QODM: A Query-Oriented Data Modeling Approach for NoSQL Databases

Xiang Li, Zhiyi Ma, Hongjie Chen
Institute of Software, School of EECS, Peking University, Beijing, China
Key Laboratory of High Confidence Software Technology, Ministry of Education of China
{lixiang13, mzy, chenhj} @sei.pku.edu.cn

Abstract—**Cloud applications on the cloud computing platform are different from traditional applications. In addition to SQL databases, cloud applications usually store their data in NoSQL databases. However, most NoSQL databases do not support join operations. Therefore, the traditional relational data modeling approach is not suitable for NoSQL databases, and it makes data query more complex in NoSQL databases. According to the characteristics of NoSQL databases, we proposed a query-oriented data modeling approach for NoSQL databases. This approach can generate data models and data schemas for NoSQL databases based on the stored data structure and data query requirements of an application. Furthermore, we defined a platform independent model of data schema for NoSQL databases. Moreover, we used ElasticInbox, a real cloud application, to evaluate the availability and platform independence of the QODM approach.**

Keywords-Query-Oriented, NoSQL, Data Modeling

I. INTRODUCTION

With the development of the Internet, the computer hardware platform is transforming from traditional closed computing platform to open internet-based computing platform. The software, which is based on the new platform, is called internetware [18]. As one of the most important representatives of the internet-based computing platform, cloud computing platforms [19] are becoming more and more popular. There are several kinds of cloud service platform model: infrastructure as a service (IaaS) platform, platform as a service (PaaS) platform, software as a service (SaaS) platform, etc. Many famous enterprises, such as Amazon and Google, provided their own cloud computing platforms for software developers based on these models. More and more developers choose to develop software on cloud computing platforms, but this brings some new challenges for software development.

Data is one of the most important parts of software, and the data in cloud is different from traditional data. The digital world is growing very fast and becomes more complex in the volume, variety, and velocity in nature. This means the database in cloud needs more capacity to store and manage vary large data, faster velocity to read and write, and more flexible schema to adapt to various data. However, traditional relational databases cannot solve these problems very well, and thus NoSQL databases [3,6,7] become the new choice for developers.

NoSQL databases are non-relational databases. Main advantages of NoSQL are following aspects [5]: 1) the ability to horizontally scale 'simple operation' throughput over many servers; 2) the ability to replicate and to distribute (partition) data over many servers; 3) a simple call level interface or protocol (in contrast to a SQL binding); 4) a weaker concurrency model than the ACID transactions of most relational (SQL) database systems; 5) efficient use of distributed indexes and RAM for data storage; 6) the ability to dynamically add new attributes to data records. NoSQL databases do not comply with ACID [8] principle, but they keep the CAP [14] theorem and BASE [15] principles.

Variety of NoSQL database systems have been developed and widely adopted in cloud computing, such as Apache Cassandra, Apache HBase, Amazon DynamoDB, Redis, Microsoft Azure Table, Google BigTable, MongoDB, etc. They can be divided into three categories: key-value stores, document stores and extensible record stores. Graph databases also are one kind of NoSQL database, and they are good at storing relations of entities. They are very different from other kinds of NoSQL databases, so we will not discuss them in this paper. We will consider the Graph database in the future.

NoSQL databases' data models are simpler than traditional relational databases. They do not have strict constraint on data structure and are used to store semi-structured data. This means only part of data structure is predefined. Most NoSQL databases do not support join operations, while the application's data that stored in them often contain association relationships. How to design an application's data model, which reserves data association relationships, satisfies the database's data model and makes data query simpler, is one of the cruxes of using NoSQL databases effectively. Some persons proposed a few of NoSQL data modeling technologies [2,4,17]. However, these data modeling technologies only considered what data model is suitable for NoSQL databases and did not consider how to design the data model of an application for NoSQL databases.

Based on the characteristics of NoSQL databases, this paper proposes a query-oriented data modeling (QODM) approach to design data model and data schema of an application for NoSQL databases. The data model defines data entities, which are needed to store, and their relationships for NoSQL databases. The data schema defines the data structure in NoSQL databases. Our contributions in this paper are as follows: 1) we define how to represent the requirements of data query; 2) we design a meta-model of the platform independent data schema for NoSQL databases; 3) we propose an approach to generate the data model and data schema based on the requirements; 4) we use a case study to show how to use our approach to design data model for a NoSQL database; 5) we evaluate the availability and platform-independence of the QODM approach by a real application.

The paper is organized as follows. Section 2 introduces some characteristics of NoSQL databases. Section 3 discusses the differences of data modeling between NoSQL databases and relational databases, a query-oriented data modeling approach, and its framework. Section 4 gives the requirements of the applications. Section 5 analyzes how to generate the data model and data schema based on the requirements. Section 6 uses a case study to illustrate how to apply the QODM approach in a real software development and evaluate availability and platform-independence of the QODM approach. Section 7 analyzes related work, and section 8 is the conclusion and future works.

