УДК 004.65

Афонін А. О., Зважій Д. В.

ОСОБЛИВОСТІ ВИКОРИСТАННЯ ДОКУМЕНТО-ОРІЄНТОВАНИХ БАЗ ДАНИХ НА ПРИКЛАДІ MONGODB

Розглянуто документо-орієнтовану базу даних MongoDB. Проаналізовано архітектуру цієї БД та особливості реалізації. На підставі результатів проведеного аналізу запропоновано конкретні сценарії використання цієї БД.

Ключові слова: СКБД, NOSQL, MongoDB, бази даних, документо-орієнтовані.

Вступ

Довгий час реляційні бази даних вважалися єдиною альтернативою. Проте з розвитком Інтернету, збільшенням обсягу інформації та кількості пристроїв почали з'являтися альтернативні *NoSQL* рішення. Спочатку це були прості сховища типу ключ-значення, а потім більш складні: графо-, атрибуто- та документо-орієнтовані. Усі вони були налаштовані на те, щоб запропонувати користувачу нові можливості, які не можуть гарантувати традиційні *SQL* рішення. Одна з таких систем – це документо-орієнтована база даних *MongoDB*.

Особливості використання MongoDB

MongoDB – це система керування базами даних із сімейства *NoSQL*, в основі якої лежить документо-орієнтована модель. На цьому варто зосередити додаткову увагу. Документо-орієнтована модель передбачає, що вся інформація зберігатиметься у вигляді документів формату *BSON*, який походить від *JSON* і розшифровується як *javascript object notation*. *BSON* – це бінарний *JSON*, цей формат дозволяє легко описувати об'єкти та інші структури даних та може бути з легкістю прочитаний людиною [1]. Документи зберігаються в колекціях. Якщо проводити паралель з реляційними базами даних, то колекції – це таблиці, документи – кортежі, а стовпчики – атрибути. Перші особливості MongoDB можна помітити вже на етапі представлення даних. Щоб краще їх зрозуміти, змоделюємо таку ситуацію: нам потрібно організувати зберігання даних соціальної мережі. Інформація, яка нас цікавить, – це адреса статті, коментарі користувачів до неї, хештеги та зображення. Спроектуємо *ER-model* для цієї предметної області. Провівши нормалізацію, отримаємо модель предметної області, зображеної на рис. 1.

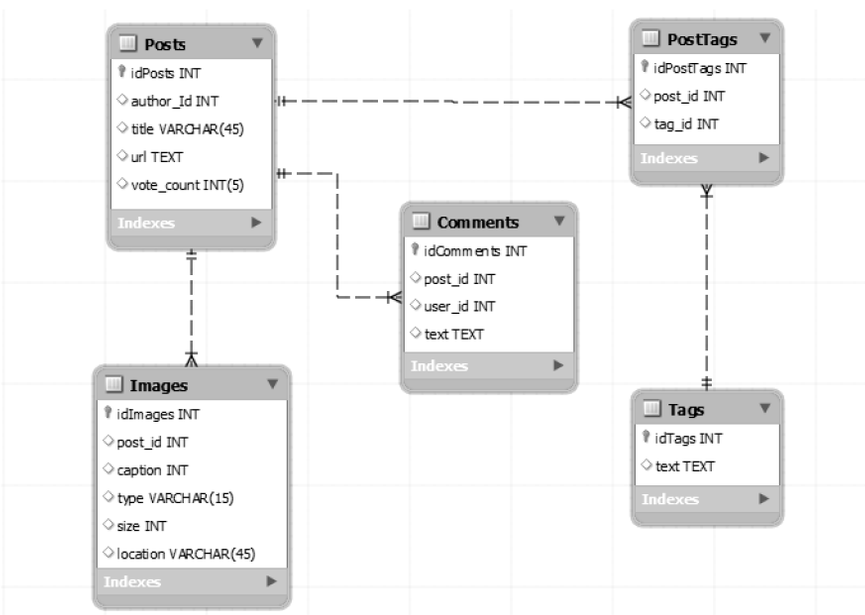

Рис. 1. Модель предметної області

Нормалізація дозволила нам побороти надлишковість даних, але за неї доведеться платити. Щоб відобразити інформацію про пост, нам доведеться виконати з'єднання ледве не всіх наших таблиць. Запит виглядатиме щось на зразок цього:

```
SELECT *
FROM Posts INNER JOIN Images ON Posts.
idPost=Image.post_id
INNER JOIN Comments ON Posts.
idPost=Comments.post_id
INNER JOIN PostTags ON Posts.
idPost=PostTags.post_id
INNER JOIN Tags ON PostTags.tags_id=Tags.
idTags
WHERE Tags.text="politics" AND Post.
vote_count>10;
```

