CDPort: A Framework of Data Portability in Cloud Platforms

Ebtesam Alomari
King Abdulaziz University
Jeddah, Saudi Arabia
ealomari0006@stu.kau.edu.sa

Ahmed Barnawi
King Abdulaziz University
Jeddah, Saudi Arabia
ambarnawi@kau.edu.sa

Sherif Sakr
King Saud bin Abdulaziz University for Health Sciences
Riyadh, Saudi Arabia
sakrs@ksau-hs.edu.sa

ABSTRACT

One of the main advantages of the cloud computing paradigm is that it simplifies the time-consuming processes of hardware provisioning, hardware procurement and software deployment. Currently, we are witnessing a proliferation in the number of cloud-hosted applications. However, one of the important challenges of the cloud computing paradigm that may hurt the growth of this technology is the interoperability and portability between cloud platforms. The developers and cloud users may lock-in to the first cloud they choose, or face difficulties when they have to move their data or software from one cloud platform to another. In this paper, we focus on the challenge of data portability between different cloud-based data storage services. In particular, we propose a common data model and a standardized API for the new generation of cloud-based NoSQL databases. The initial implementation of our framework covers three of the most popular NoSQL systems, namely, Google Datastore, Amazon SimpleDB and MongoDB. However, our framework is designed in a flexible way that it can be easily extended to support other NoSQL systems. Furthermore, our framework is equipped with tools that support the conversion, transformation and exchange of the data which is stored on the supported NoSQL databases of the framework. Finally, we describe the design and the proof-of-concept implementation of our framework using a case study.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Design, Experimentation, Performance

Keywords

Cloud, Portability, Interoperability, NoSQL Database

iiWAS2014 4-6 December, 2014, Hanoi, Vietnam

1. INTRODUCTION

Cloud computing technology represents a new paradigm for hosting software applications. This paradigm simplifies the time-consuming processes of hardware provisioning, hardware procurement and software deployment. Thus, it revolutionized the way computational resources and services are commercialized and delivered to customers. Nowadays, cloud computing is growing significantly. Cloud providers increasingly provide new services and new features to their consumers with efficient and cost-effective solutions for their problems. Consequently, the cloud has become an attractive platform for the software developers and enterprises to host their applications and systems. However, the services offered by different cloud providers are mostly incompatible with each other and do not support any standardized models or interfaces. Therefore, one of the major challenges for facilitating cloud adoption is that of the cloud interoperability and portability.

On the one hand, interoperability is defined as the ability of different heterogeneous systems to work and interact together. For clouds, interoperability means the ability for multiple cloud providers to work together and understand application formats, Service Level Agreement (SLA) templates, authentication and authorization token formats and attribute data of each other's [4]. On the other hand, cloud portability is defined as the ability of easily moving data and application components and reusing them regardless of the selected cloud provider, operating system, storage format or APIs [14]. Hence, an ideal cloud computing environment should enable customers to switch and choose among cloud providers whenever they needed without any risk or obstacles [6].

In practice, there are several reasons that can make the cloud user move from one cloud provider to another. For example, high rate of service failure or downtime, changes in the business or the technology strategy, contract termination, legal issues, or finding better alternatives with low cost [12]. The common strategy to address Cloud interoperability is the use of open standards. This consensus will make the integration and migration of services between heterogeneous cloud systems easier [12]. However, most of the existing cloud computing solutions have not been built with consideration of interoperability. In particular, they try to lock their customers into a single cloud infrastructure, platform or service in order to limit the customer ability to move his data or software to another cloud provider. This problem is called the vendor lock-in, which recognized by the Euro-

pean Network and Information Security Agency (ENISA)[1] and European Commission[2] as a high risk that entail cloud infrastructures [6].

In principle, cloud databases is currently considered as an attractive solution for software developers if they want to store the data of their applications in a scalable and highly available backend. These services are referred to as Database-as-a-Service (DBaaS). These cloud-based data storage services can be classified into two main categories: services that support traditional relational databases (RDB) (e.g., Amazon RDS, Google SQL, Microsoft Azure), and key/value pair data storage services (e.g., Amazon Simple DB, Google Data Store), which are also known as NoSQL Databases [11]. In principal, RDB systems use structured query language (SQL) as a standardized interface to access data in a relational database. On the other side, the NoSQL databases remain unstandardized, so there is no unified data access approach [10]. Thus, each cloud provider has a different way to manage and access the database, which make the data portability is a challenging task to achieve between these systems [8].