II. NOSQL DATABASES

Up to now, there are several NoSQL databases. These NoSQL databases can be divided into three kinds: key-value stores, extensible record stores and document stores, and they have different characteristics [1, 5].

For key-value stores, these NoSQL databases' data models are simple. They use a key-value pair to store data. The key is used to store the identifier of an entity, and the value is used to store data of the entity. A representative of key-value stores is Redis. Redis [12] has five types of key-value pair to store data: string pair, hash pair, list pair, set pair and sorted-set pair. String type is the basic type, and both the key and value are strings. The value of hash pair is a hash map. The value of a hash pair consists of several domains. Each domain has its name and value. All domains are indexed by the domain's name. The list pair's value is a list. Developers can use some list operations, such as pop and push, to manage the list pair's value. The set pair's value is a set. Each element of a set pair's value is unique. The sorted-set pair's value is also a set. However, each element in the set consists of a score and a value, and the set is sorted by the score of its element. Each type of key-value pair of Redis provides its own insert, delete and lookup operations. Redis does atomic updates by locking, and does asynchronous replication.

For extensible record stores, a representative is Google's BigTable. The success of BigTable motivates other extensible record stores, such Apache HBase and Apache Cassandra. The data model of BigTable [20] is rows and columns. It looks like traditional relational databases, but doesn't support join operations on tables. The basic scalability model of it is splitting both rows and columns over multiple nodes: rows are split across nodes through sharding on the primary key, and columns of a table are distributed over multiple nodes by using "column groups". The "column groups" of a table must be pre-defined, but the same "column group" of different rows could have different columns. This means that different rows in a table could have different columns, and a new column could be added in a row at any time. This is different from traditional relational databases. It brings more flexibility for developers. The basic data operations of BigTable are insert, delete, and lookup. BigTable only provides atomic operations on a row. It does not support ACID transaction. Other extensible record stores have similar characteristics with BigTable.

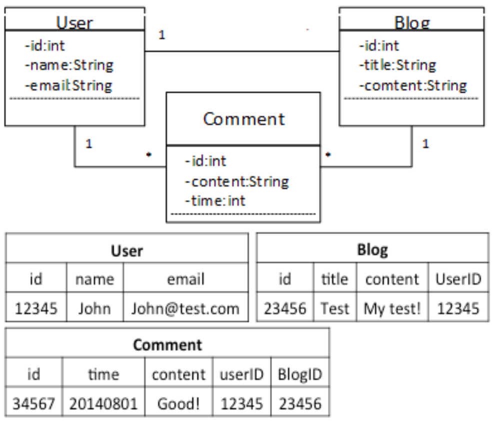
Some representatives of document stores are MongoDB and CouchDB. MongoDB [11] stores a data entity in a document. A document of MongoDB is a bson [9] document, which likes a json [16] object. A set of documents is a collection. The collection in MongoDB likes the table in MySQL, and the document likes the row of table. A document consists of a set of fields. A field represents a property of the entity. Each field has a name and value. Based on the bson-style, the value of a field could be any types, even could be a "document", which means an embedded sub-document. The basic operations of MongoDB are insert, delete, and lookup too. It only provides atomic operations on a document.

III. NOSQL DATA MODELING AND FRAMEWORK

NoSQL databases are different from relational databases. They do not support join operations, have more flexible data schema, and are more suitable for distributed storage. When designing an application's data model for a relational database, the designer only needs to analysis the entities, which need to be stored, and their relationships in problem domain. When query data from a relational database, developers can use join operations to aggregate data in database to get the queried data. However, if designing the data model for a NoSQL database, developers cannot use join operations to get the queried data, which makes data query more complex. Following is an example to show the differences of data modeling between relational databases and NoSQL databases.

An application developer wants to store the data of users, blogs and comments in a database, so he must design the data model and data schema to store them in the database. Figure 1 shows the data model and the data schema for a relational database.

Figure 1 shows the data model by the form of class diagram, which includes data entities and their relationships. In order to store this data model in a relational database, the data schema consists of three tables, which is shown in Figure 1. Developers can use join operations to aggregate these tables to get the queried data. For example, developers could use the id of User table and the userID of Comment table to join these two tables to query the data of users and related comments at the same time.



