational databases are defined later on.

In practice, a database runs on a server. A *database server* refers in this thesis to a computer that is dedicated to running a computer program that provides database services to other computer programs or computers (Wikipedia, 2014a). A *data warehouse* refers to a system capable of supporting decision-making, receiving data from multiple operational data sources (Connolly & Begg, 2005). In this thesis, a data warehouse is defined as a single repository into which users can easily insert data, from which they can easily run queries, and from which they can also produce reports and perform analysis if needed (cf. Connolly's & Begg's definition of the ultimate goal of data warehousing, 2005).

Cloud data management is a somewhat vague concept, but in this thesis, it refers to the many ways of saving and managing data in the cloud. To define the concept briefly, e.g., Wang et al. (2010), as already mentioned, write about worldwide distributed storage system as one of the enabling technologies behind cloud computing. Boles (2008) offers a technical, yet still quite clear description of cloud-based storage and its evolution, depicted in the figure 3.

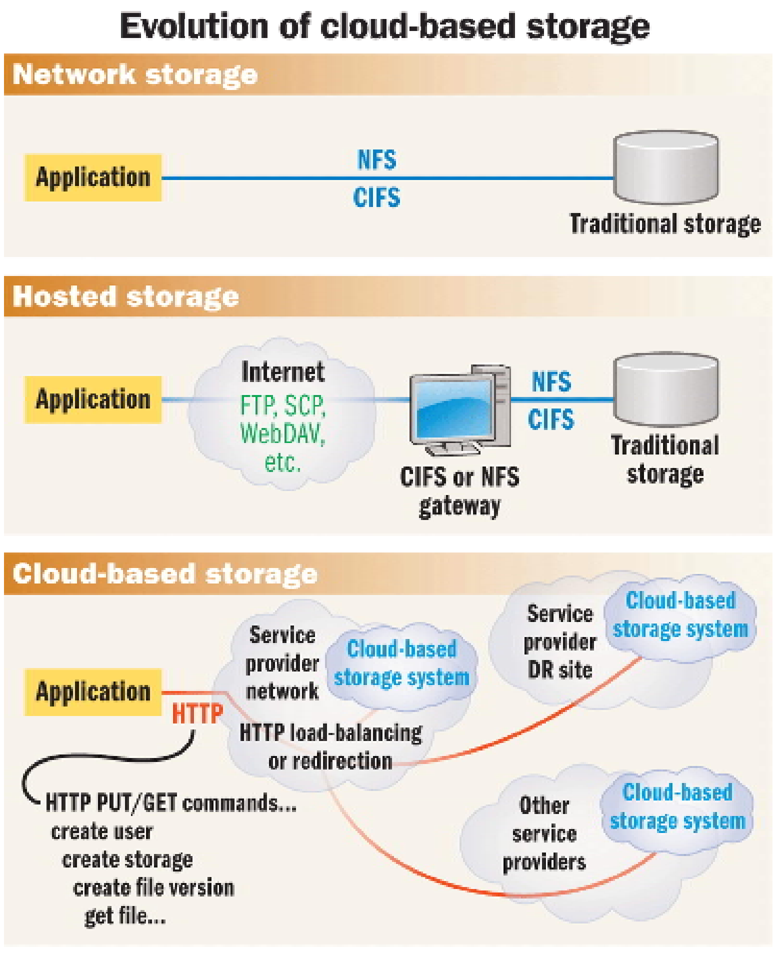

FIGURE 3 Evolution of cloud-based storage (Boles, 2008)

According to Boles (2008), simply put, storage in the cloud de-couples storage and applications, so that access to either one can be more flexible, and data storage and applications can easily scale in response to changing user demands. The industry has long been struggling with de-coupling applications from data so that each can be more flexibly managed, moved, and scaled. Network File System (NFS) and Common Internet File System (CIFS) were among the earliest ways of de-coupling applications and storage so that each could be scaled and managed more effectively. However, these protocols are complex and remain restricted to the data center in which resources can be expensive and difficult to scale. (Boles, 2008.)

The next evolution of de-coupling was to host application and data components with service providers across the web. Unfortunately, this generation of storage was often mired in the restricted scalability and complex access of traditional remote access protocols (File Transfer Protocol, FTP, Web-based Distributed Authoring and Versioning, WebDAV) and traditional storage (file and/or block). (Boles, 2008.) File-level storage refers to a storage technology that is most commonly used in storage systems that are found in hard drives, Network-Attached Storage (NAS) systems, etc. In file-level storage, the storage disk is configured with a protocol, e.g., NFS or Server Message Block (SMB) / CIFS, and the files are stored and accessed from it in bulk. In block-level storage, raw volumes of storage are created, and each block can be controlled as an individual hard drive. These blocks are controlled by server-based operating systems, and each block can be individually formatted with the required file system. (StoneFly, 2014.)

Cloud-based technology wraps traditional IT applications and infrastructure in new, simplified APIs and access semantics. APIs, or sets of application and/or storage commands, are served up as self-contained, discoverable web services that are accessed via Hypertext Transfer Protocol (HTTP) or other protocols and integrated into lightweight, easy to develop, distributed applications. This allows users to put less effort into developing complex application subroutines, and instead better serve their businesses with combinations of already available and reusable web services and data. In turn, the increased independence of these services allows each component to scale up and down in performance as end-user demands change. When distributed onto the enormous data centers of one or multiple service providers, this makes the infrastructure truly elastic. (Boles, 2008.)

Wu, Ping, Ge, Wang, and Fu (2010) mention Boles' (2008) evolution of cloud-based storage writing about four scenarios in which clouds are used. They are the aforementioned cloud service models SaaS, PaaS, and IaaS, but in addition to them Wu et al. (2010) mention *Storage as a Service (StaaS)* that facilitates cloud applications to scale beyond their limited servers. StaaS allows users to store their data at remote disks and access them anytime from any place. However, according to Wu et al. (2010), cloud storage is amorphous today, with neither a clearly defined set of capabilities nor any single architecture. Choices abound, with many traditional hosted or managed service providers (MSPs) offering block or file storage, usually alongside traditional remote access protocols, or virtual or physical server hosting. Other solutions have emerged, typified by Amazon Simple Storage Service that resembles flat databases designed to store large objects. (Wu et al., 2010.)

Boles' (2008) evolution of cloud-based storage is also mentioned in Kulkarni's, Waghmare's, Palwe's, Waykule's, Bankar's, and Koli's (2012) paper. Leaning on Storage Networking Industry Association (2009) and Curino et al. (2010), they note that cloud storage is a service model in which data is maintained, managed, and backed up remotely and made available to users over a network (typically the Internet) and that cloud storage is still amorphous (Kulkarni et al., 2012).

Arora and Gupta (2012) define some of the central concepts related to cloud data management. According to them, the different terms used for data management in the cloud differ on the basis of how data is stored and managed. *Cloud storage* is virtual storage that enables users to store documents and objects. *Data as Service (DaaS)* allows user to store data at a remote disk available through the Internet. It is used mainly for backup purposes and basic data management. Cloud storage cannot work without basic data management services, so these two terms are used interchangeably. However, *Database as a Service (DBaaS)* is one step ahead. It offers complete database functionality and allows users to access and store their database at remote disks anytime from any place through the Internet. Cloud database is a database delivered to users on-demand through the Internet from a cloud database provider's servers. While conventional DBMSs deal with structured data that is held in databases along with its metadata, cloud databases can be used for unstructured, semi-structured, or structured data. (Arora & Gupta, 2012.)

According to Dewan and Hansdah (2011), there exist at least five *cloud storage types*: unstructured data, structured data, message queues, block devices, and RDBMSs. *Unstructured* type is similar to traditional files, but has a support for accommodating large data set besides ensuring reliability and availability. A good example of unstructured storage type is Amazon Simple Storage Service. *Structured* types are non-relational data type. They are multi-dimensional data structures and designed in such a way that faster look up and access is possible. In addition, unlike relational database systems, they do not support joins and SQL queries. (Dewan & Hansdah, 2011.) In certain contexts, they can also be referred to as Non-SQL databases (Dewan & Hansdah, 2011), i.e., NoSQL databases. An example of structured storage type is Amazon SimpleDB. *Message queues* are temporary storage structures that are meant for storing messages passed between cloud application processes. *Block devices* are like traditional secondary storage media, a raw sequential order of bytes, which cloud applications can format as per their requirements of file system types. *RDBMS* store is a port of traditional RDBMS in the cloud. In RDBMS type storage, cloud applications can use SQL server instances hosted in the cloud infrastructure as if they were hosted in traditional servers. (Dewan & Hansdah, 2011.)

As for traditional databases, relational databases have been around for many years and have become the predominant choice in storing data (Wikipedia, 2014b). Next, relational databases and popular cloud databases, so-called NoSQL databases, are introduced and compared to each other.

3.2 Relational databases vs. NoSQL databases

Edgar Codd, a former IBM Fellow, is generally credited with creating the relational-database model in 1970 (Leavitt, 2010). A *relational database* is a set of tables (relations) containing data fitted into predefined categories (Leavitt, 2010; see also Connolly & Begg, 2005). Each table contains one or more data categories in columns. Each row contains a unique instance of data for the categories defined by the columns. Users can access or reassemble the data in different ways without having to reorganize the database tables. Relational databases work best with structured data, e.g., a set of sales figures that readily fits in well-organized tables. This is not the case with unstructured data, e.g., that found in word-processing documents and images. Partly in response to the growing awareness of relational databases' limitations, vendors and users are increasingly turning to NoSQL databases. (Leavitt, 2010.)

Defining what a *NoSQL database* is is not that simple. According to Pokorny (2013), the term 'NoSQL database' was chosen for a loosely specified class of non-relational data stores. Such databases (mostly) do not use SQL as their query language. The term 'NoSQL' is therefore confusing and is interpreted in the database community rather as 'not only SQL.' (Pokorny, 2013.) NoSQL can also be 'not relational' (Arora & Gupta, 2012) or 'postrelational' (Pokorny, 2013). These concepts sound like something new, but according to Leavitt (2010), non-relational databases including hierarchical, graph, and object-oriented databases have been around since the late 1960s.

The easiest way to differentiate between relational databases and NoSQL databases is to let the NoSQL data models speak for themselves, as the relational data model above. According to Leavitt (2010) and Pokorny (2013), there are three popular types of NoSQL databases: key-value stores, column-oriented databases, and document-based stores. Most simple NoSQL databases called *key-value stores* (or big hash tables) contain a set of couples (key, value). A key is in principle the same as an attribute in relational databases or a column name in SQL databases. In other words, a database is a set of named values. A key uniquely identifies a value (typically a string, but also a pointer to a place in which the value is stored), and this value can be structured or completely unstructured. In a more complex case, a NoSQL database stores combinations of couples (key, value) collected into collections. These are *column-oriented databases*. Some of these databases are composed of collections of couples (key, value) or, more generally, they look like semi-structured documents or extendable records often equipped by indexes. New attributes (columns) can be added to these collections. (Pokorny, 2013.) Finally, *document-based stores* are databases that store and organize data as collections of documents, rather than as structured tables with uniform-sized fields for each record. With these databases, users can add any number of fields of any length to a document. (Leavitt, 2010.)

Although relational databases have been around a long time, they are not perfect. Leavitt (2010) discusses some of their limitations:

Scaling. Users can scale a relational database by running it on a more powerful and expensive computer. To scale beyond a certain point though, it must be distributed across multiple servers. However, relational databases do not work easily in a distributed manner, because joining their tables across a distributed system is difficult. Also, relational databases are not designed to function with data partitioning, so distributing their functionality is a chore. (Leavitt, 2010.)

Complexity. With relational databases, users have to convert all data into tables. When the data does not fit easily into a table, the database's structure can be complex, difficult, and slow to work with. (Leavitt, 2010.)

SQL. Using SQL is convenient with structured data. However, using the language with other types of information is difficult, because it is designed to work with structured, relationally organized databases with fixed table information. SQL can entail large amounts of complex code and does not work well with modern, agile development. (Leavitt, 2010.)

Large feature set. Relational databases offer a big feature set and data integrity. However, NoSQL proponents say that database users often do not need all the features, as well as the cost and complexity they add. (Leavitt, 2010.)

NoSQL databases generally process data faster than relational databases. This stems from the fact that relational databases are usually used by businesses and often for transactions that require great precision, so they generally subject all data to the same set of *atomicity, consistency, isolation, durability (ACID)* restraints. (Leavitt, 2010.) Atomicity means that an update is performed completely or not at all (all or nothing). Consistency denotes that no part of a transaction will be allowed to break a database's rules (the result of each transaction is tables with legal data). Isolation refers to each application running transactions independently of other applications operating concurrently (transactions are independent). Durability indicates that completed transactions will persist (a database survives system failures). (Leavitt, 2010; Pokorny, 2013.) A database consistency is called in this sense strong consistency (Pokorny, 2013).

In practice, relational databases have always been fully ACID-compliant (Pokorny, 2013). However, having to perform these restraints on every piece of data makes relational databases slower. As for NoSQL databases, developers usually do not have their NoSQL databases support ACID in order to increase performance. This can cause problems when used for applications that require great precision. NoSQL databases are also often faster, because their data models are simpler. Because NoSQL databases do not have all the technical requirements that relational databases have, proponents say, most major NoSQL systems are flexible enough to better enable developers to use the applications in ways that meet their needs. (Leavitt, 2010.)

In contrast to ACID guarantees, NoSQL databases follow *basically available, soft state, eventually consistent (BASE)* guarantees (Arora & Gupta, 2012). An application works basically all the time (basically available), does not have to be consistent all the time (soft state), but the storage system guarantees that if no new updates are made to the object eventually (after the inconsistency window closes), all accesses will return the last updated value (Pokorny, 2013).

Databases that do not implement ACID fully can be only eventually consistent. In principle, if some consistency is given up, more availability can be gain and scalability of the database can be greatly improved. In contrast to ACID properties, there exists so-called *CAP theorem*, also called Brewer's theorem. It is a triple of requirements including consistency (C), availability (A), and partitioning tolerance (P). The CAP theorem states that for any system sharing data it is impossible to guarantee simultaneously all of these three properties. Particularly, in web applications based on horizontal scaling strategy, it is necessary to decide between C and A. Usually DBMSs prefer C over A and P. (Pokorny, 2013.)

As mentioned above, relational databases are not flawless. Neither are NoSQL databases. Leavitt (2010) also discusses their disadvantages or challenges:

Overhead and complexity. Because NoSQL databases do not work with SQL, they require manual query programming that can be fast for simple tasks but time-consuming for others. In addition, complex query programming for the databases can be difficult. (Leavitt, 2010.)

Reliability. Relational databases natively support ACID, while NoSQL databases do not. Hence, NoSQL databases do not natively offer the degree of reliability that ACID provides. If users want NoSQL databases to apply ACID restraints to a data set, they must perform additional programming. (Leavitt, 2010.)

Consistency. Because NoSQL databases do not natively support ACID transactions, they could also compromise consistency, unless manual support is provided. Not providing consistency enables better performance and scalability, but it is a problem for certain types of applications and transactions, e.g., those involved in banking. (Leavitt, 2010.)

Unfamiliarity with the technology. Most organizations are unfamiliar with NoSQL databases and thus may not feel knowledgeable enough to choose one or even to determine that the approach might be better for their purposes. (Leavitt, 2010.)

Limited ecostructure. Unlike commercial relational databases, many open source NoSQL applications do not yet come with customer support or management tools. (Leavitt, 2010.)

3.3 Requirements for cloud data management

Next, requirements for cloud data management are discussed. The literature sources are organized so that the more general requirements are presented first and the more specific later on. The reason for presenting general requirements for cloud computing systems is that a cloud data management system is always a part of some larger cloud computing system. They cannot be separated from each other. After the requirements are discussed, a framework of requirements for cloud data management is presented.

3.3.1 Important architectural requirements for cloud computing systems

Rimal, Jukan, Katsaros, and Goeleven (2011) consider important architectural requirements for cloud computing systems. These architectural requirements are classified according to the requirements of cloud providers, enterprises that use the cloud, and end-users. The three-layered classification of the architectural requirements of cloud systems is depicted in the figure 4. Next, these architectural requirements are discussed one at a time beginning from the provider requirements and ending to the user requirements.

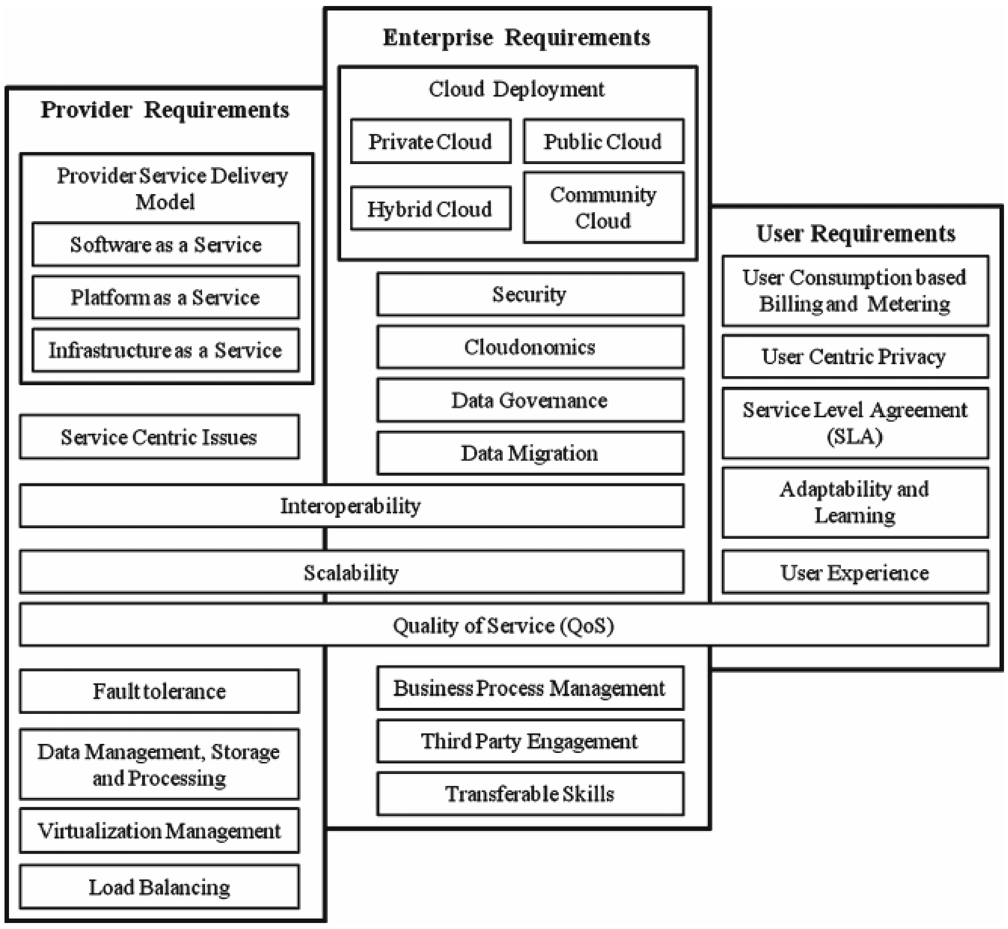

FIGURE 4 Three layered architectural requirements (Rimal et al., 2011, 6)

The provider service delivery model. As already discussed, three service delivery models can be considered in cloud systems: SaaS, PaaS, and IaaS. (Rimal et al., 2011.) They all have their advantages and disadvantages, but as they have already been discussed to some detail, they are not gone into here.

Service-centric issues. Cloud computing as a service needs to respond to real-world requirements of an enterprise's IT management. To fulfill the requirements of an enterprise's IT management, cloud architecture needs to deal with unified service-centric approach, e.g.,: Cloud services should be autonomic. Cloud systems/applications should be designed to adapt dynamically to changes in the environment with less human assistantship. Autonomic behavior of services can be used to improve the quality of services, fault-tolerance, and security. Furthermore, cloud services should be self-describing. Self-describing service interfaces can depict the contained information and functionality as reusable and context-independent way. The underlying implementation of a service can be changed simultaneously without reconfigurations when the service contract is updated. In addition, the cost composition of distributed applications should be low. (Rimal et al., 2011.)

Interoperability. Interoperability focuses on the creation of an agreed-upon framework/ontology, open data formats, or open protocols/APIs that enable easy migration and integration of applications and data between different cloud service providers and facilitates secure information exchange across platforms. For enterprises, it is important to provide interoperability between enterprise clouds and cloud service providers. (Rimal et al., 2011.)

Quality of Service (*QoS*). In general, QoS provides the guarantee of performance and availability, as well as other aspects of service quality, e.g., security, reliability, dependability, etc. SLAs play a key facilitator role to make agreed-upon QoS between service providers and end-users. (Rimal et al., 2011.)

Fault tolerance. Fault tolerance enables the systems to continue operating in the event of the failure of some of their components. In general, fault tolerance requires fault isolation to falling components, availability of reversion mode, etc. Fault-tolerant systems are characterized in terms of outages. (Rimal et al., 2011.)

Data management, storage, and processing. Data will be replicated across large geographic distances in which its availability and durability is paramount for cloud service providers. If the data is stored at untrusted hosts that can create enormous risks for data privacy. Furthermore, the cloud computing providers must ensure that the storage infrastructure is capable of providing rich query languages that are based on simple data structures to allow for scale-up and scale-down on-demand. In addition, the providers need to offer performance guarantees with the potential to allow the programmer some form of control over the storage procedures. (Rimal et al., 2011.)

In terms of storage technologies, there should be a shift from hard disk drives (HDDs) to solid-state drives (SSDs) (Graefe, 2007, as cited in Rimal et al., 2011; Lee & Kim, 2007, as cited in Rimal et al., 2011) or, since the complete replacement of hard disks is prohibitively expensive, the design of hybrid hard disks, i.e., hard disks augmented with flash memories (Lim et al., 2009, as cited in Rimal et al., 2011), as the latter provide reliable and high performance data storage. As for energy consumption, SSDs consume less power in idle state than HDDs. In addition, the programming model of data centers supported by the current (2011) industry giants, i.e., MapReduce, is not a perfect fit for all tasks.

Towards this direction, new languages and systems must be developed to realize hybrid designs among DBMSs and MapReduce-like systems. (Rimal et al., 2011.)

Virtualization management. Virtualization refers to the abstraction of logical resources away from their underlying physical resources in order to improve agility and flexibility, reduce costs, and thus enhance business value (Golden, 2008, as cited in Rimal et al., 2011). Handling a number of virtualization machines on top of operating systems and evaluating, testing servers, and deployment to the targets are some of the important concerns of virtualization. Virtualization in the cloud takes many forms, e.g., server, storage, and infrastructure virtualization. (Rimal et al., 2011.)

Scalability. Scalability deals with the ability of a software system to manage increasing complexity when given additional resources. Scalability with large data set operations is a requirement for cloud computing. Horizontal scalability is what clouds provide through load balancing and application delivery solutions. Distributed hash table (DHT), column-orientation, and horizontal partitioning are examples of horizontal scalability. Vertical scalability is related to resources used, much like the old mainframe model. (Rimal et al., 2011.)

Load balancing. Load balancing is an integral part of cloud computing and elastic scalability, which can be provided by software, hardware, or virtualware. It is the mechanism of self-regulating the workloads properly within the cloud's entities (one or more servers, hard drives, network, and IT resources). The cloud infrastructures and data centers need huge computing hardware, network, and IT resources that are always subjected to failover when the demand exceeds. Load balancing is often used to implement failover. (Rimal et al., 2011.)

Cloud deployment for enterprises. The cloud services are ubiquitous as a single point of access with four types of deployment models: public, private, community, and hybrid clouds. (Rimal et al., 2011.) They and their advantages and disadvantages are not, however, gone into here, as they have already been discussed earlier to some detail.

Security. Usually security is the focal concern in terms of data, infrastructure, and virtualization. In cloud computing, a data center holds the information that would more traditionally have been stored on the end-user's computer. This raises concerns regarding users' privacy protection, since the users do not 'own' their data. Furthermore, the move to centralized services may affect the privacy and security of users' interactions. Security threats may happen in resource provisioning and during distributed execution of user applications. In addition, new threats are likely to emerge. E.g., hackers can use the virtualized infrastructure as a launching pad for new attacks. (Rimal et al., 2011.)

Cloudonomics. The economics of cloud computing is called cloudonomics (Weinman, 2009, as cited in Rimal et al., 2011). The problem with cloud computing is the lack of cost-based transparency. It is actually very difficult to quantify the cost benefits of using traditional infrastructure vs. using remote service providers, e.g., Amazon Elastic Compute Cloud. (Rimal et al., 2011.)

Data governance. Geographical and political issues are the key requirements for an enterprise cloud. When data begins to move out of organizations, it is vulnerable to disclosure or loss. The act of moving sensitive data outside organizational boundaries may violate national regulations for privacy. Furthermore, due to the lack of interoperability among cloud platforms and the lack of standardization efforts, cloud providers cannot guarantee that a cloud user can move his data/programs to another cloud provider on-demand. Cloud computing became much more appealing if protection against data lock-in would be fully implemented. E.g., it would liberate the users from possible monopolies and guarantee the longevity of the users, since they would not be afraid of cloud providers going out of business. (Rimal et al., 2011.)

Data migration. The issue of distributing information to web users in an efficient and cost-effective manner is a challenging problem, especially under the increasing requirements emerging from a variety of modern applications, e.g., voice-over-IP and streaming media. (Rimal et al., 2011.) Content distribution networks (CDNs), e.g., Akamai, have met these challenges by providing a scalable and cost-effective mechanism for accelerating the delivery of web content, based on more or less sophisticated data migration (outsourcing) policies for the surrogate servers of a CDN (Katsaros et al., 2009, as cited in Rimal et al., 2011). To collectively address many goals, data replication in the cloud seems to be the most convenient approach (Rimal et al., 2011).

Business process management (*BPM*). Business process management systems provide a business structure, security, and consistent rules across business processes, users, organization, and territory. This classical concept is enhanced in the context of the cloud-based BPM, as cloud delivers a business operating platform for enterprises, such as combining SaaS and BPM applications, e.g., customer relationship management (CRM), enterprise resource planning (ERP), e-commerce portals, etc., which helps for the flexibility, deployability, and affordability for complex enterprise applications. When the enterprises adopt cloud-based services or business processes, the return of investment (ROI) of overall business measurement is important. (Rimal et al., 2011.)

Third party engagement. The involvement of a third party in enterprises can help for establishing a robust communication plan with a provider landscape, continuity of cloud service engagements, legal implications, potential intellectual property, cloud audit, reporting capabilities, etc. (Rimal et al., 2011.)

Transferable skills. Transferable skills deals with technology dissemination, technical supports, discussion with consulting expert groups, or offshore outsourcing that help for the adaptation and stability of systems/applications. Cloud computing comes with its own set of management tasks that need to be executed by the enterprise staff. The staff needs e.g., to monitor current computing capacity, and to increase or decrease it depending on usage. Before choosing a cloud provider system, the enterprise should have a look at the skill set of its existing workforce to identify those skills that are transferable to the new environment in order to make the transition as fluent as possible, because there is a wide variation in maturity of enterprise cloud software and services. (Rimal et al., 2011.)

