Multi-cloud and Multi-data Stores
The Challenges Behind Heterogeneous Data Models

Marcos Aurélio Almeida da Silva and Andrey Sadovykh
Research and Development, Softeam, 8 parc Ariane Immeuble Le Jupiter, SOFTEAM, 78284 CEDEX, Guyancourt, France

Keywords: Data Stores, NoSQL, Data Migration.

Abstract: The support to cloud enabled databases varies from one cloud provider to another. Developers face the task of supporting applications living in different clouds, and therefore of supporting different database management systems. To them, the challenge lies in understanding the differences in expressivity between different data stores and their impact on the application. The advent of the NoSQL movement increased the complexity of this task by leveraging the creation of a large number of cloud enabled database management systems employing slightly different data models. In this paper, we will present a model the will allow us to compare the differences in expressivity of the features supported by different databases and consider the impact of these features to different concrete deployment scenarios in multiple clouds. This model is based on the underlying data models adopted by the most used cloud database management systems. It has been developed on the FP7 JUNIPER project and will be the basis of our approach for dealing with these issues.

1 INTRODUCTION

A decade after the advent of the first cloud based solutions, it is clear to companies that migrating to cloud platforms is cost effective (Rackspace, 2013). The success of the cloud lead to the advent of multiple cloud provider offerings. As in any nascent market, there are no established standards. Economically speaking, on the one hand, the multiplication of offerings reduces the prices and makes cloud and multi-cloud applications more and more interesting to companies. On the other hand, the consequent fragmentation of the market, makes the life of cloud developers harder, since it increases the complexity of the development and maintenance of applications.

In this paper we focus on the challenges related to data stored on the cloud. The problem stems from the fact that different cloud providers support different database management systems (DMS). Developers have a variable degree of flexibility, ranging from the one they have on Infrastructure As a Service providers, in which virtually any DMS can be installed; to the one they have in the Platform as A Service providers, usually supporting a very specific subset of DMSs. Finally, in the Software as a Service providers, developers are usually only able to store data opaquely and this data is only later accessible by means of a provider specific APIs.

The consequences of this fragmentation of the support of DMSs by cloud providers are amplified by the so-called NoSQL "movement". This "movement" consists of a series of DMSs that strip the well-known SQL based relational DMSs from some for their characteristics in order to increase their performance. The problem is that different applications usually have different performance bottlenecks, which leads to different sets of optimizations that need to be applied to SQL DMSs to make them adapted to each application. This lead to the existence of a myriad of NoSQL DMSs, each one based on a slightly different set of optimizations on the SQL DMSs or even on completely different data models, fine-tuned to specific applications.

For a developer, building and maintaining a cloud application means dealing with all this fragmentation. **The main hypothesis of this paper is that in order to deal with these concerns, one, first of all, needs to understand the differences in expressivity between the data models provided by different DMSs and the potential difficulties in migrating data from one model to another.** In this paper we are going to present the main concepts behind most used cloud DMSs, and the semantic gaps between them. We are then going to present a

Aurélio Almeida da Silva M. and Sadovykh A..
Multi-cloud and Multi-data Stores - The Challenges Behind Heterogeneous Data Models.
DOI: 10.5220/0004974607030713
In *Proceedings of the 4th International Conference on Cloud Computing and Services Science* (MultiCloud-2014), pages 703-713
ISBN: 978-989-758-019-2
Copyright © 2014 SCITEPRESS (Science and Technology Publications, Lda.)

classification of the DMSs according to these concepts and we'll use this classification in a case study in which we identify the best DMSs to support parts of the data manipulated by an application.

The present work is a first step towards what we intend to achieve in the EU FP7 projects MODAClouds and JUNIPER. They intend to tackle such challenges by means of the model driven approach to allow developers to model their application on a very high level. Developers can then have their models analysed by automated tools and to have code automatically generated from it. The MODAClouds project focuses on public clouds and in providing automated tools to help data design. The JUNIPER project focuses on private clouds and in providing analysis tools to make sure that a particular set of DMSs respects a given set of real time constraints.

This paper is structured as follows. Section 2 details the context of data management in multi cloud applications and introduces the fragmentation of databases in this domain. Section 3 presents the most important concepts used in multi-cloud DMSs and a classification of the most important DMSs. Section 4 presents the trade-offs in migrating to and from different classes of DMS. Finally, in Sections 5 and 6 we present related works and conclusions of this paper.

2 MULTI-CLOUD APPLICATIONS AND THE NOSQL "MOVEMENT"

2.1 Overview

The cloud started as a way to offer all this as a service, in a "pay as you" go way. That means that the cloud provider would create a big data centre, and would use it to offer virtual machines to clients. On top of purely infrastructure driven cloud solutions, platform driven ones came into being. In this case, cloud providers do not offer virtual machines and storage device, but instead, they offer software platforms. The customer then is only responsible for installing the necessary software on the platform while the cloud provider will dimension the needed machines, storages and load balance strategies for the user's application (Khajeh-Hosseini, Greenwood, & Sommerville, 2010).