User		
id	name	email
12345	John	John@test.com

Blog			
id	title	content	UserID
23456	Test	My test!	12345

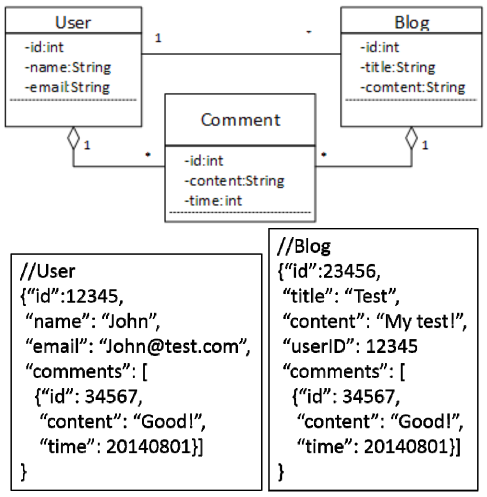
Comment				
id	time	content	userID	BlogID
34567	20140801	Good!	12345	23456

Figure 1. Data model and schema for a relational database

However, if software developers want to store data in a NoSQL database, the data model and data schema described in above are not suitable. NoSQL databases do not support join operations. If developers want to query users and their comments at the same time, they need two database visits to get users and comments data separately, then to aggregate these data to get the queried data manually. Most cloud applications' data is written once and read many times, such as tweets of Twitter. The database visits of data queries are a key feature of cloud applications performance. Therefore, developers must consider what queries they want, when they designing the data model for a NoSQL database, to reduce the database visits of data queries. It is called query-oriented data modeling.

If the developers have two data query requirements: one is querying users and related comments, and the other one is querying blogs and related comments, the data model and data schema for a NoSQL database could like Figure 2.

In Figure 2, the class diagram shows the data model. It means there are two kinds of entity needed to store: the user and blog. The comment will be a part of these two entities. The data schema for a NoSQL database is shown by json-style texts. It shows the data of users, blogs and comments are stored in two kinds of entities in a NoSQL databases. When developers store data in a NoSQL database in this way, they only need one database visits to get the data of users and related comments. Although this manner of storing data will have more redundancies than store data with a relational data model, it will make data queries simpler and faster in a NoSQL database. When the data is more complex than this example, this data modeling manner will be more suitable for NoSQL databases.



```
//User
{"id":12345,
 "name": "John",
 "email": "John@test.com",
 "comments": [
  {"id": 34567,
   "content": "Good!",
   "time": 20140801}]
}
```

```
//Blog
{"id":23456,
 "title": "Test",
 "content": "My test!",
 "userID": 12345
 "comments": [
  {"id": 34567,
   "content": "Good!",
   "time": 20140801}]
}
```

Figure 2. Data model and schema for a NoSQL database

However, except for the data model and data schema shown in Figure 2, there are other choices. For example, if developers want to query data of users, related blogs and comments at the same time, they can store the data of users, blogs and comments in only one kind of entity in a NoSQL database. However, in this way, when developers want to only query all blogs and their comments, this query will be very complex. In order to solve the problem of how to generate the data model and data schema for a NoSQL database based on data query requirements, we proposed a query-oriented data modeling approach, whose framework is shown in Figure 3.

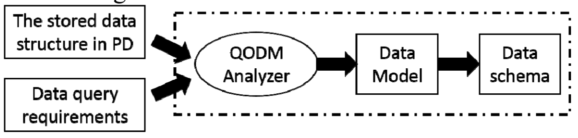


Figure 3. The framework of QODM approach

includes three phrases: 1) Describing the stored data structure in the problem domain and data query requirements of the application. 2) Based on the stored data structure and data query requirements, the QODM-Analyzer generates the data model for NoSQL databases. 3) Then, based on the data model, the QODM-Analyzer generates the data schema for NoSQL databases.

IV. DATA STRUCTURE AND QUERY REQUIREMENTS

The QODM approach needs two inputs: the application's stored data structure in the problem domain and data query requirements. Based on these two inputs, the QODM approach will generate the data model and data schema of the application for NoSQL databases.

When developing an application, at first, developers must analyze the problem domain to determine which entities are needed to store and their relationships in the problem domain, and they need to choose a data management system, such as a file system, a relational database, or a NoSQL database, to store these entities.

Then they need design the data model and data schema to store these data in the data management system that they chose. These entities, which are needed to store in a NoSQL database, and their relationships in the problem domain are called as "the stored data structure" in this paper, which is one of inputs of the QODM approach.

The unified modeling language (UML) [23] is a standard software modeling language, which is developed by the Object Management Group (OMG), and is widely used in software development. Therefore, we chose the class diagram of UML to describe the stored data structure. The *class* element is used to describe the entity and its meta-data: name and properties. The *association*, *aggregation*, *composition* and *generalization* elements are used to describe relationships of two entities in the problem domain.

Except for the stored data structure, the data query requirements are needed as an input of the QODM approach. Each data query requirement is represented by a two-tuples: <C_k, C_t>. C_k and C_t are two class sets, and classes in them are described in the stored data structure. A class in the C_k means some properties of it are a part of key of the corresponding query. A class in the C_t means some properties of it are a part of result of the corresponding query.

