

Лабораторна робота

Тема: Робота з MongoDB.

Мета: Влаштування бази даних. Документи. Встановлення та адміністрування бази даних. Додавання даних. Вибірка та фільтрація. Пагінація та сортування. Індеси. Агрегатні функції. Оператори вибірки. Оновлення даних. Видалення даних. Встановлення посилань у БД. Управління колекцією.

Хід роботи

Влаштування бази даних. Документи

Якщо в реляційних бд вміст складають таблиці, то mongodb база даних складається з **колекцій**.

Кожна колекція має своє унікальне ім'я - довільний ідентифікатор, що складається з не більше ніж 128 різних алфавітно-цифрових символів та підкреслення.

На відміну від реляційних баз даних MongoDB не використовує табличний пристрій із чітко заданою кількістю стовпців та типів даних. MongoDB є документоорієнтованою системою, в якій центральним поняттям є **документ**. Документ можна подати як об'єкт, який зберігає певну інформацію. У певному сенсі він подібний до рядків в реляційних субд, де рядки зберігають інформацію про окремий елемент. Наприклад, типовий документ:

```
1  {
2      "name": "Tom",
3      "surname": "Smith",
4      "age": "37",
5      "company": {
6          "name" : "Microsoft",
7          "salary" : "100"
8      }
9  }
```

Документ представляє набір пар ключ-значення. Наприклад, у виразі "name": "Tom" "name" є ключ, а "Tom" - значення.

Ключі є рядками. Значення можуть відрізнятися за типом даних. У цьому випадку майже всі значення також представляють рядковий тип, і лише один ключ (company) посилається на окремий об'єкт. Усього є такі типи значень:

- **String** : рядковий тип даних (для рядків використовується кодування UTF-8)
- **Array (масив)** : тип даних для зберігання масивів елементів
- **Binary data (двійкові дані)** : тип для зберігання даних у бінарному форматі
- **Boolean** : булевий тип даних, що зберігає логічні значення TRUE або FALSE, наприклад, {"married": FALSE}
- **Date** : зберігає дату у форматі часу Unix
- **Double** : числовий тип даних для зберігання чисел з плаваючою точкою
- **Integer** : використовується для зберігання цілих значень розміром 32 біта, наприклад, {"age": 29}
- **Long** : використовується для зберігання цілих значень розміром 64 біта
- **JavaScript** : тип даних для зберігання коду javascript
- **Min key/Max key** : використовуються для порівняння значень із найменшим/найбільшим елементів BSON
- **Null** : тип даних для зберігання значення Null
- **Object** : об'єкт, що містить набір властивостей
- **ObjectId** : тип даних для зберігання id документа
- **Regular expression** : застосовується для зберігання регулярних виразів
- **Decimal128** : тип даних для зберігання десяткових дробових чисел розміром 128 біт, які дозволяють вирішити проблеми з проблемою точності обчислень при використанні дробових чисел, які представляють тип Double.
- **Timestamp** : застосовується для зберігання часу

На відміну від рядків, документи можуть містити різноманітну інформацію. Так, поруч із документом, описаним вище, в одній колекції може бути інший об'єкт, наприклад:

```
1  {
2      "name": "Bob",
3      "birthday": "1985.06.28",
4      "place" : "Berlin",
5      "languages" :[
6          "english",
7          "german",
8          "spanish"
9      ]
10 }
```

Здавалося б різні об'єкти крім окремих властивостей, але вони можуть бути лише у колекції.

Ще пара важливих зауважень: у MongoDB запити мають регістрозалежність і сувору типізацію. Тобто такі два документи не будуть ідентичними:

```
1  {"age" : "28"}
2  {"age" : 28}
```

Якщо у першому випадку для ключа age визначено значення рядка, то у другому випадку значенням є число.

Ідентифікатор документа

Для кожного документа MongoDB визначено унікальний ідентифікатор, який називається `_id`. При додаванні документа до колекції цей ідентифікатор створюється автоматично. Однак розробник може сам явно задати ідентифікатор, а не покладатися на автоматично генеровані, вказавши відповідний ключ і його значення в документі.

Це поле має мати унікальне значення в межах колекції. І якщо ми спробуємо додати до колекції два документи з однаковим ідентифікатором, то додасться лише один із них, а при додаванні другого ми отримаємо помилку.

Якщо ідентифікатор не заданий явно, MongoDB створює спеціальне бінарне значення розміром 12 байт. Це значення складається з кількох сегментів: значення типу timestamp розміром 4 байти (що представляє кількість секунд з моменту початку епохи Unix), випадкове число з 5 байт та лічильник з 3 байт, який ініціалізований випадковим числом. Така модель побудови ідентифікатора гарантує з високою ймовірністю, що він матиме унікальне значення.

Встановлення та адміністрування бази даних

У цій та подальших статтях ми розглянемо базові операції з даними MongoDB із застосуванням як консольної оболонки **mongosh**, так і графічного клієнта **MongoDB Compass**. Однак у будь-якому випадку на початку роботи з сервером слід не забувати запускати сам сервер - тобто додаток **mongod**.

Починаючи працювати з MongoDB в консольній оболонці **mongosh**, насамперед треба встановити потрібну нам базу даних як поточну, щоб потім її використовувати. Для цього треба використовувати команду **use**, після якої йде назва бази даних. При цьому не важливо, чи існує така бд чи ні. Якщо її немає, MongoDB автоматично створить її при додаванні до неї даних.

Отже, запусимо консольну оболонку **mongosh** і введемо там наступну команду:

```
1 use usersdb
```



Тепер як поточний буде встановлено БД **usersdb**. При цьому не важливо, що така база даних може не існувати: якщо її не існує, то при першій операції вона створюється.

Якщо ви раптом не впевнені, а чи вже існує база даних з такою назвою, то за допомогою команди **show dbs** можна вивести назви всіх наявних бд на КОНСОЛЬ:

```
usersdb> show dbs  
admin 40.00 KiB  
config 72.00 KiB
```

```
local 72.00 KiB
test 40.00 KiB
usersdb>
```

Зверніть увагу, що в списку баз даних ще немає бд `usersdb`, тому що я з нею ще не проводив жодних операцій.

Для бази даних можна задати будь-яке ім'я, але є деякі обмеження. Наприклад, в імені не повинно бути символів `/, \, ., ", *, <, >, :, |, ?, $`. Крім того, імена баз даних обмежені 64 байтами.

Також є зарезервовані імена, які можна використовувати: `local`, `admin`, `config`. Ці імена представляють бази даних, які вже мають за промовчанням на сервері і призначені для службових цілей.

Причому як ви бачите, бд `test` у цьому списку немає, тому що я до неї ще не додав дані.

Окрім баз даних, ми можемо переглянути список усіх колекцій у поточній бд за допомогою команди

```
1 show collections
```

Отримання статистики

Використовуючи команду `db.stats()`, можна отримати статистику поточної бази даних. Наприклад, у нас як поточна встановлена база даних `test`:

```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&se...
usersdb> use test
switched to db test
test> show collections
users
test> db.stats()
{
  db: 'test',
  collections: 1,
  views: 0,
  objects: 1,
  avgObjSize: 36,
  dataSize: 36,
  storageSize: 20480,
  indexes: 1,
  indexSize: 20480,
  totalSize: 40960,
  scaleFactor: 1,
  fsUsedSize: 179737477120,
test>
ok: 1
```

Подібним чином ми можемо дізнатися всю статистику щодо окремої колекції. Наприклад, дізнаємося статистику з колекції `users:db.users.stats()`

Додавання даних

Встановивши бд, тепер ми можемо додати до неї дані. Всі дані зберігаються у бд у форматі BSON, який близький до JSON, тому нам також потрібно вводити дані в цьому форматі. І хоча у нас, можливо, на даний момент немає жодної колекції, але при додаванні до неї даних вона автоматично створюється.