The main disadvantage of clouds is that users have much less flexibility than in a "on premises" solution. Each cloud provider provides only a limited set of configurations of machines, storage and platforms, while on premise solutions allow for unlimited sets of configurations. Each cloud is also optimized for a limited range of applications, i.e. some clouds are optimized to running applications involving fast running queries and long running background processes; while others may also accept long running queries over data. One way to mitigate this heterogeneity problem is to use multiple cloud providers, putting parts of the application on each provider, trying to find the best match between the cloud and the application (Liu, Katsuno, Sun, & Li, 2011) (Singh, Kandah, & Zhang, 2011).

As one could expect, the data storages supported by each cloud provider vary from one offering to another. This is so, because data storage is nowadays a much complicated matter than it was years ago. It doesn't consist anymore of choosing between traditional relational databases or home grown file based data formats. Now, developers have a myriad of Data Management Systems (DMS), each of them optimized to a particular set of data structures. This is the result of the NoSQL "movement", which in fact intends to improve the efficiency of relational DMSs by constraining the data structures they support and the queries that they can answer.

The downside for the programmer is that designing a cloud application is not only a matter of choosing the "cheapest cloud provider", but choosing the provider that supports the DMSs backed by the best data structures to represent the application data. One still needs to think about the cost and performance costs involved in transferring data from one cloud to another, and consequently from one DMS and backed data structure to another.

The main objective of this paper is helping developers in choosing the best DMSs for their data and in understanding the **performance and expressiveness trade-offs** involved in moving data from one DMS to another. In order to do so, we intend to provide a model of the data structures and queries supported by existing NoSQL and SQL

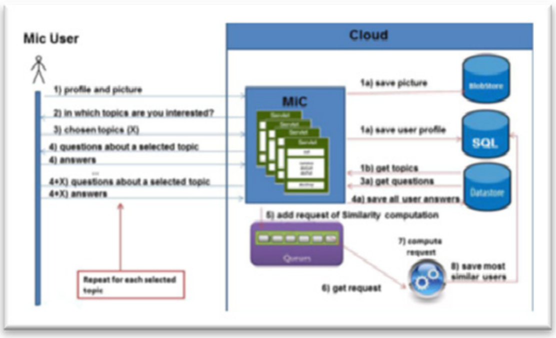
Mic User
Cloud
1) profile and picture
2) in which topics are you interested?
3) chosen topics (X)
4) questions about a selected topic
4) answers
4+X) questions about a selected topic
4+X) answers
Repeat for each selected topic
MIC
1a) save picture
BlobStore
1a) save user profile
SQL
1b) get topics
3a) get questions
4a) save all user answers
Datastore
5) add request of Similarity computation
Queues
6) get request
7) compute request
8) save most similar users

Figure 1: Overview of the MiC Application.

based DMS. Developers will then be able to develop high performance applications, losing the least as possible when moving data from one DMS to another.

2.2 Motivating Example: The MiC Application

The MiC (Meeting in the Cloud) application (F., D., M., D., & E., 2013) is a social network which allows users to maintain user profiles in which they register they topics of interests. The MiC application then groups users by similarity, allowing users to interact with their "best contacts", based on the answers given by each user in their profiles. Figure 1 overviews the main workflow of use of the MiC application.

Figure 3 presents a simplified view of the data model behind the MiC application. It stores, `Messages` posted by `UserProfiles` in `Topics` associated to `Questions`. `UserRatings` store ratings given by `UserProfiles` to `Topics`. `UserRatings` also include `Pictures` of users. Finally, and `UserSimilarity` stores pairs of similar users.

When developing this application, developers need to decide on using an infrastructure or platform as a service solution; and then on which specific provider the application is going to be deployed. When it comes to designing the data layer of the application, the developer has to decide on which DMSs will be reused and which part of the data is going to be stored on each DMS.

In order to illustrate the complexity of these choices, let us suppose the developers want to use platform as a service cloud providers, in order to reduce the cost of managing the infrastructure and to focus on the application design. Suppose they want to choose between Microsoft Azure, Heroku and Google App Engine.

Provider	DMS
Microsoft Azure	Table Service, Blob Service
Heroku	Postgres, Cloudant add-on
Google App Engine	Datastore, Blobstore

Figure 2: Comparing possible platform as a service providers for the MiC application.

Without going into the details on each DMS, Figure 3 shows that each provider includes a variety of different data stores. Each DMS supports slightly different kinds of data, with different levels of details.

