Design and Evaluation of a Hierarchical Multi-Tenant Data Management Framework for Cloud Applications

Pieter-Jan Maenhaut*†, Hendrik Moens†, Veerle Ongenae* and Filip De Turck†
*Ghent University, Faculty of Engineering and Architecture, Dept. of Industrial Technology and Construction
Valentin Vaerwyckweg 1, 9000 Ghent, Belgium
†iMinds – INTEC, Ghent University, Dept. of Information Technology
Gaston Crommenlaan 8 bus 201, 9050 Ghent, Belgium
Email: pieterjan.maenhaut@intec.ugent.be

***Abstract*—Cloud computing is a technology that enables elastic, on-demand resource provisioning. Migrating applications to the cloud can increase their elasticity, allowing them to adapt to workload changes by dynamically allocating resources. In a multi-tenant application multiple client organizations, each referred to as tenants, make use of one or more shared application instances. These shared instances must however behave like a private instance by guaranteeing both data separation and performance isolation for every tenant. In order to achieve high scalability, a multi-tenant application running on the elastic cloud requires a flexible and scalable architecture for both the computational resources and the storage resources.**

In this paper we present and evaluate the design of a data management framework which can be used to extend existing multi-tenant cloud applications in order to achieve high scalability of the storage resources. We describe the most important components, and discuss important design choices. The framework invokes data allocation algorithms in order to find a feasible allocation of tenant data resulting in a minimal operating cost and a maximal performance, while taking no more than 10 ms to execute.

I. Introduction

Multi-tenancy enables the serving of multiple client organizations, referred to as tenants, by a single, shared, application instance. Adding multi-tenancy to an application increases the utilization of available hardware resources and scaling becomes easier, resulting in lower overall application costs. Although a single instance is shared between multiple tenants, the instance needs to behave like a private instance towards every tenant by guaranteeing both data separation and performance isolation and providing support for tenant-specific customization.

Cloud computing is a technology that enables short-term elastic usage, the elimination of up-front costs, and near infinite capacity on-demand. As public cloud infrastructure providers charge for the amount of resources used, an optimal usage of available resources is desired to reduce operating costs. An elastic cloud enables applications running on the cloud to adapt to workload changes by provisioning resources automatically. Adding multi-tenancy to an application reduces the operating cost. However, as the number of tenants grows, a scalable architecture for both the application and tenant data is required.

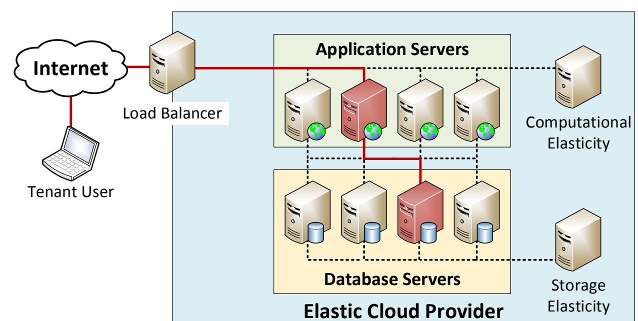

Fig. 1. A multi-tenant application running on the public cloud requires a scalable architecture for both the computational resources (application servers) and the storage resources (database servers).

Figure 1 illustrates this concept. The multi-tenant application is running on one or more application server instances, depending on the current load. If the current load on the application server instances becomes too high, additional instances must be added on the fly by a management component responsible for the elasticity of the computational resources. Similarly, the tenant data could be allocated over multiple database instances. If the current load on the database servers becomes too high, additional database instances should be added, and the existing tenant data must be reallocated. This is the task of a management component responsible for the storage elasticity. When a tenant user connects to the application, a load balancer will select one of the available application instances. The application itself however needs to connect to the proper database instance where the tenants' data is stored. Therefore, the application architecture should support the dynamic allocation of tenant data to different database servers.

In this paper, we present a hierarchical multi-tenant data management framework. This framework can be used to

extend the architecture of typical cloud applications in order to achieve high scalability of the storage resources. We illustrate this for an example typical cloud application, and describe the most important components of the framework. We discuss important design choices and evaluation results.