Як раніше говорилося, ім'я колекції - довільний ідентифікатор, що складається з не більше ніж 128 різних алфавітно-цифрових символів та підкреслення. У той же час ім'я колекції не повинно починатися з префіксу `system.`, оскільки він зарезервований для внутрішніх колекцій (наприклад, колекція `system.users` містить усі користувачі бази даних). І також ім'я не повинно містити знак долара - `$`.

Для додавання до колекції можуть використовуватися три її методи:

- **insertOne()** : додає один документ
- **insertMany()** : додає кілька документів

Допустимо, ми використовуємо базу даних test. Додамо до неї один документ:

```
1 test> db.users.insertOne({"name": "Tom", "age": 28, languages: ["english",  
    "spanish"]})
```

Документ представляє набір пар ключ-значення. У цьому випадку документ, що додається, має три ключі: name, age, languages, і кожному з них зіставляє певне значення. Наприклад, ключу languages в якості значення зіставляється масив.

Назви ключів можуть використовуватися в лапках, а можуть і без лапок.

Деякі обмеження під час використання імен ключів:

- Символ \$ не може бути першим символом у імені ключа
- Ім'я ключа не може містити символу точки.

При додаванні даних, якщо ми явно не надали значення для поля "_id" (тобто унікального ідентифікатора документа), воно генерується автоматично. Так, після виконання операції додавання консоль виведе згенерований для доданого документа ідентифікатор:

```
test> db.users.insertOne({"name": "Tom", "age": 28, languages: ["english",  
    "spanish"]})  
{  
  acknowledged: true,  
  insertedId: ObjectId("62e27b1b06adfcddf4619fc1")  
}  
test>
```

У відповіді сервера ми отримаємо об'єкт, у якого параметр insertedId буде містити ідентифікатор.

Але, в принципі, ми можемо самі встановити цей ідентифікатор при додаванні даних:

```
1 test> db.users.insertOne({"_id": 123457, "name": "Tom", "age": 28,  
    languages: ["english", "spanish"]})
```

або використовувати для ідентифікатора тип ObjectId

```
1 test> db.users.insertOne({"_id": ObjectId("62e27b1b06adfcddf4619fc6"),  
    "name": "Tom", "age": 28, languages: ["english", "spanish"]})
```

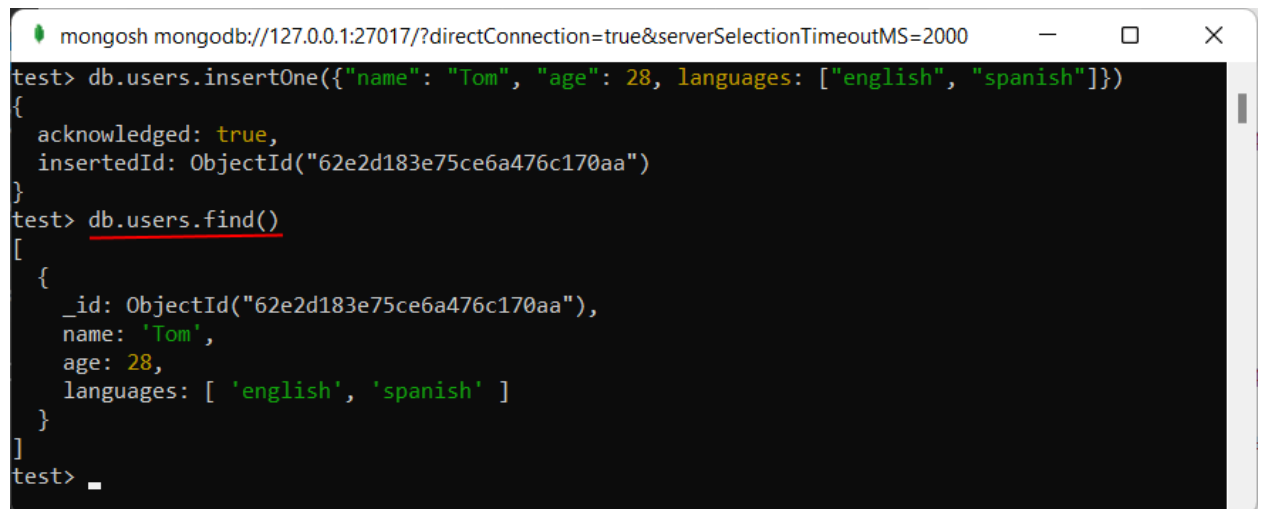
Варто враховувати, що якщо визначення ідентифікатора застосовується тип ObjectId, він повинен містити рядок 12 байт чи рядок з 24 символів.

У разі успішного додавання на консоль буде виведено ідентифікатор доданого документа.

І щоб переконатися, що документ у бд, ми виводимо його функцією `find`.

```
1 test> db.users.find()
```

За умовчанням він виводить усі документи колекції:



```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
test> db.users.insertOne({"name": "Tom", "age": 28, languages: ["english", "spanish"]})
{
  acknowledged: true,
  insertedId: ObjectId("62e2d183e75ce6a476c170aa")
}
test> db.users.find()
[
  {
    _id: ObjectId("62e2d183e75ce6a476c170aa"),
    name: 'Tom',
    age: 28,
    languages: [ 'english', 'spanish' ]
  }
]
test> _
```

Якщо потрібно додати ряд документів, то ми можемо скористатися методом **`insertMany()`**, який приймає масив об'єктів:

```
1 db.users.insertMany([{"name": "Bob", "age": 26, languages: ["english",
  "french"]},
2 {"name": "Alice", "age": 31, languages: ["german", "english"]}])
```

Після додавання консоль виводить ідентифікатори доданих документів:



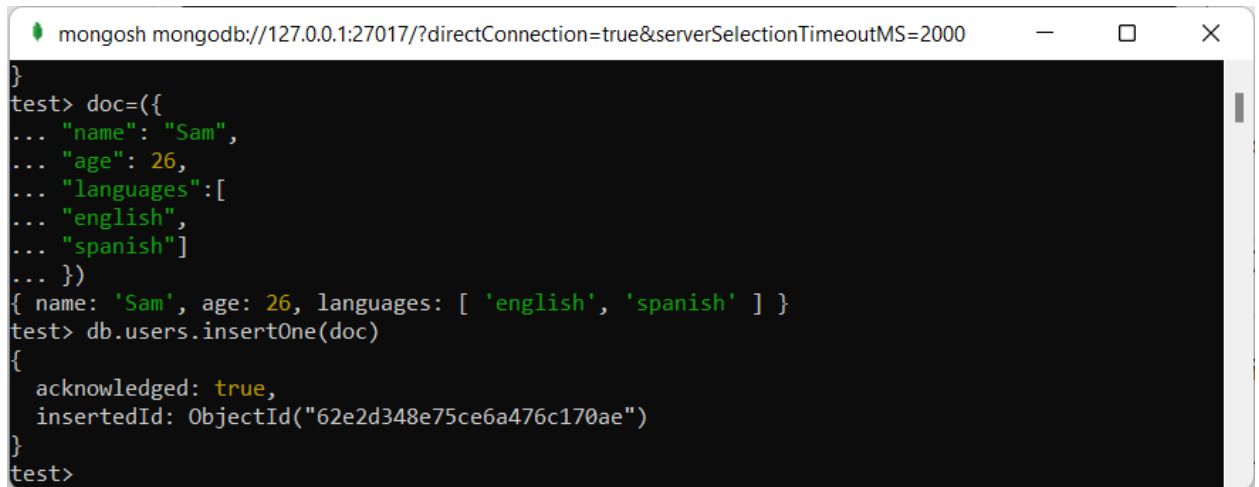
```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
]
test> db.users.insertMany([{"name": "Bob", "age": 26, languages: ["english", "french"]},
... {"name": "Alice", "age": 31, languages: ["german", "english"]}])
{
  acknowledged: true,
  insertedIds: {
    '0': ObjectId("62e2d255e75ce6a476c170ab"),
    '1': ObjectId("62e2d255e75ce6a476c170ac")
  }
}
test> _
```