For example, on the one hand, blob services support hash like structures that associate binary data to unique keys. On the other hand, table services, Google Datastores and the Cloudant add-on store multiple pieces of data associated to a single key. The former are optimal for queries on leys, while the later may support filters and more complex queries on the values associated to each key.

On top of that, each provider has different pricing strategies. The developer then needs to understand the trade-offs when designing the MiC application, in order to eventually store part of the data in one cloud and part of the data on another.

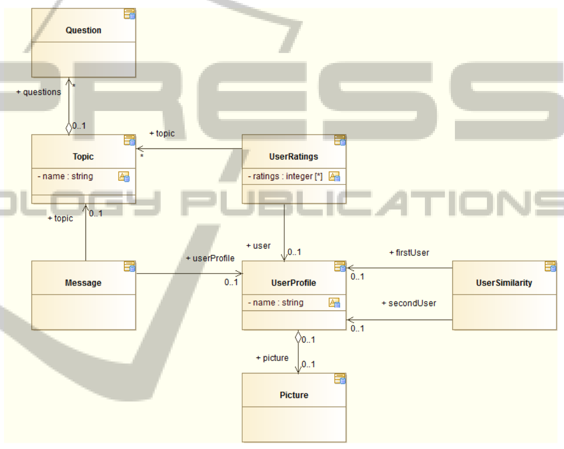

Figure 3: The data model of the MiC Application.

This paper focuses on the trade-offs involved in store different kinds of data in different DMSs, eventually in different cloud providers. The cost optimization involved in this task **is out of the scope of the present paper.**

3 CLOUD BASED DMS CONCEPTS AND COMPARISSON

In this section we present the main concepts concerning the data design in a Big Data Real-time system. These concepts are going to be presented in Section 3.1 and used to compare the most popular DMSs in Section 3.2.

3.1 Main Concepts

The main concepts related to **data structures** are summarized in Figure 5. They are based on an

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

extensive review of relational and NoSQL DMSs initially published in (SOFTEAM; University of York, 2013). The first aspect to be dealt with at this section is the **Underlying Data Abstraction** supported by the database. That is important because storing the same data under different data abstractions may lead to data loss and/or increase the complexity of the application code.

The next aspect is the one of how each system **uniquely identifies** data stored in it. This is usually done by means of a piece of data called **Key**. Notice that keys are dispensable in object oriented databases; because objects are unique by themselves, no matter the data they contain. Keys may be atomic or composed of many pieces of data (they are then called **Composite Keys**). Additionally, **File Paths** are special kinds of keys that uniquely identify documents in file systems. Finally, keys may be **Ordered** or not.

The **Values** stored in the database are represented differently from one system to another. They can represent **Single** or **Multiple** columns containing primitive types only or **Documents,** which stand for non-structured blobs of information. Finally, columns may be **Single** or **Multi-valued**

Links between different pieces of information are established differently in different kinds of database. In tuple based ones, **Foreign-Keys** are generally used, while in object oriented ones **Relationships** are used**.** A relationship is a direct link from an object to another, allowing navigation usually in constant time. Foreign-keys link two tuples by adding the key from one tuple as part of the columns represented in another. Lookups from tuples using foreign-keys may vary from logarithm time complexity in single primitive ordered keys, to linear time in non-ordered keys.

Different tools also provide different strategies for **Aggregating** data. **Tuple Spaces** and **Regions** group objects or tuples in different containers, so that items that are most accessed together (from a single region) can be retrieved more quickly. Tuple spaces differ from regions by the fact that they are also a concurrent programming mechanism: processes can **put** and **take** tuples from the tuple space, i.e. no two processes can take the same tuple at the same time. The third aggregation technique is called **Column Families.** In this case, the columns that form each tuple are grouped into families of columns that should be stored together, accelerating analysis over the whole column (e.g. summing all values).

3.2 Comparing Cloud based DMSs

Figure 5 presents and compares the main kinds of Big Data databases based on the concepts presented in the previous section and presents the main implementations for each category of database.

Distributed File Systems represent data as an association between file paths (used as keys) and documents (that represent file content). The underlying data abstraction paradigm is the object oriented one, i.e. files are not uniquely identified by their content, but only by their paths (usually, several paths may point to a single file).