The remainder of this paper is structured as follows. In the next section we discuss related work. Afterward, in Section III we present an extended architecture and in Section IV we describe the most important design decisions. In Section V, we present a discussion and evaluation and in Section VI we state our conclusions and discuss avenues for future research.

II. Related Work

The work presented in this paper extends previous work as described in [1], [2]. In [1] we introduced a hierarchical method for organizing tenants and storage of tenant data, and characterized the impact on the performance in a theoretical way. In [2], we presented multiple data allocation algorithms for finding an acceptable allocation of tenant data to different database instances. In this paper we focus on the design of a data management framework which is built on top of an optimized version of one of the previously presented data allocation algorithms.

Since a majority of applications are data driven, database management systems powering these applications form a critical component in the cloud software stack. In [3], a research overview is given for designing a scalable data management layer in the cloud. The authors introduce concepts as data fusion and data fission or partitioning for combining and splitting large databases. In this paper, we also present the design of a scalable data management system, but the framework focuses on multi-tenant applications with support for tenant-specific data policies.

For the storage of persisting application data either a relational database system providing a traditional SQL interface or a NoSQL database system could be used. NoSQL databases such as Apache Cassandra [4] are gaining popularity, as they perform better than relational SQL databases in some scenarios where only a limited subset of the SQL functionality is used. Wang et al. [5] illustrate this by describing the principles of NoSQL databases and comparing a traditional relational model with a Cassandra-based model. The performance of SQL and NoSQL databases is compared in [6]. The authors however note that NoSQL databases don't always perform better than SQL databases, as within NoSQL databases there is a wide variation in performance based on the type of operation. Applying database partitioning techniques to NoSQL databases also is still an ongoing research topic, as illustrated in [7]. The framework presented in this paper is independent of the underlying database system, and can be implemented using either SQL or NoSQL databases (or even a combination of both). As a result, it can also be used for applications which require a SQL interface or legacy applications built on top of relational databases.

Data assurance policies can be used to meet legal and business data archival requirements for both persistent data and records management. Compliance with regulatory policies on data remains a key hurdle to cloud computing [8]. Jun Li et al. [9] [10] propose a policy management service that offers scalable management of data assurance policies attached to data objects stored in a cloud environment. With GEO-DAC [11], the authors provide a policy framework that enables the expression of both the service providers' capabilities and customers' requirements, and enforcement of the agreed-upon policies in service providers' environments. The data assurance policies described in their work can add additional constraints on the allocation of tenant data and therefore the framework needs to be flexible enough to support such and other data policies.

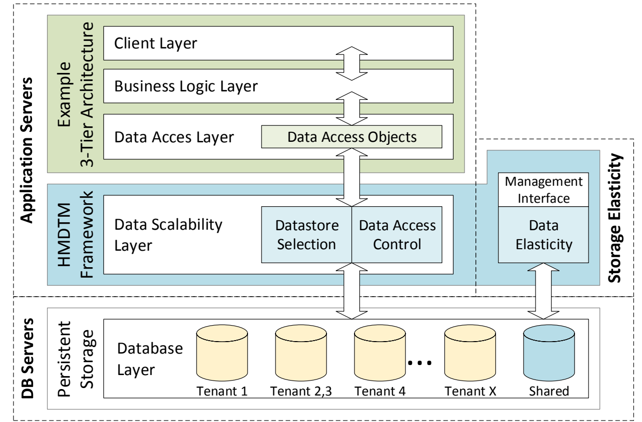

Fig. 2. Example three-tier architecture with the HMTDM framework added as an additional layer.

III. Architecture Overview

The Hierarchical Multi-Tenant Data Management (HMTDM) framework is designed to extend the architecture of typical cloud applications in order to achieve high scalability of the storage resources.

A three-tier architecture is often used for building web applications, and consists of 3 layers: the client layer, the business logic layer and the data access layer. The data access layer offers an interface to the persistent data of the application, and most web applications are using database instances for the storage of data. These database instances can be logically grouped together as a fourth layer, which we will refer to as the database layer in the application architecture. The database layer could consist of relational database instances, NoSQL database instances or a mix of both, depending on the requirements of the application.

