

ECE656 Project Report: Document-Oriented Database

Hai Jiang, Xiyue Zhang, Chang Liu

Abstract—The objective of this paper is to study the document-oriented database. A new index system is design to accelerate the performance of MongoDB by analyzing and processing the semi-structured data. The standard query is automatically generated, which improves the accuracy and the speed of finding the required information from MongoDB. Another approach of combining both MongoDB and MySQL is experimented to improve the safety of banking related transactions. The data set is spliced into two parts according to a well-designed standard. MongoDB and MySQL stores a corresponding partition respectively. Necessary coding is added to coordinate data between the two parts, which ensures a normal operation and avoids loss caused by incomplete transaction.

Index Terms – Document database, index, combination of MySQL and MongoDB.

I. INTRODUCTION

A document-oriented database or document store is a computer program designed for storing, retrieving, and managing document-oriented information, also known as semi-structured data [1]. As one of the main categories of NoSQL databases, document-oriented databases can be regarded inherently a subclass of the key-value store. However, document-oriented database provides richer experience with modern programming techniques by relying on internal structure in the document in order to extract metadata that the database engine uses for further optimization. XML databases are a specific subclass of document-oriented databases that are optimized to extract their metadata from XML documents.

In this paper, the introductions for document-oriented database are given in chapter 2. Chapter 3 illustrates the idea of an innovative index system design. Chapter 4 achieves managing data set by combining both MongoDB and MySQL.

II. DOCUMENT-ORIENTED DATABASE

A. NoSQL models

Carlo Strozzi first used the term NoSQL in 1998 as a name for his open source relational database that did not offer a SQL interface [3]. The term was reintroduced in 2009 by Eric Evans in conjunction with an event discussing open source

distributed databases. This time it did not refer to a particular system, but rather a step away from the relational model altogether (as opposed to the query language). Appropriate or not, the name attempts to describe the increasing number of distributed non-relational databases that have emerged during the second half of the 2000's [4] [5].

In general, but not ubiquitous, traits that most of the NoSQL systems share:

- They lack fixed schemas
- They avoid joins (the operation of combining relations)
- They scale horizontally

1) Key-Value

This class of storage systems arguably have the simplest data model. Rather than tables or semi structured documents, data is just organized as an associative array of entries. A unique key is used to identify a single entry and all of the three available operations use this key; delete the given key, change the entry associated with the key, insert a new key with associated data [3].

Noteworthy is that even though such a store is very straightforward to the application programmer, implementations often store data using a more traditional model behind the scenes. While the key-value-store looks and acts like an associative array, it may rely on tables, indexes and other artefacts of relational systems to be efficient in practice.

2) Document-oriented

Document-oriented databases are semi structured data storages, usually implemented without tables. The idea is to allow the client application to add and remove attributes to any single tuple without wasting space by creating empty fields for all other tuples. This means that all tuples can contain any number of fields, of any length. Even if used in a structured way, there are no constraints or schemas limiting the database to conform to a pre-set structure. This means that the application programmer gains ease of use and the possibility to create very dynamic data. The cost for all of this is the lack of a safety net and, to a certain degree, performance.

A document-oriented database can be implemented as a layer over a relational or object database. Another option is to implement it directly in a semi-structured file format, such as JSON, XML or YAML. Traditional concepts like indexes and keys are often employed in the same sense as in relational databases. By using these, one is supposed to achieve almost the same performance as would be possible in a system implemented with tables.

B. Contrast between Document Databases and Relational Databases

Document databases [2] contrast strongly with the traditional relational database (RDB). Relational databases are strongly typed during database creation, and store repeated data in separate tables that are defined by the programmer. In an RDB, every instance of data has the same format as every other, and changing that format is generally difficult. Document databases get their type information from the data itself, normally store all related information together, and allow every instance of data to be different from any other. This makes them more flexible in dealing with change and optional values, maps more easily into program objects, and often reduces database size. This makes them attractive for programming modern web applications, which are subject to continual change in place, and speed of deployment is an important issue.

C. Semi-structured data

Semi-structured data [7] is a form of structured data that does not conform with the formal structure of data models associated with relational databases or other forms of data tables, but nonetheless contains tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data. Semi-structured data is often described as “schema-less” or “self-describing”, terms that indicate that there is no separate description of the type or structure of data. Typically, when we store or program with a piece of data we first describe the structure (type, schema) of that data and then create instances of that type (or populate) the schema [6]. In semi-structured data, the entities belonging to the same class may have different attributes even though they are grouped together, and the attributes' order is not important.

1) XML

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format which is both human readable and machine-readable. It is defined by the W3C's XML 1.0 Specification [8] and by several other related specifications [9], all of which are free open standards [10].

The design goals of XML emphasize simplicity, generality and usability across the Internet.[11] It is a textual data format with strong support via Unicode for different human languages. Although the design of XML focuses on documents, it is widely used for the representation of arbitrary data structures [12] such as those used in web services. Several schema systems exist to aid in the definition of XML-based languages, while many application programming interfaces (APIs) have been developed to aid the processing of XML data.

2) JSON

JSON or JavaScript Object Notation is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML. JSON has been popularized by web services developed utilizing REST principles. There is a new breed of databases such as MongoDB and Couchbase that

store data natively in JSON format, leveraging the pros of semi-structured data architecture.

a) JSON's basic data types

- Number: a signed decimal number that may contain a fractional part and may use exponential E notation, but cannot include non-numbers like NaN. The format makes no distinction between integer and floating-point. JavaScript uses a double-precision floating-point format for all its numeric values, but other languages implementing JSON may encode numbers differently.
- String: a sequence of zero or more Unicode characters. Strings are delimited with double-quotation marks and support a backslash escaping syntax.
- Boolean: either of the values true or false
- Array: an ordered list of zero or more values, each of which may be of any type. Arrays use square bracket notation with elements being comma-separated.
- Object: an unordered collection of name/value pairs where the names (also called keys) are strings. Since objects are intended to represent associative arrays, it is recommended, though not required, that each key is unique within an object. Objects are delimited with curly brackets and use commas to separate each pair, while within each pair the colon ':' character separates the key or name from its value.
- null: An empty value, using the word null

b) JSON's Example

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

3) Metadata

Metadata is "data that provides information about other data".[18] Two types of metadata exist: structural

metadata and descriptive metadata. Structural metadata is data about the containers of data. Descriptive metadata uses individual instances of application data or the data content.

Metadata was traditionally in the card catalogs of libraries. As information has become increasingly digital, metadata is also used to describe digital data using metadata standards specific to a particular discipline. Describing the contents and context of data or data files increases their usefulness. For example, a web page may include metadata specifying what language the page is written in, what tools were used to create it, and where to find more information about the subject; this metadata can automatically improve the reader's experience.

The main purpose of metadata is to facilitate in the discovery of relevant information, more often classified as resource discovery. Metadata also helps organize electronic resources, provide digital identification, and helps support archiving and preservation of the resource. Metadata assists in resource discovery by "allowing resources to be found by relevant criteria, identifying resources, bringing similar resources together, distinguishing dissimilar resources, and giving location information." [19]

D. MongoDB

MongoDB is a cross-platform document-oriented database. Classified as a NoSQL database, MongoDB eschews the traditional table-based relational database structure in favor of JSON-like documents with dynamic schemas (MongoDB calls the format BSON), making the integration of data in certain types of applications easier and faster. MongoDB is developed by MongoDB Inc. and is published as free and open-source software under a combination of the GNU Affero General Public License and the Apache License. As of July 2015, MongoDB is the fourth most popular type of database management system, and the most popular for document stores. [14]

1) Main features

a) Document-oriented

Instead of taking a business subject and breaking it up into multiple relational structures, MongoDB can store the business subject in the minimal number of documents. For example, instead of storing title and author information in two distinct relational structures, title, author, and other title-related information can all be stored in a single document called Book [15].

b) Ad hoc queries

MongoDB supports field, range queries, regular expression searches. Queries can return specific fields of documents and also include user-defined JavaScript functions.

c) Indexing

Any field in a MongoDB document can be indexed – including within arrays and embedded documents (indices in MongoDB are conceptually similar to those in RDBMSes). Primary and secondary indices are available.