The **Key-Value Stores** represent data as simple tuples containing simple primitive keys and a single column of data. **Ordered Key-Value** stores support

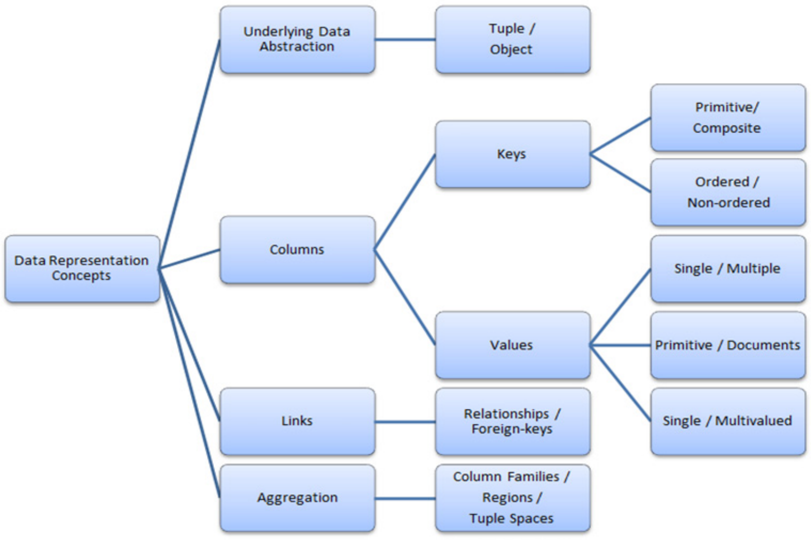

Figure 4: Kinds of Data Structure related concepts.

ordered keys, and therefore allow retrieval of ranges in linear time on the length of the range, whereas in Key-value systems this operation may be quadratic. **Document Stores** are also a special case of key-value stores in which the single column in each row (apart from the key) can store a arbitrarily complex document. Notice that these four kinds of data stores may be referred to generally as **Key-Value Stores**.

Big Tables are tuples with a primitive key and multiple columns aggregated in families and rows that can be grouped into regions. **Object Databases** represent objects which contain multiple columns (or fields) and are connected by means of relationships. **Multivalued Databases** are systems that allow more than one value to be stored at a time for a column. Expressiveness and PErformance Trade-offs.

Tuple Stores are databases that support tuple spaces. Finally, **Relational Databases** are tuple based databases supporting composite keys and foreign keys.

This section presents the trade-offs in the different data structures used by different DMSs and in the underlying limitations of these structures. Notice that complexity estimates are given relative to abstract algorithms needed to back such structures on the general case. Specific versions of sets of configuration parameters of some DMSs may achieve **better** time or space complexity for specific classes of input, which are out of the scope of this paper.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Category	**Underlying Data Abstraction**	**Keys**	**Values**	**Links**	**Aggregation**	**Examples**
Distributed File Systems	Object	Primitive (File Path)	Document	-	-	HDFS, Lustre
Key-value Store	Tuple	Primitive	Single Column	-	-	Amazon DynamoDB
Ordered Key-value Store	Tuple	Ordered	Single Column	-	-	Memcache DB, Redis
Document Store	Tuple	Primitive	Document	-	-	MongoDB, CouchDB, Riak SimpleDB
Big Table	Tuple	Primitive	Multiple Columns	-	Column Families, Regions	Google BigTable, Cassandra,
Object Database/RDF Store	Object	-	Multiple Columns	Relationships	-	Neo4j, RavenDB, FlockDB, InfiniteGraph
Multivalued databases	Tuple		Multiple Multivalued Columns		-	jBASE, Caché
Tuple store	Tuple	-	Multiple Columns	-	Tuple Spaces	Gigaspace, Javaspaces, Tarantool
Relational Database	Tuple	Composite Primitive	Multiple Columns	Foreign Keys	-	MySQL

Figure 5: Comparing cloud enabled databases.

3.3 Underlying Data Abstraction: Tuples x Objects

In practice, tuple based data models are the best suited for **data aggregation**, while the object based ones are the best suited for **navigation**. To illustrate that, let us consider a social network with tenths of millions of users. On the one hand, when storing data for computing statistical information about these users, e.g. they average age, one should prefer a tuple based data model (probably on top of a vertical partitioning scheme). On the other hand, when storing data for computing properties of the graph of friend-to-friend connections, e.g. computing a series of suggestions of products to buy for each user based on his/her friends, one should prefer an object oriented data model (probably using horizontal partitioning).

Expressiveness trade offs

Tuples are uniquely defined **by their contents**, while objects are unique **by themselves**, the main consequence of that is that two objects with the same contents will be interpreted as the same tuple, unless an internal identifier field is created to ensure uniqueness of objects.

Performance trade offs

- Navigating between tuples is a **linear time operation**[1], while it is a **constant time** operation for objects. Hashing techniques can be used to reduce this time, but they imply **extra memory cost**.
- Operations on all tuples (e.g. filtering, aggregating and bulk updates) are **cheap** on tuples but may be **expensive** on objects.

3.4 Keys

3.4.1 Primitive x Composite

Keys are used for uniquely identifying elements and for retrieving them from the database.

Expressiveness trade offs

- Migrating from composite to primitive keys has the disadvantage that the uniqueness constraint on the multiple parts of the key will not be enforced by the DMS, it thus **needs to be enforced by the application**.