User consumption-based billing and metering. The individual end-user consumption-based billing and metering in cloud systems is similar to the consumption measurement and allocation of costs of water, gas, and electricity consumption on a consumption unit basis. Cost management is important for planning and controlling decisions. It helps to check the utilized resources vs. the cost. Cost breakdown analysis, tracing the utilized activity, and adaptive cost management are important considerations as well. (Rimal et al., 2011.)

User-centric privacy. The main consideration regarding cloud computing for end-users is related to the storage of personal/enterprise sensitive data. Cloud computing brings with it the fact that most of the users' creations, data that the user would regard as his personal intellectual property, will be stored at mega data centers around the world. (Rimal et al., 2011.) In this environment, privacy becomes a major issue (Cavoukian, 2008, as cited in Rimal et al., 2011).

Service-level agreements (*SLAs*). The mutual contract between providers and users is usually called a SLA, i.e., the ability to deliver services according to predefined agreements. Many cloud providers offer SLAs, but the problem with them is that they are rather weak on user compensations on outages. (Rimal et al., 2011.)

Adaptability and learning. Cloud infrastructure must handle more resources, data, services, and users. All of this makes cloud-based enterprise application/systems more complex to control, to keep coherence between services and resources. The biggest challenge for every user is to get acquainted with applications presented by enterprises when trying to deal with clouds. (Rimal et al., 2011.)

User experience (*UX*). The notion of UX is to provide the insight into the needs and behaviors of end-users that can help to maximize the usability, desirability, and productivity of applications. UX-driven design and deployment may be the next step in the evolution of cloud computing. (Rimal et al., 2011.)

3.3.2 Cloud storage infrastructure requirements

Wu, Zhang, Lin, and Ju (2010) discuss in their paper cloud storage infrastructure requirements. According to them, there are ten critical common denominators that must be considered to make cloud storage valuable:

Elasticity. Cloud storage must be elastic to rapidly adjust the underlying infrastructure to changing subscriber demands and comply with SLAs. (Wu et al., 2010.)

Automatic. Cloud storage must have the ability to be automated so that policies can be leveraged to make underlying infrastructure changes, e.g., placing user and content management in different storage tiers and geographic locations quickly and without human intervention. (Wu et al., 2010.)

Scalability. Cloud storage needs to scale quickly and to tremendous capacities. This translates into scalability across objects, performance, users, clients, and capacity with a single namespace across all storage capacity being critical for low operating expense (OPEX) reasons. (Wu et al., 2010.)

Data Security. For private clouds, security is assumed to be tightly controlled. For public clouds, data should either be stored on a partition of a shared storage system or cloud storage providers must establish multi-tenancy policies to allow multiple business units or separate companies to securely share the same storage hardware. (Wu et al., 2010.)

Performance. A proven storage infrastructure providing fast, robust data recovery is an essential element of a cloud service. (Wu et al., 2010.)

Reliability. Enterprise users also want to make sure that their data is reliably backed up for disaster recovery purposes and that it meets pertinent compliance guidelines. (Wu et al., 2010.)

Ease of management. The need for improved manageability in the face of exploring storage capability and costs is a major benefit that enterprises are expecting from a cloud storage deployment. (Wu et al., 2010.)

Ease of data access. Ease of access to data in the cloud is critical in enabling seamless integration of cloud storage into existing enterprise workflows and to minimize the learning curve for a cloud storage adoption. (Wu et al., 2010.)

Energy efficiency. IT data centers are growing bottlenecks and approaching ceilings on available power, cooling, and flooring space. Green storage technology is the technology that enables energy efficiency and waste reduction in storage solutions leading to an overall lower carbon footprint. (Wu et al., 2010.)

Latency. Not all applications are suitable for a cloud storage model. It is important to measure and test network latency before committing to a migration. Virtual machines can introduce additional latency through the time-sharing nature of underlying hardware, and unanticipated sharing and reallocation of machines can significantly affect run times. (Wu et al., 2010.)

3.3.3 Successful cloud data management systems' wish list

Sakr et al. (2011) bring together Abouzeid's, Bajda-Pawlikowski's, Abadi's, Silberschatz's, and Rasin's (2009), as well as Cooper et al.'s (2009) cloud requirements comprising a list of features that successful cloud data management systems should have:

Availability. They have to be always accessible even on the occasions in which there is a network failure or a whole data center has gone offline. (Sakr et al., 2011; see also Cooper et al., 2009.)

Scalability. They have to be able to support very large databases with very high request rates at very low latency. They should be able to take on new tenants or handle growing tenants without much effort beyond that of adding more hardware. In particular, the system has to be able to automatically redistribute data to take advantage of the new hardware. (Sakr et al., 2011; see also Cooper et al., 2009.)

Elasticity. They have to be able to satisfy changing application requirements in both directions (scaling up or scaling down). Moreover, the system has to be able to gracefully respond to these changing requirements and quickly recover to its steady state. (Sakr et al., 2011; see also Cooper et al., 2009.)

Performance. On public cloud computing platforms, pricing is structured in such a way that one pays only for what one uses, so the vendor price increases linearly with the requisite storage, network bandwidth, and compute power. Hence, the system performance has a direct effect on its costs. Thus, efficient system performance is a crucial requirement to save money. (Sakr et al., 2011; see also Abouzeid et al., 2009.)

Multitenancy. They have to be able to support many applications (tenants) on the same hardware and software infrastructure. However, the performance of these tenants has to be isolated from each another. Adding a new tenant should require little or no effort beyond that of ensuring that enough system capacity has been provisioned for the new load. (Sakr et al., 2011; see also Cooper et al., 2009.)

Load and tenant balancing. They have to be able to automatically move load between servers so that most of the hardware resources are effectively utilized and to avoid any resource overloading situations. (Sakr et al., 2011; see also Cooper et al., 2009.)

Fault tolerance. For transactional workloads, a fault tolerant cloud data management system needs to be able to recover from a failure without losing any data or updates from recently committed transactions. Moreover, it needs to successfully commit transactions and make progress on a workload even in the face of worker node failures. For analytical workloads, a fault tolerant cloud data management system should not need to restart a query if one of the nodes involved in query processing fails. (Sakr et al., 2011; see also Abouzeid et al., 2009.)

Ability to run in a heterogeneous environment. On cloud computing platforms, there is a strong trend towards increasing the number of nodes that participate in query execution. It is nearly impossible to get homogeneous performance across hundreds or thousands of computing nodes. Part failures that do not cause complete node failure, but result in degraded hardware performance become more common at scale. A cloud data management system should be designed to run in a heterogeneous environment and has to take appropriate measures to prevent degrading performance due to parallel processing on distributed nodes. (Sakr et al., 2011; see also Abouzeid et al., 2009.)

Flexible query interface. They should support both SQL and non-SQL interface languages, e.g., MapReduce. Moreover, they should provide mechanism for allowing the user to write user defined functions (UDFs), and queries that utilize these UDFs should be automatically parallelized during their processing. (Sakr et al., 2011; see also Abouzeid et al., 2009.)

3.3.4 Cloud database management systems' wish list

According to Abadi (2009), transactional data management applications are not well suited for cloud deployment, while the characteristics of data and workloads of typical analytical data management applications are. According to Abadi (2009), analytic database systems are a likely segment of the DBMS market to move into the cloud, so he explores various available software solutions to perform the data analysis. Before dealing with them, Abadi (2009), however, lists some desired properties and features that these solutions should ideally have:

Efficiency. Given that cloud computing pricing is structured in a way so that a customer pays for only what he uses, the price increases linearly with the requisite storage, network bandwidth, and compute power. Hence, if the data analysis software product A requires an order of magnitude more compute units than the data analysis software product B to perform the same task, then the product A will cost (approximately) an order of magnitude more than the B. Efficient software has a direct effect on the bottom line. (Abadi, 2009.)

Fault tolerance. Fault tolerance in the context of analytical data workloads is measured differently than fault tolerance in the context of transactional workloads. As for transactional workloads, a fault tolerant DBMS can recover from a failure without losing any data or updates from recently committed transactions, and in the context of distributed databases, can successfully commit transactions and make progress on a workload even in the face of worker node failure. For read-only queries in analytical workloads, there are no write transactions to commit, nor updates to lose upon node failure. Hence, a fault tolerant analytical DBMS is simply one that does not have to restart a query if one of the nodes involved in query processing fails. Given the large amount of data that needs to be accessed for deep analytical queries, combined with the relatively weak compute capacity of a typical cloud compute server instance, complex queries can involve hundreds (even thousands) of server instances and can take hours to complete. Furthermore, clouds are typically built on top of cheap, commodity hardware, for which failure is not uncommon. Consequently, the probability of a failure occurring during a long-running data analysis task is relatively high. If a query must restart each time a node fails, then long, complex queries are difficult to complete. (Abadi, 2009.)

Ability to run in a heterogeneous environment. The performance of cloud computing nodes is often not consistent, with some nodes attaining orders of magnitude worse performance than other nodes (Abadi, 2009). There are a variety of reasons why this could occur, ranging from hardware failure causing degraded performance on a node (RightScale, 2008, as cited in Abadi, 2009), to an instance being unable to access the second core on a dual-core machine (Steele, 2007, as cited in Abadi, 2009), to contention for non-virtualized resources (Abadi, 2009). If the amount of work needed to execute a query is equally divided among the cloud computing nodes, there is a danger that the time to complete the query will be approximately equal to the time for the slowest computing node to complete its assigned task. A node observing degraded performance would thus have a disproportionate affect on total query latency. A system designed to run in a heterogeneous environment would take appropriate measures to prevent this from occurring. (Abadi, 2009.)

Ability to operate on encrypted data. Sensitive data may be encrypted before being uploaded to the cloud. In order to prevent unauthorized access to the sensitive data, any application running in the cloud should not have the ability to directly decrypt the data before accessing it. However, shipping entire tables or columns out of the cloud for decryption is bandwidth intensive. Hence, the ability of the data analysis system to operate directly on encrypted data so that a smaller amount of data needs to be ultimately shipped elsewhere to be decrypted could significantly improve performance. (Abadi, 2009.)

Ability to interface with business intelligence products. There are a variety of customer-facing business intelligence tools that work with database software and aid in the visualization, query generation, result dashboarding, and advanced data analysis. These tools are an important part of the analytical data management picture, since business analysts are often not technically advanced and do not feel comfortable interfacing with the database software directly. (Abadi, 2009.)

3.4 Framework of requirements for cloud data management

The aforementioned requirements for cloud data management are depicted in the tables 1 and 2. The requirements in the table 1 are requirements for cloud computing in general, while the requirements in the table 2 are more closely related to cloud data management.

In the first column are named the requirements that have been derived from the requirements or considerations described in the literature sources. E.g., data storage device type and programming model are mentioned under data management, storage, and processing in Rimal et al.'s paper (2011), but in the framework this requirement has been divided into two requirements.

In the second column are examples and illustrations of the requirements. The reason for this is simply to make the tables easier to read. E.g., service-centric issues would not open up without any explanation, and it would be an unrewarding task to go back in the text to find out what it is about. However, the purpose of the column is only to exemplify and illustrate. It does not provide all-inclusive definitions of what different requirements are about.

In the third column are mentioned the original names of the requirements or considerations described in the literature sources, as well as their respective sources. As already mentioned, some requirements are derived from 'larger' requirements or considerations.

The framework functions as a guiding principle that helps e.g., researchers and decision-makers to map what requirements should be taken into consideration when building especially cloud data management systems. The requirements are quite general and abstract, but in the author's opinion, they are still important, serving as the bedrock of virtually any cloud solution. The framework is likely to be useful to anyone building a technical solution on the cloud.

The application of the framework requires in practice technical expertise and measurements. Evaluating e.g., cloud computing system's security requires knowledge of the field. Evaluating e.g., cloud computing system's performance requires not only familiarity with the subject, but probably also benchmarking the system somehow.

TABLE 1 Cloud computing requirements

Cloud computing requirement	Examples and illustrations	Literature source
Service model	Saas, Paas, IaaS (Mell & Grance, 2011).	Service models (Mell & Grance, 2011), provider service delivery model (Rimal et al., 2011), service models (Sakr et al., 2011), service models (Wu et al., 2010).
Deployment model	Public, private, community, hybrid cloud (Mell & Grance, 2011).	Deployment models (Mell & Grance, 2011), cloud deployment for enterprises (Rimal et al., 2011), cloud deployment models (Sakr et al., 2011).
Service-centric issues	Autonomic and self-describing cloud services, low cost composition of distributed applications (Rimal et al., 2011).	Service-centric issues (Rimal et al., 2011).
Virtualization management	Server, storage, infrastructure virtualization, etc. (Rimal et al., 2011).	Virtualization management (Rimal et al., 2011).
Fault tolerance	Fault isolation to the falling components, availability of reversion mode, etc. (Rimal et al., 2011).	Fault tolerance (Rimal et al., 2011).
Privacy	Data that the user would regard as his personal intellectual property will be stored at mega data centers located around the world (Rimal et al., 2011).	User-centric privacy (Rimal et al., 2011).
Security	A data center holds the information that would more traditionally be stored on the end-user's computer (Rimal et al., 2011).	Security (Rimal et al., 2011).
Formal agreements	SLAs play a key facilitator role to make agreed-upon QoS between service providers and end-users (Rimal et al., 2011).	QoS, SLAs (Rimal et al., 2011), QoS, SLAs (Sakr et al., 2011), SLAs (Wu et al., 2010).
Transparent pricing	The economics of cloud computing, cloudonomics, cost management (Rimal et al., 2011).	Cloudonomics, user consumption-based billing and metering (Rimal et al., 2011).
Load balancing	The mechanism of self-regulating the workloads properly within the cloud's entities (Rimal et al., 2011).	Load balancing (Rimal et al., 2011).
Interoperability	The creation of an agreed-upon framework/ontology, open data format, or open protocols/APIs enabling migration and integration between cloud service providers and facilitating secure information exchange across platforms (Rimal et al., 2011).	Interoperability (Rimal et al., 2011).
Scalability	DHT, column-orientation, and horizontal partitioning (Rimal et al., 2011).	Scalability (Rimal et al., 2011).
Business process management (BPM)	Cloud-based BPM, cloud delivering business operating platforms, e.g., CRM (Rimal et al., 2011).	BPM (Rimal et al., 2011).
Third party engagement	Can help in respect of e.g., continuity of cloud service engagements and legal implications (Rimal et al., 2011).	Third party engagement (Rimal et al., 2011).
Transferable skills	Cloud computing comes with its own set of management tasks (Rimal et al., 2011).	Transferable skills (Rimal et al., 2011).
Adaptability and learning	Users have to get acquainted with applications (Rimal et al., 2011).	Adaptability and learning (Rimal et al., 2011).
User experience (UX)	UX-driven design and deployment (Rimal et al., 2011).	User experience (UX) (Rimal et al., 2011).

TABLE 2 Cloud data management requirements

Cloud data management requirement	**Examples and illustrations**	**Literature source**
Data storage device type	HDDs, SSDs, hybrid hard disks (Rimal et al., 2011).	Data management, storage, and processing (Rimal et al., 2011).
Data governance	Geographical and political issues, data disclosure or loss, sensitive data outside of an organization, data lock-in (Rimal et al., 2011).	Data governance (Rimal et al., 2011).
Data migration	Data replication in the cloud (Rimal et al., 2011).	Data migration (Rimal et al., 2011).
Programming model	MapReduce is not a perfect fit for all tasks (Rimal et al., 2011).	Data management, storage, and processing (Rimal et al., 2011).
Automatic	Underlying infrastructure changes can be made quickly and without human intervention (Wu et al., 2010).	Automatic (Wu et al., 2010).
Availability	A cloud data management system has to be always accessible (Sakr et al., 2011; see also Cooper et al., 2009).	Availability (Sakr et al., 2011; see also Cooper et al., 2009).
Scalability	Cloud storage needs to scale quickly and to tremendous capacities (Wu et al., 2010). A cloud data management system has to be able to support very large databases with very high request rates at very low latency (Sakr et al., 2011; see also Cooper et al., 2009).	Scalability (Wu et al., 2010), scalability (Sakr et al., 2011; see also Cooper et al., 2009).
Privacy	The storage of personal/enterprise sensitive data (Rimal et al., 2011).	User-centric privacy (Rimal et al., 2011).
Security	Cloud storage providers have to establish multi-tenancy policies to allow e.g., separate companies to securely share the same storage hardware (Wu et al., 2010).	Data security (Wu et al., 2010).
Elasticity	Cloud storage has to be elastic to rapidly adjust the underlying infrastructure to changing subscriber demands and comply with SLAs (Wu et al., 2010).	Elasticity (Wu et al., 2010), elasticity (Sakr et al., 2011; see also Cooper et al., 2009).
Performance	A proven storage infrastructure providing fast, robust data recovery. Important to measure and test network latency before committing to a migration. (Wu et al., 2010.)	Performance, latency (Wu et al., 2010), performance (Sakr et al., 2011; see also Abouzeid et al., 2009), efficiency (Abadi, 2009).
Multitenancy	A cloud data management system has to be able to support many applications (tenants) on the same hardware and software infrastructure (Sakr et al., 2011; see also Cooper et al., 2009).	Multitenancy (Sakr et al., 2011; see also Cooper et al., 2009).
Load and tenant balancing	A cloud data management system has to be able to automatically move load between servers and to avoid resource overloading situations (Sakr et al., 2011; see also Cooper et al., 2009).	Load and tenant balancing (Sakr et al., 2011; see also Cooper et al., 2009).
Reliability	Data is reliably backed up for disaster recovery purposes (Wu et al., 2010).	Reliability (Wu et al., 2010).
Fault tolerance	As for transactional workloads, recovering from a failure without losing any data or updates from recently committed transactions (Sakr et al., 2011; see also Abouzeid et al., 2009).	Fault tolerance (Sakr et al., 2011; see also Abouzeid et al., 2009), fault tolerance (Abadi, 2009).
Ability to run in a heterogenous environment	A cloud data management system has to take measures to prevent degrading performance due to parallel processing on distributed nodes (Sakr et al., 2011; see also Abouzeid et al., 2009).	Ability to run in a heterogenous environment (Sakr et al., 2011; see also Abouzeid et al., 2009), ability to run in a heterogenous environment (Abadi, 2009).
Ability to operate on encrypted data	The ability of the data analysis system to operate directly on encrypted data so that a smaller amount of data needs to be ultimately shipped elsewhere to be decrypted could	Ability to operate on encrypted data (Abadi, 2009).

Cloud data management requirement	Examples and illustrations	Literature source
	significantly improve performance (Abadi, 2009).	
Ability to interface with business intelligence (BI) products	Customer-facing BI tools that work with database software and aid in the visualization, query generation, result dashboarding, and advanced data analysis (Abadi, 2009).	Ability to interface with business intelligence products (Abadi, 2009).
Flexible query interface	A cloud data management system should support both SQL and non-SQL interface languages (Sakr et al., 2011; see also Abouzeid et al., 2009).	Flexible query interface (Sakr et al., 2011; see also Abouzeid et al., 2009).
Ease of management	Improved manageability in the face of exploring storage capability and costs (Wu et al., 2010).	Ease of management (Wu et al., 2010).
Ease of data access	Enabling seamless integration of cloud storage into existing enterprise workflows and minimizing the learning curve for cloud storage adoption (Wu et al., 2010).	Ease of data access (Wu et al., 2010).
Energy efficiency	Green storage technology leads to a lower carbon footprint (Wu et al., 2010).	Energy efficiency (Wu et al., 2010).

4 SMART CITIES AND THEIR DATA MANAGEMENT

This chapter is organized as follows. First, a smart city and the central concepts related to it are defined, e.g., the Internet of Things (IoT). Then, the enabling technologies of the IoT are dealt with. Next, requirements for smart city data management are discussed. Finally, a framework of requirements for integrating a smart city with a cloud infrastructure is presented.

4.1 Definition of a smart city

According to Dirks, Gurdgiev, and Keeling (2010, as cited in Chourabi et al., 2012), Dirks and Keeling (2009, as cited in Chourabi et al., 2012), and Dirks, Keeling, and Dencik (2009, as cited in Chourabi et al., 2012), more than a half of the world's population now lives in urban areas. Leaning on unfpa.org, Chourabi et al. (2012) note that this shift from a primarily rural to a primarily urban population is projected to continue for the next couple of decades.

Such enormous and complex congregations of people inevitably tend to become messy and disordered places (Johnson, 2008, as cited in Chourabi et al., 2012). Mega cities generate new kinds of problems (Chourabi et al., 2012), such as technical, physical, and material problems, e.g., difficulty in waste management, human health concerns, traffic congestions, and inadequate, deteriorating and aging infrastructures (Borja, 2007, as cited in Chourabi et al., 2012; Marceau, 2008, as cited in Chourabi et al., 2012; Toppeta, 2010, as cited in Chourabi et al., 2012; Washburn et al., 2010, as cited in Chourabi et al., 2012). Another set of problems are more social and organizational in nature. Problems of these types are associated with multiple and diverse stakeholders, high levels of interdependence, competing objectives and values, and social and political complexity. (Chourabi et al., 2012.) In this sense, city problems become wicked and tangled (Dawes, Cresswell & Pardo, 2009, as cited in Chourabi et al., 2012; Rittel &

Webber, 1973, as cited in Chourabi et al., 2012; Weber & Khademian, 2008, as cited in Chourabi et al., 2012).

The urgency around these challenges is triggering many cities around the world to find smarter ways to manage them. These cities are increasingly described with the label *smart city*. One way to conceptualize a smart city is as an icon of a sustainable and livable city. (Chourabi et al., 2012.)

According to Gibson, Kozmetsky, and Smilor (1992, as cited in Schaffers et al., 2011), the phrase 'smart city' was coined in the early 1990s to signify how urban development was turning towards technology, innovation, and globalization. Chourabi et al. (2012) note that although there is an increase in the frequency of the use of the phrase 'smart city', there is still not a clear and consistent understanding of the concept among practitioners and academia. In 2014, the situation seems to be quite the same. Piro, Cianci, Grieco, Boggia, and Camarda (2014) draw on Chourabi et al.'s (2012) paper noting that despite the term 'smart city' is very common in everyday speaking, its exact definition is still not well-established. Getting back to Chourabi et al.'s (2012) paper, they mention that only a limited number of studies have investigated and have begun to systematically consider questions related to the new urban phenomenon of smart cities. In their paper, they conceptualize a smart city by presenting several working definitions of a smart city that have been put forward and adopted in both practical and academic use, e.g.,:

A city connecting the physical infrastructure, the IT infrastructure, the social infrastructure, and the business infrastructure to leverage the collective intelligence of the city (Harrison et al., 2010, as cited in Chourabi et al., 2012).

A city striving to make itself 'smarter' (more efficient, sustainable, equitable, and livable) (Natural Resources Defense Council, as cited in Chourabi et al., 2012).

The use of smart computing technologies to make the critical infrastructure components and services of a city – which include city administration, education, healthcare, public safety, real estate, transportation, and utilities – more intelligent, interconnected, and efficient (Washburn et al., 2010, as cited in Chourabi et al., 2012).

Drawing on the various definitions of a smart city, some of them presented above, Chourabi et al. (2012) propose a framework to understand the concept of smart cities. The framework is depicted in the figure 5. Based on their exploration of literature, Chourabi et al. (2012) identify eight critical factors of smart city initiatives: management and organization, technology, governance, policy context, people and communities, economy, built infrastructure, and natural environment. These factors form the basis of an integrative framework that can be used to examine how local governments are envisioning smart city initiatives. The framework suggests directions and agendas for smart city research and outlines practical implications for government professionals. It is expected that while all factors have a two-way impact in smart city initiatives (each likely to be influenced by and is influencing other factors), at different times and in different contexts, some are more influential than others. (Chourabi et al., 2012.)

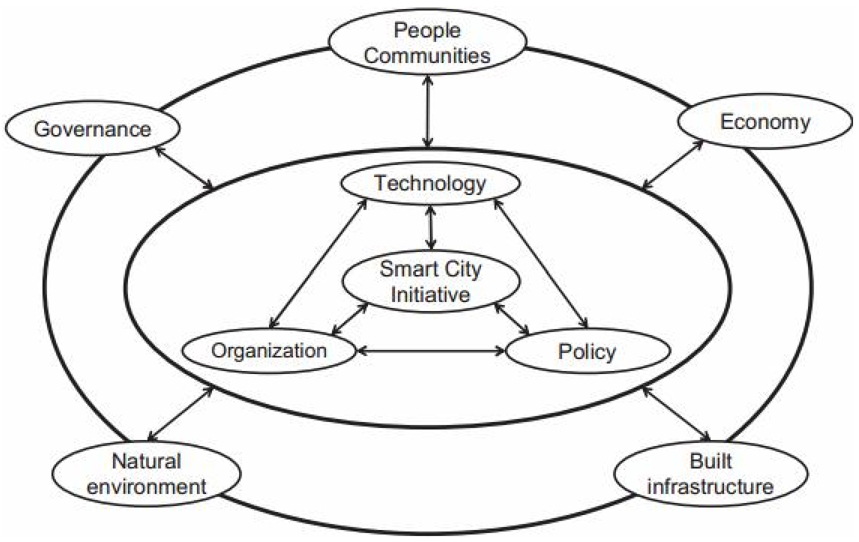

FIGURE 5 Smart city initiatives framework (Chourabi et al., 2012, 2294)

Nam and Pardo (2011) also note that the definitions of a smart city are various. According to them, the label 'smart city' is a fuzzy concept and is used in ways that are not always consistent. There is neither a single template of framing a smart city nor a one-size-fits-all definition of a smart city. (Nam & Pardo, 2011.)

Nam and Pardo (2011) present similar working definitions than Chourabi et al. (2012) above, after which they study the conceptual relatives of a smart city: a ubiquitous city, knowledge city, smart community, etc. These can be largely categorized into three dimensions: technology, people, and community. However, they are mutually connected with substantial confusion in definitions and complicated usages rather than independent of each other. E.g., in the technology dimension, the concepts of 'digital city' and 'intelligent city' can be found. (Nam & Pardo, 2011.) A digital city refers to a connected community that combines a broadband communications infrastructure, a flexible, service-oriented computing infrastructure based on open industry standards, and innovative services to meet the needs of governments and their employees, citizens, and businesses (Yovanof & Hazapis, 2009, as cited in Nam & Pardo, 2011). An intelligent city is usually used to characterize a city that has the ability to support learning, technological development, and innovation procedures. In this sense, every digital city is not necessarily intelligent, but every intelligent city has digital components. (Nam & Pardo, 2011.)

To know what all these concepts mean is not vital. After discussing them Nam and Pardo (2011) identify and clarify the key conceptual components of a smart city, as well as re-categorize and simplify them into three categories of core factors: technology (infrastructures of hardware and software), people (creativity, diversity, and education), and institution (governance and policy). This is depicted in the figure 6.

