MongoDB w akcji

Kyle **Banker**, Peter **Bakkum**, Shaun **Verch**,
Doug **Garrett**, Tim **Hawkins**

Tytuł oryginału: MongoDB in Action

Tłumaczenie: Robert Górczyński
Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-1918-9

Original edition copyright © 2016 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2017 by HELION SA.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/mongod.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres:
http://helion.pl/user/opinie/mongod
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Przedmowa 13
Podziękowania 15
O książce 17

CZĘŚĆ I ROZPOCZĘCIE PRACY 21

Rozdział 1. Baza danych dla nowoczesnej sieci WWW ***23***

1.1. Serwer opracowany na potrzeby internetu 26
1.2. Kluczowe funkcje MongoDB 27
1.2.1. Model danych oparty na dokumencie *27*
1.2.2. Zapytania ad hoc *31*
1.2.3. Indeksy *31*
1.2.4. Replikacja *32*
1.2.5. Szybkość działania i niezawodność *33*
1.2.6. Skalowanie *35*
1.3. Podstawowy serwer MongoDB i jego narzędzia 36
1.3.1. Podstawowy serwer *37*
1.3.2. Konsola JavaScript *38*
1.3.3. Sterowniki bazy danych *39*
1.3.4. Narzędzia powłoki *39*
1.4. Dlaczego MongoDB? 40
1.4.1. MongoDB kontra inne bazy danych *41*
1.4.2. Przykłady użycia i wdrożeń w środowiskach produkcyjnych *45*
1.5. Wskazówki i ograniczenia 47
1.6. Historia MongoDB 49
1.7. Zasoby dodatkowe 52
1.8. Podsumowanie 53

Rozdział 2. MongoDB i konsola JavaScript ***55***

2.1. Zagłębiamy się w powłokę MongoDB 56
2.1.1. Uruchomienie powłoki *56*
2.1.2. Bazy danych, kolekcje i dokumenty *57*
2.1.3. Wstawianie i pobieranie danych *58*
2.1.4. Uaktualnienie dokumentu *60*
2.1.5. Usunięcie danych *64*
2.1.6. Inne funkcje powłoki *65*

Kup ksi k

Pole ksi k

2.2. Tworzenie indeksów i użycie ich w zapytaniach 66
2.2.1. Utworzenie ogromnej kolekcji 66
2.2.2. Indeksowanie i metoda explain() 68
2.3. Podstawowa administracja serwerem bazy danych 73
2.3.1. Pobieranie informacji o bazie danych 73
2.3.2. Jak działają polecenia? 75
2.4. Uzyskiwanie pomocy 76
2.5. Podsumowanie 77

Rozdział 3. Tworzenie programów używających MongoDB 79

3.1. MongoDB przez pryzmat języka Ruby 80
3.1.1. Instalacja sterownika i nawiązanie połączenia z MongoDB 80
3.1.2. Wstawianie dokumentów za pomocą języka Ruby 82
3.1.3. Zapytania i kursory 83
3.1.4. Operacje uaktualnienia i usunięcia 84
3.1.5. Polecenia bazy danych 85
3.2. Jak działają sterowniki? 86
3.2.1. Generowanie identyfikatora obiektu 87
3.3. Utworzenie prostej aplikacji 89
3.3.1. Konfiguracja 90
3.3.2. Zbieranie danych 91
3.3.3. Wyświetlenie zawartości archiwum 94
3.4. Podsumowanie 96

CZĘŚĆ II TWORZENIE APLIKACJI W MONGODB 99

Rozdział 4. Dane oparte na dokumentach 101

4.1. Reguły projektowe schematu 102
4.2. Opracowanie modelu danych dla aplikacji typu e-commerce 104
4.2.1. Podstawy schematu 105
4.2.2. Użytkownicy i zamówienia 109
4.2.3. Opinie 112
4.3. Bazy danych, kolekcje i dokumenty w szczegółach 113
4.3.1. Bazy danych 113
4.3.2. Kolekcje 117
4.3.3. Dokumenty i operacje wstawiania 121
4.4. Podsumowanie 127

Rozdział 5. Tworzenie zapytań 129

5.1. Zapytania w aplikacji typu e-commerce 130
5.1.1. Produkty, kategorie i opinie o produktach 130
5.1.2. Użytkownicy i zamówienia 133
5.2. Język zapytań w MongoDB 134
5.2.1. Selektory i kryteria zapytania 135
5.2.2. Opcje zapytania 149
5.3. Podsumowanie 152

Rozdział 6. Agregacja *153*

6.1. Ogólne omówienie frameworka agregacji 154
6.2. Przykład agregacji w aplikacji typu e-commerce 156
6.2.1. Produkty, kategorie i opinie 157
6.2.2. Użytkownik i zamówienie 164
6.3. Operatory potoku agregacji 168
6.3.1. Operator $project 168
6.3.2. Operator $group 169
6.3.3. $match, $sort, $skip i $limit 171
6.3.4. Operator $unwind 171
6.3.5. Operator $out 172
6.4. Modyfikacja dokumentów 172
6.4.1. Funkcje ciągu tekstowego 173
6.4.2. Funkcje arytmetyczne 174
6.4.3. Funkcje daty i godziny 175
6.4.4. Funkcje logiczne 175
6.4.5. Operatory zbioru 176
6.4.6. Pozostałe funkcje 177
6.5. Wydajność działania potoku agregacji 178
6.5.1. Opcje potoku agregacji 179
6.5.2. Funkcja explain() frameworka agregacji 179
6.5.3. Opcja allowDiskUse 183
6.5.4. Opcja cursor w agregacji 184
6.6. Inne możliwości agregacji 185
6.6.1. Funkcje .count() i .distinct() 185
6.6.2. Funkcja modelu MapReduce 185
6.7. Podsumowanie 188

Rozdział 7. Uaktualnienia, operacje niepodzielne i usunięcia *191*

7.1. Krótkie omówienie procesu uaktualnienia dokumentu 192
7.1.1. Modyfikacja przez zastąpienie 193
7.1.2. Modyfikacja za pomocą operatora 193
7.1.3. Porównanie obu metod 194
7.1.4. Podjęcie decyzji — zastąpienie kontra operatory 194
7.2. Uaktualnienia w modelu typu e-commerce 196
7.2.1. Produkty i kategorie 196
7.2.2. Opinie o produkcie 201
7.2.3. Zamówienia 203
7.3. Niepodzielne przetwarzanie dokumentu 206
7.3.1. Zmiana stanu zamówienia 207
7.3.2. Zarządzanie produktami 209
7.4. Usunięcia i uaktualnienia w MongoDB w szczegółach 215
7.4.1. Opcje i typy uaktualnień 215
7.4.2. Operatory uaktualnienia 216
7.4.3. Polecenie findAndModify() 225

7.4.4. Usunięcie dokumentu *225*
7.4.5. Współbieżność, niepodzielność i izolacja *226*
7.4.6. Uwagi dotyczące wydajności uaktualnienia *227*
7.5. Przegląd operatorów uaktualnienia 229
7.6. Podsumowanie 230

CZĘŚĆ III ZAAWANSOWANE MONGODB 231

Rozdział 8. Indeksowanie i optymalizacja zapytania *233*
8.1. Teoria indeksowania 234
8.1.1. Prosty eksperyment *234*
8.1.2. Podstawowe koncepcje indeksowania *238*
8.1.3. Struktura B-tree *242*
8.2. Indeksowanie w praktyce 244
8.2.1. Typy indeksów *244*
8.2.2. Administracja indeksem *249*
8.3. Optymalizacja zapytania 255
8.3.1. Identyfikacja wolno wykonywanych zapytań *255*
8.3.2. Analiza wolno wykonywanych zapytań *260*
8.3.3. Wzorce zapytania *280*
8.4. Podsumowanie 282

Rozdział 9. Wyszukiwanie tekstowe *285*
9.1. Wyszukiwanie tekstowe — nie tylko dopasowanie wzorca 286
9.1.1. Wyszukiwanie tekstowe kontra dopasowanie wzorca *288*
9.1.2. Wyszukiwanie tekstowe kontra wyszukiwanie stron internetowych *288*
9.1.3. Wyszukiwanie tekstowe w MongoDB kontra dedykowane silniki wyszukiwania tekstowego *291*
9.2. Pobranie danych katalogu książek Manning 294
9.3. Zdefiniowanie indeksów wyszukiwania tekstowego 296
9.3.1. Wielkość indeksu wyszukiwania tekstowego *297*
9.3.2. Przypisanie indeksowi własnej nazwy oraz zindeksowanie wszystkich pól tekstowych kolekcji *298*
9.4. Proste wyszukiwanie tekstowe 299
9.4.1. Bardziej zaawansowane operacje wyszukiwania *300*
9.4.2. Ocena wyszukiwania tekstowego *302*
9.4.3. Sortowanie wyników na podstawie oceny wyszukiwania tekstowego *304*
9.5. Wyszukiwanie tekstowe we frameworku agregacji 304
9.5.1. Gdzie jest MongoDB in Action, Second Edition? *306*
9.6. Wyszukiwanie tekstowe w innych językach 308
9.6.1. Wskazanie języka w indeksie *309*
9.6.2. Określenie języka w dokumencie *310*
9.6.3. Podanie języka w operacji wyszukiwania *311*
9.6.4. Dostępne języki *313*
9.7. Podsumowanie 314

Rozdział 10. WiredTiger i dołączany silnik magazynu danych *315*

10.1. API Pluggable Storage Engine 315
10.1.1. Dlaczego warto używać różnych silników magazynów danych *316*
10.2. Silnik WiredTiger 318
10.2.1. Przejście do silnika WiredTiger *318*
10.2.2. Migracja bazy danych do WiredTiger *320*
10.3. Porównanie z MMAPv1 321
10.3.1. Pliki konfiguracyjne *322*
10.3.2. Skrypty wstawiania danych i przeprowadzania testu wydajności *323*
10.3.3. Wyniki testów wydajności wstawiania danych *326*
10.3.4. Skrypty sprawdzające wydajność operacji odczytu danych *327*
10.3.5. Wyniki testów wydajności odczytu danych *329*
10.3.6. Podsumowanie testów wydajności *330*
10.4. Inne przykłady dołączanych silników magazynów danych 332
10.5. Tematy zaawansowane 333
10.5.1. Jak działa dołączany silnik magazynu danych? *333*
10.5.2. Struktura danych *334*
10.5.3. Nakładanie blokad *338*
10.6. Podsumowanie 338

Rozdział 11. Replikacja *341*

11.1. Ogólne omówienie replikacji 342
11.1.1. Dlaczego replikacja ma znaczenie? *342*
11.1.2. Przykłady użycia replikacji i jej ograniczenia *344*
11.2. Zbiory replik 345
11.2.1. Konfiguracja *346*
11.2.2. Jak działa replikacja? *353*
11.2.3. Administracja *362*
11.3. Sterowniki i replikacja 372
11.3.1. Połączenia i reakcja na wystąpienie awarii *372*
11.3.2. Pewność udanego zapisu *375*
11.3.3. Skalowanie operacji odczytu *376*
11.3.4. Tagi *379*
11.4. Podsumowanie 381

Rozdział 12. Skalowanie systemu za pomocą shardingu *383*

12.1. Ogólne omówienie shardingu 384
12.1.1. Co to jest sharding? *384*
12.1.2. Kiedy należy stosować sharding? *385*
12.2. Poznajemy komponenty klastra shardingu 387
12.2.1. Shardy — pamięć masowa dla danych aplikacji *388*
12.2.2. Router mongos — przekierowywanie operacji *388*
12.2.3. Serwery konfiguracji — przechowywanie metadanych *389*
12.3. Rozproszenie danych w klastrze shardingu 389
12.3.1. Sposoby rozpraszania danych w klastrze shardingu *391*
12.3.2. Rozproszenie baz danych między shardami *392*
12.3.3. Sharding na podstawie kolekcji *392*

12.4. Budowa przykładowego klastra shardingu 394
12.4.1. Uruchomienie serwerów mongod i mongos 394
12.4.2. Konfiguracja klastra 397
12.4.3. Kolekcje shardingu 398
12.4.4. Zapis danych w klastrze shardingu 400
12.5. Wykonywanie zapytań i indeksowanie klastra shardingu 406
12.5.1. Routing zapytania 406
12.5.2. Indeksowanie w klastrze shardingu 408
12.5.3. Narzędzie explain() w klastrze shardingu 408
12.5.4. Agregacja w klastrze shardingu 410
12.6. Wybór klucza shardu 411
12.6.1. Brak równowagi podczas wykonywania operacji zapisu (hotspot) 412
12.6.2. Fragmenty niemożliwe do podziału 413
12.6.3. Kiepskie adresowanie (klucz shardu nie znajduje się w zapytaniach) 414
12.6.4. Idealny klucz shardu 415
12.6.5. Nieodłączne kompromisy podczas projektowania (aplikacja klienta poczty) 415
12.7. Sharding w produkcji 418
12.7.1. Provisioning 418
12.7.2. Wdrożenie 421
12.7.3. Obsługa i konserwacja 423
12.8. Podsumowanie 428