Тут ми знаходимо всю інформацію про пости з тегом «politics» та кількістю голосів більше десяти. Цей запит є занадто ресурсоємним та громіздким через операцію з'єднання таблиць, а враховуючи, що система отримуватиме багато схожих запитів від наших користувачів, це перетвориться на проблему.

Спроектуємо цю ж саму предметну область, використовуючи *MongoDB*.

Ось як виглядатиме наш документ:

```
{ _id: ObjectID('4bd9e8e17cefd644108961bb'),
        title: 'Adventures in Databases',
        url: 'http://example.com/databases.txt',
Listin author: 'msmith',
        vote_count: 20,
        tags: ['databases', 'mongodb', 'indexing'],
        image: {
                url: ,http://example.com/db.jpg',
                caption: '',
                type: 'jpg',
                size: 75381,
                data: "Binary"
        },
        comments: [
                { user: 'bjones',
                  text: 'Interesting article!'
                },
                { user: 'blogger',
                  text: 'Another related article is at
                  http://example.com/db.txt'
                }
        ]
}
```

Неважко помітити, що документ – це фактично множина атрибутів та їхніх значень. *MongoDB* дозволяє зберігати різні типи бінарних даних: від простих до складних, таких як масиви, зображення, відео та навіть інші документи. Наприклад, атрибут ***comments***. Це масив, комірками якого є структури, які складаються зі стрічкового атрибута ***user*** та ***text***. Документо-орієнтована модель дозволяє працювати з даними, які агреговані в цілісний об'єкт (у нашому випадку один документ – це повна структурована інформація про одну новину в соціальній мережі). Уся інформація (від коментарів до хештегів) – це один об'єкт бази даних. Така модель даних дає змогу писати лаконічні, але потужні запити. Ось *MongoDB* запит, аналогічний до згаданого вище SQL запиту:

```
db.posts.find({'tags': 'politics', 'vote_count':
{'$gt': 10}});
```

Цей запит знайде в нашій базі **db** колекцію під назвою **posts** та виконає пошук за вказаними критеріями. Результатом будуть документи, що задовольняють цю інформаційну потребу.

Іноді при проектуванні бази даних ми не можемо чітко сказати, скільки атрибутів матиме об'єкт. Для реляційної моделі даних це проблема. *MongoDB* дає краще рішення. Вона дозволяє безболісно та динамічно змінювати структуру об'єктів, зберігати документи різної структури в одній колекції.

Основні сценарії використання MongoDB

Логування та кеш

MongoDB часто використовують як допоміжну базу даних. Швидка операція вставки якісно вирізняє її серед інших конкурентів. Дуже часто в проектах доводиться зберігати логи виконаних операцій. *MongoDB* легко та ефективно з цим упорається за допомогою своєї гнучкої моделі зберігання даних. Також іноді виникає потреба в зберіганні кешу та проміжних обчислень. Зазвичай ці дані мають різну структуру, а проектувати під них окреме сховище не зовсім доцільно. У таких випадках *MongoDB* є чудовим рішенням.

Для прикладу змоделюємо типову ситуацію, коли система працює з транзакціями. Їй потрібно зберігати лог операцій. У цьому випадку найкращим чином підходить *MongoDB* (можливість уникнути затратного об'єктно-реляційного відображення, відсутність будь-якого роду проектування, можна просто використати структуру об'єкто-орієнтованої моделі, висока швидкість вставки в колекцію). У нас є клас Transaction з такими атрибутами, як: id, source, destination, value, status, transactionDate.

```
public class Transaction {
  private int id;
  private String source;
  private String destination;
```

```
    private int value;
    private String status;
}
```

Далі в нас є клас синглтон *LogSaver*, який вміє зберігати наші транзакції. Його атрибути:

```
    private static LogSaver logMaster;
    private static final String COLLECTION_
NAME="logs";
    private static MongoClient loggerClient;
    private static DB mongoDB;
```

logMaster – це одиничний представник класу LogSaver.