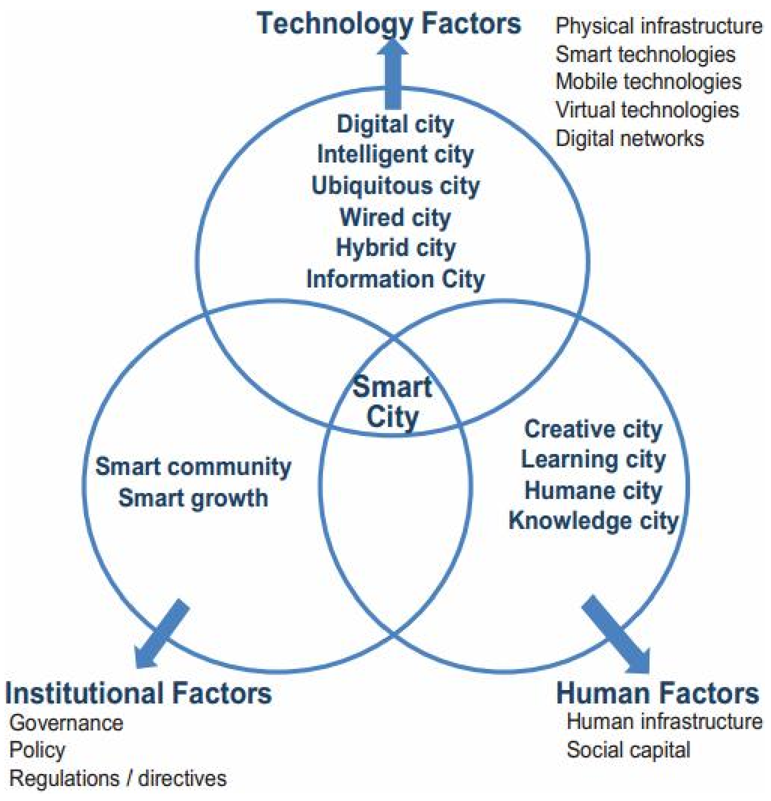

FIGURE 6 Fundamental components of a smart city (Nam & Pardo, 2011, 286)

As Nam and Pardo (2011) put it, given the connection between these factors, a city is smart when investments in human/social capital and IT infrastructure fuel sustainable growth and enhance a quality of life, through participatory governance (Caragliu, Del Bo & Nijkamp, 2009, as cited in Nam & Pardo, 2011). This is a modification of Caragliu et al.'s (2009) definition of a smart city from their paper *Smart cities in Europe* (2009). Caragliu et al.'s (2009) original definition is: we believe a city to be smart when investments in human and social capital and traditional (transport) and modern (ICT) communication infrastructure fuel sustainable economic growth and a high quality of life, with a wise management of natural resources, through participatory governance. In a newer paper, Caragliu, Del Bo, and Nijkamp (2011) mention that a smart city is still, in their opinion, quite a fuzzy concept. They go through many definitions of a smart city ending up defining it in the same way as in their previous paper.

In summary, there is no single, all-inclusive definition of a smart city, but many definitions, the aforementioned ones being, in the author's opinion, some of the best. In this thesis, a smart city is defined by Caragliu et al.'s (2009) definition. To understand better what smart cities are about, it is necessary to briefly define the central concepts related to them. Next, they are discussed briefly.

4.2 Definition of the central concepts related to a smart city

Hernández-Muñoz et al. (2011) write that most of the current city and urban developments are based on vertical ICT solutions leading to an unsustainable sea of systems and market islands. However, the recent vision of the Future Internet and its components can become building blocks to progress towards a unified urban-scale ICT platform transforming a smart city into an open innovation platform. Once major challenges of unified urban-scale ICT platforms are identified, it is clear that the future development of smart cities will be only achievable in conjunction with a technological leap in the underlying ICT infrastructure. (Hernández-Muñoz et al., 2011.)

Hernández-Muñoz et al. (2011) advocate that this technological leap can be done by considering smart cities at the forefront of the recent vision of the *Future Internet* (*FI*). Although there is no universally accepted definition of the FI, it can be approached as a socio-technical system comprising Internet-accessible information and services, coupled to the physical environment and human behavior, and supporting smart applications of societal importance (Boniface & Surridge, as cited in Hernández-Muñoz et al., 2011).

The FI can transform a smart city into an open innovation platform supporting vertical domain of business applications built upon horizontal enabling technologies (Hernández-Muñoz et al., 2011). The most relevant basic FI pillars (Towards a Future Internet Public Private Partnership: Usage Areas Workshop, 2010, as cited in Hernández-Muñoz et al., 2011) for a smart city environment are the following (Hernández-Muñoz et al., 2011):

The Internet of Things (IoT). Defined as a global network infrastructure based on standard and interoperable communication protocols where physical and virtual 'things' are seamlessly integrated into the information network. (Sundmaeker, Guillemin, Friess & Woelfflé, 2010, as cited in Hernández-Muñoz et al., 2011.) The term was probably coined by Ashton in 1999 (Ashton, 2009).

The Internet of Services (IoS). Flexible, open, and standardized enablers that facilitate the harmonization of various applications into interoperable services, as well as the use of semantics for the understanding, combination, and processing of data and information from different service providers, sources, and formats. (Hernández-Muñoz et al., 2011.)

The Internet of People (IoP). Envisaged as people becoming part of ubiquitous, intelligent networks having the potential to seamlessly connect, interact, and exchange information about themselves, their social context, and environment. (Hernández-Muñoz et al., 2011.)

It is important to highlight the bidirectional relationship between the FI and smart cities. In the one direction, the FI can offer solutions to many challenges that smart cities face. On the other direction, smart cities can provide an excellent experimental environment for the development, experimentation, and testing of common FI service enablers required to achieve 'smartness' in a variety of application domains. (Hernández-Muñoz et al., 2011; Future Internet Assembly, 2009, as cited in Hernández-Muñoz et al., 2011.)

Of all these definitions, the IoT is the most important, smart cities being one of the application fields and market sectors in which IoT solutions can provide competitive advantages over current solutions and which can play a leading role in the adoption of IoT technologies. The other application fields and market sectors of the IoT are environmental monitoring, smart business / inventory and product management, smart homes / smart building management, health-care, and security and surveillance. (Miorandi, Sicari, De Pellegrini & Chlamtac, 2012.)

As Sundmaeker et al.'s (2010, as cited in Hernández-Muñoz et al., 2011) definition of the IoT is somewhat abstract, it is useful to complete it with the definition that Miorandi et al. (2012) present in their paper: the term IoT is broadly used to refer to the resulting global network interconnecting smart objects by means of extended Internet technologies, the set of supporting technologies necessary to realize such a vision, and the ensemble of applications and services leveraging such technologies to open new business and market opportunities (Atzori, Iera & Morabito, 2010, as cited in Miorandi et al., 2012; ITU, 2005, as cited in Miorandi et al., 2012). *Smart objects* refer to the embedding of electronics into everyday physical objects, making them 'smart' and letting them seamlessly integrate within the global resulting cyberphysical infrastructure. From a conceptual standpoint, the IoT builds on three pillars, related to the ability of smart objects to be identifiable (anything identifies itself), to communicate (anything communicates), and to interact (anything interacts) – either among themselves, building networks of interconnected objects, or with end-users or other entities in the network. (Miorandi et al., 2012.)

As smart cities are one of the IoT's application fields and markets sectors, some of the enabling technologies of the IoT are next briefly discussed.

4.3 Enabling technologies of the Internet of Things

Actualization of the IoT concept into the real world is possible through the integration of several enabling technologies (Atzori et al., 2010). Atzori et al. (2010) discuss in their paper the most relevant ones:

Identification, sensing, and communication technologies. 'Anytime, anywhere, any media' has been a long time the vision pushing forward the advances in communication technologies (Atzori et al., 2010). In this context, wireless technologies have played a key role, and today the ratio between radios and humans is nearing the 1 to 1 value (Srivastava, 2006, as cited in Atzori et al., 2010). In this context, key components of the IoT will be *Radio Frequency IDentifcation (RFID)* systems (Finkenzeller, 2003, as cited in Atzori et al., 2010) that are composed of one or more reader(s) and several RFID tags. Tags are characterized by a unique identifier and are applied to objects, even persons or animals. Readers trigger the tag transmission by generating an appropriate signal that represents a query for the possible presence of tags in the surrounding area and for the reception of their IDs. Accordingly, RFID systems can be used to monitor objects in real-time, without the need of being in line-of-sight. This allows for mapping the real world into the virtual world. (Atzori et al., 2010.)

Sensor networks will also play a crucial role in the IoT. In fact, they can cooperate with RFID systems to better track the status of things, i.e., their location, temperature, movements, etc. As such, they can augment the awareness of a certain environment and thus act as a further bridge between physical and digital world. Sensor networks consist of a certain number of sensing nodes communicating in a wireless multi-hop fashion. Usually nodes report the results of their sensing to a small number of special nodes called sinks. (Atzori et al., 2010.)

Middleware. The middleware is a software layer or a set of sub-layers interposed between the technological and the application levels. Its feature of hiding the details of different technologies is fundamental to exempt the programmer from issues that are not directly pertinent to his focus, which is the development of the specific application enabled by the IoT infrastructures. The middleware is gaining more and more importance in the last years due to its major role in simplifying the development of new services and the integration of legacy technologies into new ones. This exempts the programmer from the exact knowledge of the variegate set of technologies adopted by the lower layers. As it is happening in other contexts, the middleware architectures proposed in the last years for the IoT often follow the SOA approach. The advantages of the SOA approach are recognized in most studies on middleware solutions for the IoT. (Atzori et al., 2010.)

The SOA-based architecture for the IoT middleware is depicted in the figure 7. *Applications* are on top of the architecture, exporting all the system's functionalities to the final user. This layer is not considered to be a part of the middleware, but it exploits all the functionalities of the middleware layer. *Service composition* layer is a common layer on top of a SOA-based middleware architecture. It provides the functionalities for the composition of single services offered by networked objects to build specific applications. On this layer there is no notion of devices, and the only visible assets are services. *Service management* layer provides the main functions that are expected to be available for each object and that allow for their management in the IoT scenario. A basic set of services encompasses object dynamic discovery, status monitoring, and service configuration. The IoT relies on a vast and heterogeneous set of objects, each one providing specific functions accessible through its own dialect, and there is thus a need for an *object abstraction* layer capable of harmonizing the access to the different devices with a common language and procedure. (Atzori et al., 2010.)

Trust, privacy, and security management. The deployment of automatic communication of objects in our lives represents a danger for our future. E.g., RFID tags in personal devices, clothes, and groceries could unknowingly be triggered to reply with their IDs and other information. The middleware has to include functions related to the management of trust, privacy, and security of all the exchanged data. The related functions may be either built on one specific layer of the previous ones or distributed through the entire stack, from the object abstraction to the service composition, in a manner that does not affect system performance or introduce excessive overheads. (Atzori et al., 2010.)

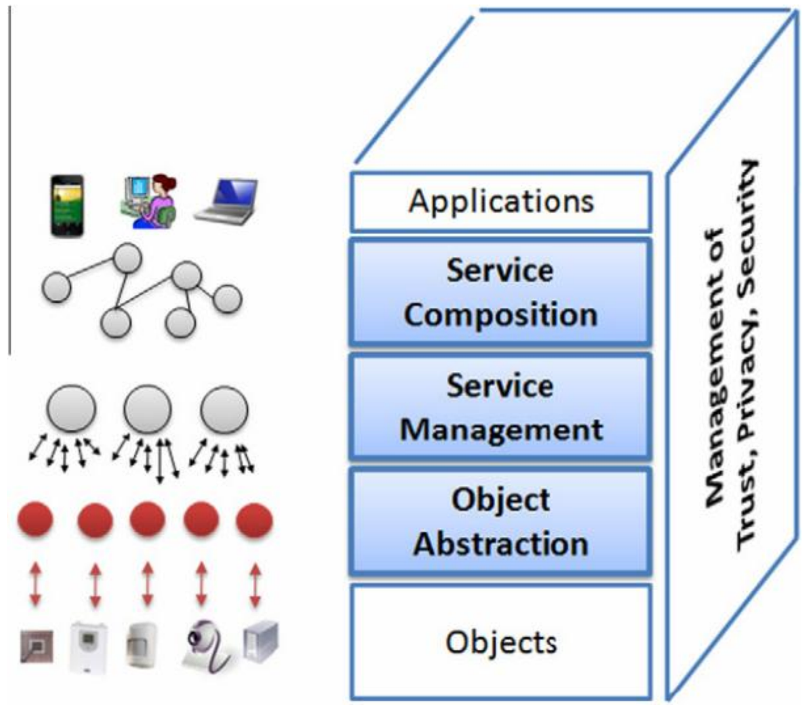

FIGURE 7 SOA-based architecture for the IoT middleware (Atzori et al., 2010, 2792)

4.4 Requirements for smart city data management

Next, requirements for smart city data management are discussed. As dealing with the requirements for cloud data management, the literature sources are organized here as well so that the more general requirements are presented first and the more specific later on. Finally, a framework to which all the central requirements for integrating a smart city with a cloud infrastructure are gathered is presented.

4.4.1 IoT Reference Architecture requirements

The IoT Reference Architecture (RA) is, among other things, designed as a reference for the generation of compliant IoT concrete architectures that are tailored to one's specific needs. The IoT RA is kept rather abstract in order to enable many, potentially different IoT architectures. The architecture consists of so-called *views* of different system aspects that can be conceptionally isolated and so-called *perspectives* that are about architectural decisions that often address concerns that are common to more than one view or even all of them.

These concerns are often related to non-functional or quality properties. (Internet of Things Architecture, 2013.) As a matter of fact, both views and perspectives are very abstract, but as perspectives address more general concerns, they are discussed below.

According to Internet of Things Architecture (2013), the perspectives that are the most important for IoT-systems based on the stakeholder requirements are evolution and interoperability, performance and scalability, availability and resilience, and trust, security and privacy:

Evolution and interoperability. The ability of the system to be flexible in the face of the inevitable change that all systems experience after deployment, balanced against the costs of providing such flexibility. This perspective addresses the fact that requirements change and software evolves sometimes rapidly and need to interoperate not only with today's technologies, but also needs to be prepared to interoperate with later technologies. (Internet of Things Architecture, 2013.)

Performance and scalability. The ability of the system to predictably execute within its mandated performance profile and to handle increased processing volumes in the future if required. Both are, compared to traditional information systems, even harder to cope with in a highly distributed scenario as the IoT. (Internet of Things Architecture, 2013.)

Availability and resilience. The ability of the system to be fully or partly operational as and when required and to effectively handle failures that could affect system availability. When dealing with distributed IoT systems in which a lot of things can go wrong, the ability of a system to stay operational and to effectively handle failures that could affect a system's availability is crucial. (Internet of Things Architecture, 2013.)

Trust, security, and privacy. They are interrelated, and often the evaluation or the improvement of one of these qualities is necessarily related to the others. Trust is a complex quality related to the extent to which a subject expects (subjectively) an IoT system to be dependable regarding all the aspects of its functional behavior. Security stands for the ability of the system to enforce the intended confidentiality, integrity, and service access policies, and to detect and recover from a failure in these security mechanisms. Privacy is about the ability of the system to ensure that the collection of personally identifying information be minimized and that collected data should be used locally wherever possible. (Internet of Things Architecture, 2013.)

4.4.2 Key system-level features that the Internet of Things needs to support

Miorandi et al. (2012) discuss in their paper the IoT's vision and concept, after which they preliminarily identify key system-level features that the IoT needs to support:

Devices heterogeneity. The IoT will be characterized by a large heterogeneity in terms of devices taking part in the system, and they are expected to present very different capabilities from the computational and communication standpoints. The management of such a high level of heterogeneity shall be supported at both architectural and protocol levels. In particular, this may question the 'thin waist' approach at the basis of IP networking. (Miorandi et al., 2012.)

Scalability. As everyday objects get connected to a global information infrastructure, scalability issues arise at different levels including naming and addressing (due to the sheer size of the resulting system), data communication and networking (due to the high level of interconnection among a large number of entities), information and knowledge management (due to the possibility of building a digital counterpart to any entity and/or phenomena in the physical realm), and service provisioning and management (due to the massive number of services / service execution options that could be available and the need to handle heterogeneous resources). (Miorandi et al., 2012.)

Ubiquitous data exchange through proximity wireless technologies. In the IoT, a prominent role will be played by wireless communications technologies that will enable smart objects to become networked. (Miorandi et al., 2012.) The ubiquitous adoption of the wireless medium for exchanging data may pose issues in terms of spectrum availability, pushing towards the adoption of cognitive/dynamic radio systems (Haykin, 2005, as cited in Miorandi et al., 2012).

Energy-optimized solutions. For a variety of IoT entities, minimizing the energy spent for communication/computing purposes will be a primary constraint. While techniques related to energy harvesting (by means of e.g., piezoelectric materials or micro solar panels) will relieve devices from the constraints imposed by battery operations, energy will always be a scarce resource to be handled with care. Thereby the need to devise solutions that tend to optimize energy usage (even at the expense of performance) will become more and more attractive. (Miorandi et al., 2012.)

Localization and tracking capabilities. As entities in the IoT can be identified and are provided with short-range wireless communications capabilities, it becomes possible to track the location (and the movement) of smart objects in the physical realm. This is particularly important for an application in logistics and product life-cycle management that are already extensively adopting RFID technologies. (Miorandi et al., 2012.)

Self-organization capabilities. The complexity and dynamics that many IoT scenarios will likely present calls for distributing intelligence in the system, making smart objects (or a subset thereof) able to autonomously react to a wide range of different situations in order to minimize human intervention. (Miorandi et al., 2012.) Following users' requests, nodes in the IoT will organize themselves autonomously into transient ad hoc networks, providing the basic means for sharing data and for performing coordinated tasks (Chlamtac, Conti & Liu, 2003, as cited in Miorandi et al., 2012). This includes the ability to perform device and service discovery without requiring an external trigger, to build overlays, and to adaptively tune protocols' behavior to adapt to the current context (Dobson et al., 2006, as cited in Miorandi et al., 2012).

Semantic interoperability and data management. The IoT will be much about exchanging and analyzing massive amounts of data. In order to turn them into useful information and to ensure interoperability among different applications, it is necessary to provide data with adequate and standardized formats, models, and semantic description of their content (metadata), using well-defined languages and formats. This will enable IoT applications to support automated reasoning, a key feature for enabling the successful adoption of such a technology on a wide scale. (Miorandi et al., 2012.)

Embedded security and privacy-preserving mechanisms. Due to the tight entanglement with the physical realm, IoT technology should be secure and privacy-preserving by design. This means that security should be considered a key system-level property, and be taken into account in the design of architectures and methods for IoT solutions. This is expected to represent a key requirement for ensuring the acceptance by users and the wide adoption of the technology. (Miorandi et al., 2012.)

4.4.3 Key requirements of a smart city software architecture

Da Silva et al. (2013) deal with several smart city software architectures noting that although the literature contains several works about the subject, a reference architecture that permeates the entire operation of a smart city has not been minimally designed yet. However, by studying different architectures, Da Silva et al. (2013) were able to present a number of key requirements that have to be met when implementing a smart city software architecture:

Objects interoperability. One of the most discussed and studied requirements is interoperability of objects in which the object is an abstraction of a sensor, actuator, or any device, able to perform some sort of computation. In fact, this is a critical requirement to the consolidation of any platform that uses a range of objects with different technical specifications and communication protocols. (Da Silva et al., 2013.) The vast majority of architectures that Da Silva et al. (2013) studied explicitly designate a module or layer to meet this requirement.

Sustainability. Due to the high coverage of all smart city areas, architectures have to include, since their conception, sustainable policies. These policies are related to environmental, economic, and social aspects of each domain. (Da Silva et al., 2013.)

Real-time monitoring. Another important feature inherent to the smart cities' context is continuous real-time monitoring. The real-time monitoring is the most valuable instrument to provide relevant information that will be used to predict phenomena. An example is the monitoring of the water level during the rainy seasons. In this situation, from an effective monitoring measures can be taken to mitigate potential inconvenience to citizens, e.g., floods and disease transmission. (Da Silva et al., 2013.)

Historical data. In the smart cities' context, all the components that compose each area of a city are constantly being modified, either by natural factors or human activities. Hence, all data picked up has the potential to become relevant, as long as it is somehow associated to other data. Therefore, it is substantial that the architectures include efficient storage and retrieval mechanisms for such data. (Da Silva et al., 2013.)

Mobility. Mobility is another key requirement that has to be explored in smart cities. Mobility means any mobile technology to capture information about the environment or act over the same. Mobility is a key ally for the implementation of real-time monitoring. (Da Silva et al., 2013.) When considering that four billion people already have smart phones (Hall, 2012, as cited in Da Silva et al., 2013), it is natural to associate mobility to the use of these devices, but other devices can also be successfully used, e.g., ZigBee and RFID (Da Silva et al., 2013).

Availability. To allow this data capture, the centralizing infrastructure has to be highly available. Hence, if a cloud computing infrastructure is used, flow control mechanisms, collision, and redundancy must be inherent to the solution. Although, the system has to continue obtaining and storing data, even acting autonomously, independently of the state of the infrastructure. (Da Silva et al., 2013.)

Privacy. All these issues of data delivery are of paramount importance to any architecture. However, one should establish privacy policies explaining what data will be captured and what will be done with these. Certainly the consolidation of these policies is a challenge that can prevent citizens, institutions, and the government to provide certain critical data. Due to the high relevance of this requirement, it is not permissible for an architecture not to satisfy it. (Da Silva et al., 2013.)

Distributed sensing and processing. It is through the sensing that a computer vision of the urban environment is obtained. The greater the number of sensors and the more dispersed they are, the higher the scope covered by the architecture. The heterogeneity of sensors influences the richness of detail and the amount of data that can be extracted from each scenario being monitored, being possible to obtain more accurate results. Situations that require preventive or corrective measures to be instantly taken demand processing in real-time, with a response time fast enough to provide bases for actions that must be performed as soon as the situation is identified, suggest the need for distributed processing, exploiting the capacity of an existing infrastructure. (Da Silva et al., 2013.)

Service composition and integrated urban management. In a systemic view, urban environments are essentially a set of complex systems available to meet the needs of their citizens. Architectures that are willing to give support to these systems should consider them as complementary in the search for an effective urban management rather than treating them isolated. Services developed to treat each system must be interoperable so that other services can reuse, group, or create a composition using them, exploring important aspects in the correlation between different systems, or even create a holistic and contextualized view of the city in which the architecture was implemented. (Da Silva et al., 2013.)

Social aspects. A smart city architecture cannot be based uniquely on technology. The main purpose in designing a smart city is to increase the quality of life of its citizens. People need to be involved and benefit from the process, otherwise the entire investment will be in vain. A smart city is also made of a change in the behavior of its citizens, and they have to feel included as a fundamental part in its deployment, feel encouraged to be a part of the solution. For this purpose, it can be created ways to stimulate and/or reward this interest. (Da Silva et al., 2013.)

Flexibility/extensibility. Changes, adaptations, and extensions should be foreseen in the architecture. Besides adding new services, new types of sensors, different data types, and urban contexts and hardware, independent operation should be addressed by the architecture, allowing it to be adaptable to different realities. (Da Silva et al., 2013.)

4.4.4 Cloud-centric Internet of Things requirements

Gubbi, Buyya, Marusic, and Palaniswami (2013) discuss a cloud-centric IoT, a conceptual IoT framework that integrates wireless sensor networks with applications, cloud computing being in the center of things providing scalable storage, computation time, and other tools to build new businesses. They mention that for the realization of a complete IoT vision, efficient, secure, scalable, and market-oriented computing and storage resourcing is essential.

As for the aforementioned cloud-centric IoT framework, Gubbi et al. (2013) note that developing IoT applications using low-level cloud programming models and interfaces, e.g., thread and MapReduce models, is complex. To overcome this, there is a need for an IoT application-specific framework for the rapid creation of applications and their deployment on cloud infrastructures (Gubbi et al., 2013). This is achieved by mapping the proposed framework to cloud APIs offered by platforms such as Aneka (Gubbi et al., 2013), a .NET-based application development PaaS that can utilize storage and compute resources of both public and private clouds (Wei, Sukumar, Vecchiola, Karunamoorthy & Buyya, 2011, as cited in Gubbi et al., 2013). It offers a runtime environment and a set of APIs that enable developers to build customized applications by using multiple programming models, e.g., task programming, thread programming, and MapReduce programming (Gubbi et al., 2013).

The new IoT application-specific framework should be able to provide support for reading data streams either from sensors directly or fetch the data from databases, easy expression of data analysis logic as functions/operators that process data streams in a transparent and scalable manner on cloud infrastructures, and if any events of interest are detected, outcomes should be passed to output streams that are connected to a visualization program. Using such a framework, the developer of IoT applications will be able to harness the power of cloud computing without knowing low-level details of creating reliable and scale applications. (Gubbi et al., 2013.)

4.5 Framework of requirements for integrating a smart city with a cloud infrastructure

The framework of requirements for integrating a smart city with a cloud infrastructure is depicted in the tables 3 and 4. The requirements in the table 3 are again requirements for cloud computing in general, while the requirements in the table 4 are more closely related to cloud data management.

In the first column are again named the cloud requirements that have been derived from the requirements or considerations described in the literature sources. However, the cloud requirements that could not be explicitly connected to the smart city requirements are not presented in the tables. In the second column are again examples and illustrations of the cloud requirements. In the third column are once again mentioned the original names of the cloud requirements or considerations described in the literature sources, as well as their respective sources. As have been noted previously, some requirements are derived from 'larger' requirements or considerations.

Next, in the fourth column are named the requirements for integrating a smart city with a cloud infrastructure. As the cloud requirements, also these requirements have been derived from the requirements or considerations described in the literature sources. E.g., fault tolerance is discussed under availability and resilience in the IoT RA (Internet of Things Architecture, 2013).

In the fifth column, there are examples and illustrations of the smart city requirements. Again, the reason for this is to make the tables easier to read. Finally, in the sixth column are mentioned the original names of the smart city requirements or considerations described in the literature sources, as well as their respective sources. This is again because some smart city requirements have been derived from 'larger' requirements or considerations.

The framework functions as a guiding principle that helps e.g., researchers and decision-makers to map, among other things, what a smart city data warehouse requires of cloud data management systems in general. An efficient integration of a smart city data warehouse with a cloud infrastructure means that requirements for smart city data management match, more or less, requirements for, or characteristics of, cloud data management. With the help of the framework, it can also be decided what are the most important requirements for some individual case. As the requirements are quite general and abstract, the application of the framework requires in practice technical expertise and measurements.

As it can be seen from the framework, the connection between smart cities and cloud computing is loose in some cases, which means that there are somewhat similar requirements both for cloud computing and smart cities, but their focus is different. E.g., under service-centric issues there is a need for a cloud computing system to be autonomic (Rimal et al., 2011). In regard to a smart city, there is also a need for autonomicity under self-organization capabilities. However, this does not refer to an autonomic cloud computing system, but to the ability of smart objects to autonomously react to different situations (Miorandi et al., 2012).