To achieve high scalability of the database layer, the HMTDM framework can be introduced as an additional abstraction layer between the data access layer and the underlying database layer. By doing so, the physical distribution of the persistent data over the physical databases is hidden towards the data access layer. The data access layer provides a generic interface towards the rest of the architecture, independent

from the selected database systems, the number of database instances and the distribution of tenant data among these instances.

Figure 2 illustrates an example three-tier architecture with the framework introduced as an additional layer and the database instances grouped inside the database layer. Other architectures are possible, but most architectures will have a clear interface between the business logic and the persistent storage, making it possible to introduce the framework in between. As can be seen in Figure 2, we have defined three main components inside the framework:

- The **Datastore Selection** component is responsible for selecting the correct database and locating the data in one of the possible databases when data from a single tenant can be divided over multiple database instances.
- The **Data Access Control** component verifies if the current tenant user has the required permissions to read and/or modify the selected data. This component can be implemented using Attribute-Based Access Control (ABAC) as illustrated in [12], [13].
- The **Data Elasticity** component is responsible for achieving high scalability of the storage resources, by increasing and/or decreasing the number of database instances and reallocating the tenant data, based on the current usage, taking into account the different constraints such as data retention policies.

The data access control component and the datastore selection component define the interface towards the data access layer. The data elasticity component on the other hand only communicates with the database layer, and is inaccessible from within the multi-tenant application. The component however provides a management interface which can be used for monitoring and configuration. As the data elasticity is the most complex and most important component of the framework, we will describe the design of this component in more detail in the remainder of this paper.

A. Data Elasticity Component

One of the best-known opportunities of migrating applications to the cloud is its elasticity, allowing applications running on the cloud to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner. This elasticity can be offered as a service by the cloud provider or a third party. For applications running on Amazon AWS, Amazon offers CloudWatch [14], whereas for applications running on Microsoft Azure, there exist some third party products such as AzureWatch [15]. Cloud providers also often provide an Application Programming Interface (API), allowing application developers to implement a custom elasticity component. These solutions however mainly handle the elasticity of computational resources (the application servers in Figure 1), and focus less on the elasticity of the storage resources (the database servers).

As a result, we introduced the data elasticity component inside the framework, responsible for achieving high scalability of the storage resources by allocating the required amount

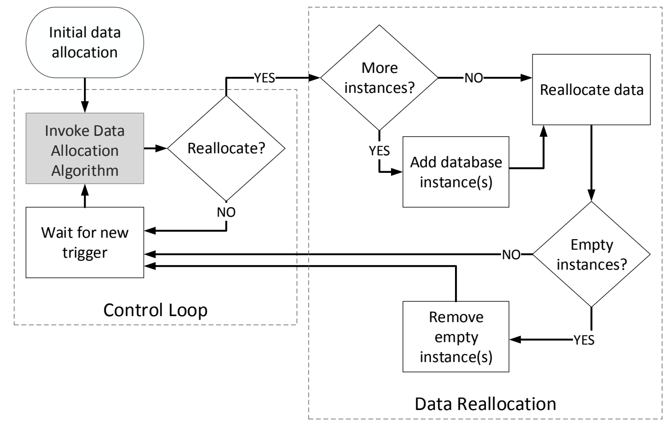

Fig. 3. The functionality of the data elasticity component, responsible for horizontal scaling of the storage resources.

of resources, in this case the number of database instances (horizontal scaling). It evaluates the current load on the different instances, adds or removes additional instances and reallocates tenant data, taking into account different constraints such as data retention policies. The data elasticity component should be implemented as a process running on a separate instance as it should have no impact on the performance of the main application.

Figure 3 illustrates the functionality of the data elasticity component. The system starts by allocating the tenant data over an initial number of database instances, and progresses into the control loop. During this control loop, the current load on the different instances is periodically evaluated. If the current load is too high, the system will add one or more additional database instances and reallocate the existing data after which the control loop continues. Similarly, if the current load is too low, the system will reallocate the existing data and remove the empty instances. The data elasticity component makes use of data allocation algorithms to find a feasible allocation of tenant data over the database instances. Re-evaluation of the current allocation of tenant data can also be triggered by a certain event, such as an overfull or underfull database instance, or when a tenant or subtenant is added or removed.