When describing the data query requirements, there is a constraint: if class B is a sub-class of class A, or class B is a part of (composition) class A, it must use class A to represent class B in all class sets. This constraint has transitivity. If the class B must be represented by the class A, and the class C must be represented by the class B, then the class C must be represented by the class A.

This constraint divides the classes, in the stored data structure, into two categories: the query classes and the included classes. A query class means a class that in a class set of a query tuple. An included class means a class that be represented by another class.

V. QODM-ANALYZER

When the stored data structure and data query requirements are got, the QODM-Analyzer will analyze them and generate the data model and data schema for NoSQL databases. Section 3 introduced that in order to make query data from NoSQL databases more efficiently, some entities are aggregated into one entity to be stored in the NoSQL database. Therefore, aggregating entities to get the data model for NoSQL databases is the first task of the QODM-Analyzer. There are four relationships in the stored data structure. The QODM-Analyzer should analyze the aggregation and association relationships to decide the related two classes of each relationship to be stored either in an aggregation entity or in an index entity in NoSQL databases.

After getting the data model for NoSQL databases, the second task is designing how to store the data model in a NoSQL databases. Up to now, there are several kinds of NoSQL database. The data models of each kind NoSQL database are different. Therefore, the strategies of storing data in different kinds of database are different. Based on the data model, the QODM-Analyzer will generate the data schema to solve how to store the data model in a NoSQL database.

A. The Data Model

The data model for NoSQL databases is a class diagram, which is generated based on the stored data structure. Therefore, the first task of the QODM-Analyzer is to determine which classes in the stored data structure need to be aggregated to an entity and which classes are needed to store their relationships in an index entity.

The goal of aggregation is to make data query simpler. Ideally, a data query only need to visit one kind of entity in the database. However, it is not a good choice. Aggregation will bring data redundancy. In figure 2, two kinds of entity will be stored in the NoSQL database: user and blog. The data of comments are stored twice: one in the user entity and the other in the blog entity. If there is also a data query requirement to query the data of users, related blogs and comments at the same time, the ideal manner is to store a new kind of entity in the database, which is aggregated by the user, blog and comment. However, in this way, the data of users and blogs will be stored twice, and the data of comments will be stored three times. If the stored data structure and data query requirements are more complex than the example introduced in section 3, the data redundancy of the ideal manner is unacceptable. The data aggregation will make data query simpler in NoSQL databases, but it cannot be abused.

The lack of aggregation will make data query complex, which is introduced in section 3. Therefore, the strategy of the QODM-Analyzer to generate the data model is aggregating data for simple queries and building index entities for complex queries. Figure 4 shows the pseudo-code of the algorithm of generating the data model.

```
Parameters:
T: query tuple (<$C_k$, $C_t$>) set
$C_{all}$: all classes in the stored data structure
Aset = $\emptyset$ : aggregation set
Iset = $\emptyset$ : index set
Aclasses = $\emptyset$ : aggregated classes set
1: sort T by elements number of $C_k \cup C_t$ of each t$\in$T
2: for each tuple t in T do
3:     if a class c in $C_k \cup C_t$ of t is not in Aclasses
4:     then for each class c in $C_k \cup C_t$ of t do
5:             Aclasses $\leftarrow$ c $\cup$ Aclasses
6:         Aset $\leftarrow$ t $\cup$ Aset
7:     else Iset $\leftarrow$ t $\cup$ Iset
```

Figure 4. The pseudo-code of generating the data model

As shown in figure 4, the first step of generating the data model is sorting all query tuples by the number of included classes of each tuple. Each tuple represents a data query. A simpler data query has fewer classes in the corresponding tuple. The sorting makes the simpler query's tuple be processed earlier.

The second step of generating the data model is to determine which query needs to build an aggregation entity, and which needs to build an index entity. This step is shown in line 2 to 7 in figure 4. All classes in a tuple relate to the corresponding query and need to be stored together to reduce database visits of the query. For each query tuple, if there is a class, which is in C_k or C_t of the query tuple, is not aggregated before, all classes in C_k and C_t of the query tuple will be aggregated to one entity to store. However, if all classes in the tuple have been aggregated before, a new index entity will be built to store relationships of these classes. In this way, if all related classes of a query are aggregated before, they will not be aggregated again. It makes the data aggregation will not be abused.

As previously mentioned, classes in the stored data structure are divided into the query classes and the included classes. The query classes have been processed in the second step. For each included class, another class in query tuples represents it. If class A represents an included class, this included class will aggregate with the class A. In this way, each included class will be aggregated with a query class.