Є ще один спосіб додавання до бд документа, який включає два етапи: визначення документа (`document = ({ ... })`) і власне його додавання:

```
1 document=({"name": "Bill", "age": 32, languages: ["english", "french"]})
2 db.users.insertOne(document)
```

Можливо, не всім буде зручно вводити в один рядок усі пари ключів та властивостей. Але інтелектуальний інтерпретатор MongoDB на основі

JavaScript дозволяє також вводити і багаторядкові команди. Якщо вираз не закінчено (з точки зору мови JavaScript), і ви натискаєте Enter, то введення наступної частини виразу автоматично переноситься на наступний рядок:



```
mongosh mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000
test> doc={
...   "name": "Sam",
...   "age": 26,
...   "languages": [
...     "english",
...     "spanish"
...   ]
... }
{ name: 'Sam', age: 26, languages: [ 'english', 'spanish' ] }
test> db.users.insertOne(doc)
{
  acknowledged: true,
  insertedId: ObjectId("62e2d348e75ce6a476c170ae")
}
test>
```

Завантаження даних із файлу

Дані бази даних mongodb можна визначати у звичайному текстовому файлі, що досить зручно, оскільки ми можемо переносити або пересилати цей файл незалежно від бази даних mongodb. Наприклад, визначимо десь на жорсткому диску файл **users.js** з таким вмістом:

```
1 db.users.insertMany([
2   {"name": "Alice", "age": 31, languages: ["english", "french"]},
3   {"name": "Lene", "age": 29, languages: ["english", "spanish"]},
4   {"name": "Kate", "age": 30, languages: ["german", "russian"]}
5 ])
```

Тобто тут за допомогою методу `insertMany` додаються три документи в колекцію `users`.

Для завантаження файлу в поточну базу даних застосовується функція `load()`, в яку як параметр передається шлях до файлу:

```
1 load("D:/users.js")
```

В даному випадку передбачається, що файл знаходиться на шляху `"D:/users.js"`.

Вибірка та фільтрація

Найпростішим способом вибірки документів з колекції є використання функції `find()`. Дія цієї функції багато в чому аналогічна до звичайного запиту `SELECT * FROM Table`, який застосовується в SQL і який витягує всі

рядки. Наприклад, щоб отримати всі документи з колекції `users`, створеної в минулій темі, ми можемо використовувати команду:

```
1 db.users.find()
```

```
test> db.users.find()
[
  {
    _id: ObjectId("62e2d183e75ce6a476c170aa"),
    name: 'Tom',
    age: 28,
    languages: [ 'english', 'spanish' ]
  },
  {
    _id: ObjectId("62e2d255e75ce6a476c170ab"),
    name: 'Bob',
    age: 26,
    languages: [ 'english', 'french' ]
  },
  {
    _id: ObjectId("62e2d255e75ce6a476c170ac"),
    name: 'Alice',
    age: 31,
```

Фільтрування даних

Однак якщо нам треба отримати не всі документи, а тільки ті, які задовольняють певну вимогу. Наприклад, ми раніше додали до бази такі документи:

```
1 db.users.insertOne({"name": "Tom", "age": 28, "languages": ["english",
  "spanish"]})
2 db.users.insertOne({"name": "Bill", "age": 32, "languages": ["english",
  "french"]})
3 db.users.insertOne({"name": "Tom", "age": 32, "languages": ["english",
  "german"]})
```

Виведемо всі документи, у яких `name=Tom`:

```
1 db.users.find({name: "Tom"})
```

Такий запит виведе нам два документи, у яких `name=Tom`.

```
test> db.users.find({ім'я: "Том"})
[
  {
    _id: ObjectId("62e2d6a5e75ce6a476c170b3"),
    ім'я: "Том",
    вік: 28,
    мови: [ 'англійська', 'іспанська' ]
  },
  {
    _id: ObjectId("62e2d348e75ce6a476c170ae"),
    _id: ObjectId("62e2d6a5e75ce6a476c170b5"),
    ім'я: "Том",
```

```
    вік: 32, s: [ 'англійська', 'іспанська' ]  
    мови: [ 'англійська', 'німецька' ]  
  }  
] _id: ObjectId("62e2d3d5e75ce6a476c170af"),  
тест>
```

Тепер складніший запит: нам треба вивести ті об'єкти, які мають name=Tom і водночас age=32. Тобто мовою SQL це міг би виглядати так: `SELECT * FROM Table WHERE Name='Tom' AND Age=32`. Даному критерію відповідає останній доданий об'єкт. Тоді ми можемо написати наступний запит:

```
1 db.users.find({name: "Tom", age: 32})
```

Фільтрування за відсутніми властивостями

Якісь документи можуть мати певну властивість, інші можуть її не мати. Що якщо ми хочемо отримати документи, в яких немає певної властивості? У цьому випадку для властивості передається значення **null**. Наприклад, знайдемо всі документи, де відсутня властивість languages:

```
1 db.users.find({languages: null})
```

Або знайдемо всі документи, де name="Tom", але властивість languages не визначена.

```
1 db.users.find({name: "Tom", languages: null})
```

Фільтрування за елементами масиву

Також легко знайти по елементу в масиві. Наприклад, наступний запит виводить усі документи, у яких у масиві languages є english:

```
1 db.users.find({languages: "english"})
```

Ускладнимо запит і отримаємо ті документи, у яких в масиві languages одночасно дві мови: "english" і "german":

```
1 db.users.find({languages: ["english", "german"]})
```

Причому саме у цьому порядку, де "english" визначено першим, а "german" - другим.

Тепер виведемо всі документи, в яких "english" у масиві languages знаходиться на першому місці:

```
1 db.users.find({"languages.0": "english"})
```

Зверніть увагу, що "languages.0" надає складну властивість і тому береться в лапки. Відповідно, якщо нам треба вивести документи, де english на другому

місці (наприклад, ["german", "english"]), то замість нуля ставимо одиницю: "languages.1".

Розглянемо складніший приклад, де елемент масиву представляє складний об'єкт. Допустимо, у нас у базі даних такі документи:

```
1 db.users.insertOne({"name": "Bob", "age": 28, friends: [{"name": "Tim"}, {"name": "Tom"}]})
2 db.users.insertOne({"name": "Tim", "age": 29, friends: [{"name": "Bob"}, {"name": "Tom"}]})
3 db.users.insertOne({"name": "Sam", "age": 31, friends: [{"name": "Tom"}]})
  db.users.insertOne({"name": "Tom", "age": 32, friends: [{"name": "Bob"}, {"name": "Tim"}, {"name": "Sam"}]})
```

Виберемо всі документи, де в масиві friends властивість name першого елемента дорівнює "Bob":

```
1 test> db.users.find({"friends.0.name": "Bob"})
```

Консольний висновок:

```
test> db.users.find({"friends.0.name": "Bob"})
[
  {
    _id: ObjectId("62e39da1c881653067e87901"),
    ім'я: "Тім",
    вік: 29,
    друзі: [ { ім'я: 'Боб' }, { ім'я: 'Том' } ]
  },
  {
    _id: ObjectId("62e39da1c881653067e87903"),
    ім'я: "Том",
    вік: 32,
    друзі: [ { ім'я: 'Боб' }, { ім'я: 'Тім' }, { ім'я: 'Сем' } ]
  }
]
тест>
```