Rozdział 13. Wdrożenie i administracja 429

13.1. Sprzęt i provisioning 430
13.1.1. Topologia klastra 430
13.1.2. Środowisko wdrożenia 432
13.1.3. Provisioning 440
13.2. Monitorowanie i diagnostyka 442
13.2.1. Rejestracja danych 442
13.2.2. Polecenia diagnostyczne MongoDB 443
13.2.3. Narzędzia diagnostyczne MongoDB 443
13.2.4. Usługa monitorowania MongoDB 446
13.2.5. Zewnętrzne aplikacje monitorowania 446
13.3. Kopia zapasowa 447
13.3.1. Narzędzia mongodump i mongorestore 447
13.3.2. Kopia zapasowa na podstawie plików danych 448
13.3.3. Kopia zapasowa tworzona za pomocą monitorowania MMS 450
13.4. Zapewnienie bezpieczeństwa 450
13.4.1. Bezpieczne środowisko 450
13.4.2. Szyfrowanie komunikacji sieciowej 451
13.4.3. Uwierzytelnianie 454
13.4.4. Uwierzytelnienie zbioru replik 457
13.4.5. Uwierzytelnianie klastra shardingu 459
13.4.6. Funkcje zabezpieczeń w korporacyjnej wersji MongoDB 459
13.5. Zadania administracyjne 459
13.5.1. Import i eksport danych 460
13.5.2. Naprawa i zmniejszenie ilości miejsca zajmowanego przez pliki danych 461
13.5.3. Uaktualnienie 462

13.6. Rozwiązywanie problemów związanych z wydajnością 463
13.6.1. Zbiór roboczy 463
13.6.2. Nagły spadek wydajności 464
13.6.3. Interakcje zapytań 465
13.6.4. Szukanie profesjonalnej pomocy 466
13.7. Lista rzeczy do sprawdzenia podczas wdrożenia 466
13.8. Podsumowanie 468

DODATKI 469

Dodatek A. Instalacja 471

A.1. Instalacja 471
A.1.1. Wdrożenie w środowisku produkcyjnym 471
A.1.2. Architektura 32-bitowa kontra 64-bitowa 472
A.2. MongoDB w systemie Linux 472
A.2.1. Instalacja z użyciem prekompilowanych plików binarnych 472
A.2.2. Użycie menedżera pakietów 473
A.3. MongoDB w systemie OS X 474
A.3.1. Instalacja z użyciem prekompilowanych plików binarnych 474
A.3.2. Użycie menedżera pakietów 475
A.4. MongoDB w Windows 475
A.4.1. Instalacja z użyciem prekompilowanych plików binarnych 476
A.5. Kompilacja MongoDB z kodu źródłowego 477
A.6. Rozwiązywanie problemów 477
A.6.1. Nieprawidłowa architektura 477
A.6.2. Brak katalogu danych 477
A.6.3. Brak uprawnień 478
A.6.4. Brak możliwości dołączenia do portu 478
A.7. Podstawowe opcje konfiguracyjne 478
A.8. Instalacja języka Ruby 480
A.8.1. Systemy Linux i OS X 480
A.8.2. Windows 481

Dodatek B. Wzorce projektowe 483

B.1. Osadzenie kontra odwołanie 483
B.2. Związek typu „jeden do wielu” 483
B.3. Związek typu „wiele do wielu” 485
B.4. Drzewo 486
B.5. Kolejki procesów roboczych 489
B.6. Atrybuty dynamiczne 490
B.7. Transakcje 491
B.8. Lokalizacja i obliczenia wstępne 492
B.9. Antywzorce 493
B.9.1. Niepoprawne indeksowanie 493
B.9.2. Bałagan w typach 494
B.9.3. Kolekcje kubełków 494
B.9.4. Ogromne, głęboko zagnieżdżone dokumenty 494

Kup ksi k   Pole ksi k

B.9.5. Jedna kolekcja dla użytkownika *494*
B.9.6. Kolekcje niemożliwe do shardingu *495*

Dodatek C. Dane binarne i GridFS *497*

C.1. Przechowywanie prostych obiektów binarnych 497
C.1.1. Przechowywanie miniatury *498*
C.1.2. Przechowywanie wartości MD5 *498*
C.2. GridFS 499
C.2.1 GridFS w języku Ruby *450*
C.2.2. GridFS i mongofiles *503*

Skorowidz *505*

3 Tworzenie programów używających MongoDB

W tym rozdziale:

- Wprowadzenie do API MongoDB na przykładzie języka Ruby.
- Omówienie sposobu działania sterowników.
- Użycie formatu BSON i protokołu sieciowego MongoDB.
- Utworzenie kompletnej przykładowej aplikacji.

Pora przejść do działań praktycznych. Wprawdzie można się wiele nauczyć, eksperymentując z powłoką MongoDB, ale prawdziwą wartość bazy danych poznasz dopiero wtedy, gdy zaczniesz budować coś opierającego się na niej. To oznacza przejście do programowania i przyjrzenie się sterownikom MongoDB. Jak wcześniej wspomniałem, firma MongoDB, Inc. oferuje oficjalnie obsługiwane i dostępne na licencji Apache sterowniki dla wszystkich najpopularniejszych języków programowania. W tej książce w przykładach wykorzystano sterownik przeznaczony dla języka Ruby, ale zaprezentowane koncepcje są uniwersalne i bardzo łatwo mogą być przeniesione do innych sterowników. Choć większość poleceń w książce została napisana w języku JavaScript, to przykłady użycia MongoDB z poziomu aplikacji będą utworzone w języku Ruby.

Programowanie w MongoDB podzieliłem na trzy etapy. W pierwszym zajmiemy się instalacją sterownika MongoDB dla języka Ruby oraz przedstawię podstawowe operacje CRUD. Ten proces powinien być krótki i wydawać Ci się znajomy, ponieważ

Kup ksi k   Pole ksi k

API sterownika jest podobne do powłoki. W drugim etapie zajmiemy się dokładniejszą analizą sterownika i wyjaśnieniem sposobu jego powiązania z MongoDB. Nie przejdziemy na zbyt niski poziom, ale i tak zobaczysz, co się dzieje w tle podczas działania sterowników. Natomiast w trzecim etapie opracujemy prostą aplikację w języku Ruby przeznaczoną do monitorowania serwisu Twitter. Pracując z rzeczywistym zbiorem danych, zaczniesz dostrzegać, jak MongoDB sprawdza się w praktyce. Ten ostatni etap położy również fundamenty pod znacznie dokładniejsze przykłady zaprezentowane w drugiej części książki.

Nie masz doświadczenia w programowaniu w języku Ruby?

Ruby to popularny i czytelny język skryptowy. Przykładowe fragmenty kodu zostały opracowane w taki sposób, aby były jak najbardziej jasne, więc nawet osoby nieposiadające doświadczenia w programowaniu w tym języku będą mogły skorzystać z omówionych przykładów. Wszystkie koncepcje języka Ruby, które mogą być trudne do zrozumienia, zostaną w książce wyjaśnione. Jeżeli chcesz poświęcić kilka minut na poznanie języka Ruby, zacznij od 20-minutowego oficjalnego samouczka, który znajdziesz na stronie *http://www.ruby-lang.org/en/documentation/quickstart/*.

3.1. MongoDB przez pryzmat języka Ruby

Słowo „sterownik” najczęściej kojarzy się z przeprowadzanymi na niskim poziomie operacjami związanymi z interfejsami. Na szczęście ze sterownikami języka MongoDB jest inaczej. Zostały zaprojektowane jako intuicyjne, charakterystyczne dla danego języka API, aby wiele aplikacji mogło używać sterownika MongoDB jako jedynego interfejsu bazy danych. API sterownika pozostaje całkiem spójne dla różnych języków, co oznacza, że programiści mogą bardzo łatwo przechodzić między niezbędnymi im językami. Wszystko to, co można zrobić w API JavaScript, będzie możliwe do wykonania również za pomocą API języka Ruby. Jeżeli jesteś programistą aplikacji, praca ze sterownikami MongoDB będzie dla Ciebie komfortowa i produktywna. Nie będziesz musiał zajmować się szczegółami implementacyjnymi na niskim poziomie.

W tym podrozdziale zajmiemy się instalacją sterownika MongoDB dla języka Ruby, nawiązaniem połączenia z bazą danych, a także wyjaśnię sposób przeprowadzania podstawowych operacji CRUD. Tym samym przygotujemy grunt pod aplikację, której budową zajmiemy się w dalszej części rozdziału.

3.1.1. Instalacja sterownika i nawiązanie połączenia z MongoDB

Sterownik dla języka Ruby możesz zainstalować za pomocą RubyGems, czyli stosowanego w języku Ruby systemu przeznaczonego do zarządzania pakietami.

Wiele nowoczesnych systemów operacyjnych ma domyślnie zainstalowany interpreter Ruby. Możesz to sprawdzić przez wydanie polecenia `ruby --version` z poziomu powłoki. Jeżeli nie masz zainstalowanego języka Ruby, szczegółowe informacje dotyczące jego instalacji znajdziesz na stronie *http://www.ruby-lang.org/en/downloads/*.

Potrzebny jest również używany przez język Ruby menedżer pakietów o nazwie RubyGems. Być może jest on już zainstalowany w Twoim systemie, sprawdź to przez

wydanie polecenia `gem --version`. Informacje dotyczące instalacji RubyGems znajdziesz na stronie *https://docs.rubygems.org/read/chapter/3*. Po przeprowadzeniu instalacji RubyGems wydaj poniższe polecenie.

```
$ gem install mongo
```

To polecenie powinno przeprowadzić instalację zarówno pakietu `mongo`, jak i `bson`[1]. Powinieneś otrzymać dane wyjściowe podobne do poniższych (numery wersji na pewno będą nowsze niż w tym fragmencie kodu).

```
Fetching: bson-3.2.1.gem (100%)
Building native extensions.  This could take a while...
Successfully installed bson-3.2.1
Fetching: mongo-2.0.6.gem (100%)
Successfully installed mongo-2.0.6
2 gems installed
```

Zachęcam Cię również do instalacji opcjonalnego pakietu `bson_ext`. To jest oficjalny pakiet zawierający przygotowaną w języku C implementację formatu BSON, co zapewni znacznie efektywniejszą obsługę danych BSON w sterowniku MongoDB. Wymieniony pakiet nie jest instalowany domyślnie, ponieważ jego instalacja wymaga kompilatora. Możesz spać spokojnie, ponieważ jeżeli nie uda Ci się zainstalować pakietu `bson_ext`, Twoje programy nadal będą działały zgodnie z oczekiwaniami.

Przystępujemy do nawiązania połączenia z bazą danych MongoDB. Przede wszystkim upewnij się o działaniu demona `mongod` oraz wydaj polecenie `mongo` w powłoce, aby sprawdzić, czy możesz nawiązać połączenie z bazą danych. Następnie utwórz plik o nazwie *connect.rb* i umieść w nim poniższy fragment kodu.

```
require 'rubygems'
require 'mongo'

$client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'tutorial')
Mongo::Logger.logger.level = ::Logger::ERROR
$users = $client[:users]
puts 'Połączono!'
```

Dwa pierwsze polecenia `require` gwarantują wczytanie sterownika. Kolejne wiersze kodu tworzą egzemplarz klienta, nawiązują połączenie z bazą danych `tutorial`, przechowują w zmiennej `$users` odwołanie do kolekcji `users` oraz wyświetlają ciąg tekstowy `Połączono!`. Umieszczenie znaku `$` przed nazwą zmiennej powoduje, że staje się ona globalna, czyli w omawianym przykładzie dostępna poza skryptem *connect.rb*. Zapisz plik i wykonaj go.

```
$ ruby connect.rb
D, [2015-06-05T12:32:38.843933 #33946] DEBUG -- : MONGODB | Adding
127.0.0.1:27017 to the cluster. | runtime: 0.0031ms
D, [2015-06-05T12:32:38.847534 #33946] DEBUG -- : MONGODB | COMMAND |
```

[1] Omówiony w kolejnym podrozdziale format BSON to format binarny zainspirowany przez JSON i używany przez MongoDB do przedstawiania dokumentów. Pakiet `bson` w języku Ruby przeprowadza serializację obiektów Ruby na postać BSON oraz deserializację z BSON na natywne obiekty języka.