In this way, each query will map an aggregated entity or an index entity. Thus, each query could be implemented by visiting the NoSQL databases only once or few times. The data model for NoSQL databases is generated.

B. The Data Schema

There are several kinds of NoSQL database. In order to adapt for different NoSQL databases, the data schema must be a platform independent model, and can be easily transformed to a special NoSQL database. Json is widely used in NoSQL database. MongoDB uses json-like documents to store data. A data object can be represented with a json-style string easily. Many programming language libraries provide functions to transform a data object to a json-style string and transform a json-style string to a data object. Therefore, the QODM approach uses json-like documents to represent the data schema. Figure 5 shows the meta-model of the data schema.

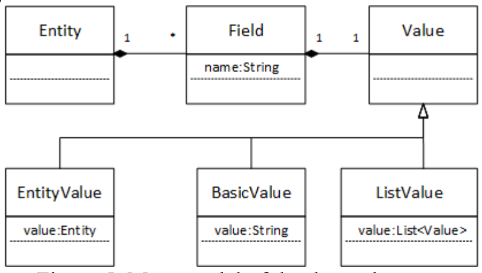


Figure 5. Meta-model of the data schema

As shown in Figure 5, each entity of the data model is represented by an Entity element. An Entity element has several Field elements. A Field element represents an Entity's property. Each Field element has a name property and a Value element. There are three kinds of Value element: EntityValue, BasicValue and ListValue. The BasicValue element represents the Value element's value is basic types, such as string, number, float, etc. The BasicValue element's value is represented by a string. The EntityValue element represents the Value element's value is an entity. It represents an embedded entity. The ListValue element represents the Value element's value is a list. Figure 6 shows the document's structure of the data scheme.

```
//Entity
{"<Field.name>": "<BasicValue.value>",
 "<Field.name>": <EntityValue.value>,
 "<Field.name>": [<ListValue.value.get(0)>,
  <ListValue.value.get(1)>,
  <ListValue.value.get(2)>,
  ...]
}
```

Figure 6. The data schema document structure

If developers want to transform the data schema to a NoSQL database, they only need to get the meta-model of the NoSQL database, and use some model transformation techniques or data mapping techniques to implement it. There are already some techniques [4,13,17] about how to map the data object model to a NoSQL database. Based on these techniques, the data scheme could be transformed for a NoSQL database easily, even to generate code for the NoSQL database. These techniques are not the core of this paper, so we will not discuss them here.

VI. CASE STUDY

Preceding part of the text has introduced the query-oriented data modeling approach. The QODM approach will get the data model and data schema at last, which can be easily transformed to a NoSQL database with the model transformation techniques or data mapping techniques. This section will introduce a case study of the QODM approach. We use a real application to show how to apply the QODM approach, and evaluate the availability and platform-independence of the QODM approach.

ElasticInbox [22] is an open source, distributed, reliable, scalable email store. All information about ElasticInbox could be gotten on the website. The goal of ElasticInbox is to provide highly available email store without a single point of failure that can run on commodity hardware and scale linearly. ElasticInbox can easily scale to millions of mailboxes, with hundreds of thousands messages in each mailbox. Up to now, users can use Rest API or POP3 to access the data in ElasticInbox. ElasticInbox uses Apache Cassandra to store the metadata and uses cloud object store, such as Amazon S3 or Azure Blob, to store the message.

ElasticInbox is a runnable application. Apache Cassandra [21] is an extensible record NoSQL database, which is used to store the metadata of ElasticInbox. Therefore, we can extract the stored data structure and data query requirements from the source code of ElasticInbox as the inputs of the QODM approach. The website of ElasticInbox shows its data model and data schema for Apache Cassandra. They can be used to evaluate whether the data model and data schema, which are generated by the QODM approach, are suitable for NoSQL databases.

A. The Stored Data Structure

ElasticInbox is written in Java. It adopts the MVC architecture. Therefore, the stored data structure could be extracted from the model part. We extract the data structure from the core.src.main.java.com.elasticinbox.core.model package. Figure 7 shows the stored data structure of ElasticInbox extracted from the source code manually.

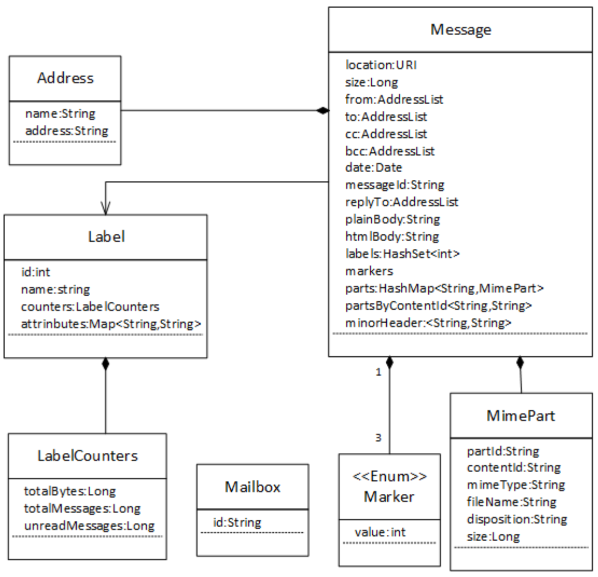


Figure 7. The stored data structure of ElasticInbox

There are seven entities that need to be stored. The stored data structure includes these entities' properties and their relationships.