Проекція

Документ може мати безліч полів, але не всі ці поля нам можуть бути потрібними та важливими при запиті. І в цьому випадку ми можемо включити у вибірку лише потрібні поля, використовуючи проекцію. Наприклад, виведемо лише порцію інформації, наприклад, значення полів "age" у всіх документів, у яких name=Tom:

```
1 db.users.find({name: "Tom"}, {age: 1})
```

Використання одиниці як параметра {age: 1} вказує, що запит повинен повернути лише зміст якості age.

```
test> db.users.find({ім'я: "Том"}, {вік: 1})
[
  { _id: ObjectId("62e2d6a5e75ce6a476c170b3"), вік: 28 },
  { _id: ObjectId("62e2d6a5e75ce6a476c170b5"), вік: 32},
  { _id: ObjectId("62e2d799e75ce6a476c170b7"), вік: 28 },
  { _id: ObjectId("62e39da1c881653067e87903"), вік: 32 }
]
тест>
```

І зворотна ситуація: ми хочемо знайти всі поля документа, крім властивості age. В цьому випадку як параметр вказуємо 0:

```
1 db.persons.find({name: "Tom"}, {age: 0})
```

При цьому треба враховувати, що навіть якщо ми зауважимо, що ми хочемо отримати тільки поле name, поле _id також буде включено до результуючої вибірки. Тому, якщо ми не хочемо бачити це поле у вибірці, то треба явно вказати: {"_id":0}

Альтернативно замість 1 та 0 можна використовувати true та false:

```
1 db.users.find({name: "Tom"}, {age: true, _id: false})
```

Якщо ми не хочемо при цьому конкретизувати вибірку, а хочемо вивести всі документи, можна залишити перші фігурні дужки порожніми:

```
1 db.users.find({}, {age: 1, _id: 0})
```

Запит до вкладених об'єктів

Попередні запити застосовувалися до простих об'єктів. Але документи можуть бути дуже складними за структурою. Наприклад, додамо до колекції users наступний документ:

```
1 db.users.insertOne({"name": "Alex", "age": 28, "company":
  {"name": "Microsoft", "country": "USA"}})
```

Тут визначається вкладений об'єкт із ключем company. І щоб знайти всі документи, у яких у ключі company вкладена властивість name=microsoft, нам потрібно використовувати оператор точку:

```
1 db.users.find({"company.name": "Microsoft"})
```

Використання JavaScript

Крім виконання запитів до бази даних, ми можемо виконувати вирази JavaScript. Наприклад, ми можемо створити якусь функцію та застосовувати її:

```
1 function sqrt(n) { return n*n; }
2 sqrt(5)
```

Консольний висновок:

```
test> function sqrt(n) { return n*n; }
[Функція: sqrt]
test> sqrt(5)
25
тест>
```

І подібні функції та вирази JavaScript ми можемо застосовувати у запитах до БД. Наприклад, знайдемо всі документи, де поле age дорівнює $\text{sqrt}(5)+3$:

```
test> db.users.find({age: sqrt(5)+3})
[
  {
    _id: ObjectId("62e2d6a5e75ce6a476c170b3"),
    ім'я: "Том",
    вік: 28,
    мови: [ 'англійська', 'іспанська' ]
  },
  { _id: ObjectId("62e2d76ae75ce6a476c170b6"), ім'я: 'Томас', вік: 28 },
  { _id: ObjectId("62e2d799e75ce6a476c170b7"), ім'я: 'Том', вік: 28 },
  {
    _id: ObjectId("62e39da1c881653067e87900"),
    ім'я: "Боб",
    вік: 28
  }
]
тест>
```

Використання регулярних виразів

Ще однією чудовою можливістю під час побудови запитів є використання регулярних виразів. Наприклад, знайдемо всі документи, у яких значення ключа name починається з літери В:

```
db.users.find({ім'я:/^В\w+/i})
```

Зразковий консольний висновок:

```
test> db.users.find({name:/^B\w+/i})
[
  {
    _id: ObjectId("62e2d6a5e75ce6a476c170b4"),
    ім'я: "Білл",
    вік: 32,
    мови: [ 'англійська', 'французька' ]
  },
  {
    _id: ObjectId("62e39dalc881653067e87900"),
    ім'я: "Боб",
    вік: 28
  }
]
тест>
```

Пошук одиночного документа

Якщо всі документи виймаються функцією `find`, то одиночний документ витягується функцією `findOne`

Наприклад, виберемо один елемент з `name="Tom"`:

```
1 test> db.users.findOne({name: "Tom"})
2 {
3   _id: ObjectId("62e2d6a5e75ce6a476c170b3"),
4   name: 'Tom',
5   age: 28,
6   languages: [ 'english', 'spanish' ]
7 }
8 test>
```

Курсори

Результат вибірки, одержуваної з допомогою функції `find`, називається **курсором**. При необхідності ми можемо передати курсор в окрему змінну:

```
1 var cursor = db.users.find()
```

Курсори інкапсують у собі набори одержуваних з бд об'єктів. Використовуючи синтаксис мови JavaScript та методи курсорів, ми можемо вивести отримані документи на екран і якось їх обробити. Наприклад:

```
1 var cursor = db.users.find()
2
3 while(cursor.hasNext()){
```

```
4     obj = cursor.next();
5     print(obj["name"]);
6 }
```

Курсор має метод **hasNext**, який показує при переборі, чи є ще в наборі документ. А метод **next** витягує поточний документ і переміщає курсор до наступного документа набору. У результаті змінної `obj` виявляється документ, до полів якого ми можемо отримати доступ.

```
test> var cursor = db.users.find()

test> while(cursor.hasNext()){
...  obj = cursor.next();
...  print(obj["name"]);
...}
Том
Боб
Сем

тест>
```

Також для перебору документів у курсорі як альтернатива ми можемо використовувати конструкцію ітератора javascript - **forEach** :

```
1  var cursor = db.users.find()
2  cursor.forEach(function(obj) {
3      print(obj.name);
4  })
```

Пагінація та сортування

MongoDB представляє ряд функцій, які допомагають керувати вибіркою із бд. Одна з них - функція **limit**. Вона задає максимально допустиму кількість одержуваних документів. Кількість передається у вигляді числового параметра. Наприклад, обмежимо вибірку трьома документами:

```
1  db.users.find().limit(3)
```

В даному випадку ми отримаємо перші три документи (якщо в колекції 3 та більше документів). Але що якщо ми хочемо зробити вибірку не спочатку, а пропустивши якусь кількість документів? У цьому нам допоможе функція **skip**. Наприклад, пропустимо перші три записи:

```
1  db.users.find().skip(3)
```


Комбінуючи обидві функції, ми можемо отримати певну кількість документів, починаючи з певного документа. Наприклад, виберемо документи з 4 по 6:

```
1 db.users.find().skip(3).limit(3)
```

MongoDB надає можливості відсортувати отриманий з бд набір даних за допомогою функції **sort**. Передаючи в цю функцію значення 1 або -1, ми можемо вказати, в якому порядку сортувати: за зростанням (1) або за зменшенням (-1). Багато в чому ця функція по дії аналогічна виразу ORDER BY SQL. Наприклад, сортування за зростанням по полю name:

```
1 db.users.find().sort({name: 1})
```

Наприклад, виведемо з бд тільки значення поля "name", відсортувавши їх за зростанням:

```
test> db.users.find({}, {name:1, _id: 0}).sort({name: 1})
[
  { name: 'Bill' },
  { name: 'Bob' },
  { name: 'Sam' },
  { name: 'Tim' },
  { name: 'Tom' },
  { name: 'Tom' },
  { name: 'Tom' },
  { name: 'Tom' },
  { name: 'Tomas' }
]
test>
```

Оператор \$ slice

\$slice є певною мірою комбінацією функцій limit і skip. Але, на відміну від них, \$slice може працювати з масивами.