Performance trade offs

- Furthermore, the DMS **may not be able** to retrieve an element my multiple keys, this may increase the cost of element retrieval if it **needs to be implemented by the application**.

3.4.2 Non-Ordered x Ordered

This feature is mainly used to **speed up retrieval operations**, it however slows down insert and deletion operations.

Expressiveness trade offs

None

Performance trade offs

- This has a great impact on the query patterns that are natively supported by database systems and on their computational cost. If keys are ordered, one can retrieve a range of keys in **linear time**, while for an unordered set, the worst case of this operation has a **quadratic time complexity**.

3.5 Values

3.5.1 Single x Multiple

The main advantage of supporting single columns is that the schema of the database normally doesn't need to be defined in advance.

Expressiveness trade offs

- Migrating from single to multiple DMSs is trivial. Conversely, multiple columns can easily be simulated as metadata attached to documents, however, in most databases, the schema associated to multiple DMSs **may be lost in this translation**.
- Single DMSs normally **do not support** queries other than given a key returning or changing the value associated to it.

Performance trade offs

- The ability of storing multiple columns for a single value is a mere convenience offered by the DMSs. Migrating from multiple to single valued databases will certainly **increase the complexity** of the application code decoding the values stored in the database or encoding sets of objects into a single value.

3.5.2 Primitive x Documents

Documents are complex structured elements, ranging from a binary blob annotated with metadata to complex trees of elements. They are mainly used to de-normalize the data model, and then increasing retrieval speed of large amounts of data.

Expressiveness trade offs

- As for single x multiple values, documents can be represented as primitive values, but this will **increase the complexity** of the application code.

Performance trade offs

- Going from primitive to document based may be trivial, but will **certainly sub utilise the resources** of the DMS.

3.5.3 Single x Multivalued

Multiple values are usually a syntactic sugar provided by DMSs as they can usually be implemented at application side without too much increase in complexity.

Expressiveness trade offs

- The ability of storing multiple values can be seen as a mere convenience offered by the DMSs, but migrating from multiple to single valued databases will certainly **increase the complexity** of the application code decoding the values stored in the database or encoding sets of objects into a single value.

Performance trade offs

None

3.6 Links: Relationships x Foreign-Keys x No Relationships

To put it simply, relationships are links between objects whereas foreign keys represent links between tuples. In order to understand the trade-offs between both cases, refer to the subsection on objects and tuples.

Expressiveness trade offs

- Migrating from a DMS without relationships to one with relationships or foreign keys is easy, but the other way **is not**. In this case one needs to **simulate relationships on the application code**.

Performance trade offs

- The trade-off here will be between finding elements in linear time or updating and deleting elements in linear time.
  - On tuple based DMSs, navigating will take linear time, which may **become a bottleneck on the long run**.
  - On Document based DMSs, one can try to de-normalize the data model, but that **will increase the complexity of the code** associated to update and delete operations.

3.7 Aggregation: Column Families X Regions or Tuple Spaces

Column families and regions are best suited to completely different use cases. The former target queries that aggregate information on a subset of columns while the later target queries that collect information on subsets of elements. These approaches imply completely different partitioning strategies: the former allocates **columns** to different nodes while the later allocates **elements** to different nodes.

Expressiveness trade offs

None

Performance trade offs

- Running queries that are not appropriate to a particular kind of database may result in **huge bottlenecks** since the DMS needs to look up data in potentially all nodes in order to answer to queries.

4 APPLICATION TO THE MIC MULTI-CLOUD CASE STUDY

In this section we use our concepts and trade off analysis to compare six DMS from three different platform as a service providers on the MiC case study presented in Section 2.2.

4.1 Method

The main objective of such comparison is to understand the trade-offs involved in representing data in one of the target DMSs. Notice that in this case study we limited ourselves to comparing the DMSs provided by large-scale generic providers addressing any kind of application as they would be the first clouds to consider for application developers. As explained in Section 2.2, the motivation for this analysis to developers, is to understand the hidden costs involved into storing parts of the application on different cloud providers.

The targeted use cases are mainly the choice of initial DMS to store data, and besides to eventually carry a migration from one data store to another. In order to do that, our method consists in classifying the data we need to represent and the data supported by the target DMSs. The comparison of the needed and available support will hopefully guide the developer into writing the code of the application and to avoid pitfalls involving eventual incompatibilities between DMSs.