In this paper, we focus on tackling the challenges of data portability between different cloud-based NoSQL data storage services. In particular, we present *CDPort* (*C*loud *D*ata *P*ortability), as a framework that supports data portability and interoperability in cloud platforms. The main contribution of this paper can be summarized as follows:

- We propose a common application programming interface (API) and a data model for the NoSQL databases that enables software developers to easily change their backend cloud-based NoSQL data storage service with zero need of changing the software code.
- We describe the design and implementation of our software tool that facilitates the data transformation between different NoSQL cloud databases. The tool is designed to minimize the data migration effort between the different cloud-based NoSQL databases especially for transforming data between databases that have different data storage models.
- We present the practicality and ease-of-use for a proof-of-concept implementation of our framework using an evaluation case study.

The remainder of this paper is organized as follows. We discuss the related work in Section 2. Section 3 describes the portability and interoperability challenges between different NoSQL database systems. Section 4 presents the design and implementation details of our *CDPort* framework. We present an evaluation case study of our framework in Section 5 before we finally conclude the paper in Section 6.

2. RELATED WORK

Several surveys[3] have reported that the cloud interoperability is considered as the second big concerns that prevent customers from adopting cloud computing (Figure 1). Therefore, there have been a set of open standards[4] that have been proposed to achieve user application and cloud provider interoperability such as Open Virtualization Format (OVF)[5], Cloud Infrastructure Management Interface (CIMI)[6], Open Cloud Computing Interface (OCCI)[7] and the Cloud Data Management Interface (CDMI)[8]. However, these standards are not enough to address the problem while the benefit from standardization is based on service models that the service provider uses and it is left to the service provider whether to follow the proposed standards. In addition, most of the proposed standards support the infrastructure as a service (IaaS) layers model more than the other layers such as platform as a service (PaaS), software as a service (SaaS) and database as a service (DBaaS) [5]. In addition, most of the existing Cloud solutions have not been built with interoperability in mind and, thus, they do not comply with any established Cloud standards, which lead to problems such as locked-in. Therefore, there has been a clear gap for finding new approaches that support a full migration process.

Figure 1: Cloud Computing Concerns

Several projects and techniques have been proposed to achieve interoperability between different cloud systems. The *mOSAIC* project [9] focuses on the lack of standard interface for resource virtualization and presents a tool for developing an application that can work on multi-cloud environments. In particular, they separated the application into application-logic layer and cloud layer, so application developer does not need to care about the infrastructure. In addition, they proposed open source API that enables the development and deployment of portable applications that use multiple clouds. They also offered multi-agent brokering system that will search for services matching the applications' request. However, the drawback of their approach is that the mOSAIC's API was designed to be event-driven and this style of programming complex for the simple programmer.

The *MODACLOUDS* project [1] has been designed to handle the problem of designing software systems for multi-Cloud environments. They are offering a framework and IDE to support developers and operators to use multi cloud and migrate their applications between clouds. Furthermore, it semi-automatically transforms from design into various platform compatible code to support the ability of moving between different providers. They are also developing an

[1] http://www.enisa.europa.eu/
[2] http://ec.europa.eu/index_en.htm
[3] http://www.northbridge.com/2011-cloud-computing-survey
[4] http://cloud-standards.org/
[5] http://www.dmtf.org/standards/ovf
[6] http://dmtf.org/sites/default/files/standards/documents/DSP0263_1.0.0.pdf
[7] http://occi-wg.org/
[8] http://www.snia.org/cdmi

integration framework between design tools and run-time. However, it is still a work in progress, and there are as yet no consideration to address the data portability problem.

The *Cloud4SOA* project [7] focuses on addressing and tackling semantic interoperability challenges at the PaaS layer to handle the lock-in problem. In addition, it focuses on eliminating the difficulties that are face by the developers when comparing competitive offerings. Therefore, they proposed an open semantic interoperability framework from which both the application developer and Cloud PaaS provider can benefit. In addition, they designed and developed a homogenized API, which covers functionalities offered by the majority of PaaS providers. In addition, there is an adapter for every PaaS provider, to enable the bridging of heterogeneous PaaS APIs, and offer the functionality of deployment and migration. Further, Cloud4SOA supports the ability to manage, monitor, migrate and search for the best PaaS offering that matches user's requirements. However, they do not handle the challenge of having a different API for each PaaS to store the data of the application.