As for cloud computing, the ability to run in a heterogeneous environment refers to the fact that it is nearly impossible to get homogeneous performance across hundreds or thousands of computing nodes, but performance should not be degraded due to e.g., failures (Sakr et al., 2011; see also Abouzeid et al., 2009). In regard to smart cities, there is also a large heterogeneity, but in terms of devices that are expected to present very different capabilities from the computational and communication standpoints (Miorandi et al., 2012). There are many kinds of mobile devices with localization and tracking capabilities that implement real-time monitoring, as well as distributed sensing and processing virtually all the time (Miorandi et al., 2012; Da Silva et al., 2013).

There is a need for interoperability both in cloud computing and smart city solutions. As for cloud computing, interoperability refers to the migration and integration of applications and data between different cloud systems (Rimal et al., 2011). In regard to smart cities, the most similar requirement is evolution and interoperability that refers to the fact that requirements change and software evolves sometimes rapidly and needs to interoperate not only with today's technologies, but possibly also with later technologies (Internet of Things Architecture, 2013).

All in all, interoperability seems to be very vital requirement in smart cities. There is a need for interoperability between proximity wireless technologies. There is also a need for semantic interoperability and data management that requires, among other things, providing data with adequate and standardized formats, models, and metadata (Miorandi et al., 2012). Objects, abstractions of sensors, actuators, or any devices, need to be interoperable. Service composition and integrated urban management refers, among other things, to the fact that services must be interoperable so that other services can reuse, group, or create a composition using them. Then again, flexibility/extensibility refers to evolution. Changes, adaptations, and extensions should be foreseen in the architecture. (Da Silva et al., 2013.)

In a number of requirements the connection is firmer. There is clearly a need for fault tolerance, privacy, (data) security, scalability, user involvement, availability, performance, and sustainability both in cloud computing solutions and smart cities. Of these requirements, especially performance and scalability are crucial requirements in regard to cloud data management. The IoT will be much about exchanging and analyzing massive amounts of data (Miorandi et al., 2012), so a cloud data management system has to be able to support e.g., very large databases with very high request rates at very low latency (Sakr et al., 2011; see also Cooper et al., 2009).

In summary, it can be interpreted that a smart city requires of a cloud infrastructure at least availability, autonomicity, scalability, performance, interoperability, and fault tolerance, as well as privacy and security. Of the 'softer' non-technical requirements user involvement and sustainability cannot be left aside.

TABLE 3 Integrating smart city requirements with general cloud computing requirements

Cloud computing requirement	Examples and illustrations	Literature source	Smart city requirement	Examples and illustrations	Literature source
Service-centric issues	Autonomic, self-describiting, low cost composition of distributed applications (Rimal et al., 2011).	Service-centric issues (Rimal et al., 2011).	Autonomicity	Distributing intelligence in the system, making smart objects able to autonomously react to a wide range of different situations in order to minimize human intervention (Miorandi et al., 2012).	Self-organization capabilities (Miorandi et al., 2012).
Fault tolerance	Fault isolation to the falling components, availability of reversion mode, etc. (Rimal et al., 2011).	Fault tolerance (Rimal et al., 2011).	Fault tolerance	The ability of a system to effectively handle failures that could affect system's availability (Internet of Things Architecture, 2013).	Availability and resilience (Internet of Things Architecture, 2013).
Privacy	Data that the user would regard as his personal intellectual property will be stored at mega data centers located around the world (Rimal et al., 2011).	User-centric privacy (Rimal et al., 2011).	Privacy	Privacy is an ability of the system to ensure that the collection of personally identifying information be minimized and that collected data should be used locally wherever possible (Internet of Things Architecture, 2013).	Trust, security, and privacy (Internet of Things Architecture, 2013), embedded security and privacy-preserving mechanisms (Miorandi et al., 2012).
Security	A data center holds the information that would more traditionally be stored on the end-user's computer (Rimal et al., 2011).	Security (Rimal et al., 2011).	Security	Security is an ability of the system to enforce the intended confidentiality, integrity, and service access policies, and to detect and recover from failure in these security mechanisms (Internet of Things Architecture, 2013).	Trust, security, and privacy (Internet of Things Architecture, 2013), embedded security and privacy-preserving mechanisms (Miorandi et al., 2012), security (Gubbi et al., 2013).
Interoperability	The creation of an agreed-upon framework/ontology, open data format, or open protocols/APIs enabling migration and integration between cloud service providers and facilitating secure information exchange across platforms (Rimal et al., 2011).	Interoperability (Rimal et al., 2011).	Evolution and interoperability	Requirements change and software evolves sometimes rapidly and needs to interoperate not only with today's technologies, but possibly also with later technologies (Internet of Things Architecture, 2013).	Evolution and interoperability (Internet of Things Architecture, 2013), ubiquitous data exchange through proximity wireless technologies, semantic interoperability and data management (Miorandi et al., 2012), objects interoperability, service composition and integrated urban management, flexibility/extensibility (Da Silva et al., 2013).
Scalability	DHT, column-orientation, and horizontal partitioning (Rimal et al., 2011).	Scalability (Rimal et al., 2011).	Scalability	The ability of the system to handle increased processing volumes in the future if required (Internet of Things Architecture, 2013).	Performance and scalability (Internet of Things Architecture, 2013), scalability (Gubbi et al., 2013).
User experience (UX)	UX-driven design and deployment (Rimal et al., 2011).	User experience (UX) (Rimal et al., 2011).	User involvement	The main purpose in designing a smart city is to increase the quality of life of its citizens. People need to be involved and benefit from the process. (Da Silva et al., 2013.)	Social aspects (Da Silva et al., 2013).

TABLE 4 Integrating smart city requirements with cloud data management requirements

Cloud data management requirement	**Examples and illustrations**	**Literature source**	**Smart city requirement**	**Examples and illustrations**	**Literature source**
Data storage device type	HDDs, SSDs, hybrid hard disks (Rimal et al., 2011).	Data management, storage, and processing (Rimal et al., 2011).	Sustainability	Minimizing IoT entities' energy spent for communication/computing purposes (Miorandi et al., 2012).	Energy-optimized solutions (Miorandi et al., 2012), sustainability (Da Silva et al., 2013).
Programming model	MapReduce is not a perfect fit for all tasks (Rimal et al., 2011).	Data management, storage, and processing (Rimal et al., 2011).	IoT application-specific framework	Developing IoT applications using low-level cloud programming models and interfaces, e.g., Thread and MapReduce, is complex. To overcome this, there is a need for an IoT application-specific framework for rapid creation of applications and their deployment on cloud infrastructures. (Gubbi et al., 2013.)	IoT application-specific framework (Gubbi et al., 2013).
Automatic	Underlying infrastructure changes can be made quickly and without human intervention (Wu et al., 2010).	Automatic (Wu et al., 2010).	Autonomicity	Distributing intelligence in the system, making smart objects able to autonomously react to a wide range of different situations in order to minimize human intervention (Miorandi et al., 2012).	Self-organization capabilities (Miorandi et al., 2012).
Availability	Cloud data management system has to be always accessible (Sakr et al., 2011; see also Cooper et al., 2009).	Availability (Sakr et al., 2011; see also Cooper et al., 2009).	Availability	The ability of the system to stay operational (Internet of Things Architecture, 2013).	Availability and resilience (Internet of Things Architecture, 2013), availability (Da Silva et al., 2013).
Scalability	Cloud storage needs to scale quickly and to tremendous capacities (Wu et al., 2010). Cloud data management system has to be able to support very large databases with very high request rates at very low latency (Sakr et al., 2011; see also Cooper et al., 2009).	Scalability (Wu et al., 2010), scalability (Sakr et al., 2011; see also Cooper et al., 2009).	Scalability	The ability of the system to handle increased processing volumes in the future if required (Internet of Things Architecture, 2013). Scalability issues at different levels (Miorandi et al., 2012).	Performance and scalability (Internet of Things Architecture, 2013), scalability (Miorandi et al., 2012), historical data (Da Silva et al., 2013).
Privacy	The storage of personal/enterprise sensitive data (Rimal et al., 2011).	User-centric privacy (Rimal et al., 2011).	Privacy	Privacy is an ability of the system to ensure that the collection of personally identifying information be minimized and that collected data should be used locally wherever possible (Internet of Things Architecture, 2013).	Trust, security, and privacy (Internet of Things Architecture, 2013), embedded security and privacy-preserving mechanisms (Miorandi et al., 2012), privacy (Da Silva et al., 2013).
Data security	Cloud storage providers have to establish multi-tenancy policies to allow e.g., separate companies to securely share the same storage hardware (Wu et al., 2010).	Data security (Wu et al., 2010).	Security	Security is an ability of the system to enforce the intended confidentiality, integrity, and service access policies, and to detect and recover from failure in these security mechanisms (Internet of Things Architecture, 2013).	Trust, security, and privacy (Internet of Things Architecture, 2013), embedded security and privacy-preserving mechanisms (Miorandi et al., 2012), security (Gubbi et al., 2013).
Performance	A proven storage infrastructure providing fast, robust data recovery. Important to measure and test network latency before committing to a migration. (Wu et al., 2010.) Efficient system performance is a crucial requirement to save money (Sakr et al., 2011; see also Abouzeid et al., 2009).	Performance, latency (Wu et al., 2010), performance (Sakr et al., 2011; see also Abouzeid et al., 2009), efficiency (Abadi, 2009).	Performance	The ability of the system to predictably execute within its mandated performance profile (Internet of Things Architecture, 2013).	Performance and scalability (Internet of Things Architecture, 2013), efficiency (Gubbi et al., 2013).
Fault tolerance	As for transactional workloads, recovering from a failure without losing any data or updates from recently committed transactions (Sakr et al., 2011; see also Abouzeid et al., 2009). A fault tolerant analytical DBMS is one that does not have to restart a query if one of the nodes involved in query processing fails (Abadi, 2009).	Fault tolerance (Sakr et al., 2011; see also Abouzeid et al., 2009), fault tolerance (Abadi, 2009).	Fault tolerance	The ability of a system to effectively handle failures that could affect system's availability (Internet of Things Architecture, 2013).	Availability and resilience (Internet of Things Architecture, 2013).

Cloud data management requirement	**Examples and illustrations**	**Literature source**	**Smart city requirement**	**Examples and illustrations**	**Literature source**
Ability to run in a heterogenous environment	A cloud data management system has to take measures to prevent degrading performance due to parallel processing on distributed nodes (Sakr et al., 2011; see also Abouzeid et al., 2009).	Ability to run in a heterogenous environment (Sakr et al., 2011; see also Abouzeid et al., 2009), ability to run in a heterogenous environment (Abadi, 2009).	Devices hetero-geneity	The IoT will be characterized by a large heterogeneity in terms of devices, which are expected to present very different capabilities from the computational and communication standpoints (Miorandi et al., 2012).	Devices heterogeneity, localization and tracking capabilities (Miorandi et al., 2012), mobility, real-time monitoring, distrib-uted sensing and processing (Da Silva et al., 2013).
Energy effi-ciency	Green storage technology leads to a lower carbon footprint (Wu et al., 2010).	Energy efficiency (Wu et al., 2010).	Sustainability	Minimizing IoT entities' energy spent for communica-tion/computing purposes (Miorandi et al., 2012).	Energy-optimized solutions (Miorandi et al., 2012), sustainabil-ity (Da Silva et al., 2013).

5 RESEARCH METHOD OF THE STUDY

This chapter is organized as follows. First, design science is briefly introduced. Then, the research process of the study is presented. Next, the central concepts of the study are dealt with. Finally, the research method of the study is explored.

5.1 Introduction of design science

There are many ways to look at design science. Gregor (2006; see also Gregor & Jones, 2007) examines the structural nature of theory in the discipline of information systems (IS) in which she finds five types of theory: analysis (what is), explanation (what is, how, why, when, and where), prediction (what is and what will be), explanation and prediction (what is, how, why, when, where, and what will be), and design and action (how to do something). The distinguishing attribute of theories for design and action is, as already mentioned, that they focus on 'how to do something.' They give explicit prescriptions on how to design and develop an artifact, whether it is a technological product or a managerial intervention. (Gregor & Jones, 2007.)

The term *artifact* is used to describe something that is artificial or constructed by humans, as opposed to something that occurs naturally (Gregor & Jones, 2007; see also Simon, 1996). According to March and Smith (1995), IT artifacts are of four types: *constructs* or concepts (form the vocabulary of a domain), *models* (a set of propositions or statements expressing the relationships among constructs), *methods* (a set of steps (an algorithm or guideline) used to perform a task), and *instantiations* (a realization of an artifact in its environment).

According to Hevner et al. (2004), two paradigms characterize much of the research in the discipline of IS: behavioral science and design science. The *behavioral science* paradigm has its roots in natural science research methods. It seeks to develop and justify theories, i.e., principles and laws, which explain or predict organizational and human phenomena surrounding the analysis, design, implementation, management, and use of information systems. (Hevner et al., 2004.) *Design science* has its roots in engineering and the sciences of the artificial (Hevner et al., 2004; see also Simon, 1996). It is fundamentally a problem-solving paradigm. It creates and evaluates IT artifacts intended to solve identified organizational problems. Such artifacts are represented in a structured form that may vary from software, formal logic, and rigorous mathematics to informal natural language descriptions. In brief, the goal of behavioral science research is truth, while the goal of design science research is utility. (Hevner et al., 2004.)

One other way to see what design science is about is to look at Järvinen and Järvinen's taxonomy of research methods in which design research belongs to the research approaches studying reality – not stressing what is reality, but stressing the utility of innovations (Järvinen, 2012). Järvinen (2012) also notes that research is normally divided to basic and applied research. The purpose of the *basic research* is to find out what is a part of reality. The knowledge of the basic research, the basic laws of the explanatory sciences, are applied to the *applied research* that e.g., design science represents. (Järvinen, 2012.)

According to Kaplan (1964, as cited in March & Smith, 1995), natural science is often viewed as consisting of two activities, discovery and justification. Discovery is the process of generating or proposing scientific claims, e.g., theories and laws. Justification includes activities by which such claims are tested for validity. Design science consists of two basic activities, building and evaluating, which parallel the discovery-justification pair from natural science. *Building* is the process of constructing an artifact for a specific purpose. *Evaluation* is the process of determining how well the artifact performs. It requires the development of metrics and the measurement of artifacts according to those metrics. (March & Smith, 1995.)

5.2 Research process of the study

Peffers, Tuunanen, Rothenberger, and Chatterjee (2007) present in their paper the *design science research methodology (DSRM)* that comprises, among other things, a nominal process model for doing design science research. A simplified version of this process model, from Ostrowski's, Helfert's, and Xie's (2012) paper, is depicted in the figure 8. The process model itself is the same in both of these papers. From Ostrowski et al.'s (2012) model is only omitted the fact that in reality, a research process may begin at almost any step of the process (Peffers et al., 2007). This study was, however, somewhat straightforward, so there is no need to discuss this matter any further.

Peffers et al.'s (2007) DSRM process model includes six steps: problem identification and motivation, definition of the objectives for a solution, design and development, demonstration, evaluation, and communication. Next, these steps and their implementation in this study are presented.

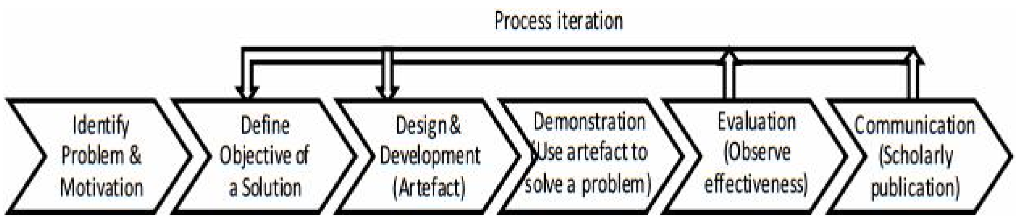
Process iteration
Identify Problem & Motivation
Define Objective of a Solution
Design& Development (Artefact)
Demonstration (Use artefact to solve a problem)
Evaluation (Observe effectiveness)
Communication (Scholarly publication)

FIGURE 8 DSRM process model (Ostrowski et al., 2012, 4075)

1. Problem identification and motivation. Define the specific research problem and justify the value of a solution. Since the problem definition will be used to develop an artifact that can effectively provide a solution, it may be useful to atomize the problem conceptually so that the solution can capture its complexity. Justifying the value of a solution accomplishes two things: it motivates the researcher and the audience of the research to pursue the solution and to accept the results and it helps to understand the reasoning associated with the researcher's understanding of the problem. Resources required for this activity include knowledge of the state of the problem and the importance of its solution. (Peffers et al., 2007.)

The Kangas project is the main urban development project of the City of Jyväskylä for the next several decades (Jyväskylän kaupunki, 2011). The Kangas area is introduced later on, but in brief, it will form a smart city in the future. This project requires implementing, but first, planning for many things. One of them is the data warehouse of the area. It was decided at the University of Jyväskylä that the data warehouse will be built on the cloud with the help of the university's hardware, network, and other resources, e.g., Eucalyptus cloud software that is introduced later on. The two candidates for the software of the data warehouse, Stardog and Neo4j, are also introduced later on.

Before implementing the data warehouse, there is, however, a need to know how a smart city data warehouse can be efficiently integrated with a cloud infrastructure in general. This requires knowledge of the requirements for smart cities, especially their data management, and the requirements for cloud computing systems, especially their data management. In the research literature exist many requirements for smart cities and their data management, as well as cloud computing systems and their data management, but before this study, there appeared to be no generalizable framework that would have integrated these requirements with each other. Hence, it was realized that this kind of framework could be useful e.g., to researchers and decision-makers.

2. Definition of the objectives for a solution. Infer the objectives of a solution from the problem definition and knowledge of what is possible and feasible. The objectives can be quantitative, e.g., terms in which a desirable solution would be better than current ones, or qualitative, e.g., a description of how a new artifact is expected to support solutions to problems not hitherto addressed. The objectives should be inferred rationally from the problem specification. Resources required for this include knowledge of the state of problems and current solutions, if any, and their efficacy. (Peffers et al., 2007.)

The main objective of this study has been to build the aforementioned framework and answer with the help of it to the main research question: how a smart city data warehouse can be efficiently integrated with a cloud infrastructure? As already mentioned, there appeared to be no such framework in existence before this study. Answering to the main research question has also required answering to the sub-questions of this study: What is cloud computing? What is cloud data management? What are the requirements for cloud data management? What are smart cities? What are the requirements for smart city data management? Getting answers to these questions has formed the sub-objectives of this study.

3. Design and development. Create the artifact. Such artifacts are potentially constructs, models, methods, or instantiations (each defined broadly) (Hevner, March & Park, 2004, as cited in Peffers et al., 2007) or 'new properties of technical, social, and/or informational resources' (Järvinen, 2007, as cited in Peffers et al., 2007). Conceptually, a design research artifact can be any designed object in which a research contribution is embedded in the design. This activity includes determining the artifact's desired functionality and its architecture and then creating the actual artifact. Resources required moving from objectives to design and development include knowledge of theory that can be brought to bear in a solution. (Peffers et al., 2007.)

In this activity, the artifact, i.e., the aforementioned framework, was created by conducting a somewhat extensive literature review. The literature review was conducted in practice so that in the course of it the requirements for smart cities, especially their data management, and the requirements for cloud computing systems, especially their data management, were looked for in many scholarly papers and other publications. Then, the most relevant of these requirements were integrated with each other to form the framework that functions as a guiding principle, being able to answer to many questions, e.g., the main research question. The framework and its operating principle have been presented earlier.

4. Demonstration. Demonstrate the use of the artifact to solve one or more instances of the problem. This could involve its use in experimentation, simulation, case study, proof, or other appropriate activity. Resources required for the demonstration include effective knowledge of how to use the artifact to solve the problem. (Peffers et al., 2007.)

The use of the artifact, i.e., the aforementioned framework, was demonstrated by solving one instance of the problem that is in this case the Kangas project. In practice, the framework was examined by the author and supervisors of this thesis, and together they chose the most important requirements for the data warehouse of the Kangas area: performance and scalability. In the next step, they are operationalized, after which two candidates for the software of the data warehouse, Stardog and Neo4j, are tested for them. The selection of Stardog and Neo4j was made solely by the author who had several reasons for their selection: their data models are suitable for smart cities, their installation and usage is possible for a person who is not a database expert and who has at most intermediate level of knowledge of various Linux distributions, their documentations are of high quality, they have out-of-the-box distributed database management system (DDBMS) functionalities, and their licenses do not prohibit benchmarking them against other database products and publishing such benchmarking results. Many other promising databases could not be utilized e.g., due to these reasons. The top candidates were AllegroGraph, Virtuoso, OWLIM, 4store, Bigdata, Apache Jena, Sesame, OrientDB, and MongoDB.

5. Evaluation. Observe and measure how well the artifact supports a solution to the problem. This activity involves comparing the objectives of a solution to actual observed results from use of the artifact in the demonstration. It requires knowledge of relevant metrics and analysis techniques. Depending on the nature of the problem venue and the artifact, evaluation could take many forms, such as quantifiable measures of system performance, e.g., response time or availability. Conceptually, such evaluation could include any appropriate empirical evidence or logical proof. At the end of this activity the researchers can decide whether to iterate back to step three to try to improve the effectiveness of the artifact or to continue on to communication and leave further improvement to subsequent projects. The nature of the research venue may dictate whether such iteration is feasible or not. (Peffers et al., 2007.)

In this step, the artifact, i.e., the aforementioned framework, is evaluated by benchmarking the performance of two candidates for the software of the data warehouse, Stardog and Neo4j, and comparing them subjectively. The original plan was to benchmark also scalability of the chosen databases (see previous step), but this proved to be too difficult. Such an attempt would have required setting up distributed databases and putting heavy load on them. Since attempts to get Stardog Cluster's beta version (Stardog's DDBMS functionality) working were not successful and the author lacks computing resources to create heavy load, the plan for benchmarking scalability of Stardog and Neo4j was abandoned. However, their performance is benchmarked by conducting several performance tests, after which the results of these tests are analyzed to decide which database performs better in this case. Then, Stardog and Neo4j are compared subjectively as well, and finally, based on all these experiences, the framework itself is evaluated. Its further improvement is left for future researchers.

6. Communication. Communicate the problem and its importance, the artifact, its utility and novelty, the rigor of its design, and its effectiveness to researchers and other relevant audiences, e.g., practicing professionals when appropriate. Communication requires knowledge of the disciplinary culture. (Peffers et al., 2007.)

The results of this study, the most important of them being the aforementioned framework and its evaluation, as well as the comparison of Stardog and Neo4j, are published as a master's thesis by the author and the University of Jyväskylä. The results are available both in electronic and paper form.

5.3 Introduction of the central concepts of the study

Next, the central concepts of the study are introduced. First, Amazon Web Services and Eucalyptus cloud software are presented. Then, the Kangas area of the City of Jyväskylä is introduced. Next, two candidates for the software of the data warehouse, Stardog and Neo4j, are dealt with. Finally, a performance testing tool, Apache JMeter, is presented.

5.3.1 Amazon Web Services (AWS)

Amazon Web Services (AWS) began offering IT infrastructure services to businesses in the form of web services, now commonly known as cloud computing, in 2006. Today, AWS provides a highly reliable, scalable, and low-cost infrastructure platform in the cloud that powers hundreds of thousands of businesses in 190 countries around the world. (Amazon Web Services, 2014a.) AWS offers currently dozens of services, with more being added each year (Amazon Web Services, 2014b). It is not possible to describe all of them here, but as far as the author knows, the most famous of these services are Amazon Elastic Compute Cloud and Amazon Simple Storage Service. Next, they are briefly introduced alongside Amazon EC2 Instance Store and Amazon Elastic Block Store that are central to Amazon Elastic Compute Cloud.

Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in the AWS cloud. A customer can use Amazon EC2 to launch as many or as few virtual servers as he needs, configure security and networking, and manage storage. Amazon EC2 provides many features, e.g., virtual computing environments (instances), preconfigured templates for instances (Amazon Machine Images, AMIs) including the operating system and additional software, various configurations of CPU, memory, storage, and networking capacity for instances (instance types), storage volumes for temporary data (instance store volumes), persistent storage volumes (Amazon Elastic Block Store volumes), and multiple physical locations (regions and availability zones) for resources. (Amazon Web Services, 2014c.) An example of an instance type that could be transformed into an instance later on is m3.medium with one vCPU (Intel Xeon E5-2670 v2 Ivy Bridge), 3.75 GiBs of RAM, and 4 GBs of SSD-based instance storage (Amazon Web Services, 2014d).

Amazon EC2 Instance Store provides temporary block-level storage for use with an instance. The size of an instance store ranges from 900 MiBs to up to 48 TiBs and varies by instance type. An instance store consists of one or more instance store volumes. When a customer launches an instance store-backed AMI, each instance store volume available to the instance is automatically mapped. Otherwise, volumes have to be formatted and mounted on the running instance before they can be used. Instance store volumes are usable only from a single instance during its lifetime. Data on instance store volumes is lost e.g., when terminating an instance. Hence, instance store volumes are ideal for temporary storage of information that changes frequently. (Amazon Web Services, 2014e.)

Amazon Elastic Block Store (Amazon EBS) also provides block-level storage volumes for use with Amazon EC2 instances. However, unlike instance store volumes, Amazon EBS volumes that are attached to an Amazon EC2 instance are exposed as storage volumes that persist independently from the life of the instance. A customer can create Amazon EBS storage volumes from 1 GiBs to 1 TiBs in size and mount them as devices on his Amazon EC2 instances. Amazon EBS volumes behave like raw, unformatted block devices. A customer can create a file system on top of these volumes or use them in any other way he would use a block device, e.g., a hard drive. Multiple volumes can be mounted on the same instance. Amazon EBS is recommended when data changes frequently and requires long-term persistence. (Amazon Web Services, 2014f.)

Amazon Simple Storage Service (Amazon S3) is a data storage infrastructure that consists of buckets and objects (Amazon Web Services, 2014g, 2014h). A *bucket* is a container for objects stored in Amazon S3. Every object is contained in a bucket. Buckets organize the Amazon S3 namespace at the highest level, identify the account responsible for storage and data transfer charges, play a role in access control, etc. *Objects* are the fundamental entities stored in Amazon S3. They consist of object data and metadata. Every object in Amazon S3 can be uniquely addressed through the combination of the web service endpoint, bucket name, key, and optionally, a version. E.g., in the URL *http://doc.s3.amazonaws.com/2006-03-01/AmazonS3.wsdl* 'doc' is the name of the bucket and '2006-03-01/AmazonS3.wsdl' is the key. Each object can contain up to 5 TBs of data, but there are no boundaries to how many objects and how much data can be stored in each bucket. (Amazon Web Services, 2014h.)

The Amazon S3 architecture is designed to be programming language-neutral, using REpresentational State Transfer (REST) and SOAP interfaces to store and retrieve objects. A customer can choose the geographical region in which Amazon S3 stores the buckets he creates. (Amazon Web Services, 2014h.) Regions refer to the fact that Amazon EC2 is hosted in multiple locations worldwide. Each *region* is a separate geographic area and has multiple, isolated locations known as *availability zones*. Each region is completely independent, and as already mentioned, each availability zone is isolated, however, they are connected in a region through low-latency links. The isolated regions achieve the greatest possible fault tolerance and stability. (Amazon Web Services, 2014i.) Certain region might be chosen for different reasons, e.g., to optimize latency, minimize costs, or address regulatory requirements (Amazon Web Services, 2014h).