B. Database Layer

The database layer consists of multiple database instances and (sub)tenants can either have a dedicated instance or share an instance with other (sub)tenants. Tenants with a large amount of data can also have multiple database instances, for example using existing technologies such as database sharding, or in some scenarios the tenant could be logically divided into multiple subtenants, and the existing tenant data can be reallocated over multiple subtenant database instances.

One database instance is shared between all tenants, and we will refer to this instance as the **tenant configuration database**. This instance holds all general tenant information such as tenant specific configuration parameters, feature con-

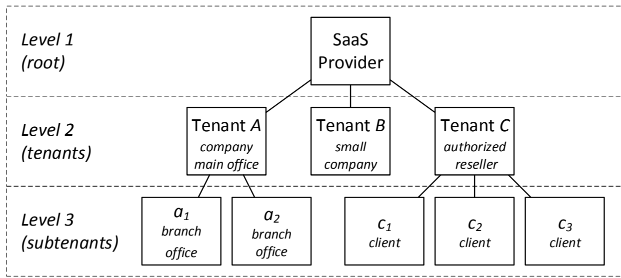

Fig. 4. Tenants and subtenants can be logically represented using the tenant tree. This figure shows an example tenant tree consisting of three levels, but more levels are possible.

figuration (if the multi-tenant variability approach from [16] is used), billing information and data assurance policies. For every tenant, a representation of the current allocation of data is also stored. The datastore selection component accesses this shared database in order to select the correct tenant database instance.

The shared tenant configuration database shouldn't become the bottleneck of the application as the amount information in it is limited, because only general information about the tenants is stored, and the most important information can be cached by the application. However, if this database becomes a bottleneck, it can be replicated or partitioned. The data elasticity component should not be used for this, as this component only handles the scalability of the other tenant database instances.

IV. Design Decisions

A. Tenant Tree

In a multi-tenant application, the tenants and subtenants can be logically represented using a tree structure, the tenant tree [1]. In this tree, every node represents a tenant or subtenant, and the root represents the application provider. Figure 4 illustrates an example tenant tree consisting of three levels, but more levels are possible.

B. Data Allocation Algorithms

As illustrated in Figure 3, the data elasticity component invokes a data allocation algorithm in order to achieve high scalability of the database layer. The goal of the data allocation algorithms is to find a possible allocation of tenant data with minimal cost. In previous work we described several data allocation algorithms that can be used for the implementation. These algorithms are either based on linear programming or permutation based and are summarized in table I. All algorithms are designed to work for a single tenant and its k subtenants (two levels of the tenant tree), but they can be applied on the full tenant tree in a recursive way as illustrated in [2].

The data allocation algorithms are using a cost function in order to evaluate a possible allocation of tenant data given certain metrics. Different metrics are possible for the design of this cost function, depending on the application's requirements and additional constraints for the storage of tenant data. A first possible metric is the number of database instances as more instances will result in a higher cost. A second possible metric is the average response time for each tenant and for the whole system, which can be calculated using the equations introduced in [1]. The cost function can also take into account the current load on the different database instances. In some scenarios, especially in heterogeneous environments where different public database providers are used, the provider may charge for moving data between 2 databases as this will use bandwidth. In this case, the total amount of data that needs to be migrated to another database instance can be counted in order to calculate the data migration cost. If the application is running on multiple instances on different physical locations, the network latency between the application server and the database server is also a useful metric. Different scenarios could use different metrics and therefore a different cost function.

TABLE I
Overview of the selected data allocation algorithms

Linear programming based	
LPDAA	Linear Programming Data Allocation Algorithm
ILPDAA	Integer Linear Programming Data Allocation Algorithm
Permutation based	
BDAA-n	Basic Data Allocation Algorithm (*n* bits)
FDAA-n	Fast Data Allocation Algorithm (*n* bits)

The linear programming based algorithms solve the data allocation problem by applying existing linear programming techniques on a mathematical model. Although they result in low execution times, the possibilities for customization of the algorithm for specific use cases are limited due to the need for linear functions for both the problem constraints and the objective function (the cost function).