COLLECTION_NAME – це назва колекції, в яку ми будемо зберігати наші операції.

loggerClient – це клас, який створюватиме зв'язок із сервером MongoDB, він за замовчуванням розташований на localhost:27017.

mongoDB – це клас самої бази даних.

Для того щоб додати транзакцію до бази даних MongoDB, ми створимо метод insertTransaction.

```
    public void insertTransactions(Transaction
transaction){
   DBCollection workingCollection = mongoDB.
getCollection(COLLECTION_NAME);
   BasicDBObject document = new
BasicDBObject();
   document.put("_id",transaction.getId());
   document.put("source",transaction.getSource());
   document.put("destination",transaction.
getDestination());
   document.put("value",transaction.getValue());
   document.put("status",transaction.getStatus());
   document.put("date",transaction.
getTransactionTime());
   workingCollection.insert(document);
}
```

На вхід нашого методу приходить об'єкт класу *Transaction*. *DBCollection* – це клас, який представляє колекцію *Mongo*. *BasicDBObject* – це наш майбутній документ. За допомогою методу ***put*** ми вставляємо значення атрибутів нашої транзакції у форматі ключ-значення. Ключ являє собою назву атрибута. Бібліотека *MongoDB* потурбується про те, щоб перетворити *BasicDBObject* у *Binary javaScript object (BSON),* а метод ***insert*** вставить її до обраної нами колекції. Для того щоб мати тестові дані, скористаємося нашим методом *insertTransaction* і додамо декілька документів до нашої колекції. На запит показати всі документи нашої колекції отримаємо:

```
{ "_id" : 7 , "source" : "Albrighton" , "destination" :
"Johnson" , "value" : 100 , "status" : "Successful" ,
"date" : { "$date" : "2015-05-18T16:22:25.832Z"}}
{ "_id" : 1 , "source" : "Albrighton" , "destination" :
"Stockton" , "value" : 101 , "status" : "Successful" ,
"date" : { "$date" : "2015-05-18T18:33:07.490Z"}}
{ "_id" : 2 , "source" : "Stockton" , "destination" :
"Johnson" , "value" : 102 , "status" : "Successful" ,
"date" : { "$date" : "2015-05-18T18:33:09.562Z"}}
{ "_id" : 3 , "source" : "Stockton" , "destination" :
"Albrighton" , "value" : 103 , "status" : "Failed" ,
"date" : { "$date" : "2015-05-18T18:33:09.941Z"}}
{ "_id" : 4 , "source" : "Johnson" , "destination" :
"Albrighton" , "value" : 104 , "status" : "Successful" ,
"date" : { "$date" : "2015-05-18T18:33:09.944Z"}}
{ "_id" : 5 , "source" : "Johnson" , "destination" :
"Stockton" , "value" : 105 , "status" : "Failed" , "date" :
{ "$date" : "2015-05-18T18:33:09.947Z"}}
```

Так само легко ми можемо додати метод *printTransactionsByStatus*, який дозволить нам вивести всі транзакції за статусом.

```
    public void printTransactionsByStatus(String
status){
   DBCollection workingCollection = mongoDB.
getCollection(COLLECTION_NAME);
   BasicDBObject searcher = new
BasicDBObject();
   searcher.put("status",status);
   DBCursor cursor = workingCollection.
find(searcher);
   while (cursor.hasNext()){
      System.out.println(cursor.next());
   }
}
```

Результат для атрибута **status** рівного "failed":

```
{ "_id" : 3 , "source" : "Stockton" , "destination" :
"Albrighton" , "value" : 103 , "status" : "Failed" ,
"date" : { "$date" : "2015-05-18T18:33:09.941Z"}}
{ "_id" : 5 , "source" : "Johnson" , "destination" :
"Stockton" , "value" : 105 , "status" : "Failed" ,
"date" : { "$date" : "2015-05-18T18:33:09.947Z"}}
```

У цьому методі ми використовуємо клас із попереднього методу, а саме *DBCollection*. Річ у тому, що запити в *MongoDB* працюють у рамках однієї заданої колекції. У нашому випадку – це колекція ***logs***. Далі ми створюємо об'єкт класу *BasicDBObject* як критерій пошуку. Результат нашого пошуку буде містити об'єкт ***cursor*** класу *DBCursor*, який має структуру ітератора. Далі ми виводимо отримані результати на екран.