Оператор \$slice має дві форми:

```
1 $slice: limit
2 $slice: skip, limit
```

Параметр limit вказує на загальну кількість документів, що повертаються. Параметр skip вказує на усунення щодо початку (як функція skip).

Наприклад, у кожному документі визначено масив мов для зберігання мов, якими говорить людина. Їх може бути і 1, і 2, і 3 і більше. І припустимо, раніше ми додали наступний об'єкт:

```
1 db.users.insertOne({"name": "Tom", "age": 32, "languages": ["english",  
"german", "spanish"]})
```

І ми хочемо при виведенні документів зробити так, щоб у вибірку потрапляла лише одна мова з масиву languages, а не весь масив:

```
1 db.users.find ({name: "Tom"}, {languages: {$slice : 1}})
```

Даний запит під час вилучення документа залишить у результаті лише першу мову з масиву languages, тобто в даному випадку english.

```
test> db.users.find ({name: "Tom"}, {languages: {$slice : 1}})  
[  
  {  
    _id: ObjectId("62e3c70079a0a7792a9de20c"),  
    name: 'Tom',  
    age: 32,  
    languages: [ 'english' ]  
  }  
]  
test>
```

Назад: нам треба залишити в масиві також один елемент, але не з початку, а з кінця. У цьому випадку необхідно передати до параметра негативне значення:

```
1 db.users.find ({name: "Tom"}, {languages: {$slice : -1}});
```

Тепер у масиві виявиться "spanish", тому що він перший з кінця в доданому елементі.

Використовуємо одразу два параметри:

```
1 db.users.find ({name: "Tom"}, {languages: {$slice : [-2, 1]}});
```

Перший параметр говорить почати вибірку елементів з кінця (оскільки негативне значення), тобто вибірка йде починаючи з другого елемента з **кінця**, а другий параметр вказує на кількість елементів масиву, що повертаються. У результаті в масиві language виявиться "german"

Індекси

При пошуку документів у невеликих колекціях ми не зазнаємо особливих проблем. Однак коли колекції містять мільйони документів, а нам треба зробити вибірку за певним полем, то пошук потрібних даних може зайняти деякий час, який може виявитися критичним для нашого завдання. І тут нам можуть допомогти індекси.

Індекси дозволяють упорядкувати дані щодо певного поля, що згодом прискорить пошук. Наприклад, якщо ми у своєму додатку або задачі зазвичай виконуємо пошук по полю `name`, то ми можемо індексувати колекцію по цьому полю.

Створення індексу

Для створення індексу використовується функція **`createIndex()`**, в яку передається об'єкт із зазначенням полів, для яких створюється індекс. Наприклад, створення індексу поля `"name"`:

```
1 db.users.createIndex({"name" : 1})
```

При створенні індексу консоль поверне нам назву індексу:

```
test> db.users.createIndex({"name" : 1})
name_1
test>
```

Тобто в прикладі вище було створено індекс з ім'ям `"name_1"` по полю `name`. MongoDB дає змогу встановити до 64 індексів на одну колекцію.

Для створення декількох індексів застосовується функція **`createIndexes()`** - до неї передається масив об'єктів, які встановлюють поля для індексів:

```
1 db.users.createIndexes([{"name" : 1}, {"age": 1}])
```

В даному випадку створюються два індекси – один для поля `name`, інший для поля `age`.

Видалення індексів

Для видалення індексів застосовується функція **`dropIndex()`**, до якої передається ім'я індексу. Наприклад, видалимо вище певний індекс `"name_1"`:

```
1 db.users.dropIndex("name_1")
```

Налаштування індексів

Якщо ми просто визначимо індекс для колекції, наприклад, `db.users.createIndex({"name" : 1})` ми все ще зможемо додавати в колекцію документи з однаковим значенням ключа `name`. Однак, якщо нам потрібно, щоб до колекції можна було додавати документ з одним і тим самим значенням ключа тільки один раз, ми можемо встановити прапор **unique** :

```
1 db.users.createIndex({"name" : 1}, {"unique" : true})
```

При цьому при додаванні унікального індексу в колекції не повинно бути документів, які мають для певного поля однакові значення.

Тепер, якщо ми спробуємо додати до колекції два документи з тим самим значенням `name`, ми отримаємо помилку.

У той же час тут є свої тонкощі. Так, документ може мати ключа `name`. У цьому випадку для документа, що додається, автоматично створюється ключ `name` зі значенням `null`. Тому при додаванні другого документа, в якому не визначено ключа `name`, буде викинуто виняток, оскільки ключ `name` зі значенням `null` вже присутній в колекції.

Також можна задати один індекс відразу для двох полів:

```
1 db.users.createIndex({"name" : 1, "age" : 1})
```

Однак у цьому випадку всі документи, що додаються, повинні мати унікальні значення для обох полів.

Крім того, тут є обмеження. Наприклад, значення поля, яким йде індексація, має бути більше 1024 байт.

Управління індексами

Усі індекси бази даних зберігаються в системній колекції **indexes**. Для звернення до неї ми можемо використовувати функцію `getIndexes`, наприклад, щоб вивести всю інформацію про індекси для конкретної колекції:

```
1 db.users.getIndexes()
```

Ця команда поверне висновок на кшталт наступного:

```
test> db.users.getIndexes()
[
  { v: 2, key: { _id: 1 }, name: '_id_' },
  { v: 2, key: { name: 1 }, name: 'name_1' }
```

```
]
test>
```

Як бачимо, тут для колекції `users` (з бд `test`) визначено 2 індекси: `id` і `name`. Поле `key` використовується для пошуку максимального та мінімального значень, для різних операцій, де треба застосовувати цей індекс. Поле `name` застосовується як ідентифікатор для операцій адміністрування, наприклад, для видалення індексу.

Агрегатні функції

Число елементів у колекції

За допомогою функції `countDocuments()` можна отримати загальну кількість документів у колекції:

```
1 db.users.countDocuments()
```

Якщо нам потрібно дізнатися не сукупну кількість документів у колекції, а лише кількість документів у конкретній вибірці, можна застосовувати функцію **`count()`**. Наприклад, підрахуємо кількість документів, які мають `name=Tom`:

```
1 db.users.find({name: "Tom"}).count()
```

Більше того, ми можемо створювати ланцюжки функцій, щоб конкретизувати умови підрахунку:

```
1 db.users.find({name: "Tom"}).skip(2).count(true)
```

Тут слід зазначити, що за умовчанням функція `count` не використовується з функціями `limit` та `skip`. Щоб їх використати, як у прикладі вище, у функцію `count` треба передати булеве значення `true`

Функція `distinct`

У колекції можуть бути документи, які містять однакові значення для одного або кількох полів. Наприклад, у кількох документах визначено `name: "Tom"`. І нам треба знайти тільки унікальні значення для одного з полів документа. Для цього ми можемо скористатися функцією **`distinct`**. Наприклад, нехай до бази даних додано такі документи:

```
1 db.users.insertOne({"name": "Tom", "age": 38, "languages": ["english",
"spanish"]})
```

```

2 db.users.insertOne({"name": "Bob", "age": 41, languages: ["english",
  "french"]})
3 db.users.insertOne({"name": "Sam", "age": 28, languages: ["english"]})
4 db.users.insertOne({"name": "Tom", "age": 22, languages: ["english",
  "german"]})

```

Виведемо всі унікальні значення по полю "name":

```

1 test> db.users.distinct("name")
2 [ 'Bob', 'Sam', 'Tom' ]
3 test>

```

Функції **min** та **max**

Функції **min** і **max** встановлює для певного поля мінімальне значення попадання у вибірку. При цьому ці функції можуть використовувати лише ті поля, для яких встановлені індекси. Наприклад, візьмемо вище випереджену колекцію `db.users` та визначимо в ній індекс для поля `age`:

```
db.users.createIndex({"age": 1})
```