2) Find or Query Data with the mongo Shell [22]

You can use the find() method to issue a query to retrieve data from a collection in MongoDB. All queries in MongoDB

have the scope of a single collection. Queries can return all documents in a collection or only the documents that match a specified filter or criteria. You can specify the filter or criteria in a document and pass as a parameter to the find () method. The find() method returns query results in a cursor, which is an iterable object that yields documents.

a) Query for All Documents in a Collection

To return all documents in a collection, call the find() method without a criteria document. For example, the following operation queries for all documents in the restaurants collection.

e.g., db.restaurants.find()

b) Specify Equality Conditions

The query condition for an equality match on a field has the following form:

{ <field1>: <value1>, <field2>: <value2>, ... }

If the <field> is a top-level field and not a field in an embedded document or an array, you can either enclose the field name in quotes or omit the quotes.

If the <field> is in an embedded document or an array, use dot notation to access the field. With dot notation, you must enclose the dotted name in quotes.

c) Query by a Top Level Field

The following operation finds documents whose borough field equals "Manhattan".

e.g., db.restaurants.find({ "borough": "Manhattan" })

The result set includes only the matching documents.

d) Query by a Field in an Embedded Document

To specify a condition on a field within an embedded document, use the dot notation. Dot notation requires quotes around the whole dotted field name. The following operation specifies an equality condition on the zip code field in the address embedded document.

e.g., db.restaurants.find({ "address.zipcode": "10075" })

The result set includes only the matching documents.

For more information on querying on fields within an embedded document, see Embedded Documents.

e) Query by a Field in an Array

The grades array contains embedded documents as its elements. To specify a condition on a field in these documents, use the dot notation. Dot notation requires quotes around the whole dotted field name. The following queries for documents whose grades array contains an embedded document with a field grade equal to "B".

e.g., db.restaurants.find({ "grades.grade": "B" })

The result set includes only the matching documents.

III. METHOD FOR SEARCHING, INDEXING, PARSING DOCUMENT DATABASE INCLUDING SUBJECT DOCUMENTS

As a powerful Document Oriented Storage, MongoDB is excel in deep query-ability. In other words, MongoDB supports dynamic queries on documents using a document-based query language that's nearly as powerful as SQL.

An example of the documents stored in MongoDB is a table of hotel policies. The schema in MongoDB is listed in Fig. 3.1.

```
'hid':88,      Hotel id
'date':20150530, Check-in Date
'enable':1, Enable policy
'price':100, Policy Price
'name':'abc', Policy Name
'position':'china', Hotel Address
'writeTime':datetime.datetime.now(), Time of write in
```

Fig. 3.1 Exemplary document-oriented schema for MongoDB

To find out the 10 cheapest policies of this hotel, an example of the query language is given in Fig. 3.2.

```
db.getCollection('hotels').find({"hid":88, "date":
{"$gte":20150501, "$lte":20150510}, "enable":1}).sort
({"price":1}).limit(10)
```

Fig. 3.2 Exemplary query for searching 10 cheapest prices

Three conditions given in this query include:

- The hotel id is “hid”: 88;
- The date is between 20150501 and 20150510
- Enable is set to 1

We can even set up index to make the searching query works more efficiently [20].

A. Shortcomings of the search engine in MongoDB

One of the basic uses of searching or parsing a document-oriented database is to associate text with the type of structure, or field, in which it is found—for example, a title, abstract, body, paragraph, table, list, and the like. By allowing these associations, complex text structures having multiple levels can be achieved. For example, portions of text can be associated with meta words indicating that text is found within a paragraph that is within a list that is within another paragraph [21]. Therefore, in order to give an efficient search query, the user needs to know the structure of the database schema.

One particular problem that arises from searching for text associated with certain meta words occurs when text contains multiple fields that overlap or enclose each other. An example of this type of problem is shown in Fig. 3.3. In Fig. 3.3, the key “grants” in the first level has a value which turns out to be an array, e.g., defined by include data into a bracket “[]”. The items in this array are key-value pairs. The values corresponding to different keys can be different data types. For example, true, “viewer”, [“public”] are Boolean, string, and array containing a string, respectively. There are also other structures in the example shown in Fig. 3.3. For example, the value of the key metadata is a dictionary type whose value

is another dictionary, which is so called complicated nesting level fields.

```
"viewCount" : 71106,
"viewLastModified" : 1364274981,
"viewType" : "tabular",
"grants" : [ {
  "inherited" : true,
  "type" : "viewer",
  "flags" : [ "public" ]
} ],
"metadata" : {
  "custom_fields" : {
    "TEST" : {
      "CFPB1" : ""
    }
  },
  "renderTypeConfig" : {
    "visible" : {
      "table" : true
    }
  },
  "availableDisplayTypes" : [ "table", "fatrow", "page" ],
  "rdfSubject" : "0",
  "rowIdentifier" : 53173967
},
```

Fig. 3.3 Exemplary document-oriented schema for MongoDB

If a search engine were queried as to whether this document has an instance of the word “Viewer” within a paragraph, it requires the specific route of its key, e.g., the user needs to know the “type” is inherited from “grants” such that a query `db.connection.find(“grants.type”:“**”) can be generated. Otherwise, MongoDB does not know where to locate “type”, and usually a “null” is output in return.`

One way of overcoming the problem of nested fields is, when creating the index, to parse the document into each separate field, and to index separately all the text stored within each field. However, this leads to duplication, because fields may overlap and different fields will then contain the same text. Thus, this solution is expensive in terms of data storage requirements, as well as time-consuming for indexing and searching purposes. In some circumstances, different objects in the same document may have different field structures, for example, every page of a website may include different information contents and thus the gathered data would be in different fields. Creating an efficient index system to track those fields is hard and requires significant computational resources to process queries.

Based upon our understanding of MongoDB data schema and the parsing queries, we design a system and method for indexing a database of documents that contains entries having nesting level information associated with the meta words so that fields nested within fields could be quickly and effectively searched.

B. System Design

Fig. 3.4 shows a document database system comprising MongoDB as the major database. An index system is developed to generate an index hash map when a new json data set is saved into Mongo DB. The index hash map contains information about the field structures of new Json data sets. When a user query is given, e.g., via a user interface, the index system analyzes the query, calls the hashmap to identify the complete route of keys, and generates the standard query language of MongoDB accordingly. Then the index system transmitted the standard query into MongoDB. In response,

MongoDB searches for the target object and outputs the search result.