5.3.2 Eucalyptus cloud software

Elastic Utility Computing Architecture for Linking Your Programs to Useful Systems (Eucalyptus) (Wolski et al., 2008) began as a research project in the Computer Science Department at the University of California, Santa Barbara, originating from the Virtual Grid Application Development Software (VGrADS) project (2003–2008) (Eucalyptus Systems, 2014a; VGrADS, 2009). The Eucalyptus project began to commercialize as an open source company in 2009 (Eucalyptus Systems, 2014a). Today, Eucalyptus is open source software for building AWS-compatible private and hybrid clouds. It leverages a customer's existing IT infrastructure to create a self-service private cloud behind his firewall. IaaS is enabled with the private cloud by abstracting the available heterogeneous compute, network, and storage resources. Eucalyptus claims to be the only solution that can transform a customer's IT infrastructure into a private cloud that works like AWS. Eucalyptus is compatible with AWS APIs, e.g., EC2, EBS, and S3. (Eucalyptus Systems, 2014b.)

Eucalyptus' services are many, and it is thus impossible to describe all of them here. However, Eucalyptus' main components can be introduced. Eucalyptus is made up of six distinct components that can be distributed in various cloud computing architectures. The six components are grouped into three separate levels, as depicted in the figure 9 below. (Eucalyptus Systems, 2014c.)

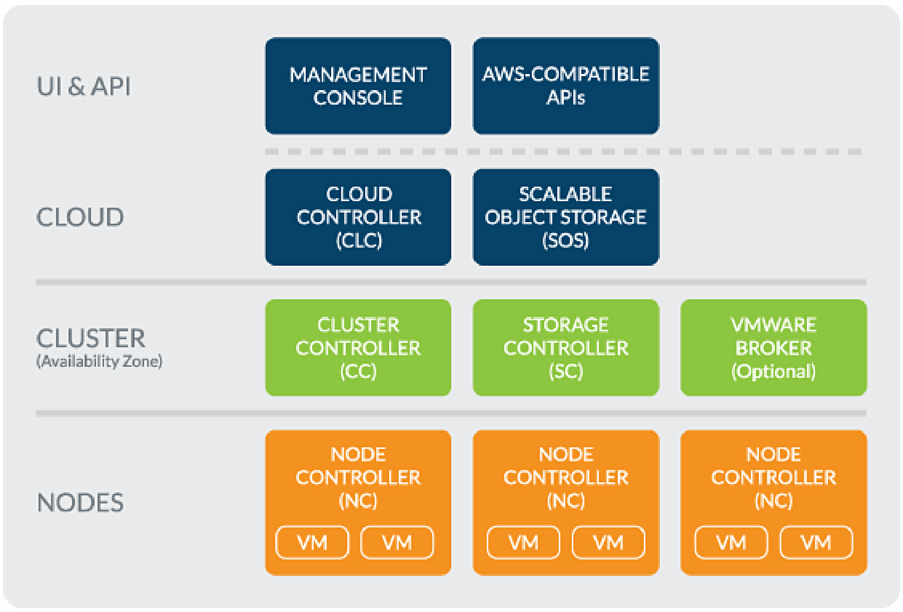

FIGURE 9 Main components of Eucalyptus (Eucalyptus Systems, 2014c)

The *Cloud Controller (CLC)* is a Java program that offers EC2-compatible SOAP and Query interfaces, as well as a web interface to the outside world, for distribution within the cloud architecture. In addition to handling incoming requests, the CLC acts as the administrative interface for cloud management and performs high-level resource scheduling and system accounting. The CLC accepts user API requests from command-line interfaces, e.g., euca2ools, or GUI-based tools (graphical user interface), e.g., the Eucalyptus Management Console, and manages the underlying compute, storage, and network resources. In brief, the CLC handles high-level authentication, accounting, reporting, and quota management. Only one CLC can exist per cloud. (Eucalyptus Systems, 2014d.)

Scalable Object Storage (SOS) is the Eucalyptus service equivalent to Amazon S3. The SOS is a pluggable service that allows infrastructure administrators the flexibility to implement scale-out storage on top of commodity resources using open source and commercial solutions that implement the S3 interface. Eucalyptus provides a basic storage implementation, Walrus, which may suit evaluation and smaller cloud deployments. For large-scale and increased performance, users are encouraged to connect the SOS to dedicated storage solutions, e.g., RiakCS. (Eucalyptus Systems, 2014d.)

A *cluster* is equivalent to an AWS availability zone, and a single Eucalyptus cloud can have multiple clusters. The *Cluster Controller (CC)* is written in C and acts as the front end for a cluster within a Eucalyptus cloud and communicates with the Storage Controller and Node Controller. The CC manages instance execution and SLAs per cluster. (Eucalyptus Systems, 2014e.)

The *Storage Controller (SC)* is written in Java and is the Eucalyptus equivalent to Amazon EBS. The SC communicates with the CC and Node Controller within the distributed cloud architecture and manages Eucalyptus block volumes and snapshots to the instances within its specific cluster. If an instance requires writing persistent data to memory outside of the cluster, it would need to write to the backend storage that is available to any instance in any cluster. The SC interfaces with storage systems including local, NFS, Internet Small Computer System Interface (iSCSI), and Storage Area Network (SAN). (Eucalyptus Systems, 2014e.)

The *VMware Broker* is an optional component that provides an AWS-compatible interface for VMware environments and physically runs on the CC within the distributed cloud computing architecture. The VMware Broker overlays existing ESX/ESXi hosts and transforms Eucalyptus Machine Images (EMIs) to VMware virtual disks. The VMware Broker mediates interactions between the CC and VMware and can connect directly to either ESX/ESXi hosts or to vCenter Server. (Eucalyptus Systems, 2014e.)

Finally, the *Node Controller (NC)* is a part of the node level of the cloud computing architecture. It is written in C and hosts the virtual machine instances and manages the virtual network endpoints. The NC downloads and caches images from the SOS, as well as creates and caches instances. (Eucalyptus Systems, 2014f.)

5.3.3 Kangas area

The Kangas area has been a place for new ideas, businesses, and innovations a long time. Already in the middle of the 1800s it was filled with water mills, and later on, there were all kinds of businesses wanting to get their share of the river Tourujoki. The Kangas paper mill was one of the first Finnish paper mills, opening in 1874. The paper mill was finally closed in 2010, and the area of 27 hectares became the property of the City of Jyväskylä in 2011. (Jyväskylän kaupunki, 2011.)

According to the City of Jyväskylä (Jyväskylän kaupunki, 2011), the main development themes of the Kangas area are based on the city's strategies, a need for new housing areas, and ideas put forward by citizens. These dreams and ideas were transformed into the desired user experience and the more general qualitative targets into concepts to enhance the everyday usability of Kangas in the future (Jyväskylän kaupunki, 2011).

The concepts can be crystallized into four words: heart, feet, sustainable, and green. *Heart* refers to the heart of the area, the old paper mill that will be a hot spot offering many kinds of services, e.g., grocery stores, restaurants, cafés, shops, a gym and day spa services, arts and crafts, and flea markets. *Feet* are about an attractive pedestrian and cycling environment, 'people first, parking underground.' *Sustainable* stands for sustainable development including a dense urban structure, an ecological way of life, and a carbon neutral city. Finally, *green* denotes recreation, water and green areas, from nature preservation areas to parks, balconies, and rooftops. (Jyväskylän kaupunki, 2011.)

As already mentioned, the Kangas project is the main urban development project of the City of Jyväskylä for the next several decades. In the future, the Kangas area will be a home to 5000 inhabitants and 2000 new jobs. (Jyväskylän kaupunki, 2011.)

5.3.4 Stardog, an RDF database

Stardog is a graph database provided by Clark & Parcia and implemented in Java. To be precise, Stardog is a resource description framework (RDF) database. (Clark & Parcia, 2014a.) According to solid IT's (2014) DB-Engines Ranking of November 2014, Stardog is the sixth most popular RDF store. In addition to the RDF data model, Stardog supports SPARQL 1.1 query language, HTTP and SNARL protocols for remote access and control, ACID, Web Ontology Language (OWL) 2, rules for inference and data analytics, Java, JavaScript, Ruby, Python, .Net, Groovy, Spring, Clojure, etc. (Clark & Parcia, 2014a, 2014b).

As already mentioned, Stardog's data model is based on RDF. RDF is a framework for expressing information about *resources* that can be anything, e.g., documents, people, physical objects, and abstract concepts. In practice, RDF consists of *statements* about resources. Statements are called *triples*, because they comprise three elements: a subject, object, and predicate. The *subject* and *object* represent some two resources that are related, while the *predicate* represents the nature of their relationship. The relationship is phrased in a directional way, from a subject to an object, and is called a *property*. *International Resource Identifiers (IRIs)* identify resources. *Literals* are basic values that are not IRIs. (W3C, 2014a.) SPARQL 1.1 is a set of specifications that provide languages and protocols to query and manipulate RDF graph content on the web or in an RDF store (W3C, 2013), e.g.,:

```
PREFIX sm: <http://www.jkl.fi#>
INSERT DATA
{ sm:Sensor1 sm:IsLocated sm:Place1 ;
            sm:SensorType "Temperature sensor" }
```

The aforementioned SPARQL query would create two triples to a RDF database, e.g., Stardog. 'Temperature sensor' is a literal, while the other resources are identified by IRIs, being thus unique:

```
<http://www.jkl.fi#Sensor1> <http://www.jkl.fi#IsLocated> <http://www.jkl.fi#Place1>
<http://www.jkl.fi#Sensor1> <http://www.jkl.fi#SensorType> Temperature sensor
```

There are three Stardog editions available: Community, Developer, and Enterprise. Community is provided free of charge with some limitations, e.g., 10 databases, 25 MBs triples per a database, etc. (Clark & Parcia, 2014b.) Developer is a 30-day trial edition of the full Enterprise that offers e.g., no data limits, support by phone or e-mail (Clark & Parcia, 2014b), and Stardog Cluster (Stardog's DDBMS functionality). In this thesis, Stardog Community was utilized.

5.3.5 Neo4j, a graph database

Neo4j is an open-source NoSQL graph database sponsored by Neo Technology and implemented in Java and Scala. With development beginning in 2003, Neo4j has been publicly available since 2007. (Neo Technology, 2014a.) According to solid IT's (2014) DB-Engines Ranking of November 2014, Neo4j is the world's most popular graph DBMS. Neo4j's data model is so-called property graph. Putting it simply, the *property graph data model* is a multigraph data structure in which graph elements, *vertices* and *edges*, can have properties/attributes (Ciglan, Averbuch & Hluchy, 2012). Neo4j's graphs can be accessed remotely via Cypher HTTP API, either directly or through one of the many available language drivers. As Stardog, Neo4j supports ACID as well. (Neo Technology, 2014a.)

As for the property graph data model, vertices can also be called *nodes* and edges *relationships*, such as in Neo4j's case. The records in Neo4j's databases are called nodes that are connected through typed, directed relationships. Nodes and relationships can also have named attributes referred to as *properties*. Furthermore, nodes can have *labels* that organize them into groups. (Neo Technology, 2014b.) In addition, so-called *identifiers* can be used to refer to parts of a pattern or a query (Neo Technology, 2014c). *Cypher* is Neo4j's declarative graph query language that allows for expressive and efficient querying and updating of the graph store (Neo Technology, 2014d), e.g.,:

```
CREATE (a:Sensor {SensorType: 'Temperature sensor'})-[:ISLOCATED]->(b:Place {PlaceID:1})
```

The aforementioned Cypher query creates a node labeled as 'Sensor' that has a property 'SensorType', its value being 'Temperature sensor'. The query also generates a node labeled as 'Place' that has a property 'PlaceID', its value being

'1'. Furthermore, the query creates a directed relationship between these two nodes, 'ISLOCATED'. The letters 'a' and 'b' are the identifiers of the two nodes. They could be something else as well, e.g., 'x' and 'y'.

There are four Neo4j subscriptions available: Community, Personal, Startup, and Enterprise. Community subscription is provided free of charge, as well as Personal subscription that can be utilized if certain criteria are met. Startup and Enterprise subscriptions are commercial subscriptions. They and Personal subscription include many features that Community subscription does not, e.g., commercial e-mail and phone support, high-performance cache, and clustering. (Neo Technology, 2014e.) In this thesis, Community subscription was utilized.

5.3.6 Apache JMeter, a testing tool

Apache JMeter is a Java-based open source desktop application that is designed to load test functional behavior and measure performance (Apache Software Foundation, 2014a). As International Software Testing Qualifications Board (ISTQB, 2014) puts it, load testing is a type of performance testing conducted to evaluate the behavior of a component or system with increasing load, e.g., numbers of parallel users and/or numbers of transactions, to determine what load can be handled by the component or system. Apache JMeter can be used to test performance both on static and dynamic resources (files, web dynamic languages, e.g., PHP, Java, and ASP.NET, Java objects, databases and queries, FTP servers, etc.) (Apache Software Foundation, 2014a).

Using Apache JMeter is simple, but it is useful to introduce some of its central concepts. To begin with, a *test plan* describes a series of steps that Apache JMeter executes when it is run. A test plan consists of one or more of the following elements: thread groups, logic controllers, sample generating controllers, listeners, timers, assertions, and configuration elements. (Apache Software Foundation, 2014b.) *Thread groups* simulate connections to a server application. The controls for a thread group allow e.g., to set the number of threads and the number of times to execute the test. Under a thread group, there can be two kinds of controllers: samplers and logical controllers. *Samplers* tell Apache JMeter to send requests to a server and wait for a response. *Logic controllers* let customize the logic that Apache JMeter uses to decide when to send requests. They can change the order of requests coming from their child elements, modify the requests themselves, cause Apache JMeter to repeat requests, etc. *Listeners* provide access to the information that Apache JMeter gathers about the test cases while it runs. By default, an Apache JMeter thread sends requests without pausing between each request, however, a delay can be specified by adding one of the available *timers* to a thread group. *Assertions*, as their name suggests, allow asserting facts about responses received from the server being tested, i.e., it can be tested that an application is returning the expected results. *Configuration elements* work closely with samplers. In general, they do not send requests, but they can add to or modify requests. (Apache Software Foundation, 2014c.) In addition, a user can, among other things, write *functions* and *variables* (Apache Software Foundation, 2014c) that can e.g., modify messages that are sent to a server application.

Apache JMeter is higly extensible. Many custom plugins are developed for it (jmeter-plugins.org, 2014a), and its results can be uploaded e.g., to BlazeMeter and Loadosophia.org. E.g., Loadosophia.org is a service for storing and analyzing performance tests (Loadosophia.org, 2014). It was utilized in this study as well.

5.4 Benchmark for comparing the performance of Stardog and Neo4j

Next, famous database benchmarks are briefly introduced, after which the benchmark of this thesis is validated and presented. Alongside it is presented the smart city ontology that served as an advice on creating Stardog's and Neo4j's schemas.

5.4.1 About famous database benchmarks

Over the years, various database benchmarks have been developed as a tool for comparing the performance of DBMSs and are frequently referred to in academic, technical, and commercial literature (Connolly & Begg, 2005). In this thesis, a *benchmark* refers to a test that serves as a basis for evaluation or comparison, e.g., of computer system performance (Merriam-Webster, 2014).

Perhaps the earliest DBMS benchmark was the Wisconsin benchmark that was developed to allow comparison of particular DBMS features (Bitton et al., 1983, as cited in Connolly & Begg, 2005). The Transaction Processing Council (TPC) was founded in 1988 (Connolly & Begg, 2005). Its benchmarks are also famous. E.g., TPC-C is based on an order entry application and TPC-H for ad hoc, decision-support environments in which users do not know which queries will be executed (Connolly & Begg, 2005). Graph databases, however, aim at different types of queries, and thus these widespread benchmarks are not adequate for evaluating their performance (Dominguez-Sal, Martinez-Bazan, Muntes-Mulero, Baleta & Larriba-Pey, 2011).

Object oriented databases share some similarities with graph databases (Dominguez-Sal et al., 2011). For object-oriented database management systems (OODBMSs), there are, among others, the Object Operations Version 1 (OO1) and OO7 benchmarks. OO1 was designed to reproduce operations that are common in the advanced engineering applications. In 1993, the University of Wisconsin released the OO7 benchmark, based on a more comprehensive set of tests and a more complex database. (Connolly & Begg, 2005.)

When discussing graph databases, it has to be borne in mind that they can be roughly divided in two: RDF and non-RDF databases, Stardog being an exemplar of the former and Neo4j for the latter. As for RDF databases, there are many benchmarks available. W3C's (2014b) webpage lists the most well-known. E.g., Berlin SPARQL Benchmark (BSBM) is used for measuring the performance of storage systems that expose SPARQL endpoints. The benchmark suite is built around an e-commerce use case. (Bizer & Schultz, 2012.) Other famous RDF database benchmarks are Lehigh University Benchmark (LUBM), University Ontology Benchmark (UOBM), SP^{2}Bench SPARQL Performance Benchmark, and DBpedia SPARQL Benchmark, but they are not gone into here (W3C, 2014b; see also Bizer & Schultz, 2008). There are many others as well.

In non-RDF databases' case, there are at least a few notable benchmarks. Ciglan et al. (2012) write that the lack of standards in the domain of graph databases makes it difficult to compare systems. However, one option that they themselves utilized is Blueprints that is a property graph model interface with provided implementations and a part of Tinkerpop, an open source graph computing framework (Tinkerpop, 2014). To the author's knowledge, with the help of Tinkerpop property graphs and even RDF graphs can be benchmarked against each other. This is also possible, as far as the author knows, with HPC Scalable Graph Analysis Benchmark (HPC-SGAB) (Dominguez-Sal et al., 2011; see also Graphanalysis.org, 2014) and Linked Data Benchmark Council's (LDBC) new benchmark, the Social Network Benchmark (SNB) (LDBC, 2014). Currently, SNB is unfinished (LDBC, 2014).

As applying the aforementioned benchmarks, e.g., Tinkerpop, to this case would have required a lot of technical expertise and as they did not seem to be suitable for this case, the author decided to create his own benchmark. Next, it is discussed in detail.

5.4.2 Smart city ontology

The smart city ontology whose classes are depicted in the figure 10 is an output of the author's imagination, supervisors' ideas and suggestions, and several ontologies and similar structures (e.g., Calabrese & Ratti, 2006; Calabrese, Colonna, Lovisolo, Parata & Ratti, 2011; Lertlakkhanakul, Choi & Kim, 2008; Wang, De, Cassar, & Moessner, 2013) that have served as examples to it. The OWL ontology was modeled with the help of Protégé, a free, Java-based open-source ontology editor and framework for building intelligent systems (Stanford Center for Biomedical Informatics Research, 2014).

Before describing the ontology, a short introduction of OWL 2 is in order. OWL 2 ontologies consist of *entities*: *classes* represent sets of individuals, *datatypes* are sets of literals, e.g., strings or integers, *object properties* and *data properties* can be used to represent relationships in the domain, *annotation properties* can be used to associate non-logical information with ontologies, axioms, and entities, and *named individuals* can be used to represent actual objects from the domain. They are all uniquely identified by IRIs. OWL 2 also provides for *anonymous individuals*, i.e., individuals that are analogous to blank nodes in RDF and that are accessible only from within the ontology they are used in. Finally, OWL 2 provides for *literals* that consist of a string called a lexical form and a datatype specifying how to interpret this string. (W3C, 2012.)

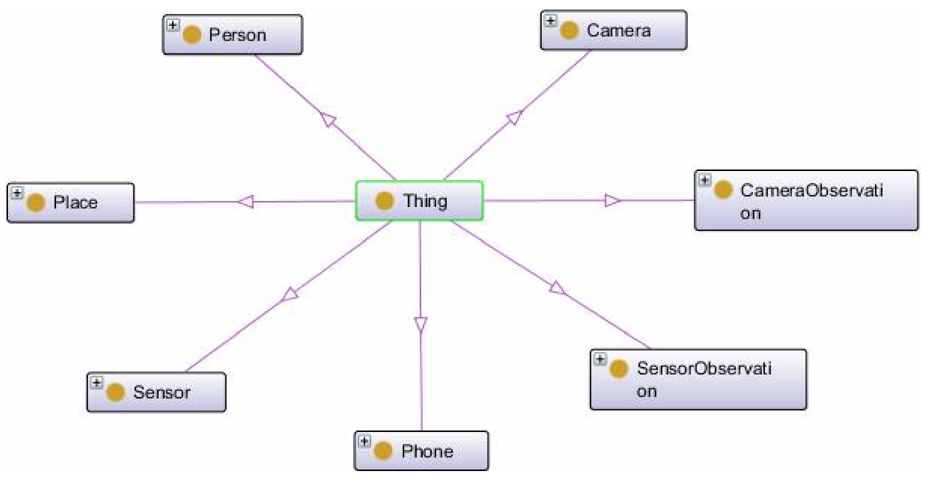

FIGURE 10 Classes of the smart city ontology visualized by Protégé

As it can be seen from the ontology, it consists of a class *Thing* and its multiple subclasses: *Place*, *Sensor*, *Camera*, *Person*, *Phone*, *SensorObservation*, and *CameraObservation*. They can all have individuals, e.g., a 'Sensor' can be 'Sensor1'. Here, the namespace of the ontology, the ontology IRI, comes into picture. In this case, it is *http://www.jkl.fi*, the domain of the City of Jyväskylä. Hence, e.g., the class 'Sensor' is identified with the IRI 'http://www.jkl.fi#Sensor' and its individual 'Sensor1' with the IRI 'http://www.jkl.fi#Sensor1'. To describe the ontology briefly otherwise, its entities have the following object properties: Sensors and cameras are located in places and observe and film observations. In addition, persons possess phones, but neither persons nor phones are located in any place. This depicts a real-life situation in which entities of a database are not all connected to each other for one reason or another. Entities also have their data properties, e.g., 'Sensor1' is a 'Temperature sensor' that is its 'SensorType' (string). The ontology is described in its entirety in the appendix 1.

The ontology is on purpose flat and simple so that it could be easily transformed into two schemas, those of Stardog's and Neo4j's. Real-life (smart city) ontologies are naturally wider and more complicated in many cases.

5.4.3 Design of the benchmark

As the ontology, also the benchmark is an output of the author's imagination, supervisors' ideas and suggestions, and several benchmarks (e.g., Vicknair et al., 2010; Jouili & Vansteenberghe, 2013) that have served as examples to it.

The benchmark was conducted on a DELL Inspiron 620 desktop computer (processor: Intel Core i3-2100 3.10 GHz, RAM: 4 GBs with a swap file of 4 GBs, operating system: Ubuntu 14.04 LTS 64-bit, Java: OpenJDK 1.7.0_65, default settings). Stardog and Neo4j were installed on Eucalyptus v.3.3.2 (instance type:

m2.2xlarge, virtual cores: 2, RAM: 4096 MBs, operating system: CentOS 6.4 (Final), Java: Java SE Development Kit 1.7.0_67, default settings). For each of the databases, the latest available versions were used: Stardog Community 2.2.2 and Neo4j Community 2.1.5. Eucalyptus ran on the university's hardware approximately 1.3 kilometers from the client computer that ran the benchmark. The network connection was a 100 Mbps LAN-based (local area network) Internet connection (Koskinen, 2012) that could not be affected.

The benchmark compared the performance of Stardog's public SPARQL endpoint (Clark & Parcia, 2014c) to Neo4j's Transactional Cypher HTTP endpoint (Neo Technology, 2014f) that are, in the author's opinion, the closest equivalents to each other. Furthermore, they are supported ways to insert data into and query it from the databases. Stardog and Neo4j were put to the test with their out-of-the-box features. The only changes were changing of their default ports to 8080, disabling Stardog's security to enable Stardog's usage as a public SPARQL endpoint (Clark & Parcia, 2014c), and configuring Neo4j to enable access from any host (Neo Technology, 2014h). The data that were inserted into the databases were not validated against any schemas. The aforementioned OWL ontology served only as an advice on creating Stardog's and Neo4j's schemas. It was not inserted into Stardog's databases at any point.

The data were inserted into the databases and queried from them with the help of Apache JMeter that acted as the 'client side' of a 'client/server' application (Halili, 2008). Hence, the performance was measured on the client side. Apache JMeter 2.12 was run in GUI mode, the only listener being Loadosophia.org Uploader 1.2.0 (jmeter-plugins.org, 2014b).

The benchmark was run twice on the morning of November 20, 2014. Stardog was put to the test first and then Neo4j. The databases were empty at the beginning of each run. Before each run, the GUIs of the databases, Stardog Web Console (Clark & Parcia, 2014d) and Neo4j Web Interface (Neo Technology, 2014g), were accessed to confirm that everything was in order. Furthermore, after each test, Loadosophia.org was visited to confirm that the results were successfully uploaded to Loadosophia.org. In addition, after the first run of the benchmark, the databases were deleted and new databases were created. In Stardog, this can be done via its GUI, while Neo4j has to be stopped before making changes to its databases. Hence, Neo4j was stopped after the first run, its database was deleted, and Neo4j was started again, which automatically created a new database. Then, Neo4j's GUI was accessed to confirm that everything was in order.

The benchmark comprised four tests that are briefly described below and in more detail in the appendix 2. Each database ran each test twice, so altogether 16 tests were run.

To begin with, the initial graph was created: First, 1000 anonymous places were created symbolizing 1000 homes and public spaces in the Kangas area. Then, 1000 temperature sensors, one for each place, were created. The sensors were imagined to measure outdoor temperature of the aforementioned places, being located outside of their windows, such as normal thermometers. Next, 250 video cameras for the first 250 places were created, one for each place. It was imagined that the cameras are located in or in the near vicinity of the places. As the sensor data, also the video data could be used e.g., for scientific purposes. Then, 1000 people were created, and finally, 1000 telephones, one for each person. The persons and their telephones were not attached to other nodes on purpose. It was imagined that this will be done later on in the future.

After the initial graph was created, it was populated by more data. The following situation simulated roughly one hour of smart city activities: 1000 temperature sensors observed 2000 temperature observations. It was imagined that each sensor observes outdoor temperature every half an hour and inserts its data into the database at the same time. In parallel, 250 video cameras filmed 7500 camera observations. It was imagined that one camera films all the time, inserting the metadata of its video data into the database every two minutes. At the same time, the actual video files are sent to someplace else, i.e., inserted into some other database that is more suitable for saving multimedia than Stardog and Neo4j.

As there were now some meaningful data in the database, it was queried. The following situation simulated roughly ten days of smart city activities: 1000 people queried their personal information, each once. It was imagined here that perhaps 10 % of the residents want to see their personal information in a day, e.g., to update it. In parallel, the average temperature of all the sensor observations was calculated 1000 times, which simulates scientific calculations on data.