При виконанні функції також необхідно використовувати функцію `hint()`, до якої передається індекс. Наприклад, виберемо всі документи, в яких поле `age` більше 30:

```

test> db.users.find().min({age:30}).hint({age:1})
[
  {
    _id: ObjectId("62e3d63a79a0a7792a9de210"),
    name: 'Tom',
    age: 38,
    languages: [ 'english', 'spanish' ]
  },
  {
    _id: ObjectId("62e3d63a79a0a7792a9de211"),
    name: 'Bob',
    age: 41,
    languages: [ 'english', 'french' ]
  }
]
test>

```

Аналогічно працює функція **max()**, яка встановлює максимальне значення. Наприклад, виберемо документи, де `age` менше 30:

```

test> db.users.find().max({age:30}).hint({age:1})
[

```

```
{
  _id: ObjectId("62e3d64079a0a7792a9de213"),
  name: 'Tom',
  age: 22,
  languages: [ 'english', 'german' ]
},
{
  _id: ObjectId("62e3d63a79a0a7792a9de212"),
  name: 'Sam',
  age: 28,
  languages: [ 'english' ]
}
]
test>
```

Оператори вибірки

Умовні оператори

Умовні оператори задають умову, якій має відповідати значення поля документа:

- **\$eq** (рівно)
- **\$ne** (не рівно)
- **\$gt** (більше ніж)
- **\$lt** (менше ніж)
- **\$gte** (більше чи одно)
- **\$lte** (менше чи одно)
- **\$in** визначає масив значень, одне з яких повинно мати поле документа
- **\$nin** визначає масив значень, які повинні мати поле документа

Наприклад, знайдемо всі документи, у яких значення ключа age менше 30:

```
1 db.users.find ({age: {$lt : 30}})
```

Аналогічно буде використання інших операторів порівняння. Наприклад, той самий ключ, тільки більше 30:

```
1 db.users.find ({age: {$gt : 30}})
```

Зверніть увагу, що порівняння тут проводиться над цілими типами, а не рядками. Якщо ключ age є строковими значеннями, то відповідно треба

проводити порівняння над рядками: `db.users.find ({age: {$gt : "30"}})`, проте результат буде тим самим.

Але уявимо ситуацію, коли нам треба знайти всі об'єкти зі значенням поля `age` більше 30, але менше 50. У цьому випадку ми можемо комбінувати два оператори:

```
1 db.users.find ({age: {$gt : 30, $lt: 50}})
```

Знайдемо користувачів, вік яких дорівнює 22:

```
1 db.users.find ({age: {$eq : 22}})
```

По суті, це аналогія наступного запиту:

```
1 db.users.find ({age: 22})
```

Зворотня операція - знайдемо користувачів, вік яких не дорівнює 22:

```
1 db.users.find ({age: {$ne : 22}})
```

Оператор `$in` визначає масив можливих виразів і шукає ключі, значення яких є в масиві:

```
1 db.users.find ({age: {$in : [22, 32]}})
```

Протилежним чином діє оператор `$nin` - він визначає масив можливих виразів і шукає ключі, значення яких відсутня в цьому масиві:

```
1 db.users.find ({age: {$nin : [22, 32]}})
```

Логічні оператори

Логічні оператори виконуються за умовами вибірки:

- **\$or** : з'єднує дві умови, і документ повинен відповідати одній з цих умов
- **\$and** : з'єднує дві умови, і документ повинен відповідати обом умовам
- **\$not** : документ повинен НЕ відповідати умові
- **\$nor** : з'єднує дві умови, і документ повинен не відповідати обом умовам

Оператор \$or

Оператор **\$or** представляє логічну операцію АБО і визначає набір пар ключ-значення, які мають бути в документі. І якщо документ має хоч одну таку пару ключ-значення, він відповідає даному запиту і витягується з бд:

```
1 db.users.find ({ $or : [{name: "Tom"}, {age: 22}]})
```

Це вираз поверне нам всі документи, у яких або `name=Tom`, або `age=22`.

Інший приклад поверне нам усі документи, у яких `name=Tom`, а `age` одно або 22, або серед значень `languages` є "german":


```
1 db.users.find ({name: "Tom", $or : [{age: 22}, {languages: "german"}]})
```

У подвираженнях `or` можна застосовувати умовні оператори:

```
1 db.users.find ({ $or : [{name: "Tom"}, {age: {$gte:30}}]})
```

У разі ми вибираємо всі документи, де `name="Tom"` чи полі `age` має значення від 30 і від.

Оператор \$and

Оператор **\$and** представляє логічну операцію І (логічне множення) та визначає набір критеріїв, яким обов'язково має відповідати документ. На відміну від оператора `$or`, документ повинен відповідати всім зазначеним критеріям. Наприклад:

```
1 db.users.find ({ $and : [{name: "Tom"}, {age: 22}]})
```

Тут документи, що вибираються, обов'язково повинні мати ім'я Tom і вік 22 - обидві ці ознаки.

Пошук по масивам

Ряд операторів призначені для роботи з масивами:

- **\$all** : визначає набір значень, які мають бути в масиві
- **\$size** : визначає кількість елементів, які мають бути в масиві
- **\$elemMatch** : визначає умову, якою повинні відповідати елементи в масиві

\$all

Оператор `$all` визначає масив можливих виразів і вимагає, щоб документи мали весь набір виразів, що визначається. Відповідно він застосовується для пошуку за масивом. Наприклад, у документах є масив `languages`, що зберігає іноземні мови, якими говорить користувач. І щоб знайти всіх людей, які говорять одночасно і англійською, і французькою, ми можемо використовувати наступне вираз:

```
1 db.users.find ({languages: {$all : ["english", "french"]}})
```

Оператор \$elemMatch

Оператор **\$elemMatch** дозволяє вибрати документи, в яких масиви містять елементи, які під певні умови. Наприклад, нехай у базі даних буде колекція,

яка містить оцінки користувачів за певними курсами. Додамо кілька документів:

```
1 db.grades.insertMany([{"student": "Tom", "courses":[{"name": "Java", "grade": 5},
  {"name": "MongoDB", "grade": 4}]},
2 {"student": "Alice", "courses":[{"name": "C++", "grade": 3}, {"name": "MongoDB",
  grade: 5}]})
```

Кожен документ має масив курсів, який у свою чергу складається з вкладених документів.

Тепер знайдемо студентів, які для курсу MongoDB мають оцінку вище 4:

```
1 db.grades.find({courses: {$elemMatch: {name: "MongoDB", grade: {$gt: 4}}}})
```

Оператор \$size

Оператор \$size використовується для знаходження документів, у яких масиви мають число елементів, що дорівнює значенню \$size. Наприклад, витягнемо всі документи, в яких в масиві languages два елементи:

```
1 db.users.find ({languages: {$size:2}})
```

Такий запит буде відповідати, наприклад, наступному документу:

```
1 {"name": "Tom", "age": 32, "languages": ["english", "german"]}
```

Оператор \$exists

Оператор \$exists дозволяє витягти лише документи, у яких певний ключ присутній чи відсутній. Наприклад, повернемо всі документи, в які є ключ company:

```
1 db.users.find ({company: {$exists:true}})
```

Якщо ми вкажемо в оператора \$exists як параметр false, то запит поверне нам лише документи, у яких не визначено ключ company.

Оператор \$type

Оператор \$type витягує лише документи, у яких певний ключ має значення певного типу, наприклад, рядок чи число:

```
1 db.users.find ({age: {$type:"string"}})
2 db.users.find ({age: {$type:"number"}})
```

Оператор \$regex

Оператор \$regex задає регулярний вираз, якому має відповідати значення поля. Наприклад, нехай поле name обов'язково має букву "b":

```
1 db.users.find ({name: {$regex:"b"}})
```

Важливо розуміти, що `$regex` приймає не просто рядки, саме регулярні висловлювання, наприклад: `name: {$regex:"om$"}-` значення `name` має закінчуватися на `"om"`.