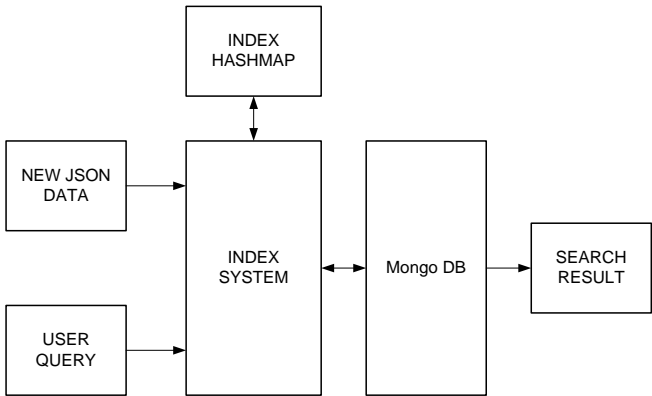


Fig. 3.4 Document database system

We experiment on a Json data set (<https://catalog.data.gov/dataset/complaint-problems-7052e>). APPENDIX (1) attaches an object of the JSON data set for illustrative purpose. As is shown in APPENDIX (1), this JSON data set is composed of various document-oriented data structures. More specifically, some simple structures include basic key-value model, e.g., "averageRating" : 0, and simple array, e.g., "rights" : ["create_datasets", "edit_others_datasets", "edit_nominations", "approve_nominations", "moderate_comments", "manage_stories", "feature_items", "change_configurations", "view_domain", "view_others_datasets", "create_pages", "edit_pages", "view_goals", "view_dashboards", "edit_goals", "edit_dashboards", "manage_provenance", "view_story", "view_unpublished_story", "view_all_dataset_status_logs"].

More complex data structures are those having nesting fields. In the example as shown below,

```
"grants" : { {
  "inherited" : true,
  "type" : "viewer",
  "flags" : [ "public" ]
} ],
```

The value of key “grants” is an array which further includes a field comprising three key-value pairs. Among them, "flags" corresponds to value which is an array type.

Moreover,

```
"metadata" : {
  "custom_fields" : {
    "TEST" : {
      "CFPB1" : ""
    }
  }
}
```

this example has multiple levels of the dictionary field, which means that the value of the key-value pair can be another key-value pair, and goes on and on into deeper levels.

The index system is capable of analyzing the data structures and stores the key-value sets into the index hashmap.

GENERIC FORM

DUPLICATE KEY	BRANCH KEYS
---------------	-------------

KEY FORM

SON KEY	PARENT KEY
---------	------------

Fig. 3.5 Hashmap structure

Fig. 3.5 illustrates the architecture of the index hash map. The index hash map includes a generic form and a key form. The generic form is a hash map that stores information about duplicate keys. The key of the generic form is the duplicate key. The value of the generic form is branch keys that are generated according to the duplicate key. For example, in the level 0 of the data set, there is a key “type”, while in the level 1 of the data set there is also a key “type”. To query the two “type”s, we need to submit different queries having different routes to the “type”. Thus, we can name the branch keys as ‘type1’ and ‘type2’ to distinguish the two ‘type’ s, which are used in the key form.

The key form is a hash map that stores all the keys in a son-parent pattern by which the relationships between their nested fields are represented. More specifically, the key of the key form is a key in a current level field, and the value of this key is the parent key in the upper level field.

1) Top level keys

We set the parent of the keys in the top level field as ‘0’. Therefore the information about the top level keys in the key form can be shown as in Fig. 3.6. It is necessary to mention that ‘0’ is a specialized label we choose to indicate the top level field, another character that is not used as a key in this data set can also be used here. Fig. 3.6 gave two keys ‘id’, ‘name’ which are inserted into the key form.

Top level keys	‘0’
----------------	-----

Example

id	‘0’
name	‘0’

Fig. 3.6 Top level keys in key form

2) Non-array nesting field

For the non-array nesting field, we generate the key form strictly following the son-parent pattern, that is, the key in the

hash map is the son key, and the value of the hash map is the parent key.

Take the “metadata” field as example,

```
"metadata" : {
  "custom_fields" : {
    "TEST" : {
      "CFPB1" : ""
    },

```

the data stored in key form can be given in Fig. 3.7.

3) Array nesting field

Array nesting field means that the value is an array, inside which there is at least one other key. In such circumstances, one or more keys in the array may have the same parent key. To label the difference, a middle key is generated to bridge the son key in the array and the parent key. The middle key can be the parent key combined with the position of son key in the array.

For example, with the data

```
"grants" : [ {
  "inherited" : true,
  "type" : "viewer",
  "flags" : [ "public" ]
} ],
```

the corresponding key-value pairs in the key form can be given as in Fig. 3.8. We can also use grant[0] to all the three pairs, because they are in the same {} which is further involved by the array.

‘metadata’	‘0’
‘custom_fields’	‘metadata’
‘TEST’	‘custom_fields’
‘CFPB1’	‘TEST’

Fig. 3.7 keys with non-array nesting fields in key form

‘grants’	‘0’
‘inherited’	‘grants[0]’
‘type’	‘grants[1]’
‘flags’	‘grants[2]’

Fig. 3.8 keys with array nesting fields in the key form

4) Duplicate keys

The duplicate keys represent two or more keys in the same document object that share the same name. For example, there is a “displayName” in the value field of “owner”, and another “displayName” can be found in the value field of “tableAuthor”. To query according to the two fields need different queries having different routes. Thus, the duplicate information is saved in the generic form. As shown in Fig. 3.5, the branch keys are generated to differentiate the keys having same names. The branch keys are used to generate the key form instead of the duplicate key itself.

GENERIC FORM

"displayName"	"displayName", "displayName1"
---------------	-------------------------------

Fig. 3.9 Generic form for duplicate key “displayName”

Fig. 3.9 and Fig. 3.10 gave an example of the duplicate “displayName”. The duplicate key “displayName” is replaced by two branch keys “displayName” and “displayName1”, which are further used to generate the key form hash map. Advantageously, the branch keys have differentiated the duplicate keys having the same name but different locations. Therefore, accurate route can be given without being messed up by the two or more possibilities.

‘owner’	‘0’
‘displayName’	‘owner’
‘tableAuthor’	‘0’
‘displayName1’	‘tableAuthor’

Fig. 3.10 Key form for duplicate key “displayName”

5) Generation of the query

Fig. 3.11 illustrates the method for generating query to MongoDB. When a query request is received, e.g., the user submits a query request including a target key, the index system called the hash map. The generic form is called at first to check if it contains the target key. If the target key is in the generic form, indicating that the target key is a duplicate key, then a branch key located in the value of the target key is read out as the new target key. If the target key is not found in the generic form, the target key is not changed.

Next, the target key is located in the hash map of key form. According to the son-parent structures, the key list is traced back until a value of the key becomes ‘0’. As a result, the accurate root of the target key is recorded to generate a standard query language. The standard query language is forwarded into MongoDB. In response, MongoDB outputs the desired data object.