Kup ksi k Pole ksi k

```
namespace=admin.$cmd selector={:ismaster=>1} flags=[] limit=-1 skip=0
project=nil | runtime: 3.4170ms
Połączono!
```

Jeżeli nie został zgłoszony wyjątek, to znaczy, że udało się nawiązać połączenie z bazą danych MongoDB z poziomu języka Ruby. W powłoce powinieneś zobaczyć wyświetlony komunikat `Połączono!`. Wprawdzie może to nie wydawać się efektowne, ale nawiązanie połączenia to pierwszy krok na drodze do użycia bazy danych MongoDB z poziomu dowolnego języka. Nawiązane tutaj połączenie wykorzystamy teraz do wstawienia pewnych dokumentów.

3.1.2. Wstawianie dokumentów za pomocą języka Ruby

W celu wykonania interesujących zapytań MongoDB najpierw trzeba przygotować pewne dane. Przechodzimy więc do utworzenia danych (to operacja oznaczona literą C w skrócie CRUD). Wszystkie sterowniki MongoDB zostały zaprojektowane do użycia najbardziej naturalnej reprezentacji dokumentu w danym języku programowania. W przypadku JavaScriptu najbardziej oczywistym wyborem są obiekty JSON, ponieważ JSON to struktura danych dokumentu. Z kolei w języku Ruby największy sens ma użycie struktury danych hash. Natywna struktura danych hash w języku Ruby różni się od obiektu JSON w kilku drobnych szczegółach. Przede wszystkim w JSON klucze i wartości są rozdzielone dwukropkiem, podczas gdy w języku Ruby — tak zwaną rakietą hash (=>)[2].

Jeżeli wykonywałeś przedstawione dotąd przykłady, możesz kontynuować dodawanie kodu do pliku *connect.rb*. Alternatywnym, eleganckim podejściem będzie wykorzystanie interaktywnej powłoki języka Ruby, czyli `irb`. Powłoka `irb` to konsola pętli **REPL** (ang. *read, evaluate, print loop*); wprowadzasz w niej kod Ruby, który następnie zostanie dynamicznie wykonany. To idealne rozwiązanie podczas eksperymentowania z kodem. Wszystkie polecenia wydawane w konsoli `irb` mogą być umieszczone w skrypcie. Dlatego też zachęcam do użycia powłoki podczas poznawania nowych koncepcji, a następnie do skopiowania poleceń, które chcesz później wykonywać w programie. Możesz uruchomić powłokę `irb` i wczytać plik *connect.rb*, dzięki temu otrzymasz natychmiastowy dostęp do połączenia, bazy danych i zainicjalizowanych w niej obiektów kolekcji. Następnie możesz uruchomić kod Ruby, a otrzymasz natychmiast dane wyjściowe. Poniżej przedstawiłem przykład tego rodzaju podejścia.

```
$ irb -r ./connect.rb
irb(main):017:0> id = $users.insert_one({"last_name" => "mtsouk"})
=> #<Mongo::Operation::Result:70275279152800 documents=[{"ok"=>1, "n"=>1}]>
irb(main):014:0> $users.find().each do |user|
irb(main):015:1* puts user
irb(main):016:1> end
{"_id"=>BSON::ObjectId('55e3ee1c5ae119511d000000'), "last_name"=>"knuth"}
{"_id"=>BSON::ObjectId('55e3f13d5ae119516a000000'), "last_name"=>"mtsouk"}
```

---

[2] Ruby 1.9 pozwala opcjonalnie na użycie dwukropka jako separatora w parze klucz-wartość, na przykład `hash = {foo: 'bar'}`, ale w przykładach przedstawionych w książce pozostanę przy rakiecie hash, aby zapewnić wsteczną zgodność.

Kup książkę Poleć książkę

```
=> #<Enumerator: #<Mongo::Cursor:0x70275279317980
@view=#<Mongo::Collection::View:0x70275279322740 namespace='tutorial.users
@selector={} @options={}>>:each>
```

Polecenie `irb` daje dostęp do powłoki. Na wprowadzanie poleceń pozwala znak zachęty w postaci > (w Twoim komputerze ten znak zachęty może mieć inną postać). W powyższym fragmencie kodu polecenia przeznaczone do wprowadzenia zostały przedstawione pogrubioną czcionką. Kiedy wykonujesz polecenie w `irb`, powłoka wyświetla zwróconą przez dane polecenie wartość (o ile taka istnieje) po znakach =>, jak możesz zobaczyć powyżej.

Przechodzimy do utworzenia pewnych dokumentów dla kolekcji `users`. Przygotujemy więc dwa dokumenty przedstawiające dwóch użytkowników: `kowalski` i `nowak`. Każdy dokument wyrażony jako struktura danych hash w języku Ruby zostaje przypisany zmiennej.

```
kowalski = {"last_name" => "kowalski", "age" => 30}
nowak = {"last_name" => "nowak", "age" => 40}
```

W celu zapisania dokumentów przekazujemy je metodzie `insert()` kolekcji. Każde wywołanie wymienionej metody zwraca unikatowy identyfikator, który następnie przechowujemy w zmiennej, aby ułatwić sobie późniejszą pracę z danymi.

```
kowalski_id = $users.insert_one(kowalski)
nowak_id = $users.insert_one(nowak)
```

Jeżeli chcesz potwierdzić zapis dokumentów, możesz wykonać kilka prostych zapytań. Jako przykład wykonamy zapytanie do kolekcji `users`, wykorzystując do tego metodę `find()`, jak możesz zobaczyć w poniższym fragmencie kodu.

```
irb(main):013:0> $users.find("age" => {"$gt" => 20}).each.to_a do |row|
irb(main):014:1* puts row
irb(main):015:1> end
=> [{"_id"=>BSON::ObjectId('55e3f7dd5ae119516a000002'), "last_name"=>"kowalski",
"age"=>30}, {"_id"=>BSON::ObjectId('55e3f7e25ae119516a000003'),
"last_name"=>"nowak", "age"=>40}]
```

Wartości zwrotne zapytań będą wyświetlane po znaku zachęty, gdy te zapytania będą wykonywane w powłoce `irb`. Jeżeli kod zostanie uruchomiony z pliku Ruby, to polecenie trzeba będzie poprzedzić metodą `p()`, aby dane wyjściowe zostały umieszczone na ekranie.

```
irb(main):013:0> p $users.find( :age => {"$gt" => 20}).to_a
```

W ten sposób z sukcesem przeprowadziłeś operację wstawienia w MongoDB dwóch dokumentów z poziomu języka Ruby. Przyjrzyjmy się nieco bliżej zapytaniom.

3.1.3. Zapytania i kursory

Skoro utworzyłeś dokumenty, przechodzimy do oferowanych przez MongoDB operacji odczytu (to operacja oznaczona literą R w skrócie CRUD). Sterownik dla języka Ruby oferuje bogaty interfejs przeznaczony do uzyskania danych i za programistę zajmuje się obsługą większości szczegółów. Zapytania przedstawione w tej sekcji są całkiem

proste. Pamiętaj jednak, że MongoDB pozwala na wykonywanie znacznie bardziej skomplikowanych zapytań, takich jak wyszukiwanie tekstowe i agregacje, którymi zajmiemy się w dalszych rozdziałach.

Zaczynamy od zapoznania się ze standardową metodą `find()`. Poniżej przedstawiłem dwie operacje wyszukiwania, które można przeprowadzić względem zbioru danych.

```
2.1.4 :020 > $users.find({"last_name" => "kowalski"}).to_a
2.1.4 :020 > $users.find({"age" => {"$gt" => 30}}).to_a
```

Pierwsze zapytanie wyszukuje wszystkie dokumenty użytkowników, w których `last_name` ma wartość `kowalski`. Natomiast w drugim zapytaniu dopasowujemy wszystkie dokumenty, w których wartość `age` jest większa niż `30`. Spróbuj wprowadzić w powłoce `irb` drugie z przedstawionych zapytań.

```
2.1.4 :020 > $users.find({"age" => {"$gt" => 30}})
 => #<Mongo::Collection::View:0x70210212601420 namespace='tutorial.users
@selector={"age"=>{"$gt"=>30}} @options={}>
```

Wynik zostaje zwrócony w obiekcie `Mongo::Collection::View` rozszerzającym `Iterable` i ułatwiającym iterację przez wyniki. Szczegółowe omówienie kursorów znajdziesz w sekcji 3.2.3. W międzyczasie spróbuj pobrać wyniki zapytania `$gt`.

```
> cursor = $users.find({"age" => {"$gt" => 30}})
  cursor.each do |doc|
    puts doc["last_name"]
  end
```

W powyższym fragmencie kodu używamy iteratora `each`, który przekazuje każdy wynik do bloku kodu. Atrybut `last_name` jest następnie wyświetlany w konsoli. Operator `$gt` użyty w zapytaniu jest operatorem MongoDB, a znak `$` nie ma żadnego związku ze znakiem `$` umieszczanym przed zmiennymi globalnymi w języku Ruby, na przykład w `$users`. Jeżeli w kolekcji znajdują się jakiekolwiek dokumenty bez `last_name`, możesz dostrzec wyświetlenie `nil` (czyli wartości typu `null` w języku Ruby), co oznacza brak wartości. Dlatego też `nil` w wyniku nie jest niczym nadzwyczajnym.

Konieczność zastanawiania się nad kursorami może okazać się niespodzianką, biorąc pod uwagę przedstawione w poprzednim rozdziale przykłady wykonywane w powłoce. Jednak powłoka używa kursorów w dokładnie taki sam sposób jak każdy sterownik języka. Różnica polega na tym, że powłoka automatycznie przeprowadza iterację przez pierwszych 20 wyników, gdy używasz wywołania `find()`. Jeśli chcesz pobrać pozostałe wyniki, możesz ręcznie kontynuować iterację przez wydanie polecenia `it`.

3.1.4. Operacje uaktualnienia i usunięcia

Przypomnij sobie z rozdziału 2., że operacja *uaktualnienia* wymaga przekazania przynajmniej dwóch argumentów: selektora zapytania oraz dokumentu uaktualnienia. Poniżej przedstawiłem prosty przykład wykorzystujący sterownik języka Ruby.

```
> $users.find({"last_name" => "kowalski"}).update_one({"$set" => {"city" => "Gliwice"}})
```

Kup ksi k Pole ksi k

Ta operacja uaktualnienia wyszukuje pierwszego użytkownika, dla którego wartością `last_name` jest `kowalski`. Jeżeli taki użytkownik zostanie znaleziony, to wartością dla `city` będzie `Gliwice`. W tej operacji wykorzystujemy operator `$set`. Możesz wykonać poniższe zapytanie, aby sprawdzić wprowadzenie zmiany.

```
> $users.find({"last_name" => "kowalski"}).to_a
```

Widok pozwala na podjęcie decyzji o tym, czy uaktualniony ma być tylko jeden dokument, czy wszystkie dopasowane do zapytania. Choćby nawet w poprzednim przykładzie znalazło się wielu użytkowników o nazwisku `kowalski`, uaktualniony będzie i tak jeden dokument. W celu wykonania operacji względem określonego użytkownika konieczne jest dodanie kolejnych warunków do selektora zapytania. Jeżeli faktycznie chcesz przeprowadzić uaktualnienie wszystkich dokumentów, to metodę `update_one()` musisz zastąpić metodą `update_many()`.

```
> $users.find({"last_name" => "kowalski"}).update_many({"$set" => {"city" => "Gliwice"}})
```

Usuwanie danych jest znacznie prostsze. Tę operację omówiłem podczas przedstawiania sposobu pracy w powłoce MongoDB, a sterownik Ruby pod tym względem działa dokładnie tak samo. Dla przypomnienia: musisz użyć po prostu metody `remove()`. Wymieniona metoda pobiera opcjonalny selektor zapytania, który usuwa jedynie dokumenty dopasowane do selektora. Jeżeli nie zostanie dostarczony selektor, usunięte będą wszystkie dokumenty w kolekcji. Poniżej pokazałem przykład usunięcia wszystkich dokumentów użytkowników, dla których atrybut `age` ma wartość równą `40` lub większą.

```
> $users.find({"age" => {"$gte" => 40}}).delete_one
```

Powyższe zapytanie spowoduje usunięcie tylko pierwszego dopasowania spełniającego podane kryteria. Jeżeli chcesz usunąć wszystkie dokumenty dopasowane do kryteriów, musisz skorzystać z następującego zapytania:

```
> $users.find({"age" => {"$gte" => 40}}).delete_many
```

W przypadku braku argumentów metoda `drop()` powoduje usunięcie wszystkich pozostałych dokumentów.

```
> $users.drop
```

3.1.5. Polecenia bazy danych

W poprzednim rozdziale poznałeś ogólnie polecenia bazy danych, a dokładnie zapoznałeś się z dwoma poleceniami `stats`. Tutaj dowiesz się, jak można wykonywać polecenia z poziomu sterownika, jako przykład wykorzystamy polecenie `listDatabases`. To jedno z wielu poleceń, które muszą być wykonywane podczas administrowania bazą danych. Wymienione polecenie będzie traktowane w sposób specjalny po włączeniu uwierzytelniania. Więcej informacji szczegółowych dotyczących uwierzytelniania bazy danych znajdziesz w rozdziale 10.