Оновлення даних

Як і інші системи управління базами даних, MongoDB надає можливість оновлення даних. Для цього є ряд функцій.

`replaceOne`

Якщо нам потрібно повністю замінити один документ іншим, також може використовуватися функція **`replaceOne`** :

```
1 db.collection.replaceOne(filter, update, options)
```

- `filter`: приймає запит на вибірку документа, який потрібно оновити
- `update`: представляє новий документ, який замінить старий при оновленні
- `options`: визначає додаткові параметри під час оновлення документів, основним з яких є параметр `upsert`.

Якщо параметр `upsert` має значення `true`, що `mongodb` оновлюватиме документ, якщо він знайдений, і створюватиме новий, якщо такого документа немає. Якщо він має значення `false`, то `mongodb` не буде створювати новий документ, якщо запит на вибірку не знайде жодного документа.

Наприклад:

```
1 db.users.replaceOne({name: "Bob"}, {name: "Bob", age: 25})
```

В даному випадку знаходимо документ, в якому `name = "Bob"`, та замінюємо його документом `{name: "Bob", age: 25}`

Після виконання операції консоль поверне на результат оновлення:

```
test> db.users.replaceOne({name: "Bob"}, {name: "Bob", age: 25})
{
  acknowledged: true,
  insertedId: null,
  matchedCount: 1,
  modifiedCount: 1,
  upsertedCount: 0
}
```

```
}
```

В отриманому результаті параметр `matchedCount` вказує кількість документів, які відповідають запиту. А параметр `modifiedCount` вказує на кількість змінених документів. Тобто в даному випадку відповідає запиту 1 документ і він був змінений.

`updateOne` та `updateMany`

Часто не потрібно оновлювати весь документ, а лише значення однієї чи кількох його властивостей. Для цього застосовуються функції **`updateOne()`** (оновлює лише один документ) та **`updateMany()`** (оновлює безліч документів).

Для оновлення окремих полів у цих функціях використовується оператор **`$set`**. Якщо документ не містить оновлюване поле, воно створюється.

```
1 db.users.updateOne({name : "Tom", age: 22}, {$set: {age : 28}})
```

Тут ми шукаємо документ з `name="Tom"` та `age=22` і встановлюємо для його властивості `age` значення 28

Якщо необхідно оновити всі документи, що відповідають певному критерію, застосовується функція **`updateMany()`** :

```
1 db.users.updateMany({name : "Tom"}, {$set: {name : "Tomas"}})
```

Якщо оновлюваного поля в документі немає, воно додається:

```
1 db.users.updateOne({name : "Tom", age: 28}, {$set: {salary : 300}})
```

Якщо треба оновити значення кількох полів, то вони передаються оператору **`$set`** через кому:

```
1 db.users.updateOne({name : "Tom"}, {$set: {name: "Tomas", age : 25}})
```

Для простого збільшення значення числового поля на певну кількість одиниць застосовується оператор **`$inc`**. Якщо документ не містить оновлюване поле, воно створюється. Цей оператор застосовується лише до числових значень.

```
1 db.users.updateOne({name : "Tom"}, {$inc: {age:2}})
```

Видалення поля

Для видалення окремого ключа використовується оператор **`$unset`** :

```
1 db.users.updateOne({name : "Tom"}, {$unset: {salary: 1}})
```

Якщо раптом такого ключа в документі не існує, то оператор не має жодного впливу. Також можна видаляти відразу кілька полів:

```
1 db.users.updateOne({name : "Tom"}, {$unset: {salary: 1, age: 1}})
```

Оновлення масивів

Оператор \$push

Оператор \$push дозволяє додати ще одне значення до існуючого. Наприклад, якщо ключ як значення зберігає масив:

```
1 db.users.updateOne({name : "Tom"}, {$push: {languages: "russian"}})
```

Вище використовувалася функція updateOne, але цей оператор також застосовується і до функцій updateMany

```
1 db.users.updateMany({name : "Tom"}, {$push: {languages: "russian"}})
```

Якщо ключ, для якого ми хочемо додати значення, не є масивом, то ми отримаємо помилку Cannot apply \$push/\$pushAll modifier to non-array.

Використовуючи оператор \$each, можна додати відразу кілька значень:

```
1 db.users.updateOne({name : "Tom"}, {$push: {languages: {$each: ["russian",  
"spanish", "italian"]}}})
```

Ще кілька операторів дозволяє налаштувати вставку. Оператор \$position задає позицію масиві для вставки елементів, а оператор \$slice вказує, скільки елементів залишити в масиві після вставки.

```
1 db.users.updateOne({name : "Tom"}, {$push: {languages: {$each: ["german",  
"spanish", "italian"], $position:1, $slice:5}}})
```

В даному випадку елементи ["german", "spanish", "italian"] будуть вставлятися в масив languages з 1-го індексу, і після вставки в масиві залишаться лише 5 перших елементів.

Оператор \$addToSet

Оператор \$addToSet подібно до оператора \$push додає об'єкти в масив. Відмінність полягає в тому, що \$addToSet додає дані, якщо їх ще немає в масиві \$push.

```
1 db.users.updateOne({name : "Tom"}, {$addToSet: {languages: "russian"}})
```

Видалення елемента з масиву

Оператор \$pop дозволяє видаляти елемент із масиву:

```
db.users.updateOne({name : "Tom"}, {$pop: {languages: 1}})
```

Вказуючи на ключ `languages` значення `1`, ми видаляємо перший елемент з кінця. Щоб видалити перший елемент спочатку масиву, треба передати негативне значення:

```
1 db.users.updateOne({name : "Tom"}, {$pop: {languages: -1}})
```

Дещо іншу дію передбачає оператор `$pull`. Він видаляє кожне входження елемента до масиву. Наприклад, через оператор `$push` ми можемо додати те саме значення масив кілька разів. І тепер за допомогою `$pull` видалимо його:

```
1 db.users.updateOne({name : "Tom"}, {$pull: {languages: "english"}})
```

А якщо ми хочемо видалити не одне значення, а відразу кілька, тоді ми можемо застосувати оператор `$pullAll`:

```
1 db.users.updateOne({name : "Tom"}, {$pullAll: {languages: ["english",  
"german", "french"]}})
```

Видалення даних

Для видалення документів у MongoDB передбачені функції **`deleteOne()`** – видаляє один документ і **`deleteMany()`** – дозволяє видалити декілька документів. Як параметр у ці функції передається фільтр документів, що видаляються.

Наприклад, видалимо документ, в якому `name="Tom"`:

```
1 db.users.deleteOne({name : "Tom"})
```

У результаті перший знайдений документ із `name=Tom` буде видалено. Після видалення консоль відображає нам об'єкт, у якому параметр `deletedCount` вказує на кількість видалених документів:

```
test> db.users.deleteOne({name: "Tom"})  
{ acknowledged: true, deletedCount: 1 }
```

Для видалення всіх документів, які відповідають фільтру, застосовується функція **`deleteMany()`** :

```
1 db.users.deleteMany({name : "Tom"})
```

Причому, як і у випадку з `find`, ми можемо задавати умови вибірки для видалення у різний спосіб (у вигляді регулярних виразів, у вигляді умовних конструкцій і т.д.):

```
1 db.users.deleteOne({name : /^T\w+/i})  
2 db.users.deleteOne({age: {$lt : 30}})
```

Щоб видалити разом усі документи з колекції, треба залишити порожнім параметр запиту:

```
1 db.users.deleteMany({})
```

Видалення колекцій та баз даних

Ми можемо видаляти не лише документи, а й колекції та бази даних. Для видалення колекцій використовується функція **drop** :

```
1 db.users.drop()
```

І якщо видалення колекції пройде успішно, консоль виведе:

```
1 true
```

Щоб видалити всю базу даних, потрібно скористатися функцією **dropDatabase()** :

```
1 db.dropDatabase()
```

Встановлення посилань у БД

У реляційних базах даних можна встановити зовнішні ключі, коли поля з однієї таблиці посилаються на поля в іншій таблиці. У MongoDB також можна встановлювати посилання. Далі ми розглянемо встановлення посилань між колекцією `users`, яка зберігає користувачів, та колекцією `companies`, яка зберігає список компаній, у яких працюють користувачі.