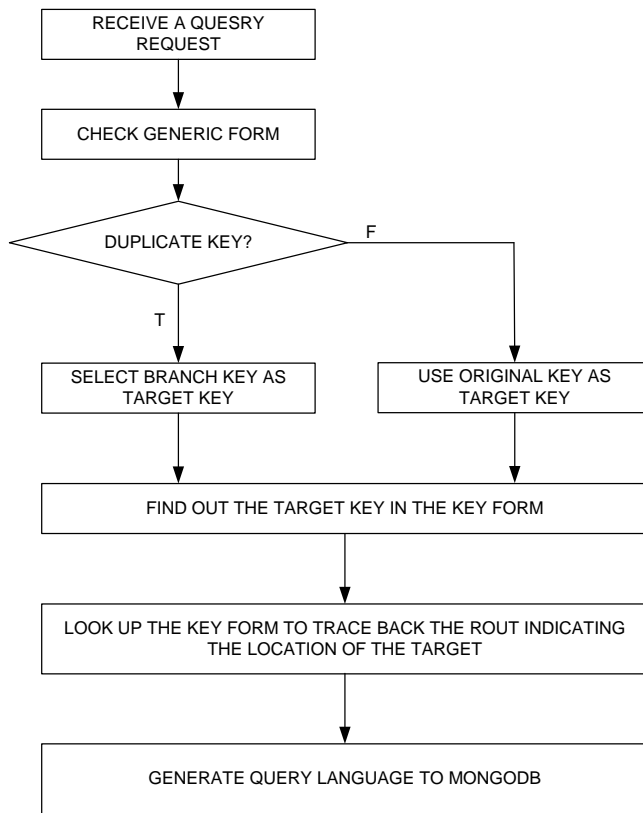


Fig. 3.11 Flowchart of method for generating query to MongoDB

6) Adaptability for new data structure

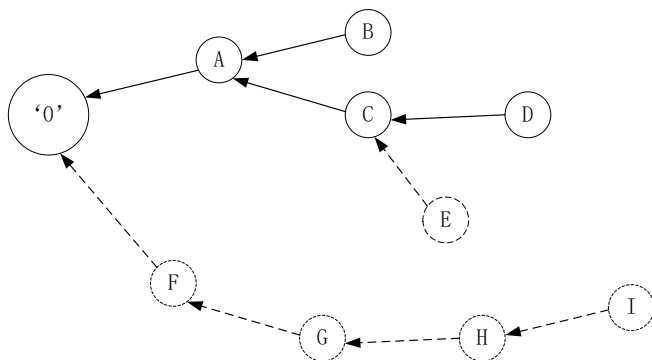


Fig. 3.12 Example of hash map structure of Key form

The innovate <son : parent> data structure as stored in the Hash Map works especially well when a new json data structure is added to the same database. Let us discuss this question in combination with Fig. 3.12. Assuming that the key form contains a data structure ['0', A, B, C, D], where '0' is the parent of A, A is the parent of both B and C, and C is the parent of D.

If a new data is updated into the database, we can identify if the son-parent relationship is different from ['0', A, B, C,

D]. If the data structure is the same, the key form is not updated.

If there is some slight different, e.g., a new key is added, we can create the relationship by generating a new son-parent key depending on the old map structure. For example, the new data object includes an E depending on C, we can simply add <E:C> to the hash map. Then, if a query request E, we can generate the complete route by tracing back E through C and A to '0'.

If the new data is a totally different data structure, then the new data mapping can be simply added to depend on '0'. For example, [F, G, H, I] can be added to depend on '0' without influencing the original data mapping.

Advantageously, the key form avoids storing repetitive data structures, which saves the storage space. More importantly, this son-parent structure is adaptable for an application that different data structures are used. For example, in the website like Facebook, tens of thousands of data are generated which may comprise different data structures. The generic form and the key form are well suitable for providing the complete route for the keys, which enhances the efficiency of the query function of MongoDB.

C. Experiment

1) Implementation

To experiment our innovative idea for improving the effectiveness and efficiency of MongoDB query function, we use python to create and manage the hash map. By importing the library PyMongo, we achieve interacting with MongoDB in python.

As mentions above, we use the data set as shown in APPENDIX (1). APPENDIX (1) contains only one object out of multiple objects. It should be noted that the actual implementation can contain data sets having different data structures.

When the new data set is required to store into MongoDB, the index system generates the hash map containing generic form and key form. We use the python code as shown in APPENDIX (2) in this step. The resulting Hash map is shown in the tables of APPENDIX (3).

Then the complete route of a standard query is generated by running the python script in APPENDIX (4). We further use the python script in APPENDIX (5) to generate the standard query that can be forwarded to MongoDB to achieve searching a target object.

2) Test Result

a) Testing code in appendix (4)

- (1) Choose "ascending" to search for its precise position based on index.

Result:

```

0
metadata
jsonQuery
  
```

order
order[0]
ascending

Therefore, the precise position (complete route) for “ascending” is:

Metadata -> jQuery -> order -> order[0] -> ascending

- (2) Choose “flags” to search for its precise position based on index.

Result:

0
grants
grants[0]
flags

Therefore, the precise position for “flags” is

grants -> grants[0] -> flags

- (3) Choose “flags1” to search for its precise position based on index. (In order to distinguish flags from other keys named flags, we add suffix “1” as the mark)

Result:

0
flags1

b) Find specific element in MongoDB

- (1) insert the json file into MongoDB

mongoimport --db test --collection test --file test.json

- (2) find a specific element without using index system in this paper

- example 1: we want to find "flags" : ["public"]

Result:

```
|> db.test.findOne({"flags" : ["public"]})  
null
```

Without providing the precise route, the output is null, which means that "flags": ["public"] is not found by the MongoDB.

- Example 2 :we want to find "ascending" .

Result:

```
|> db.test.findOne({"ascending" :false})  
null
```

- (3) find specific element with index in MongoDB

- example 1: "flags" : ["public"]

The complete route, grants -> grants[0] -> flags, is calculated by the python script in Appendix (3).

Input (generated by python script in Appendix(5):

```
|> db.test.findOne({"grants" : {$elemMatch :{  
[...           "flags" : ["public"]  
[...           }]  
[...       ]})
```

Output: (generated by MongoDB)

```
{  
  "_id" : ObjectId("56eef8a7c69f8a18f11fe46"),  
  "grants" : [  
    {  
      "inherited" : true,  
      "type" : "viewer",  
      "flags" : [  
        "public"  
      ]  
    }  
  ]  
}
```

- example2: "ascending" : false

The complete route, metadata -> jQuery -> order -> order[0] -> ascending, is calculated by the python script in Appendix (3).

Input:

```
|> db.test.findOne({ "metadata.jquery.order" :  
[...           {$elemMatch :  
[...               {"ascending" : false  
[...               }  
[...           ]})
```

Output:

```
{  
  "_id" : ObjectId("56ef327f7c69f8a18f11fe48"),  
  "metadata" : {  
    "renderTypeConfig" : {  
      "visible" : {  
        "table" : true  
      }  
    },  
    "availableDisplayTypes" : [  
      "table",  
      "fatrow",  
      "page"  
    ],  
    "rowLabel" : "Complaints",  
    "jquery" : {  
      "order" : [  
        {  
          "ascending" : false,  
          "columnFieldName" : "date_received"  
        }  
      ]  
    },  
    "rdfSubject" : "0"  
  }  
}
```

Compared to the result of two examples in (2) which fails to return any document object, MongoDB successfully find out the key-value "flags" : ["public"], and returns the corresponding document object. We have simplified the data in order to visualize the result in this report.

3) Conclusion

In MongoDB, when we want to find the specific element in the json file, especially the complex nested array json file, we may merely know the internal key and value for this

element. However, based on this conditions, the element can not be found effectively and we may even need to check it by self. Therefore, we just build the index with elements to provide the whole position of elements in the json file. With index, we can find it more efficiently.

IV. ANALYSIS ON COMBINING MYSQL AND MONGODB

A. Introduction

As the popularity of NoSQL grows, some e-commerce applications prefer a combination of MongoDB and MySQL. For example, the user-defined blog, which includes different attributes, is good for MongoDB's flexible data model. On the other hand, the banking system, which requires complex transactions, would likely be built on MySQL [23]. In this paper, the database of P2P-lending system will be redesign based on relational database and document oriented database to improve the searching efficiency and transaction security. The data will be divided into two different structured dataset. Part of dataset will be reorganized and moved to MongoDB, while other part may be left as original structure or modified based on JSON format document in MySQL.