Shirazi et al. [12] presented a design pattern to transfer data from Hbase as a column family database to Neo4j as a graph database and vice versa. However, they did not address how the user can move data between clouds that have different APIs. Furthermore, Chang et al. [2] proposed a tool for supporting the data migration from relational databases to the Google App Engine (GAE) Datastore and it supports blob data migration. In addition, they made an improvement on the *AppCfg*[9] tool that is provided by Google for uploading and downloading either data or application. They proposed a GUI for AppCfg for reading inputs from users about the information of RDBMS, GAE and blob to enable the migration. Additionally, the tool is responsible for preparing the configuration files needed by *AppCfg* and this will help in reducing the user's required effort. The proposed system architecture is separated into two parts: 1) Server side which represents an application that runs on GAE. 2) Client side application that offers the GUI for the end user. Although, their approach reduces the time and the error of migration, it focuses only on the Google Datastore and they do not solve the problem of the database portability between different cloud providers.

Bastião et al. [13] provide Service Delivery Cloud Platform (SDCP), which a cloud middleware infrastructure that uses resources from multiple cloud providers to give a group of services. They offer the SDCP-SDK to develop SDCP-based applications. In addition, they provide a common API for delivering three cloud services, which are: storage, database and notification service. For the storage layer, they offered an API that has different features which were implemented in different cloud providers. In addition, they offer encryption/decryption layer on the client side. For the database service layer, they focus on columnar data so that they create an abstraction and propose a common API for SimpleDB and Azure Table. In the implementation of the API, they used the same JPA (Java Persistence API) methods. Also, they add other methods that are specific to the Cloud databases where the JPA is usually used by Java developers to abstract the access to the databases. However, they are only concerned about columnar data.

[9]`https://developers.google.com/appengine/docs/appcfg`

3. DATA PORTABILITY CHALLENGES OF NOSQL DATABASES

Today, there are several PaaS cloud providers in the market such as Amazon web service, Google App Engine, Microsoft Azure, and many more. They provide an elastically scalable platform for developing and deploying software applications. In addition, they offer database as a service solutions where customers pay for what they use. These cloud-based database solutions can be divided into relational-based and NoSQL (*N*ot *O*nly *S*QL) systems [11] which support scalable, highly available and flexible schemas data stores. Hence, it is increasingly considered as a viable alternative to relational databases and its use in the cloud become increasingly popular. In principle, NoSQL systems can be classified based on their supported data model into three main categories [11]:

- *Key-value stores*: These systems use the simplest data model which is a collection of objects where each object has a unique key and a a set of attribute/value pairs. The main examples of this category are Amazon DynamoDB[10], Azure table[11] and Google Datastore[12].
- *Extensible record stores*: They provide variable-width tables (Column Families) that can be partitioned vertically and horizontally across multiple servers. Examples include Apache HBase[13], Amazon SimpleDB[14] and Cassandra[15].
- *Document stores*: The data model of these systems consists of objects with a variable number of attributes with a possibility of having nested objects. Examples include MongoDB[16] and CouchDB[17].

In practice, each cloud-based NoSQL service has different data models for data storage and different APIs for data access and manipulation. The absence of a unified way to store and access the data leads to the fact that the data portability between theses different platforms become a challenging task. Therefore, software developers may be locked-in to the first selected vendor or they may need to re-engineer the application to fit the new choice. Hence, the migration between platforms is becoming a costly and time consuming process. In the next section, we describe the architecture and the design details of our proposed framework, *CDPort*, that handles the data portability challenges between different cloud-based NoSQL systems.

4. CLOUD DATA PORTABILITY FRAMEWORK

4.1 Architecture

Figure 2 illustrates the architecture of our cloud data portability framework, *CDPort*. The main goal of the proposed framework is improving portability and helping developers to easily migrate the data of their applications between

[10]`https://aws.amazon.com/dynamodb/`
[11]`http://msdn.microsoft.com/en-us/library/jj553018.aspx`
[12]`https://developers.google.com/datastore/`
[13]`https://hbase.apache.org/`
[14]`https://aws.amazon.com/simpledb/`
[15]`http://cassandra.apache.org/`
[16]`http://www.mongodb.org/`
[17]`http://couchdb.apache.org/`