Ручне встановлення посилань

Ручна установка посилань зводиться до визначення значення поля `_id` одного документа полем іншого документа. Припустимо, у нас можуть бути колекції, які представляють компанії та працівників, які працюють у цих компаніях. Отже, спочатку додамо до колекції компаній документ, що представляє компанію:

```
1 db.companies.insertOne({"_id" : "microsoft", year: 1974})
```

Тепер додамо до колекції користувачів документ, який представляє працівника. У цьому документі буде поле компанії, що представляє компанію, де працює працівник. І дуже важливо, що значення для цього поля ми встановлюємо не об'єкт `company`, а значення ключа `_id` доданого вище документа:

```
1 db.users.insertOne({name: "Tom", age: 28, company: "microsoft"})
```

Тепер отримаємо документ із колекції users:

```
1 user = db.users.findOne()
```

В даному випадку мається на увазі, що доданий елемент буде єдиним в колекції. Інакше треба зробити вибірку щодо `_id` останнього доданого документа.

Після цього консоль виводить одержаний документ. І тепер знайдемо посилання на його компанію в компанії companies:

```
1 db.companies.findOne({_id: user.company})
```

І якщо документ з таким ідентифікатором виявлено, він відображається на консолі:

```
test> db.companies.insertOne({"_id" : "microsoft", year: 1974})
{ acknowledged: true, insertedId: 'microsoft' }
test> db.users.insertOne({name: "Tom", age: 28, company: "microsoft"})
{
  acknowledged: true,
  insertedId: ObjectId("62e427069fcbf14521ea012a")
}
test> user = db.users.findOne()
{
  _id: ObjectId("62e427069fcbf14521ea012a"),
  name: 'Tom',
  age: 28,
  company: 'microsoft'
}
test> db.companies.findOne({_id: user.company})
{ _id: 'microsoft', year: 1974 }
test>
```

Автоматичне зв'язування

Використовуючи функціональність **DBRef**, ми можемо встановити автоматичне зв'язування між документами. Подивимося на прикладі застосування цієї функціональності. Спочатку додамо новий документ до колекції компаній:

```
1 google = db.companies.insertOne({name : "google", year: 1998})
```


Зверніть увагу, що ми отримуємо результат додавання до змінної **google** . При додаванні нового документа генерує `_id`, який ми можемо отримати за допомогою властивості `insertedId` результату функції.

Тепер створимо новий документ для колекції `users`, у якого ключ компанії зв'яжемо з щойно доданим документом `apple`:

```
1  sam = ({name: "Sam", age: 25, company: { "$ref" : "companies", "$id" :
    google.insertedId}})
2  db.users.insertOne(sam)
```

І ми можемо протестувати:

```
1  db.companies.findOne({_id: sam.company.$id})

test> google = db.companies.insertOne({name : "google", year: 1998})
{
  acknowledged: true,
  insertedId: ObjectId("62e427c29fcbf14521ea012b")
}
test> sam = ({name: "Sam", age: 25, company: { "$ref" : "companies", "$id" :
    google.insertedId}})
{
  name: 'Sam',
  age: 25,
  company: { '$ref': 'companies', '$id': ObjectId("62e427c29fcbf14521ea012b")
}
}
test> db.users.insertOne(sam)
{
  acknowledged: true,
  insertedId: ObjectId("62e427dc9fcbf14521ea012c")
}
test> db.companies.findOne({_id: sam.company.$id})
{
  _id: ObjectId("62e427c29fcbf14521ea012b"),
  name: 'google',
  year: 1998
}
test>
```

Подивившись на прикладі, тепер розберемо організацію посилань між документами. Для зв'язування з документом `google` використовувався такий вираз `{ "$ref" : "companies", "$id" : google.insertedId}`. Формальний синтаксис `DBRef` наступний:

```
1 { "$ref" : название_коллекции, "$id": значение [, "$db" : название_бд ] }
```

Перший параметр `$ref` вказує на колекцію, де зберігається пов'язаний документ. Другий параметр вказує на значення, яке буде представляти щось типу зовнішнього ключа. Третій необов'язковий параметр вказує базу даних.

При тестуванні як запит на вибірку вказується вираз `_id: sam.company.$id`. Оскільки `sam.company` представляє тепер об'єкт `{ "$ref" : "companies", "$id" : google.insertedId }`, нам треба конкретизувати параметр `sam.company.$id`

Управління колекцією

Явне створення колекції

У попередніх темах колекція створювалася неявно автоматично під час додавання до неї перших даних. Але ми також можемо створити її явним чином, застосувавши метод **`db.createCollection(name, options)`**, де `name` назва колекції, а `options` необов'язковий об'єкт з додатковими налаштуваннями ініціалізації. Наприклад:

```
1 db.createCollection("accounts")
2 {"ok" : 1}
```

Таким чином, створюється колекція акаунтів.

Перейменування колекції

У процесі роботи, можливо, потрібно змінити назву колекції. Наприклад, якщо при першому додаванні даних у її назві була помилка. І щоб не видаляти і потім перетворювати колекцію, слід використовувати функцію `renameCollection`:

```
1 db.users.renameCollection("новое_название")
```

І якщо перейменування пройде вдало, то консоль відобразить:

```
{"ok" : 1}
```

Обмежені колекції

Коли ми надсилаємо запит до бд на вибірку, MongoDB повертає нам документи в тому порядку, як правило, в якому вони були додані. Однак такий порядок не завжди гарантується, оскільки дані можуть бути видалені,

переміщені, змінені. Тому MongoDB існує поняття **обмеженої колекції** (capped collection). Подібна колекція гарантує, що документи розташовуватимуться в тому ж порядку, в якому вони додавалися до колекції. Обмежені колекції мають фіксований розмір. І коли колекції вже немає місця, найбільш старі документи видаляються, і в кінець додаються нові дані.

На відміну від звичайних колекцій, обмежені ми можемо поставити явно. Наприклад, створимо обмежену колекцію з назвою profiles і задамо для неї розмір 9500 байт:

```
1 db.createCollection("profiles", {capped:true, size:9500})
```

І після успішного створення колекції консоль виведе:

```
{ "ok": 1 }
```

Також можна обмежити кількість документів у колекції, вказавши його у параметрі max:

```
1 > db.createCollection("profiles", {capped:true, size:9500, max: 150})
```

Однак при такому способі створення колекції слід враховувати, що якщо все місце під колекцію заповнене (наприклад, виділені нами 9500 байтів), а кількість документів ще не досягла максимуму, в даному випадку 150, то при додаванні нового документа найстаріший документ буде видалятися, а на його місце вставлятиметься новий документ.

При оновленні документів у таких колекціях слід пам'ятати, що документи не повинні зростати у розмірах, інакше оновлення не вдасться зробити.

Також не можна видаляти документи з таких колекцій, можна лише видалити всю колекцію.

Підколекції

Для спрощення організації даних у колекціях ми можемо використовувати підколекції. Наприклад, дані по колекції users треба розмежувати на профілі та облікові дані. І ми можемо використовувати для створення колекції db.users.profiles і db.users.accounts. При цьому вони не будуть пов'язані з колекцією users. Тобто в результаті будуть три різні колекції, однак

у плані логічної організації зберігання даних, можливо, для когось так буде простіше.