			Underlying Data Abstraction	*Keys*	*Values*	*Links*	*Aggregation*
Application Data							
	User Profile		Objects	Non-ordered	Multiple, Primitive, Single	Yes	Regions
	UserSimilarity		Tuple	Non-ordered	-	Yes	Regions
	Picture		Tuple	Non-ordered	Single, Primitive, Single	No	Regions
PaaS under consideration		**DMS**					
	Azure						
		Table Storage	Tuple	Primitive, Ordered	Multiple, Primitive, Single	-	Regions
		Blob Storage	Tuple	Primitive	Single, Primitive, Single	-	Regions
	Heroku						
		Postgres	Tuple	Primitive	Multiple, Primitive, Single	Foreign Keys	-
		Cloudant	Tuple	Primitive	Single, Document, Single	-	-
	Google App Engine						
		Datastore	Tuple	Primitive	Multiple, Primitive, Single	-	Regions
		Blob Storage	Tuple	Primitive	Single, Primitive, Single	-	-

Figure 6: Case study: comparing storage options for the MiC application.

Figure 6 presents the result of this comparison for both the application data model and the target cloud DMSs (detailed respectively in Sections 4.2 and 4.3.

4.2 Analysing the Application Data Model

We divide the data model presented in Figure 3 into three parts, presented on the top part of Figure 6. For the sake of simplicity we didn't include all parts of the data model in this comparison. The three parts are: (i) the `UserProfile`, concerning the user profiles and related pieces of information; (ii) the `UserSimilarity`, concerning the data store that stores the users that are similar to a given user profile; and (iii) the `Picture`, concerning the picture linked to each user profile.

We classify each part of the data model using the concepts presented in Section 3.1. User profiles are object oriented information, because two profiles may refer to the same pieces of information and still represent different users. `UserSimilarity` and Picture are different, because they should refer or belong to a specific user profile. In all cases, no key ordering is necessary in the MiC application. However, the ability to group pieces information by "region" is important in all cases: user profiles and related data a very geographically specific, and should all be located in the same geographic region to speed up computation. When it comes to values and links, user profiles should contain primitive values to represent the pieces of information that compose a user profile (e.g. name, gender, location, data of birth etc.). Pictures only contain a binary blob representing the picture, and the user similarity only contain references to similar user profiles.

4.3 Analysing Target Cloud DMSs

At the bottom part of Figure 6, we present the platform as a service providers presented in Figure 2 and their respective DMSs. We then classify the DMSs according to the same criteria used to classify the parts of the data model on the top of the table. All DMSs represent tuples, i.e. they do not support objects directly. They also enforce the use of primitive keys and most of them (except for Postgres, a relational database) do not support links or foreign-keys between tuples. Only Azure DMSs and the Google Data store support grouping elements that need to be accessed in the proximity of a geographic area. The main difference between the supported DMSs is in the supported expressiveness of columns/values: Azure Table Storage, Postgres and the Google Datastore all support multiple primitive single valued values. The Azure Blob storage, Heroku cloudant and Google Blob storage all support one single valued value, which may be a document in Cloudant, or a single value in the other ones.

4.4 Analysing Cloud Migration Scenarios

Figure 7 overviews the trade-offs that need to be faced by a developer intending to develop the MiC application and host it on the three target platform as a service providers. Notice that different DMSs have different trade-offs that need to be taken into account during application development and future maintenance and eventual migration. Let us show how this table may be useful in two eventual migration scenarios.

Based on the analyses provided in Figure 6 and Figure 7, in the next section we will analyse to specific hypothetical deployment and migration

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Trade offs:	**Azure**		**Heroku**		**Google App Engine**	
Data x DMS	**Table Storage**	**Blob Storage**	**Postgres**	**Cloudant**	**Datastore**	**Blob storage**
User Profile	Objects x Tuples No links	Objects x Tuples Multiple = Single value No links	Objects x Tuples FK for links No regions	Objects x tuples Multiple => Single value No regions No links	Objects x Tuples No links	Objects x Tuples, Multiple x Single value, No links, No regions
UserSimilarity	No links	No links	No regions	No regions	No links	No links, No regions
Picture			No regions	No regions		No regions

Figure 7: Trade-offs between DMSs.

scenarios.

4.4.1 Migration Scenario 1

Let us suppose that the developer chooses the Azure blob storage to store the user profile pictures since there are no significant differences between the required and provided expressiveness. If later the company decides to migrate to Heroku Cloudant or Google App Engine Blob Storage, the developer needs to notice that these DMSs do not support the aggregation of tuples by geographic region/access frequency. Both databases provide a generic algorithm that distributes data and then balances query answering resources. As described in Section 3.7 this may generate a performance bottleneck if data is not properly distributed. Developers should be therefore aware of this potential limitation.

4.4.2 Migration Scenario 2

In this scenario, let us suppose that the developer decided to deploy the `UserSimilarity` part of the data model in an instance of Azure Table Storage and that in the future, she or he decides to deploy part of this information in Heroku Postgres (e.g. as part of a spin off social network).