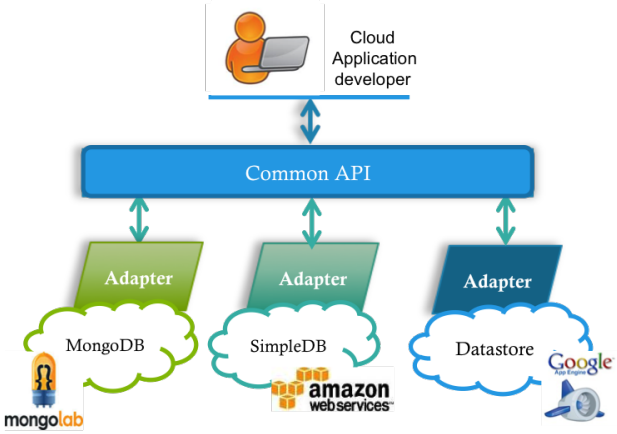

Figure 2: The Architecture of CDPort Framework

various cloud databases and various providers, without the need for re-engineering their applications to adapt them to the data model and the interface of the new cloud storage system. In principle, the framework offers a platform-independent database abstraction layer and provides a common data model that enables accessing different cloud databases through a common API. With these facilities, the software developers will not need to worry about changing the underlying platform. They can utilize the CDPort's API to connect and manage data at any of the supported cloud databases by the framework. The initial implementation of the CDPort framework supports three of the most common NoSQL cloud storage systems that cover the main different data models, namely, Google Cloud Datastore (as a representative of key-value stores), Amazon SimpleDB (as a representative of extensible record data stores) and MongoDB[18] (as a representative of document stores). Furthermore, the framework has been designed in a flexible way which makes it easy to be extended to support other databases following the supported data models.

For each of the supported cloud NoSQL databases in our framework, we have implemented an adapter that hides the variations and the specific implementation details for each of them. In particular, the adapter is responsible for receiving the requests from the user application on the framework's standard API and transparently converts these requests to the supported model and interface by the underlying database. Therefore, while each of the supported database in our framework has its own data model and API, the users of our framework will not need to know the details of these API operations or understand how they work. Furthermore, the users will not need to deal with different data models as the adapter will make the translation between them according to the standard data model supported by our framework. Hence, extending the framework to support new NoSQL system would just require the creation of a new adapter for this new system.

4.2 Data Model

As we have discussed in Section 3, NoSQL systems are classified based on their supported data model into three main categories: key-value stores, extensible record stores and document stores. These models are different in the way they store and manage data. In this section, we present these data models and discuss the differences between them. In addition, we present our proposed CDPort's data model as a unified model to manage the data portability problem.

4.2.1 *Google Cloud Datastore*

Google Datastore is one of the main representative systems of key/value stores. It supports storing data objects known as *entities* that do not need to conform to the same structure. In particular, each entity has a *kind* attribute which is used for categorization. In addition, each entity has a *key* attribute which is composed of the identifier (name or auto generated number) and the kind name. Furthermore, each entity can have one or more property. The values of these properties can be of different data type such as integer, float, date, string and Boolean. Google Datastore provides an SQL-like query language called *GQL*[19] for retrieving entities from the data store. GQL supports simple *SELECT* statement that enables defining how to order and group the results. In the Java API, the system offers the `'runQuery'` method, which receives the `queryRequest` and returns the `queryResponse`. Furthermore, Google Datastore provides different functions to create, delete, or modify entities while the lookup functionality supports retrieving entities by their key values.

4.2.2 *Amazon SimpleDB*

Amazon SimpleDB is a representative NoSQL system of extensible record stores (column family). In SimpleDB, the data is stored in set of *domains* where each domain consists of a list of *items*, each item has a group of *attributes* and the *item name* is considered as the key attribute of the item. In SimpleDB, all values of the attributes are stored as strings. In general, the SimpleDB data storage model share some similarities with the relational database where the *item* concept is similar to the *row* concept in the relational model, the *attribute* concept is similar to the concept of a *column* storage and the *domain name* concept is used as a mechanism for classifying the items which is similar to the *table* concept in the relational model. SimpleDB supports a simple query language to retrieve the items that match the specified criteria from a single domain. The query expression is similar to the standard SQL *SELECT* statement. In addition, it offer a set of operations through its API interface such as the *Select* operation that enables executing the user queries, *PutAttributes* to create or update attributes in the item, *GetAttributes* that returns all of the attributes of the item and finally *DeleteAttributes* operation which is used for deleting items.

4.2.3 *MongoDB*

MongoDB is a document-oriented database where each database has one or more collections. The collection consists of a list of documents which are similar to JSON objects[20]. Each document is composed of a set of field/value pairs.

[18]For MongoDB, we have been using MongoLab which provides MongoDB-as-a-Service and is available on `https://mongolab.com/`

[19]`https://developers.google.com/datastore/docs/apis/gql/gql_reference`

[20]`http://www.json.org/`

	Google Datastore	Amazon SimpleDB	MongoDB
Data Model	Key-value store	Column family	Document type
Categorization	Kind name	Domain name	Collection name
Key	The identifier (name or auto id) + the kind	Item name	The field (name_id)
Data Types	integer, float, date, string, Boolean	all values stored as string	string, date, array, long
Query	GQL (SQL-like)	Custom SQL	Custom API
API Operations	commit lookup runQuery	PutAttributes GetAttributes DeleteAttributes Select	insert update remove find