Przede wszystkim musimy zacząć od utworzenia w języku Ruby obiektu bazy danych odwołującego się do bazy danych `admin`. Następnie metodzie `command()` przekazujemy specyfikację zapytania.

```
> $admin_db = $client.use('admin')
> $admin_db.command({"listDatabases" => 1})
```

Zwróć uwagę na fakt, że działanie powyższego kodu zależy od zawartości skryptu *connect.rb*, ponieważ w zmiennej `$client` oczekujemy informacji o połączeniu z MongoDB. Otrzymana odpowiedź to struktura danych hash zawierająca listę wszystkich istniejących baz danych oraz informację o ilości zajmowanego przez nie miejsca na dysku.

```
#<Mongo::Operation::Result:70112905054200 documents=[{"databases"=>[
{
    "name"=>"local",
    "sizeOnDisk"=>83886080.0,
    "empty"=>false
},
{
    "name"=>"tutorial",
    "sizeOnDisk"=>83886080.0,
    "empty"=>false
},
{
    "name"=>"admin",
    "sizeOnDisk"=>1.0, "empty"=>true
}], "totalSize"=>167772160.0, "ok"=>1.0}]>
 => nil
```

Otrzymane dane mogą wydawać się inne niż podczas pracy z powłoką `irb` i sterownikiem MongoDB, ale i tak łatwo uzyskać do nich dostęp. Gdy tylko przywykniesz do przedstawiania dokumentów za pomocą struktury danych hash w języku Ruby, przejście z API powłoki odbędzie się praktycznie bezboleśnie.

Większość sterowników oferuje wygodne funkcje opakowujące polecenia bazy danych. Możesz pamiętać z poprzedniego rozdziału, że metoda `remove()` tak naprawdę nie powoduje usunięcia kolekcji. W celu pozbycia się kolekcji oraz jej wszystkich indeksów konieczne jest użycie metody `drop_collection()`.

```
> db = $client.use('tutorial')
> db['users'].drop
```

Jeżeli nadal masz mieszane uczucia dotyczące użycia MongoDB w połączeniu z językiem Ruby, nie ma w tym nic złego. Większej praktyki nabędziesz, czytając podrozdział 3.3. Teraz zajmiemy się pokrótce omówieniem sposobu działania sterowników MongoDB. To rzuci nieco światła na niektóre aspekty projektowe MongoDB oraz przygotuje Cię do efektywnego użycia sterowników.

3.2. Jak działają sterowniki?

Na tym etapie naturalne będzie to, że zastanawiasz się, co tak naprawdę dzieje się w tle podczas wykonywania poleceń za pomocą sterownika lub powłoki MongoDB. W tym podrozdziale dowiesz się, jak sterownik serializuje dane i prowadzi komunikację z bazą danych.

Kup ksi k Pole ksi k

Wszystkie sterowniki MongoDB odgrywają trzy główne funkcje. Po pierwsze: generują identyfikatory obiektów MongoDB. To są wartości domyślne przechowywane w polach _id wszystkich dokumentów. Po drugie: sterowniki przeprowadzają wszystkie konwersje między charakterystyczną dla danego języka reprezentacją dokumentu i formatem BSON, czyli formatem binarnym używanym przez MongoDB. We wcześniej przedstawionych przykładach sterownik serializował wszystkie struktury danych hash w języku Ruby na format BSON oraz deserializował na postać struktur hash dane BSON zwrócone przez bazę danych.

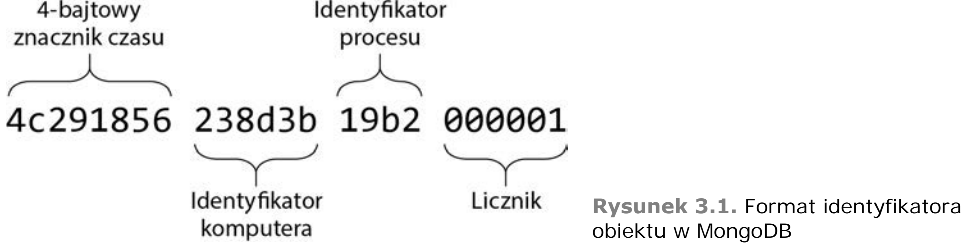
Trzecią i ostatnią funkcją sterownika jest komunikacja z bazą danych poprzez gniazdo TCP i protokół MongoDB. Nie będę się zajmował szczegółami protokołu, ponieważ to zagadnienie wykracza poza zakres tematyczny książki. Jednak styl komunikacji opartej na gnieździe — a w szczególności to, czy przeprowadzane są operacje zapisu, gdy gniazdo oczekuje na odpowiedź — jest ważny i jeszcze powrócimy do tego tematu.

3.2.1. Generowanie identyfikatora obiektu

Każdy dokument MongoDB wymaga klucza podstawowego. Ten klucz, który musi być unikatowy dla wszystkich dokumentów we wszystkich kolekcjach, jest przechowywany w polu _id dokumentu. Programiści mogą używać własnych wartości dla _id, ale w przypadku ich braku użyty będzie identyfikator obiektu MongoDB. Przed przekazaniem dokumentu do serwera sterownik sprawdza, czy istnieje pole _id. Jeżeli wymienionego pola nie ma, wygenerowany zostanie identyfikator obiektu, który następnie będzie przechowywany jako wartość _id.

Identyfikatory obiektów MongoDB są zaprojektowane w sposób zapewniający ich globalną unikatowość, co oznacza gwarancję unikatowości w określonym kontekście. Mógłbyś w tym miejscu zapytać, jak to można zagwarantować. Przeanalizujmy znacznie dokładniej jeden przykład.

Jeżeli wcześniej przeprowadzałeś operacje wstawiania dokumentów do MongoDB, to prawdopodobnie spotkałeś się już z identyfikatorem obiektu. Na początku może się on wydawać ciągiem tekstowym składającym się z losowo wybranych znaków i cyfr, na przykład 4c291856238d3b19b2000001. Prawdopodobnie nie zdawałeś sobie sprawy, że ten ciąg tekstowy to po prostu szesnastkowa reprezentacja 12 bajtów, które w rzeczywistości przechowują pewne użyteczne informacje. Wspomniane bajty mają konkretną strukturę zilustrowaną na rysunku 3.1.



Rysunek 3.1. Format identyfikatora obiektu w MongoDB

Najbardziej znaczące cztery bajty przechowują standardowy znacznika czasu (epoki) systemu UNIX[3]. Trzy kolejne bajty przechowują identyfikator komputera, a dwa następne — identyfikator procesu. Z kolei trzy ostatnie bajty przechowują lokalny dla procesu licznik inkrementowany w trakcie wygenerowania każdego identyfikatora obiektu. Obecność licznika oznacza, że identyfikatory wygenerowane w tym samym procesie i sekundzie nie będą powielone.

Dlaczego identyfikator obiektu ma przedstawiony format? Trzeba koniecznie pamiętać, że te identyfikatory są generowane przez sterownik, a nie bazę danych. To jest różnica względem wielu relacyjnych systemów baz danych, w których inkrementacja klucza podstawowego odbywa się w serwerze, a tym samym powstaje wąskie gardło dla serwera generującego klucz. Jeżeli więcej sterowników niż tylko jeden generuje identyfikator i wstawia dokument, to potrzebują one sposobu na utworzenie unikatowych identyfikatorów bez konieczności prowadzenia wzajemnej komunikacji. Dlatego też połączenie znacznika czasu, identyfikatora komputera i identyfikatora projektu praktycznie eliminuje niebezpieczeństwo, że dwa identyfikatory obiektów będą takie same.

Być może zastanawiasz się, jakie jest prawdopodobieństwo wystąpienia możliwości nałożenia się dwóch identyfikatorów. W praktyce napotkasz inne ograniczenia, zanim osiągniesz szybkość (2^{24} milionów na sekundę) wstawiania dokumentów wymaganą do nałożenia się identyfikatorów w danej sekundzie. Istnieje wręcz znikome prawdopodobieństwo, że będziesz miał do dyspozycji tak dużą liczbę sterowników rozproszonych między wieloma komputerami, aby dwa z nich mogły wygenerować ten sam identyfikator komputera. Na przykład sterownik dla języka Ruby używa przedstawionego poniżej kodu.

```
@@machine_id = Digest::MD5.digest(Socket.gethostname)[0, 3]
```

Aby powyższy kod mógł stanowić problem, musielibyśmy uruchomić proces sterownika MongoDB z takim samym identyfikatorem i mieć tę samą wartość licznika w danej sekundzie. Dlatego też w praktyce nie musisz obawiać się powielenia identyfikatora, to naprawdę jest wręcz nieprawdopodobna sytuacja.

Jedną z zalet użycia identyfikatora obiektu MongoDB jest to, że zawiera on znacznik czasu. Większość sterowników pozwala na wyodrębnienie znacznika czasu, a tym samym bez żadnych dodatkowych kosztów podaje datę i godzinę utworzenia dokumentu z dokładnością do sekundy. Wykorzystując sterownik dla języka Ruby, można wywołać metodę `generation_time()` identyfikatora obiektu i tym samym otrzymać interesującą nas wartość opakowaną obiektem `Time`.

```
irb> require 'mongo'
irb> id = BSON::ObjectId.from_string('4c291856238d3b19b2000001')
```

[3] Wiele komputerów działających pod kontrolą systemów z rodziny UNIX (to obejmuje także systemy Linux) przechowuje wartości czasu w formacie nazywanym „czas systemu UNIX" lub „czas POSIX". Wspomniana wartość jest po prostu liczbą sekund, które upłynęły od północy 1 stycznia 1970 roku nazywanej początkiem epoki. Oznacza to, że znacznik czasu można przechowywać w postaci liczby całkowitej. Na przykład data 2010-06-28 21:47:02 jest przedstawiona jako 1277761622 (lub 0x4c291856 szesnastkowo); podana wartość to liczba sekund, które upłynęły od początku epoki.

Kup ksi k Pole ksi k

```
=> BSON::ObjectId('4c291856238d3b19b2000001')
irb> id.generation_time
=> 2010-06-28 21:47:02 UTC
```

Oczywiście możesz wykorzystać identyfikatory obiektów w celu wykonywania zapytań zakresu wykorzystujących datę i godzinę utworzenia danego obiektu. Gdy na przykład chcesz pobrać wszystkie dokumenty utworzone w czerwcu 2013 roku, możesz przygotować dwa identyfikatory obiektu, których znaczniki czasu będą definiowany daty graniczne, a następnie wystarczy wykonać zapytanie zakresu dla pola `_id`. Ponieważ Ruby oferuje metody przeznaczone do generowania identyfikatorów obiektu na podstawie dowolnego obiektu `Time`, więc kod wykonujący wymienione zadanie jest bardzo prosty[4].

```
jun_id = BSON::ObjectId.from_time(Time.utc(2013, 6, 1))
jul_id = BSON::ObjectId.from_time(Time.utc(2013, 7, 1))
@users.find({'_id' => {'$gte' => jun_id, '$lt' => jul_id}})
```

Jak wcześniej wspomniałem, istnieje możliwość samodzielnego zdefiniowania wartości `_id`. To może mieć sens w sytuacjach, w których jedno pole dokumentu jest ważne i zawsze ma unikatową wartość. Na przykład w kolekcji użytkowników pole `_id` może przechowywać nazwę użytkownika, a nie identyfikator obiektu. Oba podejścia mają swoje zalety, a wybór konkretnego sprowadza się do preferencji programisty.