The MySQL and MongoDB are both popular open source database manage software. MySQL is relational database management system and it stores data in tables. Database schema should be pre-defined based on system requirements. In MySQL, related information may be stored in separate tables, but associated through the use of joins. In this way, data duplication is minimized [23]. MySQL is good for complex, multi-row transactions, but not fit for searching or updating semi-structured dataset. MongoDB is document oriented database to stores data in JSON-like documents and can vary its structure. In MongoDB, related information is stored together for fast query access [23]. MongoDB is for organizing semi-structured dataset, but not better for multi-document transactions.

Compared the feature between MySQL and MongoDB, they have both advantage and disadvantage for certain development situation. MongoDB is good for storing big data. The big data has common features that they are large-scaled and semi-structured. To make efficient use of these features, MongoDB ignores some processes in relational database such as pre-designed table, join query and some ACID transaction properties, because it will take much time for responding to these processes [24]. However, relational data model is necessary for designing many-to-many relation. Besides, transaction is widely used in some security operation like updating balance or security information. It is very important to have ACID transaction properties in case that the database rollback operation after it breaks down. So MySQL, as a relational database is fit for these applications.

B. The original database structure

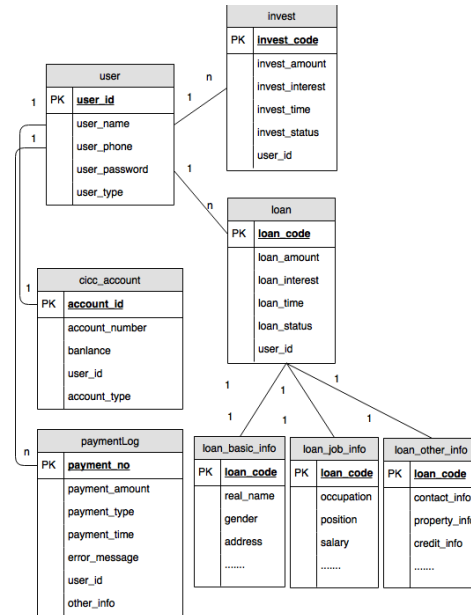


Fig.4.1 Relational Data Model

The original data in this paper is from a P2P-lending financial platform, which is for financier to make a loan and investor to invest money. The data include user information (investor and financier), loan information, investment information and payment information. Furthermore the user information is basic data about login account and payment account. The loan information records amount and interest, as well as financier detail information for investor to see. The investment information record the user applied investment amount and benefit and so on. The payment information records transfer money and account name when investing and repaying.

1) Data Model

Financiers' data attached to loan is semi-structured because financiers can be either personal or enterprise. Also financiers will fill in multiple information like contacts, property, credits when applying loan. Hence, there is no way to predict the exact amount and structure of data. Only a rough estimate of the structure is known: A list of key-value-pairs [25]. In relational schema According to 'loan_info_basic' in Fig1, one column related to one attribute and if row has no value for attribute, the column is filled with NULL. This schema is structured and need to update table column for semi-structured data. If new dataset 'loan_family_info' is added into 'loan_info_basic', the table structure will be changed. Besides, query must join tables 'loan_info_basic', 'loan_info_job' and 'loan_info_contact' to find all loan information, it take much time on joining operation. So semi-structured is hard to be organized in relational database.

2) Transaction

There are parts of the data model which do require ACID transactions and strong consistency as well as parts which do not require this functionality [25]. For example, any update operation for payment balance should be regarded as ACID transactions. However payment dataset is semi-structure. It will loss transaction properties if stored in MongoDB, while

dataset structure is hard to organized if stored in MySQL. So we should consider the tradeoff between flexible data model and transaction or combine their features to design special data structure, which can be applied to certain situation.

3) Scalability

The module of application has its business logic to generate certain query to database. Developer may change business logic when developing application at early stage. So business logic may influence schema in database. MongoDB store all the relation in one document rather than joining the document, so business logic has more influence on MongoDB than MySQL. For MySQL, Developer can add table for new business without changing original table, then write query to join new table and original table. However for MongoDB, the dataset in Embedded Data Models has fixed relation, and It is hard to reorganize document by adding new subset into original semi-structured dataset. If the dataset is in Normalized Data Models, we have to do multiple queries because MongoDB does not support join.

C. The new database structure

1) Loan Data in Embedded Models

Financier information is distributed into three table called 'loan_info_basic', 'loan_info_job' and 'loan_info_contact'. Three tables have join relation with loan table by foreign key 'loan_code'. According to Figure 3, financier dataset has some special features as below:

- Financier information is semi-structured due to diverse information they fill in when applying loan.
- Financier information show together with loan to make loan data trustworthy for investor, and financier information is always attached to loan.
- Financier information will be audited by staff before published to website, if wrong information inserted, staff will ask financier to change their information and apply again. Financier information in database is rarely changed after published to website, and all the information will be viewed frequently by investor.

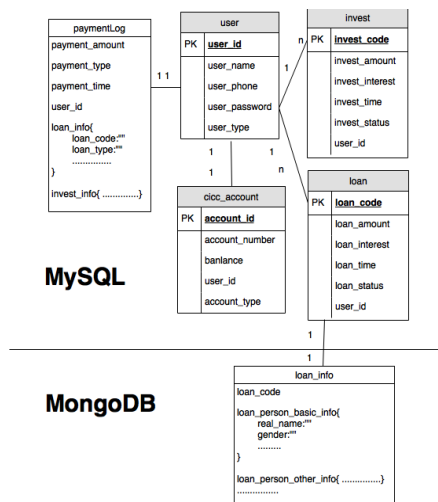


Fig. 4.2 Semi-structured Data Model

Figure 4.3 Financial Webpage
<http://demo.fero.com.cn/invest/investList.htm>

Considering these features, we can take full advantages of MongoDB to improve application performance. MongoDB has two data model for dataset, which are Normalized Data Model and Embedded Data Model. Embedded Data Models allow applications to store related pieces of information in the same database record. As a result, applications may need to issue fewer queries and provides better searching performance. In general, use embedded data models when: you have “contains” relationships between entities. In these relationships the “many” or child documents always appear with or are viewed in the context of the “one” or parent documents [27]. As financier information is semi-structured, inserted with loan information together when applying new loan and viewed with loan information on webpage frequently, so the application will increase searching speed and inserting speed for loan and financier information if financier dataset is stored as Embedded Data Model in MongoDB.

2) Payment Data Combining Transaction with JSON

RDBMS offers many intrinsic qualities that make it desirable for storing sensitive data like ordering and payment information [26]. The operation for loan, investment and payment data is a kind of multi-transaction, so all of relative dataset should be stored in MySQL. However, the payment dataset is semi-structured which is fit for MongoDB. Here is an approach to deal with this tradeoff by combining transaction feature and JSON format feature.

Payment information is in table called 'paymentLog'. Here are dome special features for payment dataset as below:

- Payment information is semi-structured due to various payment methods. For example the payment for

investment is different from lending or transfer money.

- Payment is multi-transaction operation. When repaying money from financier to investors, the application will generate many insert queries for payment data, which is a typical multi-transaction operation.
- Payment information is more frequently to be inserted into database than searched by user, because user can search the investment and loan information, which more briefly describes payment process than payment log information.