When initially deploying data on Azure, the developer needs to handle the fact that links between elements are not supported at this database. This therefore needs to be implemented in the application code (cf. Section 3.6). Since the target DMSs supports links, this does not need to be supported by the application any more. However, special care needs to be taken during the data migration with the application provided implementation of links. This should be done in order to avoid data loss during the migration or loss of functionality when part of the old application code will be fulfilled by the DMS itself.

5 RELATED WORK

The main problem addressed by this paper is the one of understanding the trade-offs between different cloud DMSs, in order to optimize the deployment of application data in multiple clouds. Past work has tried to address this problem but in different ways. We classify these works into two categories: (i) the ones that try to hide this complexity from the developer, (ii) the ones that allow the developer to work on surpassing such complexity.

We consider that approaches in the first category are not best suited to developers that need to extract the most from cloud data stores, since any black box that hides the real complexity of the DMSs is going to be efficient only in a restricted set of situations. The present work falls in the second category, but differently from other works, that try to provide tools under which the developer can himself try to bridge the semantic gap between different tools, we show explicitly the gap and the involved gaps to the developer.

In the **first category** we would put the systems that try to automatically bridge the gap between different database categories. This group starts out by the tools that facilitate the use of relational data stores by object oriented applications (DB-UML Database Modeling Tool) (Hibernate: Relational Persistence for Java and .NET) (DeMichiel, 2009).

In the non-relational word some tools try to do the same. A first set of tools (Acid House) (Kundera) (PlayORM) (DataNucleus Access Platform) (Hibernate Object/Grid Mapper) (Morphia) reuses the concepts defined by JPA, which is a very popular system of annotations over Java code (i.e. an object oriented model of data), to translate an object oriented model represented by a set of Java classes into a non-relational databases. Other tools do the same thing for relational models (Toad for Cloud) (eobjects.org MetaModel). They provide a relational SQL-based interface to non-relational NoSQL databases, allowing existing

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

relational modelling approaches to be reused to model non-relational databases. Finally, Spring Data (Spring Data), provides different interfaces for different NoSQL databases.

This comes with the drawback of the inherent loss of information in the translation process or the loss of *"object-orientedness"* in the object oriented model in some corner cases.

Other approaches do not try to hide the non-relational concepts behind relational ones, but instead, propose unified abstract modelling languages. These languages try to represent the common concepts that are present in many different non-relational stores in a uniform way. Two examples of such languages are FQL (Federated Unfied Query Language, FunQL) and UnQL (UnQL Specification). The former received this name because it was created to support "federations" of databases. A federation of databases is a set of data stores, possibly storing data under different paradigms (relational or non-relational). The FQL language is then based on SQL but is able to query non-relational data bases. Its main drawback is that it supports only data retrieval, i.e. it provides no Data Definition Language. A similar approach for dealing with federated databases can be found in (JBoss Teiid). The UnQL language stands for Unstructured Query Language. It follows a similar approach, but is limited to unstructured (and therefore non-relational databases). It is targeted only to data stores containing JSON documents.

On the **second category** we will find tools such as such as Pentaho (Pentaho) and Yahoo! Pipes (Yahoo Pipes), which are Data Integration tools. They offer visual editors that allow one to describe how data coming from different sources, following different schemas and data types can be mapped into different data types and then fed to other systems. The semantic gap between different DMSs needs to be understood and filled by the developer.

In scientific literature, some papers also discuss the differences between the offerings of cloud providers and their supported DMSs. A good example of this kind of work is (Rimal, Sch. of Bus. IT, Choi, & Lumb, 2009). In this work, the different cloud providers are described along with their features and storage solutions. However the referred paper focuses on runtime characteristics (security, load balancing, fault tolerance etc.) and not on the impact of the design time storage choices to the cloud application.

More recent works such as (Cattell, 2010), (Hecht & Jablonski, 2011) and (Moniruzzaman & Hossain, 2013) go into the concepts behind different DMSs, their runtime properties, preferred use cases and supported queries. However these works are usually restricted to some specific kinds of cloud storage (usually variations of key-valued stores), and compare tools mostly based on runtime characteristics instead of design time ones.

6 CONCLUSION

The multiplication of cloud providers has both positive and negative impacts on industrial applications. On the one hand, the increasing availability and multiplicity of cloud providers allows for the existence of clever applications profiting from the best of different providers. On the other hand, the fragmentation of the market makes developing such applications much harder. In particular, maintaining them (fixing bugs and eventually moving to other clouds) becomes much harder than for regular non-cloud applications.

In this paper we investigated this problem in the point of view of the developer that needs to design data structures that will be potentially deployed on different clouds and on different data management systems (DMS). More specifically, we investigated the main concepts behind the different DMS and the semantic gap between different databases.