B. The Data Query Requirements

ElasticInbox provides Rest API and POP3 interface. Users could use one of them to visit data in ElasticInbox. Therefore, the data query requirements could be extracted from them. We extracted the data query requirements from the Rest API. All implementations of Rest API are in the rest.src.main.java.com.elasticinbox.rest.v2 package.

There are five java files in the package. The data query functions are only in three of them: in the LabelResource.java file, there is only one data query function: query the data of Message based on the properties of Mailbox and Label; in the MailboxResource.java file, there is one data query function: query the data of Label based on the properties of Mailbox and Label; in the SingleMessageResource.java file, there are five data query functions, but they all are querying the data of Message based on the properties of Mailbox and Message. Therefore, finally, we extracted three data query requirements from the source code of ElasticInbox. Figure 8 shows these data query requirements by the data query tuples.

Q1: <{Mailbox, Label}, {Message}>
Q2: <{Mailbox}, {Label}>
Q3: <{Mailbox, Message}, {Message}>

Figure 8. The data query requirements

C. The Data Model and Data Schema

With the stored data structure and the data query requirements we extracted from the source code of ElasticInbox, the QODM approach will generate the data model and data schema of ElasticInbox for NoSQL databases.

Based on the sorting of data model generating algorithm, Q2 will be processed at first, and Q1 will be processed at last. For Q2, the queried Label entity is not aggregated before. The Mailbox and the Label will be aggregated to a new entity to store. For Q3, the queried Message entity is not aggregated before. The Mailbox and the Message will be aggregated to another new entity to store. For Q1, the related Mailbox entity, Label entity and Message entity have been aggregated before, so a new index entity will be created to store the relationships between Mailbox, Label and Message. Figure 9 shows the data model that is generated by the QODM approach.

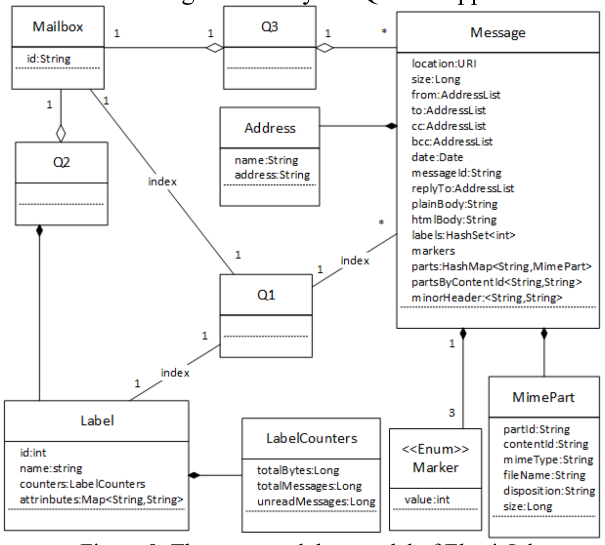


Figure 9. The generated data model of ElasticInbox

In Figure 9, there are three entities needed to be stored in NoSQL databases. Q2 and Q3 are two aggregated entities. Q2 includes Mailbox entity and all properties of related Label entities. Q3 includes Mailbox entity and all properties of related Message entities. Q1 is an index entity, which includes the relationships between Mailbox, Label and Message entities. The following is the data schema of Q1:

```
"MailboxID" : <mailboxid>
"LabelID" : <labelid>
"messages" : [
 <messageID0>,
 <messageID1>,
 …]
```

D. Evaluation

ElasticInbox is a runnable application. It uses Apache Cassandra, an extensible record NoSQL database, to store the metadata of mailbox. The data schema of ElasticInbox for Apache Cassandra is provided on the website. It can be used to verify whether the data model and data schema, which are generated by the QODM approach, are suitable for NoSQL databases. In this way, we evaluate availability of the QODM approach.

Actually, ElasticInbox uses five column families to store its data: Accounts CF, MessageMetadata CF, MessageBlob CF, IndexLables CF and Counters CF.