Також у класі *LogSaver* передбачений метод *closeConnection*.

```
public void closeConnection(){
   loggerClient.close();
}
```

Він відповідає за звільнення ресурсів *Mongo*. Також ми можемо запрограмувати декілька

додаткових методів, які допоможуть нам отримувати потрібну інформацію. Наприклад, отримати останніх 10 транзакцій *getLastNTransactions*.

```
public void getLastNTransactions(int number){
   DBCollection workingCollection = mongoDB.
getCollection(COLLECTION_NAME);
   DBCursor cursor = workingCollection.find().
sort(new BasicDBObject("date",-1)).
limit(number);
   while (cursor.hasNext()){
      System.out.println(cursor.next());
   }
}
```

Результат для n дорівнює 4

```
{ "_id" : 5 , "source" : "Johnson" , "destination" :
"Stockton" , "value" : 105 , "status" : "Failed" , "date" :
{ "$date" : "2015-05-18T18:33:09.947Z"}}
{ "_id" : 4 , "source" : "Johnson" , "destination" :
"Albrighton" , "value" : 104 , "status" : "Successful" ,
"date" : { "$date" : "2015-05-18T18:33:09.944Z"}}
{ "_id" : 3 , "source" : "Stockton" , "destination" :
"Albrighton" , "value" : 103 , "status" : "Failed" ,
"date" : { "$date" : "2015-05-18T18:33:09.941Z"}}
{ "_id" : 2 , "source" : "Stockton" , "destination" :
"Johnson" , "value" : 102 , "status" : "Successful" ,
"date" : { "$date" : "2015-05-18T18:33:09.562Z"}}
```

Цей метод дуже схожий на попередній. Різниця в тому, що ми використовуємо метод ***sort*** по атрибуту ***date***. Атрибут-1 гарантує нам обернений порядок сортування. Також можна зробити *MapReduce* функцію, яка дозволить вести статистику успішних та неуспішних транзакцій наших користувачів за певний проміжок часу.

```
public void getTransactionStatistic(String status){
   DBCollection workingCollection = mongoDB.
getCollection(COLLECTION_NAME);
   String mapFunction ="function(){" +
         "emit(this.source,this.status);" +
         "}";
   String reduceFunction ="function(key, values) {" +
         "var count = 0;" +
         "var st "\"+status+"\";" +
         "for(var i=0;i<values.length;i++) {" +
         "   if(values[i]==st)
count++;" +
         "}" +
         "return count;" +
         "}";
   MapReduceCommand cmd = new MapReduce
Command(workingCollection, mapFunction,
reduceFunction,
         null, MapReduceCommand.OutputType.
INLINE, null);
   MapReduceOutput out = workingCollection.
mapReduce(cmd);
   for (DBObject o : out.results()) {
      System.out.println(o.toString());
   }
}
```

Результат:

```
{ "_id" : "Albrighton" , "value" : 2.0}
{ "_id" : "Johnson" , "value" : 1.0}
{ "_id" : "Stockton" , "value" : 1.0}
```

Функція ***map*** створює пари ключ-значення з атрибутів ***source*** як ключа та ***status*** як значення. Функція **reduce** проходить по вибірці значень ***values*** та порівнює зі стрічкою st. У разі успіху збільшуємо лічильник на 1. Якщо змінити значення ***status*** на *Failed,* отримаємо:

```
{ "_id" : "Albrighton" , "value" : 0.0}
{ "_id" : "Johnson" , "value" : 1.0}
{ "_id" : "Stockton" , "value" : 1.0}
```