In MongoDB, write operation is atomic at document level, and no single write operation can atomically affect more than one document or more than one collection. Normalizing the data would split the data across multiple collections and would require multiple write operations that are not atomic collectively [27]. However, we can take advantages of JSON format document in MongoDB. As JSON data model is semi-structured, 'paymentLog' table can add columns to store these semi-structured dataset as JSON data model, in this way it does not have to define schema first and is easily adapt to changes in the future. In general, JSON data model can be added to RDBMSs, gaining some of the traditional benefits of relational systems in the bargain [28]. So the application can insure the payment transaction while updating payment dataset structure.

3) User Data for Business Logic Scalability

Database architecture to use is one that should be made in the early stages of application design. That decision will be influenced by answering questions about the data, the expected behavior of the application, and ultimately the business requirements [25]. So developer should make it clear that which kind of query or business logic the application needs, then choose appropriate strategy to solve certain problem.

If it is uncertain that what relation will be added or updated in the future, or we prefer scalable business, it is better to design relational schema first in MySQL at early stage. So user login account information is still as relational data model in MySQL even though it does need complex transaction, and it is flexible to write join query to show the relation among user account table, loan table, investment table and payment table, if there is new transfer business from account to account in future, we can design new table for new business and query joining these tables with user login account table. However, we can also move user login account information to MongoDB as Normalized Data Models, but it does not improve much searching speed, because MongoDB will do multiple queries on searching key in login account first and then searching new document by this key. Also it takes more time on connecting database across MySQL and MongoDB.

D. Implement

1) Data Migration Module

Before loading data from MySQL to MongoDB, we should analyze which part of data should be migrated, and how to redesign the new data structure. New models must be identical to original models in terms of semantic and

conceptual structure. In general, the dataset with huge data volume should be migrated, and existing relations have to be adequately represented without loss or distortion of data. Here is a framework named NoSQLayer to perform data and model migration from MySQL to NoSQL databases, more specifically MongoDB, in a transparent and automatic manner [29]. According to the principle of NoSQLayer, we write loading.py to reorganize the data from MySQL and load data into MongoDB as below:

```
query = 'select * from loan_info_basic'
cursor.execute(query)
for row in cursor:
    loanInfo = db.loan_info.find_one({'loanCode':row['loanCode']})
    if loanInfo:
        print "loading failed"
    |
    exit()

loanInfoNew = []
query = 'select * from loan_info_basic'
cursor.execute(query)
for row in cursor:
    loanInfo={}
    loanInfo['loanCode']=row['loanCode']
    loanInfo['basicInfo']= initData(row,['id','loanCode'])
    loanInfoNew.append(loanInfo)
db.loan_info.insert(loanInfoNew)
```

Fig. 4.4 Main Scripts for data migration module

2) Business module

According to the business requirement for P2P-lending system, There are some main operation including searching information, applying for loan, investing, effecting loan and repaying loan. So we write modules for these operations as below:

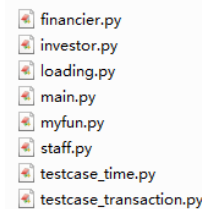


Figure 4.5 Programs for business module

E. Evaluation

1) Searching performance

As the loan_info data model is changed and moved from MySQL into MongoDB, The program called testcase_time.py is designed to compare searching time for financier information in MySQL and MongoDB respectively. If the loan_info dataset stored in MongoDB, we will only write one searching query rather than more queries in MySQL:

```
def findLoanDetailInfo1(cursor):
    query = "select * from loan_info_basic where loanCode ='2'"
    cursor.execute(query)
    loanBasic = cursor.fetchone()
    query = "select * from loan_info_job where loanCode ='201'"
    cursor.execute(query)
    loanJob = cursor.fetchone()
    query = "select * from loan_info_contact where loanCode ="
    cursor.execute(query)
    loanContact = cursor.fetchone()
```

Figure 4.6 Searching Scripts for MySQL

```
def findLoanDetailInfo2(db):
    loanInfo = db.loan_info.find({'loanCode':'201506010010'})
```

Figure 4.7 Searching Scripts for MongoDB

```
t1 = timeit.Timer(lambda:findLoanDetailInfo1(cursor))
print t1.timeit(1)

t2 = timeit.Timer(lambda:findLoanDetailInfo2(db))
print t2.timeit(1)
```

Figure 4.8 Measure Scripts for Time Measurement

TABLE 4.1 The searching time for MySQL and MongoDB

MySQL	MongoDB
0.0046s	0.000075s

As shown in the table, it takes much less time to search financier information in MySQL than MongoDB. Because financier information is stored as Embedded Data Models in MongoDB rather than Relational Data Model in MySQL, so compared with MySQL, MongoDB only need one query to search all the financier information.

2) Transaction performance

One of multi-transaction in P2P-lending system is loan repay process. When money is repaid from financier to investor, Payment information for different users should be inserted into database together at once. The program called testcase_transaction.py is designed to generate multi-transaction result, and the result can be viewed in database directly.

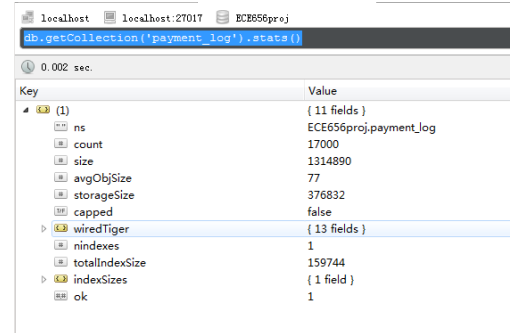
```
def insertPayment1(cursor,cnx):
    for i in range(0,50000):
        print "mysql",i
        paymentId = str(i)
        query = "insert into payment_log (ID,
        cursor.execute(query)
        print("inserting")
        cnx.commit()
```

Figure 4.9 Insert payment data for MySQL

```
def insertPayment2(db):
    payments=[]
    for i in range(0,50000):
        print "mongodb",i
        payment={}
        payment['ID'] = str(i)
        payment['PROJECT_NAME'] = "123"
        payment['PAYMENT_AMOUNT'] = 123
        payments.append(payment)
    print("inserting")
    db.payment_log.insert(payments)
```

Figure 4.10 Insert payment data for Mongo

If we force the application above to exit, we will see different result. In MySQL, there is no data because the multi-transaction rollback inserted data. However, in MongoDB, 17000 of 50000 payment information is inserted as below, so multi-transaction in MySQL is significant to complex business logic.



Key	Value
ns	{ 11 fields }
count	ECE656proj.payment_log
size	17000
avgObjSize	1314890
storageSize	77
capped	376832
wiredTiger	false
nindexes	{ 13 fields }
totalIndexSize	1
indexSizes	159744
ok	{ 1 field }

Figure 4.11 Window for MongoDB Result

F. Conclusion

In this paper, relational data model is reorganized into a semi-structured data model to improve application performance. We have analyzed advantages of MySQL and MongoDB and their feature on data model, transaction and scalability. Then we write scripts to reorganize dataset and implement operations on these datasets based on business requirement. In general, MySQL can guarantees ACID transaction such as payment, while MongoDB can improve searching speed by redesign flexible data model such as the financier information and its related dataset. The Evaluation result confirm that searching speed increased by 98% in MongoDB, and multi-transaction is insured in MySQL.