Finally, data was inserted into the database and queried from it at the same time. The following situation did not simulate any real-life smart city situation: 1000 people queried their personal information, each once. In parallel, 1000 people updated their personal information, each once. A very long character string was added to each person's information, symbolizing detailed information of some kind, e.g., a self-description and other social media data.

The aforementioned figures might look small compared e.g., to the City of London that produces at least 1160000 observations in an hour (Boyle, Yates & Yeatman, 2013). There reside about 9000000 people in London (CIA, 2014). The Kangas area will be a home to 5000 people in the future (Jyväskylän kaupunki, 2011). On the basis of a simple calculation, (1160000 / 9000000) * 5000, they could produce about 644 observations in an hour. Hence, the chosen figures are reasonable.

As the ontology, the benchmark is on purpose simplified to keep it more manageable. In reality, e.g., sensors and cameras do not necessarily insert their data straight into a database. E.g., in SmartSantander repeaters and gateways gather the information that is sensed by sensors (Krčo et al., 2013). However, the author deemed more fruitful to simulate many concurrent users than a few gateways.

5.4.4 Definition of the performance in the benchmark

The concept of 'performance' has been mentioned several times in this thesis, e.g., in the framework of requirements for integrating a smart city with a cloud infrastructure. As for the performance of databases, however, there appears to be no practical definition available. Hence, the concepts of 'performance' are briefly discussed below, after which the performance in the benchmark of this thesis is defined.

According to Menascé (2002), QoS is a combination of several qualities or properties of a service, e.g., response time and throughput. *Response time* can be defined as the interval between a user's request and the system's response, but this is a simplistic definition, since the requests and the responses are not instantaneous, e.g., the system takes time outputting the response. There are (at least) two possible definitions of the response time: it can be defined as either the interval between the end of a request submission and the beginning of the corresponding response from the system or as the interval between the end of a request submission and the end of the corresponding response from the system. (Jain, 1991.) The former definition is also known as *latency*, although it has to be emphasized that there are many definitions of latency as well. The response time of a system generally increases, as the load on the system increases (Jain, 1991). QoS takes into account not only the average response time, but also the percentile (e.g., 95th percentile) of the response time (Menascé, 2002).

Throughput is defined as the rate (requests per unit of time) at which the requests can be serviced by the system. E.g., for transaction processing systems, the throughput is measured in transactions per second (TPS). The throughput of a system generally increases, as the load on the system initially increases. After a certain load, the throughput stops increasing. In most cases, it may even begin decreasing. (Jain, 1991.)

In Apache JMeter, response time is called *elapsed time*. Apache JMeter measures the elapsed time from just before sending the request to just after the last response has been received. Apache JMeter does not include the time needed to render the response, nor does Apache JMeter process any client code, e.g., JavaScript. Apache JMeter measures the latency from just before sending the request to just after the first response has been received. Hence, the time includes all the processing needed to assemble the request, as well as assembling the first part of the response that in general will be longer than one byte. The throughput is calculated as requests / unit of time. The time is calculated from the start of the first sample to the end of the last sample. This includes any intervals between samples, as it is supposed to represent the load on the server. The formula is: throughput = number of requests / total time. (Apache Software Foundation, 2014d.) In Loadosophia.org, the throughput is measured in terms of TPS. In Apache JMeter, a *transaction* means an operation that a user wants Apache JMeter to perform, e.g., get a webpage, login to a webpage, insert data into a database, etc. (jmeter-plugins.org, 2014c).

Performance is measured in the benchmark of this thesis in terms of response time and throughput (TPS). The smaller the response time and the larger the throughput (TPS), the better the performance of the database. The most important performance indicators of the benchmark are the 90th percentiles of different response times and different throughputs (TPS).

6 RESULTS AND CONCLUSIONS

This chapter is organized as follows. First, the results of the benchmark are presented and analyzed. Then, Stardog and Neo4j are compared subjectively. Finally, the framework of requirements for integrating a smart city with a cloud infrastructure is evaluated.

6.1 Results of the benchmark

Next, the durations of the tests are briefly discussed. Then, the results of the benchmark are presented and analyzed one part at a time. The figures '1' and '2' in the tables 6–13 refer to the first and second run of each test, while the letters 'a', 'b', 'c', and 'd' refer to the actual tests. E.g., 'Neo4j, 1a' means the first run of the first test by Neo4j, 'Stardog, 2b' the second run of the second test by Stardog, etc.

6.1.1 About the durations of the tests

A stopwatch was used to measure the durations of the tests during some of the last rehearsals of the benchmark. They were identical with the final benchmark. The most accurate of these measurements are described in the table 5. The stopwatch was not used during the final benchmark, as it was assumed that Apache JMeter would measure the durations of the tests correctly. Unfortunately, this did not happen in all the tests, which is discussed further below.

As for the durations of all the tests, it is in order to stress that they varied somewhat every time the tests were run. In the final benchmark, the client and servers were not running anything else than Apache JMeter and Stardog or Neo4j, so the variation might be a consequence of network traffic that changed throughout the day. Another reason might be some unknown software that the author was not aware of that ran on the background consuming resources.

TABLE 5 Durations of the tests measured by the stopwatch

Part of the benchmark	Stardog	Neo4j
1. Create the graph	15 min 14 s	3 min 46 s
2. Write queries	35 min 12 s	7 min 41 s
3. Read queries	8 s	9 s
4. Read and write queries	3 min 26 s	1 min 37 s

6.1.2 Create the graph

The first part of the benchmark is described in the tables 6 and 7. During a single test, the initial graph was created by one virtual user. According to Loadosophia.org, the tests ran in Stardog's case on average 15 minutes (min) 25 seconds (s), the first test being 43 s faster than the second one, and in Neo4j's case on average 3 min 37 s, the second test being 4 s faster than the first one. The stopwatch gave roughly the same figures as Loadosophia.org according to which all the tests were valid for comparison.

During a single test, altogether 4250 transactions were processed. The HTTP code '2xx' indicates that there were no errors during the tests. The other figures, the minimum response time, average response time, maximum response time, 90th percentile of the response time, and average throughput indicate clearly that Neo4j performed better than Stardog in the first part of the benchmark.

TABLE 6 Properties and results of the first part of the benchmark

Properties	**Stardog, 1a**	**Stardog, 2a**	**Average**
Started at	20.11.2014 7.44.57	20.11.2014 8.57.55	-
Test duration	0:15:03	0:15:46	0:15:25
Transactions count	4250	4250	4250
HTTP codes presence	2xx	2xx	-
Minimum response time, ms	53	46	49.5
Average response time, ms	211	218	214.5
Maximum response time, ms	525	619	572
90th percentile of the response time, ms	290	300	295
Average throughput (TPS)	4.70653	4.4926	4.599565
Average virtual users	1	1	1
Maximum virtual users	1	1	1
Properties	**Neo4j, 1a**	**Neo4j, 2a**	**Average**
Started at	20.11.2014 10.22.35	20.11.2014 11.10.58	-
Test duration	0:03:39	0:03:35	0:03:37
Transactions count	4250	4250	4250
HTTP codes presence	2xx	2xx	-
Minimum response time, ms	33	29	31
Average response time, ms	50	49	49.5
Maximum response time, ms	374	443	408.5
90th percentile of the response time, ms	88	89	88.5
Average throughput (TPS)	19.4064	19.7674	19.5869
Average virtual users	1	1	1
Maximum virtual users	1	1	1

This can also be seen from the table 7 in which are listed the transaction groups and their average response times in milliseconds (ms). The sizes of the messages that created the initial graph were roughly the same, so it is not surprising that their response times were also roughly of the same size. What is significant, however, is that Neo4j's average response times were roughly four times better than Stardog's.

TABLE 7 Transactions of the first part of the benchmark

Transaction group	**Count**	**Fraction**	**Stardog, 1a**	**Stardog, 2a**	**Average**
Create a camera	250	5.9 %	224.012	234.86	229.436
Create a person	1000	23.5 %	219.295	227.686	223.4905
Create a phone	1000	23.5 %	226.081	236.168	231.1245
Create a place	1000	23.5 %	189.448	204.158	196.803
Create a sensor	1000	23.5 %	211.415	219.296	215.3555
Transaction group	**Count**	**Fraction**	**Neo4j, 1a**	**Neo4j, 2a**	**Average**
Create a camera	250	5.9 %	51.748	51.668	51.708
Create a person	1000	23.5 %	49.068	48.614	48.841
Create a phone	1000	23.5 %	52.213	51.111	51.662
Create a place	1000	23.5 %	51.809	49.961	50.885
Create a sensor	1000	23.5 %	51.297	52.074	51.6855

The differences in the 90th percentiles of the response times are also remarkable. A percentile indicates a value below which a given percentage of observations in a group of observations fall (Wikipedia, 2014c). The 90th percentile of a response time denotes that 90 % of transactions were processed below some value that is in this case ms. Hence, if the averages of the 90th percentiles of the response times are to be trusted, Neo4j performed roughly three times better than Stardog. Furthermore, according to the averages of the average throughputs, Neo4j was roughly four times faster than Stardog.

6.1.3 Write queries

The second part of the benchmark is described in the tables 8 and 9. During a single test, the initial graph was populated by more data by hundreds of parallel virtual users, the maximum being 1250. According to Loadosophia.org, Stardog ran the test on average 33 min 39 s, the first test being 1 min 47 s faster than the second one, while Neo4j completed the test on average in 7 min 20 s, the second test being 7 seconds faster than the first one. As for Stardog's results, Loadosophia.org complained about the duration difference, while Neo4j's results were in order. The stopwatch gave roughly the same durations.

During a single test, altogether 9500 transactions were processed. There were no errors during these tests either. What is notable is that the minimum response times are straight away many times larger than in the first part of the benchmark, which is an indication of the fact that these tests created heavier load than the first ones. Stardog's average minimum response time is slightly better than Neo4j's, although the difference is so small that it could easily be explained by network traffic especially, because Neo4j's tests were ran later on during the morning when there might have been more traffic on the network. The average and maximum response times, however, are dreadfully larger than in the first part of the benchmark, which could be explained at least by the fact that there were many parallel users inserting data into the databases, so more transactions were put on the line. In any case, if the averages of these figures are to be trusted, Stardog processed a transaction on average in approximately 92.7 s, while Neo4j in approximately 19.5 s. The maximum amount of time it took for Stardog to process a transaction was approximately 261.5 s, while Neo4j processed a transaction at worst in approximately 60.8 s.

TABLE 8 Properties and results of the second part of the benchmark

Properties	**Stardog, 1b**	**Stardog, 2b**	**Average**
Started at	20.11.2014 8.03.52	20.11.2014 9.24.40	-
Test duration	0:32:46	0:34:33	0:33:39
Transactions count	9500	9500	9500
HTTP codes presence	2xx	2xx	-
Minimum response time, ms	178	356	267
Average response time, ms	85006	100358	92682
Maximum response time, ms	252031	270885	261458
90th percentile of the response time, ms	245793	264948	255370.5
Average throughput (TPS)	4.83215	4.58273	4.70744
Average virtual users	308.006	311.532	309.769
Maximum virtual users	1250	1250	1250
Properties	**Neo4j, 1b**	**Neo4j, 2b**	**Average**
Started at	20.11.2014 10.37.54	20.11.2014 11.21.31	-
Test duration	0:07:24	0:07:17	0:07:20
Transactions count	9500	9500	9500
HTTP codes presence	2xx	2xx	-
Minimum response time, ms	319	329	324
Average response time, ms	19638	19457	19547.5
Maximum response time, ms	60974	60533	60753.5
90th percentile of the response time, ms	57172	57882	57527
Average throughput (TPS)	21.3964	21.7391	21.56775
Average virtual users	312.8	311.929	312.3645
Maximum virtual users	1250	1250	1250

The average response times of the transaction groups are also interesting. During a single test, both sensor and camera observations were created in parallel. In theory, the two thread groups that created these observations should have started exactly at the same time, but during all the rehearsals the thread group that created the sensor observations always seemed to start first, as it was located in the test plan before the other thread group that created the camera observations. This second thread group always seemed to start after the test had run perhaps 10–20 s. Then, the thread groups ran in parallel until all the sensor observations were created, after which only the camera observations were being created. What can be seen from the table 9 is that it took substantially more time to create the sensor observations. The reason for this is probably that the performance of the first thread group suffered when the second one started running. At this time, the load was the heaviest, as both observations were being created at the same time.

TABLE 9 Transactions of the second part of the benchmark

Transaction group	**Count**	**Fraction**	**Stardog, 1b**	**Stardog, 2b**	**Average**
Create a camera observation	7500	78.9 %	65909.4855	69207.0129	67558.2492
Create a sensor observation	2000	21.1 %	180853.176	197005.767	188929.472
Transaction group	**Count**	**Fraction**	**Neo4j, 1b**	**Neo4j, 2b**	**Average**
Create a camera observation	7500	78.9 %	14784.1891	14620.1492	14702.1692
Create a sensor observation	2000	21.1 %	43207.145	42716.2775	42961.7113

The differences in the averages of the 90th percentiles of the response times are again significant. According to them, Neo4j performed almost 4.5 times better than Stardog. Despite the load, the average throughputs remained quite the same, rising only slightly. If their averages are to be trusted, Neo4j was over 4.5 times faster than Stardog.

6.1.4 Read queries

The third part of the benchmark is described in the tables 10 and 11. During a single test, data were queried from the databases on average by hundreds of parallel virtual users, at most 1861.5. Stardog got through the tests in 2 s, while Neo4j in 1 s. During a single test, altogether 2000 transactions were processed. There were no errors. As for Stardog's results, Loadosophia.org complained about the VU (virtual user) difference, while Neo4j's tests were again in order.

In regard to the durations of the tests, something probably went wrong, as the results are contradictory. Stardog could not have processed all the transactions in 2 s if there were on average only 104.7 and at most 186 parallel users reading the database. As for Neo4j, the figures are more realistic, however, at least the durations of the tests are distorted. Neo4j could not have processed all the transactions in 1 s if the average response time was on average 3.02 s and the maximum response time on average 5.12 s. Hence, the tests lasted very probably more than 2 s in reality. As it can be seen from the table 5, according to the stopwatch, Stardog ran its test in 8 s and Neo4j in 9 s. The difference of 1 s could easily be explained by the inaccuracy of the measuring method. However, assuming that the tests had run 8 s, the throughput would have been approximately 2000 / 8 = 250 TPS. If they had run 9 s, the throughput would have been approximately 2000 / 9 ≈ 222.2 TPS.

TABLE 10 Properties and results of the third part of the benchmark

Properties	**Stardog, 1c**	**Stardog, 2c**	**Average**
Started at	20.11.2014 8.43.46	20.11.2014 10.06.17	-
Test duration	0:00:02	0:00:02	0:00:02
Transactions count	2000	2000	2000
HTTP codes presence	2xx	2xx	-
Minimum response time, ms	3	3	3
Average response time, ms	60	66	63
Maximum response time, ms	190	202	196
90th percentile of the response time, ms	125	148	136.5
Average throughput (TPS)	1000	1000	1000
Average virtual users	97.3333	112	104.66665
Maximum virtual users	174	198	186
Properties	**Neo4j, 1c**	**Neo4j, 2c**	**Average**
Started at	20.11.2014 10.50.26	20.11.2014 11.35.07	-
Test duration	0:00:01	0:00:01	0:00:01
Transactions count	2000	2000	2000
HTTP codes presence	2xx	2xx	-
Minimum response time, ms	303	173	238
Average response time, ms	3250	2785	3017.5
Maximum response time, ms	5063	5173	5118
90th percentile of the response time, ms	4931	4634	4782.5
Average throughput (TPS)	2000	2000	2000
Average virtual users	1624	1570.5	1597.25
Maximum virtual users	1866	1857	1861.5

The average response times of the transaction groups are interesting if they are to be trusted at all. Making calculations on data and returning the results might have been by and large as fast as querying other information, although it has to be borne in mind that the results of the calculation in question did not change at all. The thread group that queried the people's personal information was located in the test plan before the other thread group that counted the average temperature of all the sensor observations. Hence, the first thread group probably started first and ran a moment alone, after which the second thread group started running, during which time the load was the heaviest. The first thread group probably finished its work first. It is hard to say if its performance suffered when the second thread group started running.

TABLE 11 Transactions of the third part of the benchmark

Transaction group	**Count**	**Fraction**	**Stardog, 1c**	**Stardog, 2c**	**Average**
Count the average temperature	1000	50 %	60.617	53.065	56.841
Show my personal information 1	1000	50 %	69.243	84.684	76.9635
Transaction group	**Count**	**Fraction**	**Neo4j, 1c**	**Neo4j, 2c**	**Average**
Count the average temperature	1000	50 %	3453.28	3022.264	3237.772
Show my personal information 1	1000	50 %	3013.303	2565.132	2789.2175

All in all, it is dubious to compare Stardog's and Neo4j's results to each other in this case. According to the averages of the 90th percentiles of the response times, Stardog would have performed roughly 35 times better than Neo4j, while according to the averages of the average throughputs, Neo4j would have performed twice as good as Stardog. The latter figures are probably more realistic than the former, but either way, based on the available results, it is impossible to say which database performed better in these tests. The author is inclined to believe that Stardog and Neo4j ran their tests in 8–9 s, so their throughputs would have been the aforementioned ones.

6.1.5 Read and write queries

The fourth part of the benchmark is described in the tables 12 and 13. During a single test, data were inserted into and queried from the databases at the same time on average by hundreds, or as Loadosophia.org claims, thousands of parallel virtual users, at most 1799. According to Loadosophia.org, all the tests ran 2 s. In these tests, Loadosophia.org complained about the VU difference both in Stardog's and Neo4j's case.

Loadosophia.org's results are again somewhat distorted. At least the durations of the tests are questionable, which can be seen by comparing them to the figures of the table 5. Hence, also the average throughputs are dubious. Excluding the durations of the tests and average throughputs, the author is, however, inclined to believe Loadosophia.org's results.

According to the minimum response times, Stardog seems to have processed some individual transactions faster than Neo4j, but this could easily be a consequence of e.g., changing network traffic. However, the average and maximum response times speak in the favor of Neo4j. Both are roughly twice as good as Stardog's. Stardog processed a transaction on average in approximately 75.2 s, while Neo4j in approximately 36.2 s. The maximum amount of time it took for Stardog to process a transaction was approximately 204.3 s, while Neo4j processed a transaction at worst in approximately 88.5 s.

TABLE 12 Properties and results of the fourth part of the benchmark

Properties	**Stardog, 1d**	**Stardog, 2d**	**Average**
Started at	20.11.2014 8.46.31	20.11.2014 10.09.21	-
Test duration	0:00:02	0:00:02	0:00:02
Transactions count	2000	2000	2000
HTTP codes presence	2xx	2xx	-
Minimum response time, ms	3	3	3
Average response time, ms	70819	79675	75247
Maximum response time, ms	194614	213886	204250
90th percentile of the response time, ms	155861	172625	164243
Average throughput (TPS)	1000	1000	1000
Average virtual users	1024	1190.33	1107.165
Maximum virtual users	1641	1765	1703
Properties	**Neo4j, 1d**	**Neo4j, 2d**	**Average**
Started at	20.11.2014 10.53.53	20.11.2014 11.41.01	-
Test duration	0:00:02	0:00:02	0:00:02
Transactions count	2000	2000	2000
HTTP codes presence	2xx	2xx	-
Minimum response time, ms	5	6	5.5
Average response time, ms	36542	35784	36163
Maximum response time, ms	88412	88610	88511
90th percentile of the response time, ms	73667	70962	72314.5
Average throughput (TPS)	1000	1000	1000
Average virtual users	1187.33	1249	1218.165
Maximum virtual users	1813	1785	1799

The fact that Neo4j's average response times are roughly twice as good as Stardog's can be seen from the table 13 as well. In these tests, the first thread group queried the people's personal information, while the second one altered them. The sizes of the messages that the first thread group sent were very different than those of the second thread group. The first thread group ran probably a moment alone, after which the second one started running, during which time the load was the heaviest. The first thread group probably came out first. Based on the figures, inserting data into the databases seems to have taken in Stardog's case roughly twice as much time as querying it. In Neo4j's case, the equivalent figure is approximately 1.54.

TABLE 13 Transactions of the fourth part of the benchmark

Transaction group	**Count**	**Fraction**	**Stardog, 1d**	**Stardog, 2d**	**Average**
Show my personal information 2	1000	50 %	47706.897	56724.83	52215.8635
Update my personal information	1000	50 %	97180.578	106867.513	102024.046
Transaction group	**Count**	**Fraction**	**Neo4j, 1d**	**Neo4j, 2d**	**Average**
Show my personal information 2	1000	50 %	29261.35	27711.71	28486.53
Update my personal information	1000	50 %	43900.032	43996.26	43948.146

The differences in the averages of the 90th percentiles of the response times are once again notable. Neo4j processed transactions over twice as fast as Stardog. As for the averages of the average throughputs, Stardog ties with Neo4j if one looks at Loadosophia.org's results, but as already mentioned, the durations of the tests, and thus the average throughputs are probably distorted, which is easy to see looking at e.g., the average and maximum response times. However, if the times measured by the stopwatch are taken into account, Stardog might have processed 2000 / 206 s ≈ 9.71 TPS and Neo4j 2000 / 97 s ≈ 20.62 TPS. Hence, Neo4j might have been roughly twice as fast as Stardog.

6.1.6 Summary of the results

In summary, based on all the aforementioned figures and their interpretation, Neo4j seemed to perform better than Stardog in the benchmark. In the first part of the benchmark in which the initial graph was created by one virtual user, Neo4j performed 3–4 times better than Stardog measured by the averages of the 90th percentiles of the response times and the averages of the average throughputs. In the second part of the benchmark, the graph was populated by more data by hundreds of parallel virtual users. Here, Neo4j performed roughly 4.5 times better than Stardog if the averages of the 90th percentiles of the response times and the averages of the average throughputs are to be trusted. In the third part of the benchmark, the databases were queried by hundreds of parallel virtual users. Unfortunately, the results that Loadosophia.org gave are somewhat distorted, but taking into account the durations of the tests measured by the stopwatch, Neo4j and Stardog performed roughly speaking equally well in the tests. Finally, in the fourth part of the benchmark, data were inserted into and queried from the databases at the same time. The results that Loadosophia.org provided are again somewhat distorted, but if the averages of the 90th percentiles of the response times are to be trusted, Neo4j performed twice as good as Stardog. Same conclusion can be reached by taking into account the durations of the tests measured by the stopwatch.

6.2 Subjective comparison of Stardog and Neo4j

Stardog and Neo4j are both great databases, which is in a nutshell the reason why they were chosen to be compared to each other in this thesis. The other reasons have been already mentioned in the previous chapter, but be it underlined that both databases are easy to install and use, they have a good array of out-of-the-box features, their data models and query languages are logical and easy to learn, and that of the two products, Neo4j seems to be more mature. Next, some of the features of Stardog and Neo4j are discussed in more detail and compared to each other as far as this is possible based on their brief experiment. The author emphasizes that he was not familiar with Stardog and Neo4j before making this thesis.

To begin with, both Stardog's and Neo4j's GUIs are very intuitive. Neo4j Web Interface is perhaps slightly more simplistic than Stardog Console. E.g., there are no security settings in Neo4j Web Interface, while in Stardog Console users, roles, and permissions can be managed. Then again, in Neo4j Web Interface it is possible to visualize the graph, while in Stardog Console there is no such option. This feature came in handy when checking out relationships of data. All things considered, both GUIs have their advantages and disadvantages, so it is impossible to say which one is better. In practice, they are tools among others. Some changes have to be made to Stardog's and Neo4j's configuration files, which requires shutting down and restarting the servers.

In regard to the data models and query languages, the author found Neo4j's property graph and Cypher slightly easier to learn and use than RDF and SPARQL despite the fact that he was familiar with RDF prior to writing this thesis. Of these concepts, RDF is the oldest and most mature. SPARQL has been around many years as well, but then again, so has Neo4j with its technologies. In the author's knowledge, all these concepts have been evolving to this day and still continue to do so. Comparing them otherwise is difficult. It cannot be said that RDF and SPARQL are superior to property graph and Cypher or vice versa. Both data models and query languages have their pros and cons. What is the 'best' depends on the situation.

Both data models are in any case suitable for smart cities. The author is of the opinion that Stardog or Neo4j could be a linchpin of the Kangas area's data warehouse. Both products could be used for saving structured data, e.g., sensor readings, and unstructured data, e.g., people's personal information. However, as far as the author knows, neither Stardog nor Neo4j are suitable for saving multimedia. They do not seem to comprise such data types that e.g., image, audio, and video files could be saved to them. Hence, if Stardog or Neo4j were used in the Kangas project, multimedia would have to be saved to some other system, saving only its metadata to Stardog or Neo4j. Multimedia could be saved e.g., in Neo4j's case to some high write performance key-value store, as Hunger (2014) suggests. E.g., a video camera could send video files to such a system. Simultaneously, it could send complete, fine-grained metadata of these files to Neo4j.

Otherwise it can be said that Neo4j seems to be in many respects more mature product than Stardog. Neo4j's documentation seems to be wider and of higher quality than Stardog's. There seems to be a larger community, and thus more support behind Neo4j than Stardog. Yet another sign of product maturity is that Neo4j has offered DDBMS functionality some time, while Stardog Cluster is in beta version at the moment. In addition, Neo4j seems to be very extendable. E.g., Neo4j SPARQL Plugin enables inserting RDF data into a database and querying it (Neo Technology, 2013). Unfortunately, it seems to offer no more functionalities at the moment. Moreover, it is probably not the best option performance-wise (Hunger, 2014). Then again, also Stardog appears to be very extendable. As Neo4j, it seems to be a very versatile product supporting many approaches and technologies.

6.3 Evaluation of the framework

The main objective of this study has been to build the framework of requirements for integrating a smart city with a cloud infrastructure and answer with the help of it to the main research question: how a smart city data warehouse can be efficiently integrated with a cloud infrastructure? An efficient integration of a smart city data warehouse with a cloud infrastructure means that requirements for smart city data management match, more or less, requirements for, or characteristics of, cloud data management. With the help of the framework, it can also be decided what are the most important requirements for some individual case.

All in all, the framework is a guiding principle and functions as it, in the author's opinion, quite well. The framework has several strengths: It gathers together many cloud computing, cloud data management, and smart city requirements that are relevant in themselves. Paying attention to them is important when building especially smart city data management systems. It is naturally impossible to invest in every requirement, but with the help of the framework, it can be decided what are the most important requirements for some individual case. E.g., in this study the most important requirement ended up to be performance.

The framework has also some weaknesses: The connection between some requirements is so loose that it can be argued if it is worthwhile to present their connection. The requirements are quite general and abstract, so the application of the framework requires in practice technical expertise and measurements. E.g., to say something about performance and scalabity of some particular system, there is a need for experts on these issues. Furthermore, performance and scalability probably have to be operationalized and the system tested for them. The biggest weakness of the framework might be that it does not offer any baseline for technical comparison. One might measure e.g., performance in terms of response time and throughput, but the framework cannot answer if the measured performance is good or bad, i.e., it does not provide specifications. Unfortunately, the author could not find any usable and generalizable specifications of this kind in the literature. If the author found some specifications, they were so application-specific that they could not be generalized in any way. Assuming that some kind of generalizable specifications could be found somewhere, the framework would greatly benefit from them, as they would make its requirements more concrete than verbal examples and illustrations.