3.3. Utworzenie prostej aplikacji

Przystępujemy teraz do utworzenia prostej aplikacji przeznaczonej do archiwizacji oraz wyświetlania komunikatów publikowanych w serwisie Twitter. Możesz sobie wyobrazić, że to jest komponent większej aplikacji pozwalającej użytkownikom na zachowanie kart wyników wyszukiwania zgodnych z ich zainteresowaniami. Ten przykład pokaże, jak łatwo można używać danych JSON pochodzących z API serwisu Twitter oraz jak konwertować je na postać dokumentów MongoDB. Jeżeli do budowy tej aplikacji wykorzystalibyśmy relacyjną bazę danych, konieczne byłoby wcześniejsze przygotowanie schematu prawdopodobnie składającego się z wielu tabel, a następnie zadeklarowanie tych tabel. Jednak w przypadku użycia MongoDB nie istnieją tego rodzaju wymagania, nadal możemy zachować bogatą strukturę dokumentów zawierających komunikaty serwisu Twitter oraz będziemy mogli efektywnie wykonywać do nich zapytania.

Budowanej aplikacji nadamy nazwę TweetArchiver, będzie składała się z dwóch komponentów: jeden będzie odpowiedzialny za archiwizację komunikatów, a drugi za ich wyświetlanie. Komponent wyświetlający będzie wywoływał API wyszukiwania serwisu Twitter, a następnie przechowywał znalezione komunikaty. Z kolei komponent wyświetlający pokaże wyniki w przeglądarce WWW.

[4] Przedstawiony przykład w rzeczywistości nie działa, został pokazany jako rodzaj ćwiczenia do przemyślenia. Na tym etapie powinieneś mieć już wiedzę wystarczającą do utworzenia danych, do których będzie można wykonywać zapytania zwracające pewne dane. Dlaczego nie poświęcić chwili czasu i nie wypróbować tego samodzielnie?

Kup ksi k Pole ksi k

3.3.1. Konfiguracja

Aplikacja wymaga czterech bibliotek języka Ruby. Kod źródłowy w repozytorium dla tego rozdziału zawiera plik o nazwie *Gemfile* z listą niezbędnych bibliotek. Przejdź do katalogu roboczego dla tego rozdziału i upewnij się, że po wydaniu polecenia `ls` widzisz plik *Gemfile*. Teraz wszystkie wymagane biblioteki możesz zainstalować z poziomu powłoki systemu za pomocą wymienionych poniżej poleceń.

```
$ gem install bundler
$ bundle install
```

W ten sposób gwarantujemy instalację biblioteki `bundler`. Kolejnym krokiem jest instalacja pozostałych bibliotek za pomocą narzędzi przeznaczonych do zarządzania pakietami Bundler. Narzędzia `bundler` są dość powszechnie używane w świecie języka Ruby i pomagają w zapewnieniu, że używane biblioteki będą dopasowane do wskazanych wersji, czyli wersji wykorzystanych w przykładowych fragmentach kodu.

W naszym pliku *Gemfile* zostały wymienione biblioteki `mongo`, `twitter`, `bson` i `sinatra`, więc one wszystkie będą zainstalowane. Z biblioteki `mongo` korzystaliśmy już wcześniej, ale dołączamy ją, aby mieć pewność, że używamy odpowiedniej wersji. Biblioteka `twitter` jest konieczna do prowadzenia komunikacji z API serwisu Twitter. Natomiast `sinatra` to framework przeznaczony do uruchomienia prostego serwera WWW w języku Ruby; nieco dokładniej zajmiemy się nim w sekcji 3.3.3.

Kod źródłowy omawianego przykładu przedstawię oddzielnie, zaprezentowane tutaj wprowadzenie ma ułatwić jego zrozumienie. Zachęcam Cię do eksperymentowania i wypróbowywania nowych rozwiązań, aby jak najwięcej skorzystać z przedstawionych przykładów.

Dobrym rozwiązaniem jest przygotowanie pliku konfiguracyjnego, który będzie mógł być współdzielony między skryptami archiwizacyjnym i wyświetlającym komunikaty. Utwórz więc plik o nazwie *config.rb* (lub skopiuj go z dostarczonego przez mnie kodu źródłowego) i umieść w nim następujący kod:

```
DATABASE_HOST   = 'localhost'
DATABASE_PORT   = 27017
DATABASE_NAME   = "twitter-archive"
COLLECTION_NAME = "tweets"
TAGS = ["#MongoDB", "#Mongo"]
CONSUMER_KEY    = "zastąp mnie własnym"
CONSUMER_SECRET = "zastąp mnie własnym"
TOKEN           = "zastąp mnie własnym"
TOKEN_SECRET    = "zastąp mnie własnym"
```

Na początku podajemy nazwy bazy danych i kolekcji, które będą używane w aplikacji. Następnie przechodzimy do zdefiniowania tablicy wyrażeń wyszukiwania przekazywanych do API serwisu Twitter.

Twitter wymaga rejestracji bezpłatnego konta i aplikacji, aby można było uzyskać dostęp do API. Wspomnianą rejestrację przeprowadzasz na stronie *https://apps.twitter.com/*. Po zarejestrowaniu aplikacji powinieneś zobaczyć stronę z informacjami o uwierzytelnieniu, prawdopodobnie na karcie kluczy API. Będziesz musiał kliknąć przycisk tworzący przeznaczony dla Ciebie token dostępu. Wartości wyświetlone na stronie wprowadź w odpowiednich miejscach przedstawionego powyżej fragmentu kodu.

Kup ksi k Pole ksi k

3.3.2. Zbieranie danych

Kolejnym krokiem jest przygotowanie skryptu archiwizującego komunikaty serwisu Twitter. Pracę zaczniemy od klasy `TweetArchiver`. Twoje zadanie polega na utworzeniu egzemplarza tej klasy z wyrażeniem wyszukiwania. Później wywołujesz metodę `update()` egzemplarza `TweetArchiver`, co powoduje wykonanie wywołania API serwisu Twitter i zapisanie wyników w kolekcji bazy danych MongoDB.

Na początek spójrz na konstruktor klasy.

```
def initialize(tag)
  connection = Mongo::Connection.new(DATABASE_HOST, DATABASE_PORT)
  db         = connection[DATABASE_NAME]
  @tweets    = db[COLLECTION_NAME]
  @tweets.ensure_index([['tags', 1], ['id', -1]])
  @tag = tag
  @tweets_found = 0
  @client = Twitter::REST::Client.new do |config|
    config.consumer_key        = API_KEY
    config.consumer_secret     = API_SECRET
    config.access_token        = ACCESS_TOKEN
    config.access_token_secret = ACCESS_TOKEN_SECRET
  end
end
```

Metoda `initialize()` tworzy egzemplarz połączenia, obiekt bazy danych oraz obiekt kolekcji, w którym będą przechowywane komunikaty serwisu Twitter.

Definiujemy złożony indeks wykorzystujący pola `tags` (kolejność rosnąca) i `id` (kolejność malejąca). Ponieważ zamierzamy wykonywać zapytania dotyczące konkretnego tagu oraz wyświetlać wyniki w kolejności od najnowszych do najstarszych, więc indeks obejmujący pola `tags` (kolejność rosnąca) i `id` (kolejność malejąca) pozwala, aby zapytanie wykorzystało te pola podczas filtrowania i sortowania wyników. Jak możesz zobaczyć, kolejność w indeksie wskazujemy za pomocą liczby `1` (*rosnąca*) i `-1` (*malejąca*). Nie przejmuj się, jeśli to wszystko nie ma jeszcze dla Ciebie sensu. Indeksami zajmiemy się znacznie dokładniej w rozdziale 8.

Trzeba skonfigurować klienta serwisu Twitter z użyciem informacji uwierzytelniania pochodzących z pliku *config.rb*. Ten krok spowoduje powiązanie tych wartości z biblioteką `twitter`, która będzie je wykorzystywać w trakcie wywoływania API Twitter. Ruby ma unikatową składnię często stosowaną w tego rodzaju konfiguracji; zmienna `config` jest przekazywana do bloku kodu Ruby, w którym przypisujemy jej wartości.

MongoDB pozwala na wstawienie danych niezależnie od ich struktury. W przypadku relacyjnej bazy danych każda tabela potrzebuje doskonale zdefiniowanego schematu, który z kolei wymaga wcześniejszego zaplanowania wartości, jakie będą przechowywane. W przyszłości serwis Twitter może zmienić używane API. W takim przypadku zostaną zwrócone inne wartości, które będą wymagały zmiany schematu, jeśli chcesz przechowywać te wartości dodatkowe. Ten problem nie występuje w przypadku MongoDB. Pozbawiony schematu projekt tej bazy danych pozwala na zapis dokumentu otrzymanego z API serwisu Twitter bez przejmowania się dokładnym formatem.

Biblioteka `twitter` dla języka Ruby zwraca strukturę danych hash, więc można ją przekazać bezpośrednio obiektowi kolekcji MongoDB. W klasie `TweetArchiver` umieść przedstawioną poniżej metodę egzemplarza.

```
def save_tweets_for(term)
  @client.search(term).each do |tweet|
    @tweets_found += 1
    tweet_doc = tweet.to_h
    tweet_doc[:tags] = term
    tweet_doc[:_id] = tweet_doc[:id]
    @tweets.insert_one(tweet_doc)
  end
end
```

Przed zapisaniem każdego dokumentu komunikatu serwisu Twitter wprowadzamy dwie drobne modyfikacje. W celu uproszczenia późniejszych zapytań wyrażenie wyszukiwania dodajemy do atrybutu `tags`. W polu `_id` można zapisać identyfikator komunikatu, w ten sposób zastępujemy klucz podstawowy kolekcji i gwarantujemy, że każdy komunikat będzie dodany tylko jednokrotnie. Następnie zmodyfikowany dokument przekazujemy metodzie `save()`.

W celu użycia kodu zdefiniowanego w klasie potrzebny jest nam jeszcze pewien kod dodatkowy. Przede wszystkim trzeba skonfigurować sterownik MongoDB, aby nawiązywał połączenie z właściwym demonem `mongod` oraz używał wybranej przez nas bazy danych i kolekcji. To jest prosty kod, który powielasz za każdym razem, gdy wykorzystujesz MongoDB. Następnym krokiem jest konfiguracja biblioteki `twitter` i podanie w niej własnych danych uwierzytelniających. Ten krok jest niezbędny, ponieważ serwis Twitter pozwala na użycie jego API tylko zarejestrowanym programistom. W listingu 3.1 możesz zobaczyć uaktualnioną wersję metody `update()`, która wyświetla informacje zwrotne oraz wywołuje `save_tweets_for()`.

Listing 3.1. Kod zawarty w pliku archiver.rb. To jest klasa przeznaczona do pobierania komunikatów serwisu Twitter oraz archiwizacji ich w bazie danych MongoDB

```
$LOAD_PATH << File.dirname(__FILE__)
require 'rubygems'
require 'mongo'
require 'twitter'
require 'config'

class TweetArchiver

def initialize(tag)
    client =
      Mongo::Client.new(["#{DATABASE_HOST}:#{DATABASE_PORT}"],:database =>
      "#{DATABASE_NAME}")
    @tweets    = client["#{COLLECTION_NAME}"]
    @tweets.indexes.drop_all
    @tweets.indexes.create_many([
      { :key => { tags: 1 }},
      { :key => { id: -1 }}
      ])
    @tag = tag
    @tweets_found = 0

    @client = Twitter::REST::Client.new do |config|
      config.consumer_key        = "#{API_KEY}"
      config.consumer_secret     = "#{API_SECRET}"
```

Utworzenie nowego egzemplarza klasy TweetArchive.

Konfiguracja klienta serwisu Twitter, aby używał wartości znalezionych w pliku config.rb.

```
    config.access_token        = "#{ACCESS_TOKEN}"
    config.access_token_secret = "#{ACCESS_TOKEN_SECRET}"
  end
end

def update
  puts "Rozpoczęcie wyszukiwania w serwisie Twitter dla '#{@tag}'..."
  save_tweets_for(@tag)
  print "#{@tweets_found} Komunikaty zostały zapisane.\n\n"
end

private
def save_tweets_for(term)
  @client.search(term).each do |tweet|
    @tweets_found += 1
    tweet_doc = tweet.to_h
    tweet_doc[:tags] = term
    tweet_doc[:_id] = tweet_doc[:id]
    @tweets.insert_one(tweet_doc)
  end
end
end
```

Konfiguracja klienta serwisu Twitter, aby używał wartości znalezionych w pliku config.rb.

Opakowanie dla metody save_tweets_for().

Wyszukiwanie za pomocą klienta serwisu Twitter i zapisanie wyników w MongoDB.