The permutation based algorithms iterate over different feasible allocations of the tenant data (permutations), and for every valid permutation the corresponding cost is calculated. For a single permutation, every subtenant is represented by n bits. If $n = 1$, every subtenant can either store all of its data in a dedicated datastore, or move all data to the datastore of the parent tenant. When $n > 1$, subtenants can divide data over a dedicated datastore and the datastore of the parent tenant. Larger values for n will result in a larger search space, but at the cost of higher execution times.

The Basic Data Allocation Algorithm (BDAA) iterates over all possible permutations within the defined search space, but large values for k (the number of subtenants) and n (the number of bits used to represent a single subtenant) can rapidly make the algorithm unusable as it would take too much time to evaluate all permutations. As a result, the Fast Data Allocation Algorithm (FDAA) was designed to reduce the complexity of the algorithm. This algorithm prunes the search space in different ways, for example by removing small

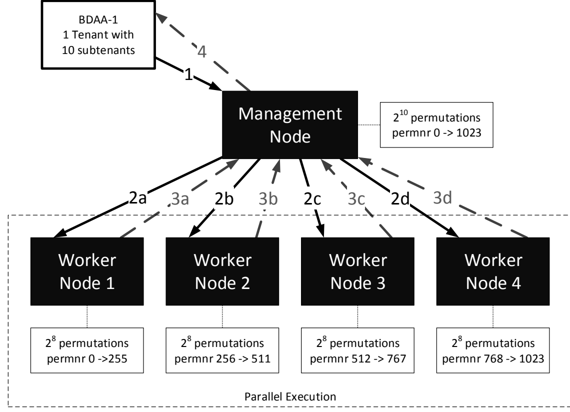

Fig. 5. Parallel execution of BDAA-1 for a single tenant with 10 subtenants, using 4 worker nodes.

tenants from the input data and adding their data to the parent node, reducing the number of permutations, and by cutting search paths that have a little influence on the calculated cost.

For the implementation of the data elasticity component, we selected the FDAA. To minimize the overhead due to the reallocation of tenant data, the cost function of the algorithm was configured to include the number of records that needs to be migrated to a different database instance as a metric. In our initial design, we used the FDAA-2 ($n = 2$), but later we decided to use the FDAA-1 instead, in order to keep data belonging to a single (sub)tenant together. By doing so, not only the number of permutations is reduced (reducing the execution time of the data allocation algorithms), but existing database partitioning techniques can also be applied, for example by partitioning based on the (sub)tenant ID, and the complexity of the system is strongly reduced as described in more detail in Section V.

C. Support for Parallelization

Although the FDAA already reduces the complexity of the BDAA, we added optional support for parallelization to further reduce the execution time of the permutation based algorithms. As the permutation based algorithms enumerate over a set of different feasible permutations, the evaluation of the permutations can be divided over different server instances. This however introduces the need for an extra instance, the *management node*, whose task is to divide the permutations over different *worker nodes* and combine the results in order to select the best permutation. The management node implements the functionality illustrated in Figure 3, whereas worker nodes only implement the selected data allocation algorithm. Figure 5 illustrates this concept for the parallel execution of BDAA-1 for a single tenant with 10 subtenants, using 4 worker nodes. The management node and the dynamic provisioning of the worker nodes can be implemented on top of an elastic cloud by invoking the provisioning functions through the cloud API.

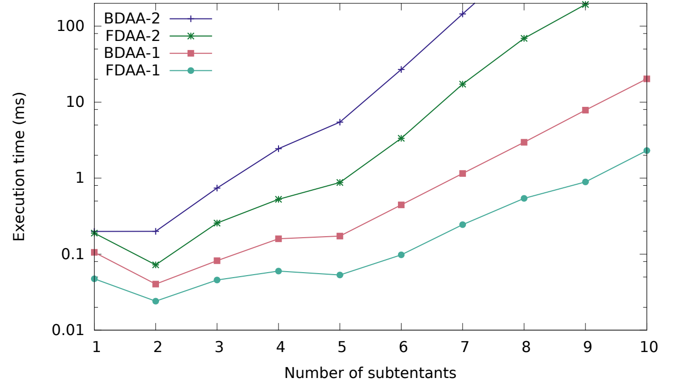

Fig. 6. A comparison of the average execution times of the selected data allocation algorithms for an increasing number of subtenants.