REFERENCES

- [1] Wikipedia Website for documented-oriented database, https://en.wikipedia.org/wiki/Document-oriented_database
- [2] Buneman, Peter. "Semistructured data." Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. ACM, 1997.
- [3] Lith A, Mattsson J. Investigating storage solutions for large data[J]. Chalmers University of Technology, 2010: 1-70.
- [4] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 4.
- [5] Lakshman A, Malik P. Cassandra: a decentralized structured storage system[J]. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40.
- [6] Abiteboul S, Buneman P, Suciu D. Data on the Web: from relations to semistructured data and XML[M]. Morgan Kaufmann, 2000.
- [7] Buneman P. Semistructured data[C]//Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems. ACM, 1997: 117-121.
- [8] "XML 1.0 Specification". World Wide Web Consortium. Retrieved 2010-08-22.
- [9] "XML and Semantic Web W3C Standards Timeline", <http://www.dblab.ntua.gr/~bikakis/XML%20and%20Semantic%20Web%20W3C%20Standards%20Timeline-History.pdf>, 2012-02-04.
- [10] "W3C DOCUMENT LICENSE", <https://www.w3.org/Consortium/Legal/2002/copyright-documents-20021231>.

- [11] Yergeau F, Bray T, Paoli J, et al. Extensible markup language (XML) 1.0[J]. W3C Recommendation, third edition, February, 2004.
- [12] XML London 2013, <http://xmllondon.com/2013/presentations/fennell/>.
- [13] JSON, https://en.wikipedia.org/wiki/JSON#cite_note-11.
- [14] DB-Engines Ranking, <http://db-engines.com/en/ranking>.
- [15] Hoberman S. Data Modeling for MongoDB: Building Well-Designed and Supportable MongoDB Databases[M]. Technics Publications, 2014.
- [16] MongoDB. Introduction to Replication, <https://docs.mongodb.org/manual/core/replication-introduction/>, MongoDB.
- [17] MongoDB. Introduction to Sharding, <https://docs.mongodb.org/manual/sharding/>, MongoDB.
- [18] Metadata, <http://www.merriam-webster.com/dictionary/metadata>.
- [19] Greenberg J. Understanding metadata and metadata schemes[J]. Cataloging & classification quarterly, 2005, 40(3-4): 17-36.
- [20] Practicing Index in MongoDB, <https://cnodejs.org/topic/555bf91ee684c4c8088a0c0f>.
- [21] Burrows, Michael. "System and method for search, index, parsing document database including subject document having nested fields associated start and end meta words where each meta word identify location and nesting level." U.S. Patent No. 6,963,869. 8 Nov. 2005.
- [22] MongoDB document, <https://docs.mongodb.org/manual/tutorial/query-documents/>.
- [23] <https://www.mongodb.com/compare/mongodb-mysql>
- [24] Aboutorabi, Seyyed Hamid, et al. "Performance evaluation of SQL and MongoDB databases for big e-commerce data." Computer Science and Software Engineering (CSSE), 2015 International Symposium on. IEEE, 2015.
- [25] Ebel, Marius, and Martin Hulin. "Combining relational and semi-structured databases for an inquiry application." Multidisciplinary Research and Practice for Information Systems. Springer Berlin Heidelberg, 2012. 73-84.
- [26] Ploetz, Aaron. A Study of MongoDB and Oracle in an E-commerce Environment. Diss. Regis University, 2013.
- [27] <https://docs.mongodb.org/manual/core/data-model-design/>
- [28] <https://docs.mongodb.org/manual/core/data-modeling-introduction>
- [29] Chasseur, Craig, Yinan Li, and Jignesh M. Patel. "Enabling JSON Document Stores in Relational Systems." WebDB. 2013.
- [30] Rocha, Leonardo, et al. "A Framework for Migrating Relational Datasets to NoSQL." Procedia Computer Science 51 (2015): 2593-2602.

APPENDIX (1)

JSON DATA IN THE EXPERIMENT

(ONLY ONE OUT OF MULTIPLE OBJECTS IS SHOWN)

```
[{
  "id" : "25ei-6bcr",
  "name" : "Credit Card Complaints",
  "averageRating" : 0,
  "createdAt" : 1337199939,
  "displayType" : "table",
  "downloadCount" : 5307,
  "indexUpdatedAt" : 1414647847,
  "newBackend" : false,
  "numberOfComments" : 0,
  "oid" : 2956892,
```

```
  "publicationAppendEnabled" : false,
  "publicationDate" : 1364274981,
  "publicationGroup" : 342069,
  "publicationStage" : "published",
  "rowIdentifierColumnId" : 53173967,
  "rowsUpdatedAt" : 1364274443,
  "rowsUpdatedBy" : "dfzt-mv86",
  "tableId" : 756116,
  "totalTimesRated" : 0,
  "viewCount" : 71106,
  "viewLastModified" : 1364274981,
  "viewType" : "tabular",
  "grants" : [ {
    "inherited" : true,
    "type" : "viewer",
    "flags" : [ "public" ]
  } ],
  "metadata" : {
    "custom_fields" : {
      "TEST" : {
        "CFPB1" : ""
      }
    },
    "renderTypeConfig" : {
      "visible" : {
        "table" : true
      }
    },
    "availableDisplayTypes" : [ "table", "fatrow", "page" ],
    "rdfSubject" : "0",
    "rowIdentifier" : 53173967
  },
  "owner" : {
    "id" : "dfzt-mv86",
    "displayName" : "CFPB Administrator",
    "roleName" : "publisher",
    "screenName" : "CFPB Administrator",
    "rights" : [ "create_datasets", "edit_others_datasets",
      "edit_nominations", "approve_nominations",
      "moderate_comments", "manage_stories", "feature_items",
      "change_configurations", "view_domain",
      "view_others_datasets", "create_pages", "edit_pages",
```



```

"view_goals",      "view_dashboards",      "edit_goals",
"edit_dashboards", "manage_provenance",      "view_story",
"view_unpublished_story", "view_all_dataset_status_logs" ]
    },
    "rights": [ "read" ],
    "tableAuthor": {
        "id": "dfzt-mv86",
        "displayName": "CFPB Administrator",
        "roleName": "publisher",
        "screenName": "CFPB Administrator",
        "rights": [ "create_datasets", "edit_others_datasets",
"edit_nominations",      "approve_nominations",
"moderate_comments", "manage_stories", "feature_items",
"change_configurations",      "view_domain",
"view_others_datasets",      "create_pages",      "edit_pages",
"view_goals",      "view_dashboards",      "edit_goals",
"edit_dashboards", "manage_provenance", "view_story",
"view_unpublished_story", "view_all_dataset_status_logs" ]
    },
    "flags": [ "default" ]
}]

```

APPENDIX (2)

PYTHON CODE FOR READING THE JSON FILE AND SETTING UP THE HASH MAP

```

import pandas
import json
import re
import pickle
import sys
sys.setrecursionlimit(100) # 10000 is an
example, try with different values

from collections import OrderedDict

global generic
generic = {}
global Form
Form = {}

'''
to build a dict generic storing the
metadatas have the same name
'''
def Generic(key):
    genericValue = []
    if generic.has_key(key) == 0:
        genericValue.append(key)
        value = key + "1"
        genericValue.append(value)
        generic[key] = genericValue