Pozostało jeszcze utworzenie skryptu przeznaczonego do wykonania kodu klasy `Tweet` ↪`Archive` dla każdego z wyszukiwanych wyrażeń. Utwórz plik o nazwie *update.rb* (lub skopiuj z dostarczonych przeze mnie materiałów) i umieść w nim poniższy fragment kodu.

```
$LOAD_PATH << File.dirname(__FILE__)
require 'config'
require 'archiver'

TAGS.each do |tag|
  archive = TweetArchiver.new(tag)
  archive.update
end
```

Teraz uruchom skrypt, wydając polecenie:

```
$ ruby update.rb
```

Otrzymasz kilka komunikatów informujących o znalezieniu odpowiednich komunikatów w serwisie Twitter i zapisaniu ich w bazie danych. Działanie skryptu możesz zweryfikować przez przejście do powłoki MongoDB i wykonanie zapytań bezpośrednio do kolekcji.

```
> use twitter-archive
switched to db twitter-archive
> db.tweets.count()
30
```

Najważniejsze tutaj jest to, że jedynie kilka wierszy kodu[5] wystarczyło do przygotowania aplikacji przeznaczonej do zarządzania komunikatami pochodzącymi z serwisu Twitter i ich przechowywania. Kolejnym zadaniem jest wyświetlenie wyników.

[5] Istnieje możliwość przygotowania rozwiązania wymagającego jeszcze mniejszej ilości kodu. Jego opracowanie pozostawiam jednak jako ćwiczenie dla czytelników.

Kup ksi k Pole ksi k

3.3.3. Wyświetlenie zawartości archiwum

Framework sieciowy Sinatra wykorzystamy do opracowania prostej aplikacji odpowiedzialnej za wyświetlanie wyników. Sinatra pozwala na zdefiniowanie punktów końcowych aplikacji sieciowej oraz bezpośrednie wskazanie odpowiedzi. Potęga tego frameworka kryje się w jego prostocie. Na przykład zawartość strony głównej aplikacji można zdefiniować, jak pokazałem w poniższym fragmencie kodu.

```
get '/' do
  "odpowiedź"
end
```

Powyższy fragmentu kodu wskazuje, że żądania `GET` wykonywane do punktu końcowego `/` aplikacji mają zwrócić klientowi wartość `odpowiedź`. Za pomocą przedstawionego formatu można opracowywać pełne aplikacje sieciowe z wieloma punktami końcowymi, które będą wykonywały dowolny kod w języku Ruby przed udzieleniem odpowiedzi. Więcej informacji na temat frameworka Sinatra oraz jego pełną dokumentację znajdziesz w witrynie *http://www.sinatrarb.com/*.

Przechodzimy teraz do pliku o nazwie *viewer.rb*, który umieścimy w katalogu, gdzie znajdują się pozostałe skrypty. Teraz utwórz podkatalog o nazwie *views* i umieść w nim plik o nazwie *tweets.erb*. Po wykonaniu tych kroków struktura projektu powinna przedstawiać się, jak pokazałem poniżej.

```
- config.rb
- archiver.rb
- update.rb
- viewer.rb
- /views
  - tweets.erb
```

Także na tym etapie pliki możesz utworzyć samodzielnie lub skopiować z materiałów przygotowanych przeze mnie. Przejdź do pliku *viewer.rb* i umieść w nim kod przedstawiony w listingu 3.2.

Listing 3.2. Plik viewer.rb. Aplikacja frameworka Sinatra przeznaczona do wyświetlania komunikatów serwisu Twitter pobranych z archiwum

```
$LOAD_PATH << File.dirname(__FILE__)
require 'rubygems'
require 'mongo'
require 'sinatra'      <-- ❶ Wymagane biblioteki.
require 'config'
require 'open-uri'

configure do
  client = Mongo::Client.new(["#{DATABASE_HOST}:#{DATABASE_PORT}"], :database
    => "#{DATABASE_NAME}")
  TWEETS = client["#{COLLECTION_NAME}"]      <-- ❷ Utworzenie kolekcji dla komunikatów.
end

get '/' do
  if params['tag']
    selector = {:tags => params['tag']}      <-- ❸ Dynamiczne przygotowanie selektora zapytania...
  else
```

```
    selector = {}    ← ❹ ... lub użycie pustego.
  end

  @tweets = TWEETS.find(selector).sort(["id", -1])    ← ❺ Wykonanie zapytania.
  erb :tweets    ← ❻ Wygenerowanie widoku.
end
```

W pierwszych wierszach mamy polecenia wczytujące niezbędne biblioteki oraz plik konfiguracyjny ❶. Następnie w kodzie znajduje się blok konfiguracyjny odpowiedzialny za nawiązanie połączenia z MongoDB i przechowywanie w stałej `TWEETS` odwołania do kolekcji `tweets` ❷.

Najbardziej interesujący kod aplikacji zaczyna się w linii `get '/' do`. Kod w tym bloku obsługuje żądania wykonywane do głównego adresu URL aplikacji. Zaczynamy od przygotowania selektora zapytania. Jeżeli parametr `tags` adresu URL został podany, tworzymy selektor zapytania ograniczający zbiór wynikowy do wskazanych tagów ❸. W przeciwnym razie tworzymy pusty selektor zwracający wszystkie dokumenty w kolekcji ❹. Kolejnym krokiem jest wykonanie zapytania ❺. W tym momencie powinieneś już wiedzieć, że wynik przypisany zmiennej `@tweets` nie jest faktycznym wynikiem, ale kursorem. Iterację przez ten kursor przeprowadzimy w widoku.

Ostatni wiersz ❻ powoduje wygenerowanie pliku widoku o nazwie *tweets.erb* (patrz listing 3.3).

Listing 3.3. Plik tweets.erb. Dokument HTML z osadzonym kodem w języku Ruby, przeznaczony do wyświetlania komunikatów serwisu Twitter

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <style>
    body {
      width: 1000px;
      margin: 50px auto;
      font-family: Palatino, serif;
      background-color: #dbd4c2;
      color: #555050;
    }
    h2 {
      margin-top: 2em;
      font-family: Arial, sans-serif;
      font-weight: 100;
    }
  </style>
</head>
<body>
<h1>Archiwum komunikatów serwisu Twitter</h1>
<% TAGS.each do |tag| %>
  <a href="/?tag=<%= URI::encode(tag) %>"><%= tag %></a>
<% end %>
<% @tweets.each do |tweet| %>
  <h2><%= tweet['text'] %></h2>
  <p>
    <a href="http://twitter.com/<%= tweet['user']['screen_name'] %>">
```

Kup ksi k Pole ksi k

```
      <%= tweet['user']['screen_name'] %>
    </a>
    on <%= tweet['created_at'] %>
  </p>
  <img src="<%= tweet['user']['profile_image_url'] %>" width="48" />
<% end %>
</body>
</html>
```

Większość kodu w listingu 3.3 to po prostu zwykły kod HTML z dodatkiem **ERB** (ang. *embedded Ruby*, czyli osadzony Ruby). Aplikacja frameworka Sinatra uruchamia plik *tweets.erb* poprzez procesor ERB i cały kod Ruby znaleziony między znacznikami `<%` i `%>` przetwarza w kontekście aplikacji.

Najważniejsze fragmenty znajdują się blisko końca — to dwa iteratory. Pierwszy przeprowadza iterację przez listę tagów, co ma na celu utworzenie łączy ograniczających zbiór wynikowy do wskazanego tagu. Natomiast drugi iterator, zaczynający się od kodu `@tweets.each`, przeprowadza iterację przez wszystkie komunikaty serwisu Twitter w celu wyświetlenia tekstu komunikatu, daty jego utworzenia oraz obrazu profilu użytkownika, który opublikował dany komunikat. Wyniki możesz zobaczyć po uruchomieniu aplikacji za pomocą poniższego polecenia.

```
$ ruby viewer.rb
```

Jeżeli aplikacja zostanie uruchomiona bez błędów, otrzymasz standardowe komunikaty podczas uruchamiania programu zbudowanego na podstawie frameworka Sinatra, które będą podobne do przedstawionych poniżej.