V. Discussion and Evaluation

Multi-tenant applications should provide a clear isolation of tenant data. As a result, queries for tenant data should not return data belonging to multiple tenants. However, when data belonging to a single tenant is distributed among different database instances, some complex queries could become unusable as they would have to access different database instances, therefore only simple select, update and delete queries could be used, and the business logic needs to implement the merging of results from different instances, for example by implementing the bottom-up search method as described in [1]. Keeping data belonging to a single tenant together, for example by using $n = 1$ in the permutation based algorithms, eliminates this problem, and when database partitioning techniques are applied, the system even supports queries over multiple tenants as the separate database instances appear towards the application as a single virtual database instance.

For the evaluation, the framework was implemented in Java and all simulations were executed on a Linux server with an Intel Core i5 CPU (2.80 GHz) and 4 GiB of memory. Figure 6 illustrates the average execution times for the different data allocation algorithms for an example scenario with an increasing number of subtenants. As can be seen from this figure, using $n = 1$ instead of $n = 2$ reduces the execution of the algorithms, and the FDAA has lower execution times than the BDAA.

Using the FDAA-1 instead of the FDAA-2 reduces the search space of the data allocation algorithms, but the calculated cost of the selected optimal solution differs only a little bit as illustrated in Figure 7. In this simulation, the cost function of the algorithms was configured to return a valid allocation of tenant data resulting in a minimum average response time with a minimum number of database instances. The tenant and its subtenants were assigned an initial amount of data and each iteration the amount of data for each tenant was slightly increased. Similar experiments were executed with different iterations and different cost functions, but these experiments provided similar results. Using the FDAA instead

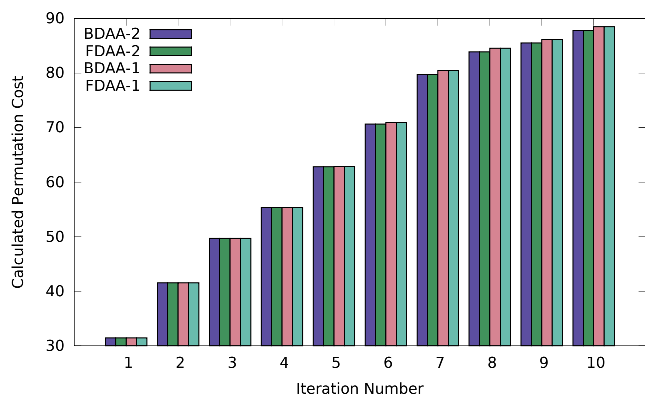

Fig. 7. The calculated cost of the selected permutation for the different data allocation algorithms over different iterations in one of the executed experiments.

of the BDAA has no influence on the calculated cost, as both algorithms are returning the same permutation. The extra cost of using the FDAA-1 instead of the FDAA-2 is strongly compensated by the advantages of keeping data belonging to a single tenant together as discussed above.

Although searching for the optimal allocation can happen offline, scenarios with a very high number of subtenants could still make the algorithm unusable as it would take too much time to find the optimal allocation of tenant data. However, by adding parallelization and using multiple worker nodes, the execution time can be heavily reduced. An elastic cloud provider offers a good base for the implementation of the data elasticity component and the creation and deletion of worker nodes, and when there is no hard time constraint on the re-evaluation of the current data allocation, the cloud bidding pricing model based could even be used to minimize the cost of running the worker nodes.

VI. Conclusions

Scalable multi-tenant applications on the public cloud require a scalable architecture for both the application and data. In this paper, we presented the design of a hierarchical multi-tenant data management framework, which can be used to extend existing cloud applications in order to achieve high scalability of the storage resources. We introduced three main components inside the framework. The data elasticity component is the most important component as it is responsible for achieving high scalability of the database layer by allocating tenant data to multiple database instances. During the control loop of this component, a data allocation algorithm is invoked in order to find a feasible allocation of tenant data with minimal cost. When a new allocation is found and reallocation is recommended, the system reallocates the tenant data and adds or removes additional database instances if required.