```

```

else:
    genericValue = generic.get(key)
    count = len(genericValue)
    value = key +str(count)
    genericValue.append(value)
    generic[key] = genericValue

'''
to check whether the value of a metadata
is a dict or a list
'''
def nestedCheck(v):
    if isinstance(v,dict):
        return 1
    elif isinstance(v,list):
        return 2
    else:
        return 0

'''
to build a dict Form storing the meta and
its value
'''
def setHashMap(son,parent):
    if Form.has_key(son) == 0:
        Form[son] = parent

    else:
        Generic(son)
        #print generic.get(son)
        meta = generic.get(son)
        #print meta
        son = meta[(len(meta)-1)]
        #print son
        Form[son] =parent

'''
to build two stacks,stack_k and
stack_v,storing metadata and its value,for
DFS.
'''
def
set_stack(parentx,valuex,stack_k,stack_v
):
    for keyx in valuex:
        setHashMap(keyx,parentx)
        stack_k.append(keyx)
        stack_v.append(valuex.get(keyx))

with open('1.json','r') as data_file:
    stack_v = []
    stack_k = []
    data = json.load(data_file)
    items = data[0]
    for key in items:
        setHashMap(key,0)
        stack_k.append(key)
        stack_v.append(items.get(key))
        while(len(stack_k) != 0):
            parent = stack_k.pop()

```

```

        value = stack_v.pop()
        if nestedCheck(value) == 1:

set_stack(parent,value,stack_k,stack_v)
        elif nestedCheck(value) == 2:
            for i in
range(len(value)):
                valueT = value[i]
                if nestedCheck(valueT)
== 1:
                    parent1 =
str(parent) + str([i])
                    #key2 =
list(Form.keys())[list(Form.values()).in
dex(parent)]
                    #Form[key2] =
parent1

setHashmap(parent1,parent)

set_stack(parent1,valueT,stack_k,stack_v
)

'''
export Form as filename.pickle,
export generic as filename2.pickle
'''
with open('filename.pickle', 'wb') as
handle:
    pickle.dump(Form,
handle,pickle.HIGHEST_PROTOCOL)
    with open('filename2.pickle', 'wb') as
f:
        pickle.dump(generic,
f,pickle.HIGHEST_PROTOCOL)
        #with open('filename2.pickle', 'rb')
as f:
            #b = pickle.load(f)
            #print b
        #with open('filename.pickle', 'rb') as
handle:
            #b = pickle.load(handle)
            #print b

```

APPENDIX (3)

HASH MAP FOR DATA EXAMPLE IN APPENDIX(1)

<GENERIC FORM>

Duplicate Key	Branch Key
'displayName'	'displayName','displayName1'
'rights'	'rights','rights1','rights2'
'flags'	'flags','flags1'
'roleName'	'roleName','roleName1'
'id'	'id','id1','id2'
'screenName'	'screenName','screenName1'

<KEY FORM>

KEY (Son Key)	VALUE (Parent Key)
'grants[0]'	'grants'
'publicationStage'	0
'newBackend'	0
'visible'	'renderTypeConfi g'
'grants'	0
'publicationDate'	0
'viewLastModified'	0
'owner'	0
'table'	'visible'
'id1'	0
'id'	'owner'
'createdAt'	0
'availableDisplayTypes'	'metadata'
'publicationAppendEnabled'	0
'flags1'	0
'roleName1'	'tableAuthor'
'id2'	'tableAuthor'
'order[0]'	'order'
'publicationGroup'	0
'displayName1'	'tableAuthor'
'displayType'	0
'rowsUpdatedBy'	0
'inherited'	'grants[0]'
'flags'	'grants[0]'
'type'	'grants[0]'
'tableId'	0
'screenName'	'owner'
'columnFieldName'	'order[0]'
'jsonQuery'	'metadata'
'description'	0
'oid'	0
'rdfSubject'	'metadata'
'ascending'	'order[0]'
'rights1'	0
'rights2'	'tableAuthor'
'name'	0
'viewCount'	0
'numberOfComments'	0
'displayName'	'owner'
'screenName1'	'tableAuthor'
'rights'	'owner'
'metadata'	0
'rowLabel'	'metadata'
'totalTimesRated'	0
'renderTypeConfig'	'metadata'
'indexUpdatedAt'	0
'downloadCount'	0
'roleName'	'owner'
'rowsUpdatedAt'	0
'viewType'	0
'order'	'jsonQuery'
'tableAuthor'	0
'averageRating'	0

APPENDIX (4)

PYTHON CODE FOR GENERATING COMPLETE ROUTE FOR A PARTICULAR KEY

```
import pandas
import json
import re
import sys
import re
import pickle

enterinput = raw_input()
output = []
metadata = enterinput

with open('filename.pickle', 'rb') as handle:
    b = pickle.load(handle) #import filename.pickle as b,referencing to Form
    #print b
    with open('filename2.pickle', 'rb') as f:
        d = pickle.load(f) #import filename2.pickle as b,referencing to generic

        if b.has_key(metadata) == 0:
            print "error:this is not such key."
        else:
            output.append(metadata)
            #print output

            while metadata != 0: # to check the metadata reach to outer
                if b.has_key(b.get(metadata)) == 1:
                    metadata = b.get(metadata)

            output.append(metadata)

            #for item in reversed(output):
            #print item

            else:
                if b.get(metadata) == 0:
                    metadata = b.get(metadata)

            output.append(metadata)
            else:
                #the metadata has the same value
                metadata = metadata[:-3]

            output.append(metadata)
```

```
#else:
#break
```

```
for item in reversed(output):
#export the output reversely
```

```
PRINT ITEM
```

APPENDIX (5)

PYTHON CODE FOR GENERATING STANDARD QUERY FOR MONGODB

```
import pandas
import json, ast
import re
import sys
import re
import pickle
import pymongo
from bson.json_util import dumps
from pymongo import MongoClient
client = MongoClient() #connect mongodb
db = client.test
post = db.test

enterinput = raw_input()
enterinput = eval(enterinput)
output = []
metadata = enterinput[0]
metadataValue = enterinput[1]

with open('filename.pickle', 'rb') as handle:
    b = pickle.load(handle) #import filename.pickle as b,referencing to Form
    #print b
    with open('filename2.pickle', 'rb') as f:
        d = pickle.load(f) #import filename2.pickle as b,referencing to generic

        if b.has_key(metadata) == 0:
            print "error:this is not such key."
        else:
            output.append(metadata)
            #print output

            while metadata != 0: # to check the metadata reach to outer
                if b.has_key(b.get(metadata)) == 1:
                    metadata = b.get(metadata)

            output.append(metadata)

            #for item in reversed(output):
            #print item

            else:
                if b.get(metadata) == 0:
                    metadata = b.get(metadata)

            output.append(metadata)
            else:
                #the metadata has the same value
                metadata = metadata[:-3]

            output.append(metadata)
```



```

output.append(metadata)

        else:
            if b.get(metadata)
== 0:
                metadata =
b.get(metadata)

output.append(metadata)
        else:
#the metadata has the same value
                metadata =
metadata[:-3]

output.append(metadata)

##MongoDB export
length = len(output)
output = output[::-1]
Str = ""
for items in output:
    Str = Str + str(items)
m = re.search(r"\[|\]", Str)
#judge whether output include array

string = ""
i = 1
if m == None: #the normal
condition without nest
    while(i != length):
#concatenate the string
        item = output[i]
        string = string +
str(item) + "."
        i = i+1
        normal = string[:-1]
        value =
post.find_one({normal : metadataValue})
    else: # the condition with
nest
        while(i != (length-2)):
#concatenate the string
            item = output[i]
            string = string +
str(item) + "."
            i = i+1
            nest = string[:-1]
            value =
post.find_one({nest : {"$elemMatch" :
{output[-1] : metadataValue}}})

print dumps(value) #convert
bson to json

```