```
$ ruby viewer.rb
[2013-07-05 18:30:19] INFO  WEBrick 1.3.1
[2013-07-05 18:30:19] INFO  ruby 1.9.3 (2012-04-20) [x86_64-darwin10.8.0]
== Sinatra/1.4.3 has taken the stage on 4567 for development with backup from
WEBrick
[2013-07-05 18:30:19] INFO  WEBrick::HTTPServer#start: pid=18465 port=4567
```

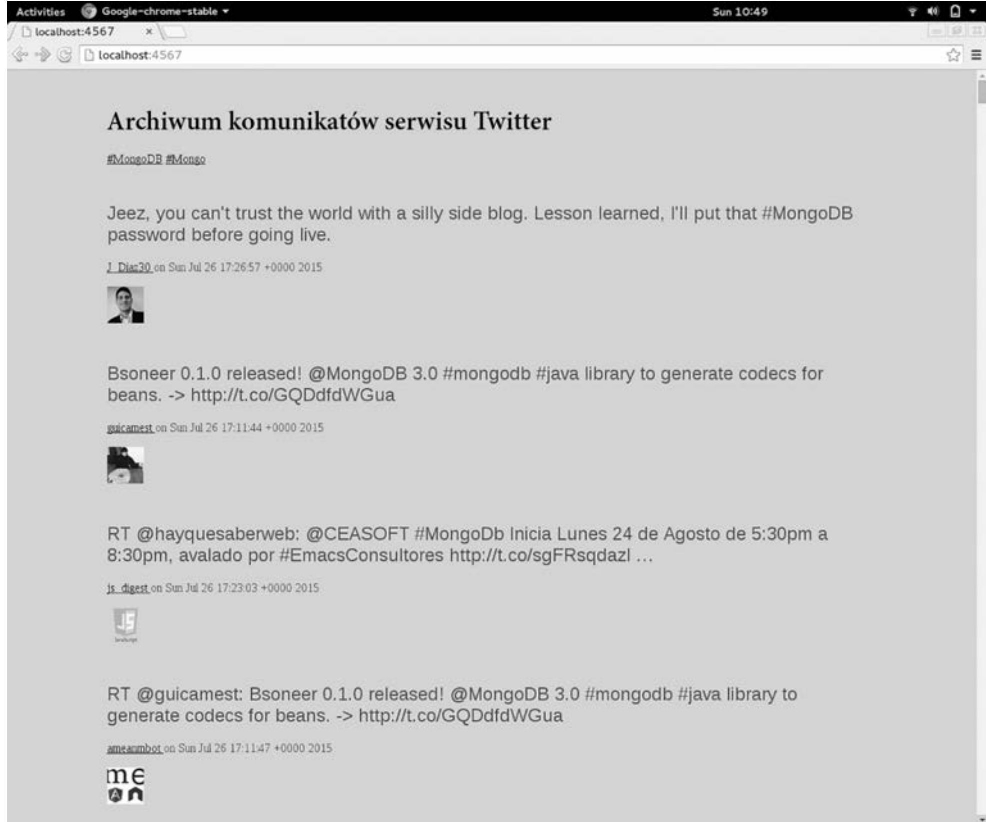
Teraz w przeglądarce WWW przejdź pod adres *http://localhost:4567*. Wyświetlona strona powinna być podobna do pokazanej na rysunku 3.2. Spróbuj kliknąć dowolne łącze na początku strony, aby tym samym zawęzić wyniki do wybranego tagu.

W ten sposób zbudowaliśmy pewnego rodzaju aplikację. Wprawdzie jest bardzo prosta, ale pokazała, jak łatwo można wykorzystać MongoDB. Nie musiałeś wcześniej definiować schematu, mogłeś wykorzystać zalety indeksów drugorzędnych w celu przyśpieszenia zapytań i uniknięcia wstawiania powielających się danych. Ponadto zastosowałeś względnie prostą integrację z użytym językiem programowania aplikacji.

3.4. Podsumowanie

W tym rozdziale poznałeś podstawy pracy z MongoDB za pomocą języka programowania Ruby. Zobaczyłeś, z jaką łatwością można przedstawiać dokumenty MongoDB w języku Ruby, a także jak podobne są API CRUD języka Ruby i powłoki MongoDB. W pewnym stopniu zagłębiliśmy się w bazę danych, poznałeś ogólny sposób działania

Kup ksi k Pole ksi k



Rysunek 3.2. Dane wyjściowe naszej aplikacji wygenerowane przez przeglądarkę WWW

sterowników, a także dokładną konstrukcję identyfikatora obiektów w MongoDB. Poruszyłem także kwestie formatu BSON i protokołu sieciowego używanego przez MongoDB. Na końcu zajęliśmy się budową prostej aplikacji, aby pokazać wykorzystanie MongoDB do pracy z rzeczywistymi danymi. Wprawdzie użycie MongoDB w rzeczywistym świecie będzie często wiązało się ze znacznie większym poziomem skomplikowania niż pokazany w rozdziale, ale perspektywa tworzenia aplikacji opierających działanie na bazie danych jest już dla Ciebie w zasięgu ręki.

Począwszy od rozdziału 4., będziemy wykorzystywać całą zdobytą dotąd wiedzę. W szczególności przyjrzymy się możliwości zbudowania aplikacji typu e-commerce wykorzystującej bazę danych MongoDB. Ponieważ to jest duży projekt, skoncentrujemy się wyłącznie na kilku kwestiach back endu. Przedstawię pewne modele danych dla tej domeny, a także pokażę, jak wstawiać i pobierać dane w przypadku tego rodzaju aplikacji.

Kup ksi k   Pole ksi k

Część II

Tworzenie aplikacji w MongoDB

W drugiej części książki zagłębimy się w oparty na dokumentach model danych w MongoDB, stosowany język zapytań oraz operacje CRUD (tworzenie, odczyt, uaktualnianie i usuwanie danych).

Do wymienionych tematów podejdziemy konkretnie przez progresywne projektowanie modelu danych dla aplikacji typu e-commerce oraz operacji CRUD niezbędnych do zarządzania tego rodzaju danymi. W poszczególnych rozdziałach zajmę się omówieniem wybranego tematu od początku do końca. Najpierw przedstawię przykłady wraz z odpowiednim fragmentem aplikacji typu e-commerce, a następnie zaprezentuję dokładne omówienie. W trakcie pierwszej lektury rozdziału możesz skoncentrować się wyłącznie na przykładach aplikacji e-commerce, a szczegółowe wyjaśnienie tematu pozostawić na później lub też na odwrót.

W rozdziale 4. poznasz wybrane reguły projektowe stosowane podczas przygotowywania schematu, a następnie opracujesz podstawowy model danych dla aplikacji typu e-commerce przeznaczony do przechowywania informacji o produktach, kategoriach, użytkownikach, zamówieniach oraz opiniach użytkowników o produktach. Następnie zobaczysz, jak MongoDB organizuje dane na poziomie bazy danych, kolekcji i dokumentów. Ten rozdział zawiera również podsumowanie podstawowych typów danych formatu BSON.

Kup ksi k		Pole ksi k

W rozdziale 5. zajmiemy się językiem zapytań MongoDB. Dowiesz się, jak wykonywać najczęściej stosowane zapytania do modelu danych opracowanego w rozdziale 4. Później w dalszej części rozdziału dokładnie omówię semantykę operacji pobierania danych przez zapytanie.

Rozdział 6. został poświęcony agregacji. Najpierw zobaczysz, jak można przeprowadzać proste operacje grupowania. Natomiast w dalszej części rozdziału zagłębimy się we framework agregacji MongoDB.

Tematem rozdziału 7. są operacje uaktualniania i usuwania danych przeprowadzane w MongoDB. Ponadto poznasz powody utworzenia w rozdziale 4. tego konkretnego modelu danych dla budowanej aplikacji typu e-commerce. Zobaczysz, jak można obsługiwać hierarchię kategorii oraz jak w sposób transakcyjny zarządzać magazynem produktów. Szczegółowo omówię operacje uaktualniania i przedstawię przy okazji polecenie `findAndModify` oferujące naprawdę potężne możliwości.

Kup ksi k
Pole ksi k

Skorowidz

A

administracja, 429
  indeksem, 249
  serwerem, 73
adresowanie, 414
AGPL, gnu-affero general public license, 36
agregacja, 51, 153
  w aplikacji, 156
  w klastrze shardingu, 410
alokacja, 114
analityka, 46
analiza wolno wykonywanych zapytań, 260
antywzorce, 493
API, application programming interface, 315
API Pluggable Storage Engine, 315, 334
API silnika magazynu danych, 52
aplikacja, 99
  biblioteki Ruby, 90
  wyświetlenie zawartości archiwum, 94
  zbieranie danych, 91
aplikacje
  monitorowania, 446
  sieciowe, 45
  typu e-commerce, 30, 104, 130, 156, 196
architektura, 432
  32-bitowa, 472
  64-bitowa, 472
archiwum, 94
atrybuty dynamiczne, 490
automatyczne indeksowanie, 292
automatyzacja MMS, 441
awaria, 359, 367, 427
  serwera konfiguracji, 428
  serwera mongos, 428

B

baza danych, 42, 113
  oparta na dokumentach, 45
bezpieczeństwo, 51, 450
  kluczy, 454
bezpieczne środowisko, 450
blokada, 338, 435
  optymistyczna, 196
brak
  katalogu danych, 477
  możliwości dołączenia do portu, 478
  uprawnień, 478
budowa klastra shardingu, 394
buforowanie, 46
  planu zapytania, 279

C

chmura, 440
ciągi tekstowe, 123, 173
CRUD, create, read, update, delete, 55, 99
CSV, Comma-Separated Values, 40

D

dane
  binarne, 497
  oparte na dokumentach, 101
data i godzina, 124, 175
dedykowane silniki wyszukiwania tekstowego, 291
defragmentacja, 254
demon mongod, 37
denormalizacja, 161
deskryptory pliku, 437
diagnostyka, 442
dodanie
  indeksu, 263, 421
  shardu, 421, 424
dokładne dopasowanie, 280
dokument, 25
  Google, 384
  JSON, 27
  kategorii, 108
  użytkownika, 111
dokumenty, 101
  głęboko zagnieżdżone, 494
  modyfikowanie, 172
  niepodzielne przetwarzanie, 206
  ograniczenia, 121, 125
  określenie języka, 310
  serializacja, 121
  typy, 121
  uaktualnienia, 192
  usuwanie, 225
  wstawiania danych, 126
  wykluczanie, 301
  z określonym kluczem, 140
  zagnieżdżone, 107
dopasowanie
  częściowe, 133
  subdokumentów, 140
  wyrażenia, 301
  wzorca, 286, 288
dostępne języki, 313
dowiązanie symboliczne, 439
drzewo, 486
  Log Structure Merge, 332
dynamiczne tworzenie atrybutów, 30
dysk, 434
działanie
  metody explain(), 261
  replikacji, 353
  silnika magazynu danych, 333
  sterowników, 86
dziennik zdarzeń, 353
  replikacji, 358

E

efektywność indeksu, 240
eksport danych, 460
ERB, embedded Ruby, 96

F

finalizacja zamówienia, 207, 209
format
 BSON, 123, 149
 dokumentu, 25
 JSON, 57
framework agregacji, 50, 154
funkcja
 $add, 175
 $addToSet, 170
 $allElementsTrue, 176
 $and, 176
 $anyElementTrue, 176
 $avg, 170
 $cmp, 176
 $concat, 174
 $cond, 176
 $dayOfMonth, 175
 $dayOfWeek, 175
 $dayOfYear, 175
 $divide, 175
 $eq, 176
 $first, 170
 $gt, 176
 $gte, 176
 $hour, 175
 $ifNull, 176
 $last, 170
 $let, 177
 $literal, 177
 $lt, 176
 $lte, 176
 $map, 177
 $max, 170
 $meta, 177
 $millisecond, 175
 $min, 170
 $minute, 175
 $mod, 175
 $month, 175
 $multiply, 175
 $ne, 176
 $not, 176
 $or, 176
 $push, 170
 $second, 175
 $setDifference, 176
 $setEquals, 176
 $setIntersection, 176
 $setIsSubset, 176
 $setUnion, 176
 $size, 177
 $strcasecmp, 174
 $substr, 174
 $subtract, 175
 $sum, 170
 $toLower, 174
 $toUpper, 174
 $week, 175
 $year, 175
 .count(), 185
 .distinct(), 185
 ensureIndex(), 52
 explain(), 179, 181, 182
 map(), 187
 reduce(), 187
funkcje, 27
 $group, 170
 arytmetyczne, 174
 ciągu tekstowego, 173
 daty i godziny, 175
 logiczne, 175
 modelu MapReduce, 185
 powłoki, 65
 zabezpieczeń, 459

G

generowanie identyfikatora
 obiektu, 87
GNU-AGPL, 37
GridFS, 499

H

hierarchia kategorii, 198
historia MongoDB, 49

I

identyfikator obiektu, 87
import danych, 460
indeks, 466
 klucza złożonego, 281
 obejmujący pojedynczy
 klucz, 238
 w postaci pojedynczego
 klucza, 280
 złożony, 235
indeksowanie, 68, 233, 234, 416
 klastra shardingu, 406
 offline, 254
 pól tekstowych kolekcji, 298
 przestrzenne, 49, 51
 reguły, 238
 typy indeksów, 244
 w klastrze shardingu, 408
 w tle, 253
indeksy, 31, 66
 hash, 247
 pokrywające, 282
 przestrzenne, 32
 przestrzenne, 248
 rzadkie, 49
 rzadkie, 245
 składające się z wielu
 kluczy, 246
 unikatowe, 244
 wyszukiwania tekstowego,
 296, 298
 złożone, 238
informacje o bazie danych, 73
instalacja, 471
 języka Ruby, 480
 sterownika dla języka Ruby,
 80
 w systemie Linux, 472
 w systemie OS X, 474
 w systemie Windows, 475
interakcje zapytań, 465
izolacja, 226

J

JavaScript
 operatory zapytań, 145
język
 Ruby, 80
 zapytań, 134
JSON, JavaScript Object
 Notation, 25, 57

K

kategorie, 130
klaster, 421
  shardingu, 389, 394, 406, 430
klucz shardu, 411, 412, 415
kolejki procesów roboczych, 489
kolekcja, 58, 66, 117
  kubełków, 494
  niemożliwe do shardingu, 495
  replset.minvalid, 354
  shardingu, 398
  specjalne, 118
  TTL, 50, 120
  użytkowników, 133, 494
kompilacja, 477
komponenty
  klastra shardingu, 387
  shardingu, 394
kompresja, 50
  danych, 462
konfiguracja
  klastra, 394, 397
  uwierzytelniania podstawowego, 456
  zbioru replik, 346
  magazynów danych, 322
  shardingu, 423
konsola
  javascript, 38, 55
  sieciowa, 445
kopia zapasowa, 254, 447–450
  klastra shardingu, 426
kopiowanie plików danych, 448
kryteria zapytania, 135
księgowanie, 49, 438
kursory, 83
  agregacji, 159

L

liczby, 123
Linux, 472
lista produktów, 132
lokalizacja, 492
LSM, long-structured merge-trees, 32
LVM, logical volume manager, 435

M

macierz dysków, 435
magazyn, 210, 212
magazyny danych typu
  klucz-wartość, 41, 42
  proste, 41
  zaawansowane, 42
mechanizm równoważenia obciążenia, 427
menedżer
  pakietów, 473, 475
  woluminów logicznych, 435
metadane, 389
metoda
  currentOp(), 252
  explain(), 52, 68, 71, 261
  find(), 130
  findOne(), 130
  hint(), 277
migawka, 449
migracja bazy danych, 320
migracje, 404
MMAPv1, 321
MMS, mongodb management system, 441
model
  MapReduce, 185, 188
  skalowalności, 42
model danych, 27, 42
  klucz-wartość, 41
  oparty na dokumencie, 27
  pozbawiony schematu, 29
  w aplikacji, 157
  w relacyjnej bazie danych, 29
modyfikacja
  dokumentów, 172