We described and evaluated the design of the data elasticity component and discussed important design choices. For the implementation of the data elasticity component, we selected the FDAA-1 in order to keep data belonging to a single tenant together. Doing so not only reduces execution times, but also introduces other advantages and enables the usage of existing database partitioning techniques. We also introduced optional support for parallelization to further reduce the execution time of the algorithms, and described a strategy for implementing the data elasticity component and the dynamic provisioning of the worker nodes on top of an elastic public cloud.

The presented framework can be extended to build a generic middleware layer for hosting multi-tenant applications on the public cloud. The design, implementation and evaluation of this generic middleware will be the focus of our future work.

References

[1] P.-J. Maenhaut, H. Moens, M. Decat, J. Bogaert, B. Lagaisse, W. Joosen, V. Ongenae, and F. D. Turck, "Characterizing the performance of tenant data management in multi-tenant cloud authorization systems," in *Proceedings of the 14th Network Operations and Management Symposium (NOMS2014)*, Krakow, Poland, May 2014.

[2] P.-J. Maenhaut, H. Moens, V. Ongenae, and F. De Turck, "Scalable user data management in multi-tenant cloud environments," in *Proceedings of the 10th International Conference on Network and Service Management 2014 (CNSM2014)*, Rio de Janeiro, Brazil, Nov. 2014.

[3] D. Agrawal, A. El Abbadi, S. Das, and A. J. Elmore, "Database scalability, elasticity, and autonomy in the cloud," in *Proceedings of the 16th International Conference on Database Systems for Advanced Applications - Volume Part I*, ser. DASFAA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 2–15.

[4] The Apache Cassandra project. [Online]. Available: http://cassandra.apache.org

[5] G. Wang and J. Tang, "The nosql principles and basic application of cassandra model," in *Proceedings of the 2012 International Conference on Computer Science Service System (CSSS)*, Aug 2012, pp. 1332–1335.

[6] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases," in *Communications, Computers and Signal Processing (PACRIM), 2013 IEEE Pacific Rim Conference on*, 2013, pp. 15–19.

[7] Z. Chen, S. Yang, S. Tan, G. Zhang, and H. Yang, "Hybrid range consistent hash partitioning strategy – a new data partition strategy for nosql database," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, July 2013, pp. 1161–1169.

[8] M. Henze, M. Grossfengels, M. Koprowski, and K. Wehrle, "Towards data handling requirements-aware cloud computing," in *Proceedings of the 2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom)*, vol. 2, Dec. 2013, pp. 266–269.

[9] J. Li, S. Singhal, R. Swaminathan, and A. Karp, "Managing data retention policies at scale," in *Proceedings of the 2011 IFIP/IEEE International Symposium on Integrated Network Management (IM2011)*, 2011, pp. 57–64.

[10] J. Li, S. Singhal, R. Swaminathan, and A. Karp, "Managing data retention policies at scale," *IEEE Transactions on Network and Service Management*, vol. 9, no. 4, pp. 393–406, 2012.

[11] J. Li, B. Stephenson, H. Motahari-Nezhad, and S. Singhal, "GEODAC: A data assurance policy specification and enforcement framework for outsourced services," *IEEE Transactions on Services Computing*, vol. 4, no. 4, pp. 340–354, 2011.

[12] M. Decat, B. Lagaisse, D. Van Landuyt, B. Crispo, and W. Joosen, "Federated authorization for software-as-a-service applications," in *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, vol. 8185. Springer, September 2013, pp. 342–359.

[13] M. Decat, B. Lagaisse, and W. Joosen, "Middleware for efficient and confidentiality-aware federation of access control policies," *Journal of Internet Services and Applications*, vol. 5, pp. 1–15, February 2014.

[14] Amazon Cloudwatch - Cloud and Network Monitoring Services. [Online]. Available: http://aws.amazon.com/cloudwatch/

[15] Autoscaling and monitoring for Windows Azure applications. [Online]. Available: http://www.paraleap.com/azurewatch

[16] H. Moens, E. Truyen, S. Walraven, W. Joosen, B. Dhoedt, and F. De Turck, "Cost-effective feature placement of customizable multi-tenant applications in the cloud," *Journal of Network and Systems Management*, Feb. 2013.