7 SUMMARY

The Kangas project is the main urban development project of the City of Jyväskylä for the next several decades. In brief, the Kangas area will form a smart city in the future, being a home to 5000 inhabitants and offering 2000 new jobs. This project requires implementing, but first, planning for many things. One of them is the data warehouse of the area. It was decided at the University of Jyväskylä that the data warehouse will be built on the cloud with the help of the university's hardware, network, and other resources, e.g., Eucalyptus cloud software.

Before implementing the data warehouse of the Kangas area, there is a need to know how a smart city data warehouse can be efficiently integrated with a cloud infrastructure in general. This requires knowledge of the requirements for smart cities, especially their data management, and the requirements for cloud computing systems, especially their data management. In the research literature exist many such requirements, but before this study, there appeared to be no framework that would have integrated them with each other. It was thus realized that this kind of framework could be useful e.g., to researchers and decision-makers. Hence, the main objective of this study has been to build such a framework and answer with the help of it to the main research question: how a smart city data warehouse can be efficiently integrated with a cloud infrastructure?

This thesis represents *design science* that is fundamentally a problem-solving paradigm that creates and evaluates IT artifacts intended to solve identified organizational problems. Design science consists of two basic activities, building and evaluating. *Building* is the process of constructing an artifact for a specific purpose. *Evaluation* is the process of determining how well the artifact performs. The aforementioned framework, i.e., the framework of requirements for integrating a smart city with a cloud infrastructure, is the *artifact* of this study.

The framework functions as a guiding principle that helps e.g., researchers and decision-makers to map, among other things, what a smart city data warehouse requires of cloud data management systems in general. An efficient integration of a smart city data warehouse with a cloud infrastructure means that requirements for smart city data management match, more or less, requirements for, or characteristics of, cloud data management. With the help of the framework, it can also be decided what are the most important requirements for some individual case.

As it can be seen from the framework, the connection between smart cities and cloud computing is loose in some cases, but in a number of requirements the connection is firmer. With the help of the framework, it can be interpreted that a smart city requires of a cloud infrastructure at least availability, autonomicity, scalability, performance, interoperability, and fault tolerance, as well as privacy and security. Of the 'softer' non-technical requirements user involvement and sustainability cannot be left aside.

The framework functions as a guiding principle quite well. The evaluation of the framework revealed that it has several strengths: It gathers together many cloud computing, cloud data management, and smart city data management requirements that are relevant in themselves. With the help of the framework, it can also be decided what are the most important requirements for some individual case. The framework has also some weaknesses: Its requirements are quite general and abstract, so the application of the framework requires in practice technical expertise and measurements. The biggest weakness of the framework might be that it does not offer any baseline for technical comparison, i.e., specifications of its requirements. Assuming that some kind of generalizable specifications could be found somewhere, the framework would greatly benefit from them, as they would make its requirements more concrete than mere verbal examples and illustrations. Hence, improving the framework e.g., in this way provides a subject for further study.

The use of the framework was demonstrated by choosing the most important requirements for the data warehouse of the Kangas project: performance and scalability. Of these requirements, performance was operationalized, after which Stardog and Neo4j were tested for it. They were installed on a Eucalyptus cloud and a benchmark was built that inserted data and queried it from the databases with the help of Apache JMeter, a performance testing tool. The benchmark compared the performance of Stardog's public SPARQL endpoint to Neo4j's Transactional Cypher HTTP endpoint. The most important performance indicators of the benchmark were the 90th percentiles of different response times and different throughputs (TPS).

Neo4j performed better than Stardog in the benchmark. In the first part of the benchmark in which the initial graph was created by one virtual user, Neo4j performed 3–4 times better than Stardog measured by the averages of the 90th percentiles of the response times and the averages of the average throughputs. In the second part of the benchmark, the graph was populated by more data by hundreds of parallel virtual users. Here, Neo4j performed roughly 4.5 times better than Stardog if the averages of the 90th percentiles of the response times and the averages of the average throughputs are to be trusted. In the third part of the benchmark, the databases were queried by hundreds of parallel virtual users. Unfortunately, the results that Loadosophia.org gave are somewhat distorted, but taking into account the durations of the tests measured by the stopwatch, Neo4j and Stardog performed roughly speaking equally well in the tests. Finally, in the fourth part of the benchmark, data were inserted into and queried from the databases at the same time. The results that Loadosophia.org provided are again somewhat distorted, but if the averages of the 90th percentiles of the response times are to be trusted, Neo4j performed twice as good as Stardog. Same conclusion can be reached by taking into account the durations of the tests measured by the stopwatch.

The benchmark has naturally its limitations. First of all, it only compared the performance of Stardog's public SPARQL endpoint to Neo4j's Transactional Cypher HTTP endpoint. It has to be stressed, however, that there are also other ways to access Stardog and Neo4j, and they are possibly faster than the aforementioned endpoints. Furthermore, the benchmark only compared Stardog Community to Neo4j Community, i.e., the free-of-charge editions of Stardog and Neo4j. Both databases offer also their enterprise editions that are meant for heavier use. Benchmarking or studying their possibilities otherwise provides another subject for further study. If one is interested to see e.g., how much Stardog and Neo4j can take with their DDBMS functionalities enabled, the benchmark of this thesis can easily be extended. In addition, the settings and messages of the benchmark can always be questioned especially, because some of its results were distorted for one reason or the other: The client computer should have had enough memory, but it still could have run out of memory. Apache JMeter and/or Loadosophia.org Uploader could have malfunctioned. Some element(s) of the test plan could have caused the problem. The author could have made a mistake, etc.

In regard to the elements of the test plan, everything possible was made to ensure that they were as similar and error-free as possible. The tests were run many times, after which it was checked out that the created graphs were similar and error-free. All the test plan elements were saved to only one file that comprised the whole test plan, which made comparing the elements to each other as easy as possible. It also made running the tests very straightforward. There was only a need to activate certain elements and deactivate some others, as well as change some minor settings.

Stardog and Neo4j were compared subjectively as well. In summary, both databases are easy to install and use, they have a good array of out-of-the-box features, their data models and query languages are logical and easy to learn, and of the two products, Neo4j seems to be more mature. Both Stardog's and Neo4j's data models are suitable for smart cities. As such, Stardog or Neo4j could be a linchpin of the Kangas area's data warehouse. Both products could be used for saving structured and unstructured data, however, they probably are not suitable for saving multimedia. Hence, if Stardog or Neo4j were used in the Kangas project, multimedia would have to be saved to some other system, saving only its metadata to Stardog or Neo4j. Whether this is feasible and meaningful is also worthwhile to study.

LITERATURE SOURCES

Abadi, D. J. (2009). Data Management in the Cloud: Limitations and Opportunities. *IEEE Data Engineering Bulletin, 32*(1), 3–12. Retrieved October 4, 2014, from http://cs-www.cs.yale.edu/homes/dna/papers/abadi-cloud-ieee09.pdf

Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A. & Rasin, A. (2009). HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proceedings of the VLDB Endowment, 2*(1), 922–933. doi:10.14778/1687627.1687731

Amazon Web Services (2014a). About AWS. Retrieved November 10, 2014, from http://aws.amazon.com/about-aws

Amazon Web Services (2014b, September 18). Getting Started with AWS - Getting Started with AWS. Retrieved November 10, 2014, from http://docs.aws.amazon.com/gettingstarted/latest/awsgsg-intro/gsg-aws-intro.html

Amazon Web Services (2014c, November 7). What Is Amazon EC2? - Amazon Elastic Compute Cloud. Retrieved November 10, 2014, from http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

Amazon Web Services (2014d). AWS | Amazon EC2 | Instance Types. Retrieved November 10, 2014, from http://aws.amazon.com/ec2/instance-types

Amazon Web Services (2014e, November 7). Amazon EC2 Instance Store - Amazon Elastic Compute Cloud. Retrieved November 10, 2014, from http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/InstanceStorage.html

Amazon Web Services (2014f, November 7). Amazon Elastic Block Store (Amazon EBS) - Amazon Elastic Compute Cloud. Retrieved November 10, 2014, from http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AmazonEBS.html

Amazon Web Services (2014g, November 8). What Is Amazon S3? - Amazon Simple Storage Service. Retrieved November 10, 2014, from http://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html

Amazon Web Services (2014h, November 8). Introduction to Amazon S3 - Amazon Simple Storage Service. Retrieved November 10, 2014, from http://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html

Amazon Web Services (2014i, November 7). Regions and Availability Zones - Amazon Elastic Compute Cloud. Retrieved November 10, 2014, from http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html

Apache Software Foundation (2014a, November 2). Apache JMeter - Apache JMeter™. Retrieved November 12, 2014, from http://jmeter.apache.org

Apache Software Foundation (2014b, November 2). Apache JMeter - User's Manual: Building a Test Plan. Retrieved November 12, 2014, from http://jmeter.apache.org/usermanual/build-test-plan.html

Apache Software Foundation (2014c, November 2). Apache JMeter - User's Manual: Elements of a Test Plan. Retrieved November 12, 2014, from http://jmeter.apache.org/usermanual/test_plan.html

Apache Software Foundation (2014d, November 2). Apache JMeter - User's Manual: Glossary. Retrieved November 13, 2014, from http://jmeter.apache.org/usermanual/glossary.html

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. & Zaharia, M. (2010). A View of Cloud Computing. *Communications of the ACM, 53*(4), 50–58. doi:10.1145/1721654.1721672

Arora, I. & Gupta, A. (2012). Cloud Databases: A Paradigm Shift in Databases. *IJCSI International Journal of Computer Science Issues, 9*(4), 77–83. Retrieved May 1, 2014, from http://www.ijcsi.org/papers/IJCSI-9-4-3-77-83.pdf

Ashton, K. (2009, June 22). That 'Internet of Things' Thing - RFID Journal. Retrieved November 3, 2014, from http://www.rfidjournal.com/articles/view?4986

Atzori, L., Iera, A. & Morabito, G. (2010). The Internet of Things: A survey. *Computer Networks, 54*(15), 2787–2805. doi:10.1016/j.comnet.2010.05.010

Bizer, C. & Schultz, A. (2008). Benchmarking the Performance of Storage Systems that expose SPARQL Endpoints. In *Proceedings of the 4th International Workshop on Scalable Semantic Web knowledge Base Systems (SSWS 2008)*, Karlsruhe, Germany, October, 2008. Retrieved October 1, 2014, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.161.7773

Bizer, C. & Schultz, A. (2012, March 23). Berlin SPARQL Benchmark. Retrieved November 13, 2014, from http://wifo5-03.informatik.uni-mannheim.de/bizer/berlinsparqlbenchmark/spec/index.html

Boles, J. (2008, November 7). What is cloud-based storage?, part 1 - Infostor. Retrieved April 30, 2014, from http://www.infostor.com/index/articles/display/4979101322/articles/infostor/top-news/what-is_cloud-based.html

Boyle, D. E., Yates, D. C. & Yeatman, E. M. (2013). Urban Sensor Data Streams: London 2013. *IEEE Internet Computing, 17*(6), 12–20. doi:10.1109/MIC.2013.85

Brown, E. (2011, October 25). Final Version of NIST Cloud Computing Definition Published. Retrieved April 30, 2014, from http://www.nist.gov/itl/csd/cloud-102511.cfm

Calabrese, F. & Ratti, C. (2006). Real Time Rome. *Networks and Communication studies, 20*(3–4), 247–258. Retrieved October 3, 2014, from http://senseable.mit.edu/papers/pdf/2006_Calabrese_Ratti_Rome_IGU-GISC.pdf

Calabrese, F., Colonna, M., Lovisolo, P., Parata, D. & Ratti, C. (2011). Real-Time Urban Monitoring Using Cell Phones: A Case Study in Rome. *IEEE*

Transactions on Intelligent Transportation Systems, 12(1), 141–151. doi:10.1109/TITS.2010.2074196

Caragliu, A., Del Bo, C. & Nijkamp, P. (2009). *Smart cities in Europe* (Serie Research Memoranda 0048). VU University Amsterdam, Faculty of Economics, Business Administration and Econometrics. Retrieved May 2, 2014, from http://ideas.repec.org/p/dgr/vuarem/2009-48.html

Caragliu, A., Del Bo, C. & Nijkamp, P. (2011). Smart Cities in Europe. *Journal of Urban Technology, 18*(2), 65–82. doi:10.1080/10630732.2011.601117

Chourabi, H., Nam, T., Walker, S., Gil-Garcia, J. R., Mellouli, S., Nahon, K., Pardo, T. A. & Scholl, H. J. (2012). Understanding Smart Cities: An Integrative Framework. In *2012 45th Hawaii International Conference on System Science (HICSS-45)* (2289–2297), Maui, HI, January 4–7, 2012. doi:10.1109/HICSS.2012.615

CIA (2014, September 8). The World Factbook. Retrieved November 13, 2014, from https://www.cia.gov/library/publications/the-world-factbook/geos/uk.html

Ciglan, M., Averbuch, A. & Hluchy, L. (2012). Benchmarking traversal operations over graph databases. In *2012 IEEE 28th International Conference on Data Engineering Workshops (ICDEW)* (186–189), Arlington, VA, April 1–5, 2012. doi:10.1109/ICDEW.2012.47

Clark & Parcia (2014a, October 15). Stardog 2.2.2 Docs: Stardog Docs. Retrieved November 12, 2014, from http://docs.stardog.com

Clark & Parcia (2014b, October 15). Stardog: The Enterprise Graph Database. Retrieved November 12, 2014, from http://stardog.com

Clark & Parcia (2014c, October 15). Stardog 2.2.2 Docs: FAQ. Retrieved November 13, 2014, from http://docs.stardog.com/faq

Clark & Parcia (2014d, October 15). Stardog 2.2.2 Docs: Web Console. Retrieved November 13, 2014, from http://docs.stardog.com/console

Connolly, T. M. & Begg, C. E. (2005). *Database Systems: A practical Approach to Design, Implementation, and Management* (4th ed.). Harlow, UK: Addison-Wesley.

Cooper, B. F., Baldeschwieler, E., Fonseca, R., Kistler, J. J., Narayan, P. P. S., Neerdaels, C., Negrin, T., Ramakrishnan, R., Silberstein, A., Srivastava, U. & Stata, R. (2009). Building a Cloud for Yahoo! *IEEE Data Engineering Bulletin, 32*(1), 36–43. Retrieved November 7, 2014, from http://sites.computer.org/debull/A09mar/cooper1.pdf

Da Silva, W. M., Alvaro, A., Tomas, G. H. R. P., Afonso, R. A., Dias, K. L. & Garcia, V. C. (2013). Smart Cities Software Architectures: A Survey. In Shin, S. Y. & Maldonado, J. C. (eds.), *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)* (1722–1727), Coimbra, Portugal, March 18–22, 2013. doi:10.1145/2480362.2480688

Dewan, H. & Hansdah, R. C. (2011). A Survey of Cloud Storage Facilities. In *7th World Congress on Services (IEEE SERVICES 2011)* (224–231), Washington, DC, July 4–9, 2011. doi:10.1109/SERVICES.2011.43

Dominguez-Sal, D., Martinez-Bazan, N., Muntes-Mulero, V., Baleta, P. & Larriba-Pey, J. L. (2011). A Discussion on the Design of Graph Database Benchmarks. In Nambiar, R. & Poess, M. (eds.), *Performance Evaluation, Measurement and Characterization of Complex Systems: Second TPC Technology Conference, TPCTC 2010, Singapore, September 13–17, 2010. Revised Selected Papers* (25–40). Berlin, Germany: Springer. doi:10.1007/978-3-642-18206-8_3

Eucalyptus Systems (2014a). Story of Eucalyptus Cloud Software | AWS Cloud Integration | Eucalyptus. Retrieved November 10, 2014, from https://www.eucalyptus.com/about/story

Eucalyptus Systems (2014b). Eucalyptus Datasheet. Retrieved November 10, 2014, from https://www.eucalyptus.com/sites/all/files/ds-eucalyptus-iaas.en.pdf

Eucalyptus Systems (2014c). Distributed Cloud Computing Architecture and Components | Eucalyptus. Retrieved November 10, 2014, from https://www.eucalyptus.com/eucalyptus-cloud/iaas/architecture

Eucalyptus Systems (2014d). Eucalyptus Architecture: Cloud Level | Eucalyptus. Retrieved November 10, 2014, from https://www.eucalyptus.com/eucalyptus-cloud/iaas/architecture/cloud-level

Eucalyptus Systems (2014e). Eucalyptus Architecture: Cluster Level | Eucalyptus. Retrieved November 10, 2014, from https://www.eucalyptus.com/eucalyptus-cloud/iaas/architecture/cluster-level

Eucalyptus Systems (2014f). Eucalyptus Architecture: Node Level | Eucalyptus. Retrieved November 10, 2014, from https://www.eucalyptus.com/eucalyptus-cloud/iaas/architecture/node-level

Graphanalysis.org (2014, October 14). GraphAnalysis.org: High Performance Computing for solving large-scale graph problems. Retrieved November 13, 2014, from http://www.graphanalysis.org

Gregor, S. & Jones, D. (2007). The Anatomy of a Design Theory. *Journal of the Association for Information Systems, 8*(5), 312–335. http://aisel.aisnet.org/jais/vol8/iss5/1

Gregor, S. (2006). The Nature of Theory in Information Systems. *MIS Quarterly, 30*(3), 611–642. Retrieved September 26, 2014, from http://www.jstor.org/stable/25148742

Gubbi, J., Buyya, R., Marusic, S. & Palaniswami, M. (2013). Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems, 29*(7), 1645–1660. doi:10.1016/j.future.2013.01.010

Halili, E. H. (2008). *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Birmingham, UK: Packt Publishing. Retrieved November 13, 2014, from http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=333399

Hayes, B. (2008). Cloud Computing. *Communications of the ACM, 51*(7), 9–11. doi:10.1145/1364782.1364786

Hernández-Muñoz, J. M., Vercher, J. B., Muñoz, L., Galache, J. A., Presser, M., Hernández Gómez, L. A. & Pettersson, J. (2011). Smart Cities at the Forefront of the Future Internet. In Domingue, J., Galis, A., Gavras, A., Zahariadis, T., Lambert, D., Cleary, F., Daras, P., Krco, S., Müller, H., Li, M.–S., Schaffers, H., Lotz, V., Alvarez, F., Stiller, B., Karnouskos, S., Avessta, S. & Nilsson, M. (eds.), *The Future Internet: Future Internet Assembly 2011: Achievements and Technological Promises* (447–462). Berlin, Germany: Springer. doi:10.1007/978-3-642-20898-0_32

Hevner, A. R., March, S. T., Park, J. & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly, 28*(1), 75–105. Retrieved September 26, 2014, from http://www.jstor.org/stable/25148625

Hunger (2014, July 30 – August 1). Problems with Neo4j's SPARQL Plugin, etc. [Msgs 2, 4]. Messages posted to https://groups.google.com/forum/#!topic/neo4j/o9JL3YZKov8

Internet of Things Architecture (2013). *Internet of Things – Architecture IoT-A: Deliverable D1.5 – Final architectural reference model for the IoT v3.0.* Retrieved May 2, 2014, from http://www.iot-a.eu/public/public-documents/d1.5/at_download/file

ISTQB (2014). Glossary: Standard Glossary of Terms used in Software Testing Version 2.4. Retrieved November 12, 2014, from http://www.istqb.org/downloads/finish/20/145.html

Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. New York, NY: Wiley.

Järvinen, P. (2012). *On research methods* (4th ed.). Tampere, Finland: Opinpajan kirja.

jmeter-plugins.org (2014a). JMeter Plugins :: JMeter-Plugins.org. Retrieved November 12, 2014, from http://jmeter-plugins.org

jmeter-plugins.org (2014b). Documentation :: JMeter-Plugins.org. Retrieved November 13, 2014, from http://jmeter-plugins.org/wiki/LoadosophiaUploader

jmeter-plugins.org (2014c). Documentation :: JMeter-Plugins.org. Retrieved November 13, 2014, from http://jmeter-plugins.org/wiki/TransactionsPerSecond

Jouili, S. & Vansteenberghe, V. (2013). An empirical comparison of graph databases. In *2013 International Conference on Social Computing (SocialCom)* (708–715), Alexandria, VA, September 8–14, 2013. doi:10.1109/SocialCom.2013.106

Jyväskylän kaupunki (2011). Jyväskylän Kangas Kangas in english. Retrieved November 11, 2014, from http://www3.jkl.fi/blogit/kangasjyvaskyla/?page_id=489

Koskinen, J. (2012). Network Guide: Central Finland Student Housing Foundation. Retrieved November 13, 2014, from http://www.koas.fi/index.php/download_file/view/460/337

Krčo, S., Fernandes, J., Sanchez, L., Natti, M., Theodoridis, E., Vučković, D., Casanueva, J., Galache, J. A., Gutiérrez, V., Santana, J. R. & Sotres, P. (2013). SmartSantander – a smart city experimental platform. *Electrotechnical Review, 79*, 268–272. Retrieved November 13, 2014, from http://www.ltfe.org/wp-content/uploads/2012/11/SmartSantander-VITEL12-v4.pdf

Kulkarni, G., Waghmare, R., Palwe, R., Waykule, V., Bankar, H. & Koli, K. (2012). Cloud storage architecture. In *7th International Conference on Telecommunication Systems, Services, and Applications (TSSA 2012)* (76–81), Bali, Indonesia, October 30–31, 2012. doi:10.1109/TSSA.2012.6366026

LDBC (2014). Social Network Benchmark | LDBCouncil. Retrieved November 13, 2014, from http://ldbcouncil.org/developer/snb

Leavitt, N. (2010). Will NoSQL Databases Live Up to Their Promise? *Computer, 43*(2), 12–14. doi:10.1109/MC.2010.58

Lertlakkhanakul, J., Choi, J. W. & Kim, M. Y. (2008). Building data model and simulation platform for spatial interaction management in smart home. *Automation in Construction, 17*(8), 948–957. doi:10.1016/j.autcon.2008.03.004

Loadosophia.org (2014). Loadosophia.org. Retrieved October 3, 2014, from http://loadosophia.org

March, S. T. & Smith, G. F. (1995). Design and natural science research on information technology. *Decision Support Systems, 15*(4), 251–266. doi:10.1016/0167-9236(94)00041-2

Mell, P. & Grance, T. (2011). *The NIST Definition of Cloud Computing: Recommendations of the National Institute of Standards and Technology* (Special Publication 800-145). Gaithersburg, MD: NIST. Retrieved April 30, 2014, from http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

Menascé, D. A. (2002). QoS Issues in Web Services. *IEEE Internet Computing, 6*(6), 72–75. doi:10.1109/MIC.2002.1067740

Merriam-Webster (2014). Benchmark - Definition and More from the Free Merriam-Webster Dictionary. Retrieved November 13, 2014, from http://www.merriam-webster.com/dictionary/benchmark

Miorandi, D., Sicari, S., De Pellegrini, F. & Chlamtac, I. (2012). Internet of things: Vision, applications and research challenges. *Ad Hoc Networks, 10*(7), 1497–1516. doi:10.1016/j.adhoc.2012.02.016

Nam, T. & Pardo, T. A. (2011). Conceptualizing Smart City with Dimensions of Technology, People, and Institutions. In Bertot, J., Nahon, K., Chun, S. A., Luna-Reyes, L. & Atluri, V. (eds.), *Proceedings of the 12th Annual International Digital Government Research Conference: Digital Government Innovation in Challenging Times (dg.o '11)* (282–291), College Park, MD, June 12–15, 2011. doi:10.1145/2037556.2037602

Neo Technology (2013, June 20). Neo4j Sparql Plugin v0.2-SNAPSHOT. Retrieved December 14, 2014, from http://neo4j-contrib.github.io/sparql-plugin

Neo Technology (2014a). What is a Graph Database? - Neo4j Graph Database. Retrieved November 12, 2014, from http://neo4j.com/developer/graph-database

Neo Technology (2014b). Online Training: Getting Started with Neo4j - Neo4j Graph Database. Retrieved November 12, 2014, from http://neo4j.com/graphacademy/online-course

Neo Technology (2014c, November 10). 8.5. Identifiers - - The Neo4j Manual v2.1.5. Retrieved November 12, 2014, from http://neo4j.com/docs/stable/cypher-identifiers.html

Neo Technology (2014d, November 10). 7.1. What is Cypher? - - The Neo4j Manual v2.1.5. Retrieved November 12, 2014, from http://neo4j.com/docs/stable/cypher-introduction.html

Neo Technology (2014e). Subscriptions - Neo4j Graph Database. Retrieved November 12, 2014, from http://neo4j.com/subscriptions

Neo Technology (2014f, November 13). 19.1. Transactional Cypher HTTP endpoint - - The Neo4j Manual v2.1.5. Retrieved November 13, 2014, from http://neo4j.com/docs/milestone/rest-api-transactional.html

Neo Technology (2014g, November 13). Chapter 27. Web Interface - - The Neo4j Manual v2.1.5. Retrieved November 13, 2014, from http://neo4j.com/docs/stable/tools-webadmin.html

Neo Technology (2014h, December 5). 25.1. Securing access to the Neo4j Server - - The Neo4j Manual v2.1.6. Retrieved December 11, 2014, from http://neo4j.com/docs/stable/security-server.html

Ostrowski, Ł., Helfert, M. & Xie, S. (2012). A Conceptual Framework to Construct an Artefact for Meta-Abstract Design Knowledge in Design Science Research. In *2012 45th Hawaii International Conference on System Science (HICSS-45)* (4074–4081), Maui, HI, January 4–7, 2012. doi:10.1109/HICSS.2012.51

Peffers, K., Tuunanen, T., Rothenberger, M. A. & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems, 24*(3), 45–77. doi:10.2753/MIS0742-1222240302

Piro, G., Cianci, I., Grieco, L. A., Boggia, G. & Camarda, P. (2014). Information centric services in Smart Cities. *Journal of Systems and Software, 88*, 169–188. doi:10.1016/j.jss.2013.10.029

Pokorny, J. (2013). NoSQL databases: a step to database scalability in web environment. *International Journal of Web Information Systems, 9*(1), 69–82. doi:10.1108/17440081311316398

Ranjan, R., Buyya, R. & Parashar, M. (2012). Special section on autonomic cloud computing: technologies, services, and applications. *Concurrency and Computation: Practice and Experience, 24*(9), 935–937. doi:10.1002/cpe.1865

Rimal, B. P., Jukan, A., Katsaros, D. & Goeleven, Y. (2011). Architectural Requirements for Cloud Computing Systems: An Enterprise Cloud Approach. *Journal of Grid Computing, 9*(1), 3–26. doi:10.1007/s10723-010-9171-y

Rouvinen, J. (2013). *Peeking inside the cloud*. Master's thesis in information technology. University of Jyväskylä. Retrieved April 30, 2014, from http://urn.fi/URN:NBN:fi:jyu-201306282050