The Accounts CF and Counters CF store the data of labels of each mailbox account. The Accounts CF stores what labels an account has, and the Counters CF stores stats of each label of an account. The keys of these two column families both are the mailbox address, which is the key property of Mailbox in the data model. The stored data of these two column families is same with the Q2 data schema. Actually, two column families will bring more database visits, when query the data of label, than the data schema generated by QODM approach. Therefore, these two column families are corresponding to the Q2 data schema.

The MessageMetadata CF stores the message metadata of each account. The key of MessageMetadata CF is the mailbox address, which is the key property of Mailbox in the data model. It has the same function and structure with the Q3 data schema.

The InboxLabels CF is used to store all message ids of each label of an account. It uses the mailbox address and label id as its key. This column family stores the relationships of mailbox, label and message, which is the same with the Q1 data schema.

ElasticInbox uses the MessageBlob CF to store message data, which should be stored in a cloud object store. This means ElasticInbox not only treats Apache Cassandra as a NoSQL database, but also as a cloud object store. The message data is not included in the stored data structure of ElasticInbox, which only includes the metadata of ElasticInbox. Thus, it is reasonable that there is no data schema corresponding to the MessageBlob CF.

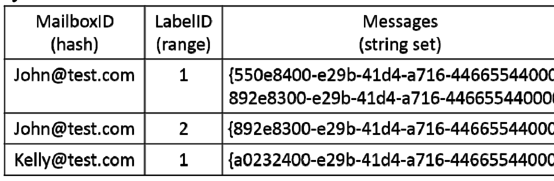
In conclusion, there is one-to-one correspondence between the real data schema of ElasticInbox and the generated data schema. The data model and data schema, which are generated by the QODM approach, are suitable for NoSQL databases.

E. Platform Independence

As aforementioned, the generated data schema of ElasticInbox corresponds to the real data schema for Apache Cassandra. This means the generated data schema is suitable for Apache Cassandra, and it could be transformed to the Apache Cassandra schema by suitable model transformation techniques or data mapping techniques. Except Apache Cassandra, because the generated data schema is a platform independent model, it also could be transformed to other NoSQL databases. For example, if the developers want to store metadata of ElasticInbox in Amazon DynamoDB, the generated data schema could be transformed to DynamoDB schema too.

Amazon DynamoDB [10] is a NoSQL database service. It is used to store semi-structured data. For each piece of data, except the primary key, its scheme is unconstrained. The data model of DynamoDB has a secondary structure: table and item. A table is a set of items, and an item is a piece of data, which is an entity in the table. This data model looks like traditional relational databases: table and row. Being same with traditional relational databases, in a table, the primary key's type of each item is same and predefined. A difference of data model between DynamoDB and traditional relational databases is that non-key attributes have no predefined and constant schema in DynomoDB. The primary key of DynamoDB could consist of two key attributes: hash type attribute and range type attribute. DynamoDB builds an unordered hash index on the hash type attribute and a sorted range index on the range type attribute. The hash type attribute is required and the range type attribute is optional. DynamoDB supports six types of data to store: number, string, binary, number set, string set, binary set.

Figure 10 shows a sample of the Q1 entity data schema in DynamoDB.



MailboxID (hash)	LabelID (range)	Messages (string set)
John@test.com	1	{550e8400-e29b-41d4-a716-446655440000, 892e8300-e29b-41d4-a716-446655440000}
John@test.com	2	{892e8300-e29b-41d4-a716-446655440000}
Kelly@test.com	1	{a0232400-e29b-41d4-a716-446655440000}

Figure 10. A sample of Q1 data schema in DynamoDB

The hash type attribute of primary key of Q1 in DynamoDB is the id of Mailbox, and the range type attribute is the id of Label. The messageIds of Messages, which are related to the Mailbox id and the Label id, are stored in a string set in DynamoDB. Each messageId is an UUID string.

VII. RELATED WORKS

With the development of NoSQL databases, several researchers and developers discussed the skills of using NoSQL databases. Some famous NoSQL database vendors provide developer guides about their database products, such as Amazon DynamoDB [24]. However, these guides are only useful for their own databases. These guides always only introduce what data model is suitable for there own databases, and they do not discuss how to design the data model of an application. The QODM approach proposed in this paper could generate platform independent data model and data schema of an application for NoSQL databases. It will not lock developers in any NoSQL database.

Luca Cabibbo [4] discussed about how to map the data object model to NoSQL databases. However, he did not talk about how to design a suitable data model for NoSQL databases. Aaron Schram and Kenneth M. Anderson [2] introduced their own experiences about migrating data from MySQL, a relational database, to Apache Cassandra. They only discussed how to design the data model of their own application for NoSQL databases, and did not discuss how to do it for other applications. The Hibernate [13] and Paolo Atzeni [17] also provided their data mapping frameworks for NoSQL databases. They provided their programming frameworks to map the data object to a NoSQL database. However, these frameworks only support few NoSQL databases, and the developer must design the data model and decide the data mapping strategy.