The present work is a first step on the direction of providing some automated support to developers and is going to be extended as part of the FP7 projects MODAClouds and JUNIPER. As future works, we are currently working on providing automated tools for analysing data models and proposing better data structures, and verifying if they respect a given set of real-time constraints on a multi-cloud setting. The extension of the model presented in this paper with other concerns unrelated to data structures (i.e. support to transactions, programming language integration etc.) is also under consideration.

ACKNOWLEDGEMENTS

The research reported in this article is partially supported by the European Commission grant no. FP7-ICT-2011-8- 318484 (MODAClouds).

The research reported in this article is partially supported by the European Commission grant no. FP7-ICT-2011-8- 318763 (JUNIPER).

REFERENCES

Acid house. (n.d.). Retrieved november 8, 2013, from https://github.com/eiichiro/acidhouse

Cattell, R. (2010). Scalable SQL and NoSQL data stores. *ACM SIGMOD Record* , 12-27.

DataNucleus Access Platform. (n.d.). Retrieved November 8, 2013, from http:// www.datanucleus.org/

DB-UML Database Modeling Tool. (n.d.). Retrieved November 8, 2013, from http://argouml-db.tigris.org/

DeMichiel, L. (2009). *JSR 131:Java Persistence API, Version 2.0.* Sun Microsystems.

eobjects.org MetaModel. (n.d.). Retrieved November 8, 2013, from http://metamodel.eobjects.org/ index.html

F., G., D., L., M., S. Y., D., A., & E., D. N. (2013). An Approach for the Development of Portable Applications on PaaS Clouds. *Proceedings of the 3rd International Conference on Cloud Computing and Service Science (CLOSER 2013)*, (pp. 591-601).

Federated Unfied Query Language, FunQL. (n.d.). Retrieved November 8, 2013, from http://funql.org/

Han, J., Haihong, E., Le, G., & Du, J. (2011). Survey on NoSQL database . *Pervasive Computing and Applications (ICPCA), 2011 6th International Conference on* , (pp. 363 - 366). Port Elizabeth .

Hecht, R., & Jablonski, S. (2011). NoSQL evaluation: A use case oriented survey . *Cloud and Service Computing (CSC), 2011 International Conference on* , (pp. 336-341). Hong Kong .

Hibernate Object/Grid Mapper. (n.d.). Retrieved November 8, 2013, from http://www.hibernate.org/ subprojects/ogm.html

Hibernate: Relational Persistence for Java and .NET. (n.d.). Retrieved November 8, 2013, from http://hibernate.org

JBoss Teiid. (n.d.). Retrieved November 8, 2013, from http://www.jboss.org/teiid/

Khajeh-Hosseini, A., Greenwood, D., & Sommerville, I. (2010). Cloud Migration: A Case Study of Migrating an Enterprise IT System to IaaS . *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on* , (pp. 450 - 457). Miami, FL .

Kundera. (n.d.). Retrieved November 8, 2013, from https://github.com/impetus-opensource/Kundera

Liu, T., Katsuno, Y., Sun, K., & Li, Y. (2011). Multi Cloud Management for unified cloud services across cloud sites . *IEEE International Conference on Cloud Computing and Intelligence Systems (CCIS)*, (pp. 164-169). Beijing.

Moniruzzaman, A. B., & Hossain, S. A. (2013). NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. *International Journal of Database Theory and Application*, 1-14.

Morphia. (n.d.). Retrieved November 8, 2013, from http://code.google.com/p/morphia/

Pentaho. (n.d.). Retrieved November 8, 2013, from http://www.pentaho.com/

PlayORM. (n.d.). Retrieved November 8, 2013, from https://github.com/deanhiller/playorm

Rackspace. (2013, February 13). *88 per cent of cloud users point to cost savings, according to Rackspace Survey.* Retrieved June 2013, from http:// blog.rackspace.co.uk/in-the-industry/88-per-cent-of-cloud-users-point-to-cost-savings-according-to-rackspace-survey/

Rimal, B., Sch. of Bus. IT, K. U., Choi, E., & Lumb, I. (2009). A Taxonomy and Survey of Cloud Computing Systems. *Fifth International Joint Conference on INC, IMS and IDC, 2009. NCM '09.*, (pp. 44-51). Seoul.

Singh, Y., Kandah, F., & Zhang, W. (2011). A secured cost-effective multi-cloud storage in cloud computing . *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, (pp. 619-624). Shanghai.

SOFTEAM; University of York. (2013). *D5.1 – Foundations for MDE of Big Data Oriented Real-Time Systems.*

Spring Data. (n.d.). Retrieved November 8, 2013, from http://www.springsource.org/spring-data

Toad for Cloud. (n.d.). Retrieved November 8, 2013, from http://toadforcloud.com/index.jspa

UnQL Specification. (n.d.). Retrieved November 8, 2013, from http://www.unqlspec.org

Yahoo Pipes. (n.d.). Retrieved November 8, 2013, from http://pipes.yahoo.com/pipes/