Table 1: Comparison between NoSQL Data Models

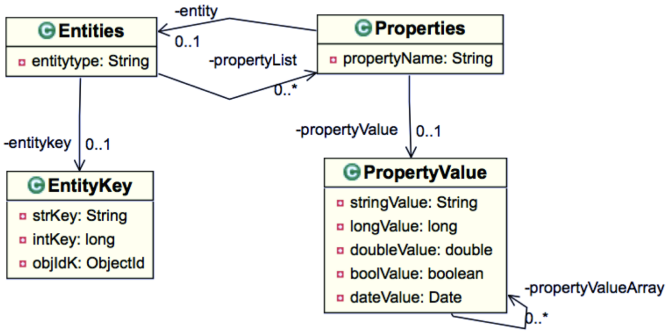

Figure 3: CDPort's Data Model

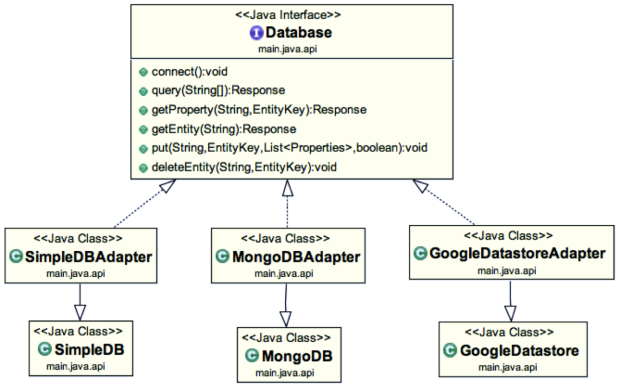

Figure 4: CDPort's API Class Diagram

MongoDB supports different data types, such as string, date, array and long. Each document is identified by the field *'name _id'* that must be unique in the collection. In addition, MongoDB supports querying through a single collection and enables retrieving the document that matches the criteria. In MongoDB API, the `find` operation is used to return all fields of all documents that match the query. The `insert` method enables adding new documents into a collection while both the `update` and `save` methods with the `upsert` flag support adding a new document if it does not exist. The `remove` operation is used for deleting one or more documents. Table 1 summarizes the comparison between the different NoSQL data models.

4.2.4 CDPort Data Model

Figure 3 illustrates our proposed CDPort's data model that attempts to unify the concepts for interfacing with the different NoSQL models and provides a standardized interface for them. In particular, in our data model, data objects are stored as entities where each entity stores the following information:

- *Entity type*: It is a mandatory categorization attribute for the entity.
- *Key*: It represents an identification attribute for the entity which must be unique through the entity type. This key attribute can be of data type string, integer or object ID so that it can be compatible with the different NoSQL systems.
- *Property*: Which is a descriptive key-value attribute that store a string, long, float, date, Boolean or an array data value. Apparently, each entity can have multiple descriptive properties.

By using our proposed data model, the software developer will not need to deal with the details of the different data models of NoSQL systems as our framework can provide them with the facility of easily and transparently moving the backend of their applications between different NoSQL systems without the need to change the data management interface code in their software as we will show next.

4.3 API Implementation

The CDPort's API has been designed to hide the programmatic differences between the different NoSQL database systems and provides a unified way to interface with them. For example, it supports automatic configuration to the connection to the underlying database. In particular, each adapter offers a contract-centric interface that receives the needed data as parameters to configure the connection. Hence, users need only to use the `connect` method which will automatically configure the credential and configuration information. To illustrate, if the user decides to change the backend of its software application from Google Datastore to Amazon SimpleDB, he only needs to change the connection object in his code, i.e, he need to create an instance from the class of the new Database. In particular, in order to connect to a SimpleDB database, the user needs to create an instance from the `SimpleDBAdapter` class as follows:

```
Database myDB = new SimpleDBAdapter(accKey, secKey,
region);
myDB.connect();
```