Також ми можемо все підсумувати і дізнатися, хто скільки грошей отримав після таких маніпуляцій.

```
public void getProfit(){
   DBCollection workingCollection = mongoDB.
getCollection(COLLECTION_NAME);
   String mapFunction ="function(){" +
         "var obj={val: this.value," +
         "         stat: this.status" +
         "};" +
         "emit(this.destination,obj);" +
         "}";
   String reduceFunction ="function(key, values) {" +
         "var sum = 0;" +
         "var object = {val:0 ," +
         "              stat: \"Failed\"};" +
         "values.forEach( function(value) {" +
         " if(object.stat!=value.stat)
object.val+=value.val;" +
         " " +
         " }" +
         ");" +
         "return {profit:object.val};" +
         "}";
   MapReduceCommand cmd = new MapReduce
Command(workingCollection, mapFunction,
reduceFunction,
         null, MapReduceCommand.OutputType.
INLINE, null);
   MapReduceOutput out = workingCollection.
mapReduce(cmd);
   for (DBObject o : out.results()) {
```

```
        System.out.println(o.toString());
    }
}
```

Результат:

```
{ "_id" : "Albrighton" , "value" : { "profit" : 104.0}}
{ "_id" : "Johnson" , "value" : { "profit" : 202.0}}
{ "_id" : "Stockton" , "value" : { "profit" : 101.0}}
```

Функція *getProfit* дещо відрізняється від попередньої *Map-Reduce* функції. Перша відмінність помітна вже у функції ***map***. Враховуючи, що нас цікавлять успішні транзакції та сума грошей, нам недостатньо одного атрибута, щоб передати обидва значення. Тому ми створюємо структуру, яка міститиме необхідні нам атрибути. У методі ***reduce*** ми створюємо об'єкт ***object*** з аналогічною структурою. І для того, щоб відсіяти непотрібні об'єкти, потрібно встановити атрибуту об'єкта ***object*** значення «Successful» або «Failed». Якщо придивитися, то Map-Reduce схожий на GROUP BY в SQL. Але річ у тому, що завдяки розпаралеленню можна досягти кращих результатів зі швидкодії на надзвичайно великій кількості даних, на що, на жаль, *GROUP BY* розраховувати не може. *MongoDB* надає потужний набір запитів, який нічим не поступається *SQL*, а за рахунок особливої архітектури вони стають ще ефективнішими.

Висновки

У статті розглянуто MongoDB як типового представника NoSQL баз даних, що завдяки своїй особливій архітектурі має багато переваг у порівнянні з традиційними SQL рішеннями. Основні з них: гнучкість схеми, масштабованість та легкість у використанні. Усе це, безумовно, сприяє її швидкій популяризації. Але не можна забувати, що кожен випадок особливий, тому потрібно відповідально підходити до вибору цільової СКБД. Слід чітко розуміти, які саме дані ви будете зберігати, та прогнозувати «вузькі місця», які виникатимуть під час роботи системи. У статті проаналізовано переваги конкретної СКБД, що дасть змогу правильно зробити вибір і досягти максимальної ефективності системи.

Список літератури

1. Banker K. MongoDB in action / K. Banker. – NY : Manning, 2012. – 288 p.
2. Seguin K. The Little MongoDB Book [Electronic resource] / K. Seguin. – 34 p. – Mode of access: http://github.com/karlseguin/the-little-mongodb-book. – Title from the screen.

A. Afonin, D. Zvazhii

SPECIFICS OF USE OF DOCUMENT-ORIENTED DATABASES BASED ON MONGODB EXAMPLE

MongoDB is a typical representative of NoSQL databases. Due to its special architecture, it has many advantages over traditional SQL solutions. Highlights include: flexible scheme, scalability and ease of use. This certainly contributes to its rapid popularization. But do not forget that each case is unique, so you need a responsible approach to the selection of the target database. We need to clearly understand what data you store and predict the "bottlenecks" that will arise during the system operation. Understanding the strengths and benefits of a particular DBMS will help to make the right choices and achieve maximum system efficiency.

Keywords: DBMS, NOSQL, MongoDB, databases, document-oriented.

Матеріал надійшов 20.09.2015