Sakr, S., Liu, A., Batista, D. M. & Alomari, M. (2011). A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Communications Surveys & Tutorials, 13*(3), 311–336. doi:10.1109/SURV.2011.032211.00087

Schaffers, H., Komninos, N., Pallot, M., Trousse, B., Nilsson, M. & Oliveira, A. (2011). Smart Cities and the Future Internet: Towards Cooperation Frameworks for Open Innovation. In Domingue, J., Galis, A., Gavras, A., Zahariadis, T., Lambert, D., Cleary, F., Daras, P., Krco, S., Müller, H., Li, M.–S., Schaffers, H., Lotz, V., Alvarez, F., Stiller, B., Karnouskos, S., Avessta, S. & Nilsson, M. (eds.), *The Future Internet: Future Internet Assembly 2011: Achievements and Technological Promises* (431–446). Berlin, Germany: Springer. doi:10.1007/978-3-642-20898-0_31

Simon, H. A. (1996). *The Sciences of the Artificial* (3rd ed.). Cambridge, MA, London, UK: MIT Press. Retrieved September 26, 2014, from http://courses.washington.edu/thesisd/documents/Kun_Herbert%20Simon_Sciences_of_the_Artificial.pdf

solid IT (2014). DB-Engines Ranking - popularity ranking of database management systems. Retrieved November 12, 2014, from http://db-engines.com/en/ranking

Stanford Center for Biomedical Informatics Research (2014). protégé. Retrieved November 13, 2014, from http://protege.stanford.edu

StoneFly (2014). What is File Level Storage vs. Block Level Storage? : Education : Resource Center : StoneFly's iSCSI.com. Retrieved May 1, 2014, from http://www.iscsi.com/resources/File-Level-Storage-vs-Block-Level-Storage.asp

TinkerPop (2014, November 8). TinkerPop. Retrieved November 13, 2014, from http://www.tinkerpop.com

Vaquero, L. M., Rodero-Merino, L., Caceres, J. & Lindner, M. (2009). A Break in the Clouds: Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review, 39*(1), 50–55. doi:10.1145/1496091.1496100

VGrADS (2009, September 30). The VGrADS Project — VGrADS at Rice University. Retrieved November 10, 2014, from http://vgrads.rice.edu

Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y. & Wilkins, D. (2010). A Comparison of a Graph Database and a Relational Database: A Data Provenance Perspective. In Cunningham, H. C., Ruth, P. & Kraft, N. A. (eds.), *Proceedings of the 48th Annual Southeast Regional Conference (ACM SE '10)*, Oxford, MS, April 15–17, 2010. doi:10.1145/1900008.1900067

W3C (2012, December 11). OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). Retrieved November 13, 2014, from http://www.w3.org/TR/2012/REC-owl2-syntax-20121211

W3C (2013, March 21). SPARQL 1.1 Overview. Retrieved November 12, 2014, from http://www.w3.org/TR/sparql11-overview

W3C (2014a, June 24). RDF 1.1 Primer. Retrieved November 12, 2014, from http://www.w3.org/TR/rdf11-primer

W3C (2014b, October 17). RdfStoreBenchmarking - W3C Wiki. Retrieved November 13, 2014, from http://www.w3.org/wiki/RdfStoreBenchmarking

Wang, L., Laszewski, G. V., Younge, A., He, X., Kunze, M., Tao, J. & Fu, C. (2010). Cloud Computing: a Perspective Study. *New Generation Computing, 28*(2), 137–146. doi:10.1007/s00354-008-0081-5

Wang, L., Tao, J., Kunze, M., Castellanos, A. C., Kramer, D. & Karl, W. (2008). Scientific Cloud Computing: Early Definition and Experience. In *10th IEEE International Conference on High Performance Computing and Communications (HPCC '08)* (825–830), Dalian, China, September 25–27, 2008. doi:10.1109/HPCC.2008.38

Wang, W., De, S., Cassar, G. & Moessner, K. (2013). Knowledge Representation in the Internet of Things: Semantic Modelling and its Applications. *Automatika – Journal for Control, Measurement, Electronics, Computing and Communications, 54*(4), 388–400. doi:10.7305/automatika.54-4.414

Wikipedia (2014a, October 6). Database server - Wikipedia, the free encyclopedia. Retrieved November 2, 2014, from http://en.wikipedia.org/wiki/Database_server

Wikipedia (2014b, September 30). Relational database - Wikipedia, the free encyclopedia. Retrieved November 2, 2014, from http://en.wikipedia.org/wiki/Relational_database

Wikipedia (2014c, December 15). Percentile - Wikipedia, the free encyclopedia. Retrieved January 19, 2015, from http://en.wikipedia.org/wiki/Percentile

Wolski, R., Grzegorczyk, C., Nurmi, D., Obertelli, G., Rajagopalan, S., Soman, S., Youseff, L. & Zagorodnov, D. (2008). EUCALYPTUS: An Elastic Utility Computing Architecture for Linking Your Programs to Useful Systems. Retrieved September 30, 2014, from http://cdn.oreillystatic.com/en/assets/1/event/7/EUCALYPTUS%20-%20Elastic%20Utility%20Computing%20Architecture%20for%20Linking%20Your%20Programs%20To%20Useful%20Systems%20Presentation.ppt

Wu, J., Ping, L., Ge, X., Wang, Y. & Fu, J. (2010). Cloud Storage as the Infrastructure of Cloud Computing. In *2010 International Conference on Intelligent Computing and Cognitive Informatics (ICICCI 2010)* (380–383), Kuala Lumpur, Malaysia, June 22–23, 2010. doi:10.1109/ICICCI.2010.119

Wu, J., Zhang, J., Lin, Z. & Ju, J. (2010). Recent Advances in Cloud Storage. In Zou, Y., Yu, F., Jia, Z. & Li, Z. (eds.), *Proceedings of the Third International Symposium on Computer Science and Computational Technology (ISCSCT '10)* (151–154), Jiaozuo, China, August 14–15, 2010. Retrieved May 1, 2014, from http://www.academypublisher.com/proc/iscsct10/papers/iscsct10p151.pdf

Zhang, Q., Cheng, L. & Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications, 1*(1), 7–18. doi:10.1007/s13174-010-0007-6

APPENDIX 1: SMART CITY ONTOLOGY

```
<?xml version="1.0"?>


<!DOCTYPE rdf:RDF [
   <!ENTITY www "http://www.jkl.fi#" >
   <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>


<rdf:RDF xmlns="http://www.jkl.fi#"
    xml:base="http://www.jkl.fi"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:www="http://www.jkl.fi#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
   <owl:Ontology rdf:about="http://www.jkl.fi"/>



   <!--

///////////////////////////////////////////////////////////////////////
/////////////////
   //
   // Object Properties
   //

///////////////////////////////////////////////////////////////////////
/////////////////
    -->




   <!-- http://www.jkl.fi#Films -->

   <owl:ObjectProperty rdf:about="&www;Films">
      <rdfs:domain rdf:resource="&www;Camera"/>
      <rdfs:range rdf:resource="&www;CameraObservation"/>
   </owl:ObjectProperty>
```

```
    <!-- http://www.jkl.fi#IsLocated -->

    <owl:ObjectProperty rdf:about="&www;IsLocated">
        <rdfs:domain rdf:resource="&www;Camera"/>
        <rdfs:range rdf:resource="&www;Place"/>
        <rdfs:domain rdf:resource="&www;Sensor"/>
    </owl:ObjectProperty>



    <!-- http://www.jkl.fi#Observes -->

    <owl:ObjectProperty rdf:about="&www;Observes">
        <rdfs:domain rdf:resource="&www;Sensor"/>
        <rdfs:range rdf:resource="&www;SensorObservation"/>
    </owl:ObjectProperty>



    <!-- http://www.jkl.fi#Possesses -->

    <owl:ObjectProperty rdf:about="&www;Possesses">
        <rdfs:domain rdf:resource="&www;Person"/>
        <rdfs:range rdf:resource="&www;Phone"/>
    </owl:ObjectProperty>



    <!--
///////////////////////////////////////////////////////////////////////
////////////////
    //
    // Data properties
    //
///////////////////////////////////////////////////////////////////////
////////////////
     -->




    <!-- http://www.jkl.fi#CameraObservationData -->

    <owl:DatatypeProperty rdf:about="&www;CameraObservationData">
        <rdfs:domain rdf:resource="&www;Camera"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>
```

```
<!-- http://www.jkl.fi#CameraObservationTime -->

<owl:DatatypeProperty rdf:about="&www;CameraObservationTime">
   <rdfs:domain rdf:resource="&www;CameraObservation"/>
   <rdfs:range rdf:resource="&xsd;dateTime"/>
</owl:DatatypeProperty>



<!-- http://www.jkl.fi#CameraType -->

<owl:DatatypeProperty rdf:about="&www;CameraType">
   <rdfs:domain rdf:resource="&www;Camera"/>
   <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>



<!-- http://www.jkl.fi#PersonAge -->

<owl:DatatypeProperty rdf:about="&www;PersonAge">
   <rdfs:domain rdf:resource="&www;Person"/>
   <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>



<!-- http://www.jkl.fi#PersonDescription -->

<owl:DatatypeProperty rdf:about="&www;PersonDescription">
   <rdfs:domain rdf:resource="&www;Person"/>
   <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>



<!-- http://www.jkl.fi#PersonName -->

<owl:DatatypeProperty rdf:about="&www;PersonName">
   <rdfs:domain rdf:resource="&www;Person"/>
   <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

```
<!-- http://www.jkl.fi#PersonSex -->

<owl:DatatypeProperty rdf:about="&www;PersonSex">
   <rdfs:domain rdf:resource="&www;Person"/>
   <rdfs:range>
      <rdfs:Datatype>
         <owl:oneOf>
            <rdf:Description>
               <rdf:type rdf:resource="&rdf;List"/>
               <rdf:first>Female</rdf:first>
               <rdf:rest>
                  <rdf:Description>
                     <rdf:type rdf:resource="&rdf;List"/>
                     <rdf:first>Male</rdf:first>
                     <rdf:rest rdf:resource="&rdf;nil"/>
                  </rdf:Description>
               </rdf:rest>
            </rdf:Description>
         </owl:oneOf>
      </rdfs:Datatype>
   </rdfs:range>
</owl:DatatypeProperty>



<!-- http://www.jkl.fi#PhoneNumber -->

<owl:DatatypeProperty rdf:about="&www;PhoneNumber">
   <rdfs:domain rdf:resource="&www;Phone"/>
   <rdfs:range rdf:resource="&xsd;integer"/>
</owl:DatatypeProperty>



<!-- http://www.jkl.fi#PhoneType -->

<owl:DatatypeProperty rdf:about="&www;PhoneType">
   <rdfs:domain rdf:resource="&www;Phone"/>
   <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>



<!-- http://www.jkl.fi#PlaceAddress -->

<owl:DatatypeProperty rdf:about="&www;PlaceAddress">
   <rdfs:domain rdf:resource="&www;Place"/>
   <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>
```

```
<!-- http://www.jkl.fi#PlaceDescription -->

<owl:DatatypeProperty rdf:about="&www;PlaceDescription">
  <rdfs:domain rdf:resource="&www;Place"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>



<!-- http://www.jkl.fi#SensorObservationTemperature -->

<owl:DatatypeProperty rdf:about="&www;SensorObservationTemperature">
  <rdfs:domain rdf:resource="&www;SensorObservation"/>
  <rdfs:range rdf:resource="&xsd;decimal"/>
</owl:DatatypeProperty>



<!-- http://www.jkl.fi#SensorObservationTime -->

<owl:DatatypeProperty rdf:about="&www;SensorObservationTime">
  <rdfs:domain rdf:resource="&www;Sensor"/>
  <rdfs:range rdf:resource="&xsd;dateTime"/>
</owl:DatatypeProperty>



<!-- http://www.jkl.fi#SensorType -->

<owl:DatatypeProperty rdf:about="&www;SensorType">
  <rdfs:domain rdf:resource="&www;Sensor"/>
  <rdfs:range rdf:resource="&xsd;string"/>
</owl:DatatypeProperty>



<!--
///////////////////////////////////////////////////////////////////////////
/////////////////
  //
  // Classes
  //
///////////////////////////////////////////////////////////////////////////
/////////////////
   -->



<!-- http://www.jkl.fi#Camera -->

<owl:Class rdf:about="&www;Camera"/>
```

```
    <!-- http://www.jkl.fi#CameraObservation -->

    <owl:Class rdf:about="&www;CameraObservation"/>



    <!-- http://www.jkl.fi#Person -->

    <owl:Class rdf:about="&www;Person"/>



    <!-- http://www.jkl.fi#Phone -->

    <owl:Class rdf:about="&www;Phone"/>



    <!-- http://www.jkl.fi#Place -->

    <owl:Class rdf:about="&www;Place"/>



    <!-- http://www.jkl.fi#Sensor -->

    <owl:Class rdf:about="&www;Sensor"/>



    <!-- http://www.jkl.fi#SensorObservation -->

    <owl:Class rdf:about="&www;SensorObservation"/>
</rdf:RDF>



<!-- Generated by the OWL API (version 3.5.0) http://owlapi.sourceforge.net -->
```

APPENDIX 2: APACHE JMETER TEST PLAN

General settings		
Test plan / element	**Test plan / element name**	**Settings**
Test plan	Test plan	Run thread groups consecutively (i.e., run groups one at a time): yes (during the first part of the test when the initial graph is created), no (otherwise)
Loadosophia.org uploader	Loadosophia.org uploader	Initiate active test: yes, upload to project: Stardog/Neo4j, test title: 1a–1d, 2a–2d
HTTP header manager	HTTP header manager (Stardog)	Content-Type: application/sparql-update (write queries), Accept: application/sparql-results+json, Content-Type: application/sparql-query (read queries)
HTTP request defaults	HTTP request defaults (Stardog)	Web server IP: 130.234.208.101, port number: 8080, implementation: Java, path: /db/update (write queries), /db/query (read queries)
HTTP header manager	HTTP header manager (Neo4j)	Accept: application/json; charset=UTF-8, Content-Type: application/json
HTTP request defaults	HTTP request defaults (Neo4j)	Web server IP: 130.234.208.102, port number: 8080, implementation: Java, path: /db/data/transaction/commit

1. Create the graph

Element	**Element name**	**Settings**
Thread group	Create places (Stardog/Neo4j)	Number of threads (users): 1, ramp-up period (in seconds): 1, loop count: 1000
Counter	Counter	Start: 1, increment: 1, maximum: 1000, reference name: ID1
Function	RandomString	Generates a random string
HTTP request	Create a place	Method: POST

Description of the body data

The query creates 1000 places that have IDs, names (PlaceDescription), and addresses.

Body data (Stardog)

```
PREFIX sm: <http://www.jkl.fi#>
INSERT DATA
{ sm:Place${ID1} sm:PlaceDescription "Place ${ID1}" ;
                    sm:PlaceAddress "${__RandomString(30,abcdefghijklmnopqrstuvwxyz1234567890,)}" }
```

Body data (Neo4j)

```
{
  "statements" : [
    {
      "statement" : "CREATE (a:Place {props})",
      "parameters" : {
        "props" : {
          "PlaceID" : ${ID1},
          "PlaceDescription" : "Place ${ID1}",
          "PlaceAddress" : "${__RandomString(30,abcdefghijklmnopqrstuvwxyz1234567890,)}"
        }
      }
    }
  ]
}
```

Element	**Element name**	**Settings**
Thread group	Create sensors (Stardog/Neo4j)	Number of threads (users): 1, ramp-up period (in seconds): 1, loop count: 1000
Counter	Counter	Start: 1, increment: 1, maximum: 1000, reference name: ID2
HTTP request	Create a sensor	Method: POST

Description of the body data

The query creates 1000 temperature sensors that have IDs and that are located in the aforementioned 1000 places.

Body data (Stardog)

```
PREFIX sm: <http://www.jkl.fi#>
INSERT DATA
{ sm:Sensor${ID2} sm:IsLocated sm:Place${ID2} ;
                    sm:SensorType "Temperature sensor" }
```

Body data (Neo4j)

```
{
  "statements" : [
    {
      "statement" : "MATCH (a:Place {PlaceID: {PlaceID}}) CREATE (b:Sensor {props})-[:ISLOCATED]->(a)",
      "parameters" : {
        "PlaceID" : ${ID2},
        "props" : {
          "SensorID" : ${ID2},
          "SensorType" : "Temperature sensor"
        }
      }
    }
  ]
}
```

Element	Element name	Settings
Thread group	Create cameras (Stardog/Neo4j)	Number of threads (users): 1, ramp-up period (in seconds): 1, loop count: 250
Counter	Counter	Start: 1, increment: 1, maximum: 250, reference name: ID3
HTTP request	Create a camera	Method: POST

Description of the body data

The query creates 250 video cameras that have IDs and that are located in the 250 first places of the aforementioned 1000 places.

Body data (Stardog)

```
PREFIX sm: <http://www.jkl.fi#>
INSERT DATA
{ sm:Camera${ID3} sm:IsLocated sm:Place${ID3} ;
                          sm:CameraType "Video camera" }
```

Body data (Neo4j)

```
{
   "statements" : [
      {
         "statement" : "MATCH (a:Place {PlaceID: {PlaceID}}) CREATE (b:Camera {props})-[:ISLOCATED]->(a)",
         "parameters" : {
            "PlaceID" : ${ID3},
            "props" : {
               "CameraID" : ${ID3},
               "CameraType" : "Video camera"
            }
         }
      }
   ]
}
```

Element	Element name	Settings
Thread group	Create persons (Stardog/Neo4j)	Number of threads (users): 1, ramp-up period (in seconds): 1, loop count: 1000
Counter	Counter	Start: 1, increment: 1, maximum: 1000, reference name: ID4
Function	RandomString	Generates a random string
Function	Random	Generates a random number
Function	chooseRandom	Chooses a single random value from a list of arguments
HTTP request	Create a person	Method: POST

Description of the body data

The query creates 1000 people that have IDs, names, ages, and sexes.

Body data (Stardog)

```
PREFIX sm: <http://www.jkl.fi#>
INSERT DATA
{ sm:Person${ID4} sm:PersonName "${__RandomString(20,abcdefghijklmnopqrstuvwxyz1234567890,)}" ;
                         sm:PersonAge ${__Random(1,65,)} ;
                         sm:PersonSex "${__chooseRandom(Male,Female,Random)}" }
```

Body data (Neo4j)

```
{
   "statements" : [
      {
         "statement" : "CREATE (a:Person {props})",
         "parameters" : {
            "props" : {
               "PersonID" : ${ID4},
               "PersonName" : "${__RandomString(20,abcdefghijklmnopqrstuvwxyz1234567890,)}",
               "PersonAge" : ${__Random(1,65,)},
               "PersonSex" : "${__chooseRandom(Male,Female,Random)}"
            }
         }
      }
   ]
}
```

Element	Element name	Settings
Thread group	Create phones (Stardog/Neo4j)	Number of threads (users): 1, ramp-up period (in seconds): 1, loop count: 1000
Counter	Counter	Start: 1, increment: 1, maximum: 1000, reference name: ID5
Counter	Counter	Start: 1234567890, increment: 1, reference name: NUMBER
Function	RandomString	Generates a random string
HTTP request	Create a phone	Method: POST

Description of the body data

The query creates 1000 phones that have IDs, numbers, and types, and that the aforementioned persons possess.

Body data (Stardog)

```
PREFIX sm: <http://www.jkl.fi#>
INSERT DATA
{ sm:Person${ID5} sm:Possesses sm:Phone${ID5} } ;
INSERT DATA
{ sm:Phone${ID5} sm:PhoneNumber ${NUMBER} ;
                        sm:PhoneType "${__RandomString(20,abcdefghijklmnopqrstuvwxyz1234567890,)}" }
```

Body data (Neo4j)

```
{
   "statements" : [
      {
         "statement" : "MATCH (a:Person {PersonID: {PersonID}}) CREATE (a)-[:POSSESSES]->(b:Phone {props})",
         "parameters" : {
            "PersonID" : ${ID5},
            "props" : {
               "PhoneID" : ${ID5},
               "PhoneNumber" : ${NUMBER},
               "PhoneType" : "${__RandomString(20,abcdefghijklmnopqrstuvwxyz1234567890,)}"
            }
         }
      }
   ]
}
```

2. Write queries

Element	Element name	Settings
Thread group	Create sensor observations (Stardog/Neo4j)	Number of threads (users): 1000, ramp-up period (in seconds): 1, loop count: 2
Counter	Counter	Start: 1, increment: 1, maximum: 1000, reference name: ID6
Counter	Counter	Start: 1, increment: 1, maximum: 2000, reference name: ID7
Counter	Counter	Start: 201410251200, increment: 1, reference name: TIME1
Function	chooseRandom	Chooses a single random value from a list of arguments
HTTP request	Create a sensor observation	Method: POST

Description of the body data

The query creates 2000 sensor observations that the aforementioned 1000 sensors observe. The sensor observations have IDs, and they are created on October 25, 2014, 12 p.m. and beyond (SensorObservationTime). They also have some values, i.e., temperatures

Body data (Stardog)

```
PREFIX sm: <http://www.jkl.fi#>
INSERT DATA
{ sm:Sensor${ID6} sm:Observes sm:SensorObservation${ID7} } ;
INSERT DATA
{ sm:SensorObservation${ID7} sm:SensorObservationTime ${TIME1} ;
                                        sm:SensorObservationTemperature ${__chooseRandom(13,14,15,Random)} }
```

Body data (Neo4j)

```
{
   "statements" : [
      {
         "statement" : "MATCH (a:Sensor {SensorID: {SensorID}}) CREATE (a)-[:OBSERVES]->(c:SensorObservation {props})",
         "parameters" : {
            "SensorID" : ${ID6},
            "props" : {
               "SensorObservationID" : ${ID7},
               "SensorObservationTime" : ${TIME1},
               "SensorObservationTemperature" : ${__chooseRandom(13,14,15,Random)}
            }
         }
      }
   ]
}
```

Element	Element name	Settings
Thread group	Create camera observations (Stardog/Neo4j)	Number of threads (users): 250, ramp-up period (in seconds): 1, loop count: 30
Counter	Counter	Start: 1, increment: 1, maximum: 250, reference name: ID8
Counter	Counter	Start: 1, increment: 1, maximum: 7500, reference name: ID9
Counter	Counter	Start: 201410251200, increment: 1, reference name: TIME2
Function	RandomString	Generates a random string
HTTP request	Create a camera observation	Method: POST

Description of the body data

The query creates 7500 camera observations that the aforementioned 250 cameras film. The camera observations have IDs, and they are created on October 25, 2014, 12 p.m. and beyond (CameraObservationTime). They also have some values, i.e., hyperlinks to so

Body data (Stardog)

```
PREFIX sm: <http://www.jkl.fi#>
INSERT DATA
{ sm:Camera${ID8} sm:Films sm:CameraObservation${ID9} } ;
INSERT DATA
{ sm:CameraObservation${ID9} sm:CameraObservationTime ${TIME2} ;
                                            sm:CameraObservationData "${__RandomString(20,abcdefghijklmnopqrstuvwxyz1234567890,)}" }
```

Body data (Neo4j)

```
{
   "statements": [
      {
         "statement" : "MATCH (a:Camera {CameraID: {CameraID}}) CREATE (a)-[:FILMS]->(c:CameraObservation {props})",
         "parameters" : {
            "CameraID" : ${ID8},
            "props" : {
               "CameraObservationID" : ${ID9},
               "CameraObservationTime" : ${TIME2},
               "CameraObservationData" : "${__RandomString(20,abcdefghijklmnopqrstuvwxyz1234567890,)}"
            }
         }
      }
   ]
}
```

3. Read queries

Element	**Element name**	**Settings**
Thread group	Show my personal information 1 (Stardog/Neo4j)	Number of threads (users): 1000, ramp-up period (in seconds): 1, loop count: 1
Counter	Counter	Start: 1, increment: 1, maximum: 1000, reference name: ID10
HTTP request	Show my personal information 1	Method: POST

Description of the body data

The query reads the personal information of the aforementioned 1000 people.

Body data (Stardog)

```
PREFIX sm: <http://www.jkl.fi#>
SELECT *
WHERE { sm:Person${ID10} ?b ?c }
```

Body data (Neo4j)

```
{
   "statements" : [
      {
         "statement" : "MATCH (a:Person {PersonID: {PersonID}}) RETURN a",
         "parameters" : {
            "PersonID" : ${ID10}
         }
      }
   ]
}
```

Element	**Element name**	**Settings**
Thread group	Count the average temperature (Stardog/Neo4j)	Number of threads (users): 1000, ramp-up period (in seconds): 1, loop count: 1
HTTP request	Count the average temperature	Method: POST

Description of the body data

The query counts the average temperature of all the sensor observations 1000 times.

Body data (Stardog)

```
PREFIX sm: <http://www.jkl.fi#>
SELECT (AVG(?b) AS ?Average)
WHERE
{ ?a sm:SensorObservationTemperature ?b }
```

Body data (Neo4j)

```
{
   "statements" : [
      {
         "statement" : "MATCH (a:SensorObservation) RETURN avg(a.SensorObservationTemperature)"
      }
   ]
}
```

4. Read and write queries

Element	**Element name**	**Settings**
Thread group	Show my personal information 2 (Stardog/Neo4j)	Number of threads (users): 1000, ramp-up period (in seconds): 1, loop count: 1
Counter	Counter	Start: 1, increment: 1, maximum: 1000, reference name: ID11
HTTP request	Show my personal information 2	Method: POST

Description of the body data

The query reads the personal information of the aforementioned 1000 people.

Body data (Stardog)

```
PREFIX sm: <http://www.jkl.fi#>
SELECT *
WHERE { sm:Person${ID11} ?b ?c }
```

Body data (Neo4j)

```
{
  "statements" : [
    {
      "statement" : "MATCH (a:Person {PersonID: {PersonID}}) RETURN a",
      "parameters" : {
        "PersonID" : ${ID11}
      }
    }
  ]
}
```

Element	**Element name**	**Settings**
Thread group	Update my personal information (Stardog/Neo4j)	Number of threads (users): 1000, ramp-up period (in seconds): 1, loop count: 1
Counter	Counter	Start: 1, increment: 1, maximum: 1000, reference name: ID12
Counter	Counter	Start: 1, increment: 1, reference name: ID13
*	Character string	10000 characters long comprising the same numbers and characters than RandomString functions above
HTTP request	Update my personal information	Method: POST

Description of the body data

The query updates the personal information of the aforementioned 1000 people. The query adds a new property to each person's information (Description, PersonDescription) that is randomized a little by the counter ID13.

Body data (Stardog)

```
PREFIX sm: <http://www.jkl.fi#>
INSERT DATA
{ sm:Person${ID12} sm:Description "${ID13}*" }
```

Body data (Neo4j)

```
{
  "statements" : [
    {
      "statement" : "MATCH (a:Person {PersonID: {PersonID}}) SET a.PersonDescription = {PersonDescription}",
      "parameters" : {
        "PersonID" : ${ID12},
        "PersonDescription" : "${ID13}*"
      }
    }
  ]
}
```