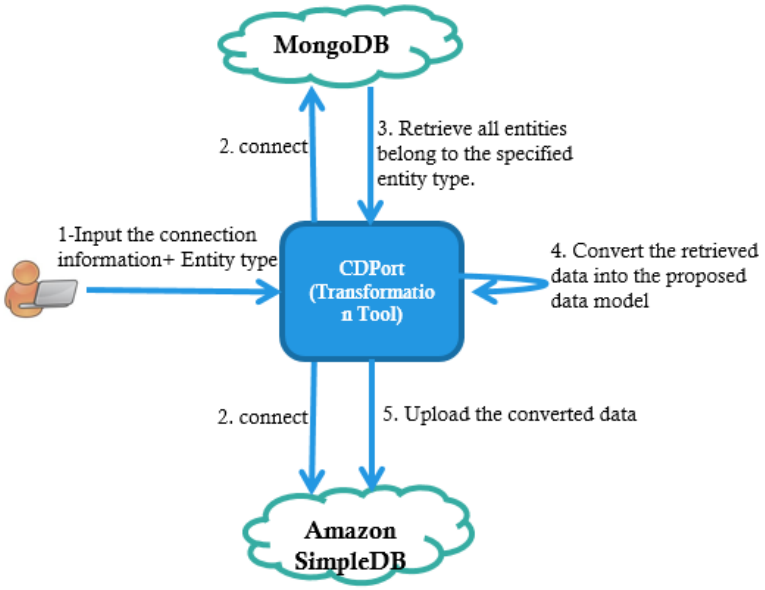

Figure 5: CDPort's Data Exchange Model

where `accKey` and `secKey` represents the credential information to connect the SimpleDB database and `region` represents the location of the database. When the user decides to change the connection of his backend data store to MongoDB, he will just need to change the creation of the instance of his database class to `MongoDBAdapter` instead of the `SimpleDBAdapter` class and sends the client URL information (`MongoClientURI`) as a parameter as follows:

```
Database myDB = new MongoDBAdapter(MongoClientURI);
myDB.connect();
```

Furthermore, the API offers the main CRUD (Create, Read, Update, Delete) operations that are needed to access and manage the data. In particular, the supported methods are:

- *Put*: This method is used for adding a new entity with its identification information and name. The method uses a Boolean parameter which indicates that the entity should be replaced if it exists when the value of the parameter is set to `true`.
- *getEntity*: This method is used for retrieving a specific or all entities that belong to a same specified entity type.
- *getProperty*: Which is used for retrieving a specific property or all properties for a specified entity.
- *deleteEntity*: It helps to remove an existing specific or all entities of a specific entity type.
- *Query*: This method receives the query statement in a form of a simple `SELECT` statement and returns the result back for the user.

In principle, the adapter of each supported database works as a middleware that translates between the operation of CDPort's common API and the operation of its own database. For instance, the `Query` method in CDPort's API receives the string query statement from the user in the form of a simple *SELECT* statement, the adapter class will be responsible for converting this statement to be executed by the supported operations of its own database. To illustrate, The adapter of the Google Datastore will use the `runQuery` method to execute the CDPort's query statement (`QS`) as follows:

```
GqlQuery.Builder query =
GqlQuery.newbuilder.setQueryString(QS);
RunQueryRequest request =
RunQueryRequest.newbuilder.setGqlQuery(query).build();
RunQueryResponse response= datastore.runQuery(request);
```

In case of the SimpleDB adapter, the select method will receive the string of query statement (`QS`) and just execute it as follows:

```
SelectRequest selectRequest = new SelectRequest(QS);
SelectResult result = sdb.select(selectRequest);
```

However, in the case of MongoDB adapter, the input query statement string will be parsed and translated into the `find` method which is used for retrieving data. In particular, the SELECT statement string will be parsed to identify the entity type and the filtering condition if it exists as follows:

```
DBCollection coll = db.getCollection(entityType));
DBCursor cursor=coll.find();
```

These examples show how all these details will be hidden from the CDPort's users who will benefit from easy data portability and portability by dealing with a unified data model and API interface.

4.4 Data Transformation and Exchange

The CDPort framework is equipped with a module which is responsible for data transformation and exchange between the the NoSQL storage system which follows any of the CDPort's supported data models. In particular, the user of the tool need to provide the connection information for the target target source and destination databases. In addition, the user needs to specify the entity types which represent the target of the migration. The CDPort framework uses this information to retrieve the data from the source database, convert it into the CDPort's data model and then migrate