VIII. CONCLUSION AND FUTURE WORK

According to the characteristics of NoSQL databases, the paper proposes a query-oriented data modeling approach for NoSQL databases. The approach discusses how to design an application's data model and data schema for NoSQL databases, can generate platform independent data models and data schemas for NoSQL databases. Then developers could use some suitable model transformation techniques or data mapping techniques to transform the data schema for a special NoSQL database, such as the sample shown in Figure 11. The inputs of QODM approach only are the application's stored data structure in problem domain and data query requirements.

We defined the structure of two-tuples to represent data query requirements. Based on the stored data structure and data query tuple, we proposed an algorithm to generate the data model and data schema for NoSQL databases. Moreover, we defined a platform independent model of data schema for NoSQL databases.

We used ElasticInbox, which is a real application using Apache Cassandra to store some data, to evaluate the QODM approach we proposed. Based on the comparison between the generated data schema and the real data schema of ElasticInbox, we evaluated the availability of QODM approach. Moreover, we transformed the generated data schema for Amazon DynamoDB to evaluate platform-independence of the QODM approach.

Graph databases are good at storing relationships, and they are very different from other NoSQL databases. Therefore, the QODM approach does not include graph databases. In the future, we will pay attention to study data modeling approach to graph databases. Then, we will try to integrate the QODM approach and graph databases data modeling approach to a unified data modeling approach.

ACKNOWLEDGMENT

The work supported by the National Natural Science Foundation of China (No. 61272159) and Beijing Natural Science Foundation (No. 4122036).

REFERENCES

[1] Bugiotti Francesca, and Luca Cabibbo. A Comparison of Data Models and APIs of NoSQL Datastores. 2013, www.bugiotti.it/downloads/publications/noamSEBD13.pdf

[2] Aaron Schram and Kenneth M. Anderson. MySQL to NoSQL: data modeling challenges in supporting scalability. In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity. ACM, New York, NY, USA, 2012, 191-202.

[3] Moniruzzaman, A. B. M., and Syed Akhter Hossain. NoSQL Database: New Era of Databases for Big data Analytics-Classification, Characteristics and Comparison. International Journal of Database Theory & Application 6.4,2013.

[4] Luca Cabibbo. ONDM: an Object-NoSQL Datastore Mapper. Faculty of Engineering, Roma Tre University. Retrieved June 15th 2013.

[5] Rick Cattell. Scalable SQL and NoSQL data stores. SIGMOD Rec. 39, 4 , May 2011, 12-27.

[6] Christof Strauch, Ultra-Large Scale Sites, and Walter Kriha. NoSQL databases. Lecture Notes, Stuttgart Media University, 2011.

[7] Shen DR, Yu G, Wang XT, Nie TZ, Kou Y.Survey on NoSQL for management of big data. Ruan Jian Xue Bao/Journal of Software, 2013,24(8):1786− 1803.

[8] ACID. http://zh.wikipedia.org/zh-cn/ACID, 2014.

[9] Bson. http://bsonspec.org, 2014.

[10] Giuseppe DeCandia, et al. Dynamo: amazon's highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, New York, NY, USA, 2007,205-220.

[11] MongoDB. http://www.mongodb.org/, 2014.

[12] Redis. http://redis.io/, 2014.

[13] Hibernate. http://hibernate.org/, 2014.

[14] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33, 2, June 2002, 51-59.

[15] Dan Pritchett. BASE: An Acid Alternative. Queue 6, 3, May 2008, 48-55. DOI=http://doi.acm.org/10.1145/1394127.1394128

[16] Json. http://www.json.org, 2014.

[17] Paolo Atzeni, Francesca Bugiotti, and Luca Rossi.Uniform access to non-relational database systems: the SOS platform. In Proceedings of the 24th international conference on Advanced Information Systems Engineering, Springer-Verlag, Berlin, 2012, 160-174.

[18] Mei, Hong, and Xuan-Zhe Liu. Internetware: An emerging software paradigm for Internet computing. Journal of computer science and technology 26.4, 2011, 588-599.

[19] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A Taxonomy and Survey of Cloud Computing Systems. In Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC., Washington, DC, USA, 2009, 44-51.

[20] Fay Chang, et al. Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26, 2, Article 4, June 2008, 26 pages.

[21] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44, 2, April 2010, 35-40.

[22] ElasticInbox. http://www.elasticinbox.com, 2014.

[23] Unified Modeling Language (v2.4.1). http://www.omg.org/spec/UML/2.4.1/, 2011.

[24] Amazon DynamoDB Developer Guide. http://docs.aws.amazon.com/zh_cn/amazondynamodb/latest/developerguide/Introduction.html, 2012.