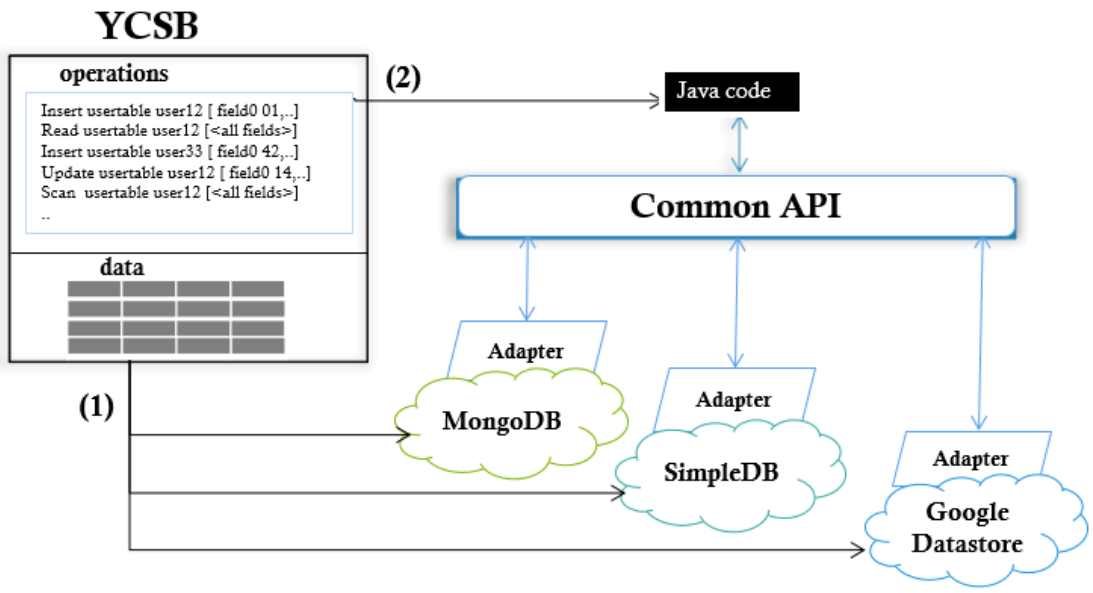

Figure 6: The Architecture of CDPort's Evaluation Case Study

it to the destination database. Currently, the implementation of the conversion tool is being considered to support the three supported NoSQL systems by our framework (Google Data store, SimpleDB and MongoDB). However, the tool is designed in a flexible way which is easy to be extended to support other data models and NoSQL systems. Figure 5 illustrates an example of the data migration process between MongoDB and SimpleDB databases as follows:

1. The user provides the credential information of the MongoDB database instance (client URL information) and the Amazon SimpleDB instance (Access Key and Secret Key).
2. The credential information is used to establish the connection with the source and target databases.
3. The user provides the information of the target entities for the migration process. The Entity Type in SimpleDB is represented by the *domain name* and in MongoDB by the *collection name*. All entities belong to the specified entity types will be retrieved from source database.
4. The retrieved data is converted into the CDPort's data model format and stored in XML format.
5. The retrieved data is converted to the data model of the target database after which it gets uploaded.

One of the current limitation of our migration framework is that in the case when migrating the data from any database to an Amazon SimpleDB database, all values get converted into a string (the only supported data type by SimpleDB) and all data typing information is lost. We will address this limitation as part of our future work.

5. EVALUATION

To evaluate the practicality and ease-of-use of our approach, we have implemented a proof-of-concept[21] case study for our framework by using the Yahoo! Cloud Serving Benchmark (YCSB)[22] [3]. In general, YCSB has been mainly designed to measure and compare the performance characteristics of different cloud database systems. In our context, we have used it to validate the usage of our proposed standardized data model and API. In principle, YCSB enables generating workload based on specified properties. The workload defines the data that will be loaded into the database during the loading phase, and the operations that will be executed, during the transaction phase, against the loaded data. Each record of the generated data consists of a group of fields and has a string primary key (e.g., "user123"). The fields of records are named "*field0*", "*field1*",...,"*fieldn*" while the value of each field is a random string of ASCII characters of length L. The benchmark offers a package of standard workloads named "*core workload*". However, the user can specify different properties of the workloads such as defining the length of the field, the number of records and fields that he wants them to be constructed by YCSB. The operations of the workload represents a mix of the standard CRUD operations where the user can control the frequency of the generated operations of each operation type.

To simulate the case of developing portable java application on different NoSQL systems, we have used the YCSB workload generator to generate a workload with simple data set and 10 random operations. The generated operations represent a balanced mix of read, insert, scan and update operations. We have uploaded the generated dataset into the three supported databases (SimpleDB, MongoDB and Google Datastore). Then, we represented the YCSB's generated workload using the methods of our CDPort's API (Figure 6). In particular, we started by executing the generated workload over the SimpleDB database using the `SimpleDBAdapter` of the framework. On the second step, we assumed that we need to change the backend of our workload to the MongoDB database, therefore, we have changed the used adapter to the `MongoDBAdapter` with the required credential

[21]The source code of our implementation is available on `https://github.com/CDPort`

[22]`https://github.com/brianfrankcooper/YCSB/`

and configuration information (the `connect` method). We have noticed that this was the only change we need to make in our code and the workload can be successfully executed over the MongoDB backend with no errors. Finally, we followed the same scenario with the Google Datastore backend and similarly we only needed to change the adapter class to create an instance of the new Database that we need to use as our backend without any need to make other changes in the rest of the application's code and also without the need to take any consideration regarding the difference between data models of the underlying database.

The outcome of this case study shows the practicality and ease-of-use of the CDPoart's standard data model and API. Therefore, we believe that the design of our framework represents an important step towards tackling the challenges of the data portability and interoperability between NoSQL systems. The framework also supports its user to mitigate the risks of platform lock-in in addition to transparently hiding all the variations between the different supported data models and APIs of the different NoSQL systems.

6. CONCLUSION

In this paper, we have presented an approach for tackling some of the challenges that arises as a result of the data portability issues between different cloud-based storage services. We have presented the design and the implementation of the *CDPort* framework with a unified API and data model that hides the variation between three main categories of NoSQL database systems. The current implementation of the system supports three NoSQL databases which are representatives of the three main NoSQL categories, namely, Amazon SimpleDB, MongoDB and Google Datastore. The design of the *CDPort* framework has been made in a flexible way that make it easy to extend to support other databases. In addition, we offer a tool for data transformation and exchange that facilitate migrating the data even between the different NoSQL data models. The evaluation case study for our proof-of-concept validates the proposed framework and shows its practicality and ease-of-use from the developer side. As a future work, we plan to extend the implementation of our framework to provide native adapters for others NoSQL systems in addition to supporting other NoSQL complex data models such as the graph model (e.g. Neo4J [23]).

7. REFERENCES

[1] Danilo Ardagna, Elisabetta Di Nitto, Giuliano Casale, Dana Petcu, Parastoo Mohagheghi, Sébastien Mosser, Peter Matthews, Anke Gericke, Cyril Ballagny, Francesco D'Andria, Cosmin-Septimiu Nechifor, and Craig Sheridan. MODACLOUDS: A Model-Driven Approach for the Design and Execution of Applications on Multiple Clouds . In *ICSE Workshop on Modeling in Software Engineering (MISE),*, pages 50 – 56, 2012.

[2] Yao-Chung Chang, Ruay-Shiung Chang, and Yudy Chen. Simplifying Data Migration from Relational Database Management System to Google App Engine Datastore. In *Advanced Technologies, Embedded and Multimedia for Human-centric Computing*, pages 887–895, 2013.

[3] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[4] Piyush Harsh, Florian Dudouet, Roberto G. Cascella, Yvon Jégou, and Christine Morin. Using open standards for interoperability issues, solutions, and challenges facing cloud computing. In *CNSM*, pages 435–440, 2012.

[5] Grace A. Lewis. Role of Standards in Cloud-Computing Interoperability. In *HICSS*, pages 1652–1661, 2013.

[6] Nikolaos Loutas, Eleni Kamateri, Filippo Bosi, and Konstantinos A. Tarabanis. Cloud Computing Interoperability: The State of Play. In *CloudCom*, pages 752–757, 2011.

[7] Dana Petcu. Portability and Interoperability between Clouds: Challenges and Case Study. In *ServiceWave*, pages 62–74, 2011.

[8] Dana Petcu, Georgiana Macariu, Silviu Panica, and Ciprian Craciun. Portable Cloud applications - From theory to practice. *Future Generation Comp. Syst.*, 29(6):1417–1430, 2013.

[9] Dana Petcu, Beniamino Di Martino, Salvatore Venticinque, Massimiliano Rak, Tamás Máhr, Gorka Esnal Lopez, Fabrice Brito, Roberto Cossu Miha Stopar, Svatopluk Łperka, and Vlado Stankovski. Experiences in building a mOSAIC of clouds. *Journal of Cloud Computing*, 2(12), 2013.

[10] Sherif Sakr. Cloud-Hosted Databases: Technologies, Challenges and Opportunities. *Cluster Computing*, 2013.

[11] Sherif Sakr, Anna Liu, Daniel M. Batista, and Mohammad Alomari. A Survey of Large Scale Data Management Approaches in Cloud Environments. *IEEE Communications Surveys and Tutorials*, 13(3):311–336, 2011.

[12] Mahdi Negahi Shirazi, Chin Kuan Ho, and Hossein Dolatabadi. Design Patterns to Enable Data Portability between Clouds' Databases. In *ICCSA Workshops*, pages 117–120, 2012.

[13] Luís A. Bastião Silva, Carlos Costa, and José Luís Oliveira. A common API for delivering services over multi-vendor cloud resources. *Journal of Systems and Software*, 86(9):2309–2317, 2013.

[14] Zhizhong Zhang, Chuan Wu, and David Wai-Lok Cheung. A survey on cloud interoperability: taxonomies, standards, and practice. *SIGMETRICS Performance Evaluation Review*, 40(4):13–22, 2013.

[23] `http://www.neo4j.org/`