  przez zastąpienie, 193
  za pomocą operatora, 193
monitorowanie, 442
  MMS, 450

N

na wystąpienie awarii, 372
nakładanie blokad, 50, 338
naprawa, 50, 461
naprawa po awarii, 359, 367, 427
narzędzia
  diagnostyczne, 443
  do profilowania zapytań, 257
  powłoki, 39
narzędzie
  bsondump, 445
  Bsondump, 40
  mongodump, 39, 447
  mongoexport, 40, 460
  mongofiles, 503
  mongoimport, 40, 460
  mongooplog, 40
  mongoperf, 40
  mongorestore, 39, 447
  mongosniff, 40, 444
  mongostat, 40, 444
  mongotop, 40, 444
niepodzielność, 226
niepoprawne indeksowanie, 493
nieprawidłowa architektura, 477
niezawodność, 33
normalizacja, 23
NTP, network time protocol, 438

O

obliczenia
  średniej liczby opinii, 159
  wstępne, 492
obsługa
  dla silnika magazynu danych, 52
  LSM, 32
  shardingu, 49, 423
  SSL, 453
  stemmingu, 292
  wielu języków, 292
ocena wyszukiwania tekstowego, 302, 304
odporność na awarie, 427
odwołanie, 483
ograniczenia, 47, 151
  replikacji, 344
określenie języka w dokumencie, 310

opcja
allowDiskUse, 183
cursor w agregacji, 184
limit, 151
skip, 151
opcje
dokumentu
konfiguracyjnego, 363
konfiguracyjne, 478
pliku konfiguracyjnego, 319
potoku agregacji, 179
replikacji, 362
uaktualnień, 215
zapytania, 149
operacja
typu upsert, 216
uaktualnienia, 61, 64
upsert, 204
zastąpienia, 61
operacje
CRUD, 99
odczytu, 83, 377
uaktualnienia, 84
usunięcia, 84
wstawiania, 121
zapisu, 51, 412
operator
$addToSet, 221, 229
$all, 137
$bit, 222, 229
$each, 219, 221, 230
$exists, 140
$group, 169
$in, 137
$inc, 217, 229
$isolated, 230
$limit, 171
$match, 171
$nin, 137
$or, 139
$out, 162, 172
$pop, 221, 229
$project, 162, 168, 173
$pull, 223, 229
$pullAll, 223, 229
$push, 219, 229
$pushAll, 219, 229
$rename, 218, 229
$set, 61, 217, 229
$setOnInsert, 229
$skip, 171
$slice, 219, 230
$sort, 171, 220, 230
$unset, 217, 229
$unwind, 162, 171
operatory
boolowskie, 138
frameworka agregacji, 156
potoku agregacji, 168
uaktualnienia, 216, 229
uaktualnienia tablicy, 219
zapytania, 145
zapytania
nieskategoryzowane, 148
zbiorów, 136, 176
opinie, 112, 159
o produkcie, 201
o produktach, 130
opracowanie modelu danych,
104
optymalizacja zapytania, 255,
269
optymistyczna kontrola
współbieżności, 196
OS X, 474
osadzenie, 483

P

PaaS, platform-as-a-service, 26
pamięć masowa, 385
dla danych aplikacji, 388
pamięć RAM, 433
para klucz-wartość, 379
partycjonowanie ręczne, 423
pewność udanego zapisu, 375
pętla REPL, 82
plan zapytania
buforowanie, 279
wyświetlanie, 277
planowe wyłączenie serwera,
343
plik
archiver.rb, 92
connect.rb, 82
tweets.erb, 95
viewer.rb, 94
pliki
BSON, 448
danych, 114
GridFS, 503
konfiguracyjne, 322
pobieranie
danych, 58, 294
danych z magazynu, 210
określonego zakresu, 134
tablicy, 144
podstawy schematu, 105
podsumowanie sprzedaży, 165
podział, 404, 413
fragmentów danych, 422
pojedynczy węzeł, 430
pole _id, 58
polecenia, 75
bazy danych, 85
diagnostyczne, 443
SQL, 156
stats, 85
polecenie
irb, 83
repairDatabase, 461
połączenia
kontrolne, 359
z pojedynczym węzłem, 372
zbioru replik, 373
pominięcie, 151
pomoc, 76, 466
potok agregacji, 154, 168, 178
wydajność działania, 178
powłoka, 55, 56
irb, 82
powtórzenie profilowania, 263
pozycja strony, 289
predykat zapytania, 59
preferencje odczytu, 377
priorytety zbiorów replik, 49
problemy, 477
z wydajnością, 463
procesor, 433
produkty, 130
profilowanie, 263
danych wyjściowych, 258
zapytań, 257
programowanie, 79
zwinne, 46
projekcje, 149
provisioning, 418, 430, 440
przechowywanie
metadanych, 389
miniatury, 498
obiektów binarnych, 497
wartości MD5, 498

przekazanie predykatu do zapytania, 59
przekierowywanie operacji, 388
przywracanie danych, 448

R

RAID, 435
RDBMS, relational database management systems, 44
reakcja na wystąpienie awarii, 372
reguły
  indeksowania, 238
  projektowe schematu, 102
rejestracja, 118
  danych, 46, 442
rejestrowanie transakcji, 34
relacyjne bazy danych, 42, 44
REPL, read, evaluate, print loop, 82
replikacja, 32, 341
  dziennik zdarzeń, 353, 358
  konfiguracja, 346
  ograniczenia, 344
  opcje, 362
  strategie wdrożeń, 370
  trwale wstrzymana, 358
  typu główny-podległy, 358
  wycofanie, 360
  zatwierdzenie, 360
rodziny baz danych, 42
router mongos, 388
routing zapytania, 406
rozłożenie
  obciążenia, 386
  użycia pamięci masowej, 385
rozproszenie danych, 389–392
rozwiązywanie problemów, 477
równoważenie obciążenia, 427
Ruby
  API CRUD, 82, 96
  GridFS, 500
  instalacja sterownika, 80
  operacje uaktualnienia, 84
  operacje usunięcia, 84
  tworzenie aplikacji, 89
  wstawianie dokumentów, 82
  zapytania, 83

S

schemat, 29
  reguły projektowe, 102
schematy zmienne, 47
selektor dopasowania, 135
selektory, 135
serwer, 26
  mongod, 394
  MongoDB, 36
  mongos, 394
serwery konfiguracji, 389
sharding, 49, 51, 383, 384
  istniejącej kolekcji, 422
  kolekcje, 398
  kolekcji, 394
  konserwacja, 423
  monitorowanie, 423
  na podstawie kolekcji, 392
  na podstawie nadawcy, 416
  na podstawie odbiorcy, 417
  obsługa, 423
  podgląd danych, 403
  pustej kolekcji, 399
  rezygnacja, 425
  uwzględniający tagi, 50
  w produkcji, 418
silnik
  JavaScript V8, 51
  MMAPv1, 51, 326
  WiredTiger, 315–318, 326
silniki
  magazynów danych, 316, 332
  wyszukiwarek
    internetowych, 291
skalowanie, 35
  operacji odczytu, 376
  pionowe, 36
  poziome, 36
  systemu, 383
skrypty
  do importu danych, 460
  sprawdzające wydajność, 327
  wstawiania danych, 323
slug, 106
słownik stop-listy, 312
sortowanie, 150, 281, 304
  w zapytaniu, 131
spadek wydajności, 464
sprawdzenie stanu shardów, 401
sprzedaż, 165
SSL, secure sockets layer, 452
SSL w klastrze, 453
stan
  shardów, 401
  zamówienia, 207
  zbioru replik, 367
standardowe operatory uaktualnienia, 216
stany węzłów zbioru replik, 368
stemming, 311, 372
sterownik dla języka Ruby, 80
sterowniki
  bazy danych, 39
  języków, 39
stosowanie shardingu, 385
strategie wdrożeń, 370
struktura
  B-tree, 242, 243, 335
  danych, 334
  Log Structure Merge, 332
  typu B-tree, 337
  związku, 108
symulacja rejestracji, 118
system
  kolekcji, 121
  operacyjny Linux, 472
  operacyjny OS X, 474
  operacyjny Windows, 475
  plików, 436
szybkość działania, 33
szyfrowanie komunikacji sieciowej, 451

Ś

średnia
  liczba opinii, 159
  ocena produktu, 196
środowisko
  produkcyjne, 45, 471
  wdrożenia, 432

T

tablice, 142, 219
tagi, 379
teoria indeksowania, 234
test wydajności, 323, 326, 330
  odczytu danych, 329
thrashing, 464

topologia klastra, 430
topologie wdrożenia, 418
transakcje, 491
TTL, time-to-live, 120
tworzenie
  aplikacji, 89, 99
  baz danych, 58
  indeksów, 66, 249, 251
  kolekcji, 66
  kopii zapasowej, 448, 450
  migawki, 449
  programów, 79
  zapytań, 129
typ datetime, 124
typy
  danych, 149
  indeksów, 244
  uaktualnień, 215
  wirtualne, 125

U

uaktualnianie
  opinii, 203
  skomplikowanych danych, 62
  wielu dokumentów, 215
uaktualnienia, 215, 462
  dokumentu, 60, 192
  ilości produktów, 205
  pozycyjne, 223
  w aplikacji, 196
unikatowość, 414
uruchomienie
  komponentów shardingu, 394
  powłoki, 56
  serwerów, 394
usługa monitorowania, 446
ustalanie pozycji strony, 289
ustawienie
  nearest, 377
  primary, 377
  primaryPreferred, 377
  secondary, 377
  secondaryPreferred, 377
usunięcie, 215
  danych, 64
  dokumentu, 225
  indeksu, 249
  shardu, 424
  użytkownika, 457
uwierzytelnianie, 454
  klastra shardingu, 459
  podstawowe, 456
  usługi, 454
  użytkownika, 455
  plikiem klucza, 458
  X509, 459
  zbioru replik, 457
użycie
  funkcji obszarów, 292
  indeksów, 66
  menedżera pakietów, 473, 475
  pamięci masowej, 385
  replikacji, 344
  zindeksowanego klucza, 266
użytkownicy, 109, 133

W

waga
  pola, 303
  słowa, 303
wartości rozdzielone przecinkami, 40
wczytywanie danych, 295
wdrożenia, 45, 418, 421, 429, 466
wdrożenie w środowisku produkcyjnym, 471
wersje MongoDB, 49
wielkość
  dziennika zdarzeń replikacji, 358
  indeksu wyszukiwania tekstowego, 297
Windows, 475
WiredTiger, 315
współbieżność, 226
wstawianie
  danych, 58, 126
  dokumentów, 82
wybór klucza shardu, 411
wydajność, 463, 464
  działania potoku agregacji, 178
  odczytu danych, 329
  operacji odczytu danych, 327
  silników magazynów danych, 326
  uaktualnienia, 227
wyjątek localhost, 458
wykluczanie dokumentów, 301
wyniki
  stemmingu, 312
  testów wydajności, 326
wyrażenia regularne, 146
wystąpienie awarii, 367
wyszukanie
  najlepszych klientów, 166
  stron internetowych, 288
wyszukiwanie tekstowe, 50, 285, 288, 291
  dedykowane silniki, 291
  dopasowanie wzorca, 286
  indeksy, 296
  korzyści, 292
  koszty, 292
  proste, 299
  w innych językach, 308
  we frameworku agregacji, 304
  zaawansowane, 300
  znaki wieloznaczne, 298
wyszukiwanie zaawansowane, 302
wyświetlanie
  listy produktów, 132
  planów zapytania, 277
  oceny wyszukiwania tekstowego, 303
  zawartości archiwum, 94
wzorce
  projektowe, 483
  zapytania, 280

Z

zadania administracyjne, 459
zakresy, 135
zamówienia, 109, 133, 164, 203
  autoryzacja, 208
  finalizacja, 207
  finalizacja, 209
  weryfikacja, 208
zanik zasilania, 343
zapis, 375, 412
  danych, 400
zapytania, 83, 129, 406
  $and, 49
  $text, 51
  ad hoc, 31

do shardu klastra, 407
filtrujące, 187
w aplikacji, 130
zakresu, 281
zakresu, 67
zarządzanie
bazami danych, 113
kolekcjami, 117
magazynem, 212
produktami, 209
zasoby dodatkowe, 52
zastąpienie, 193, 194
zbiory, 137, 176
replik, 345, 430
zbiór roboczy, 463
zegar, 438
zewnętrzne aplikacje
monitorowania, 446
zliczanie opinii, 160
złączanie kolekcji, 161
złączenia, 163
zmaterializowany widok, 173
zmiana stanu zamówienia, 207
znacznik czasu BSON, 354
znaki wieloznaczne, 298
związek typu
„Jeden do wielu”, 108, 483
„Wiele do wielu”, 108, 485

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

Bazy danych są kluczowymi elementami systemów informatycznych. Choć zwykle pojęcie to kojarzy się z relacyjnymi bazami danych i skomplikowanymi zapytaniami pisanymi w języku SQL, istnieją również zupełnie inne, bardzo wartościowe rozwiązania. Właśnie takim jest MongoDB — rozwijany na zasadach *open source* nierelacyjny system zarządzania bazą danych napisany w języku C++. Dane są tu składowane jako obiekty JSON, co umożliwia intuicyjne, bezproblemowe ich przetwarzanie w aplikacji. MongoDB jest dynamicznie rozwijanym projektem. Charakteryzuje się dużą skalowalnością, elastycznością i wszechstronnością.

Książka ta jest przeznaczona dla programistów i administratorów baz danych, którzy chcą poznać MongoDB od podstaw. Ten świetnie napisany przewodnik okaże się również bezcenną pomocą dla średnio zaawansowanych użytkowników systemu. Przedstawiono tu zarówno podstawy MongoDB, jak i zaawansowane metody optymalizacji, skalowania bazy i administrowania nią. Nie zabrakło opisu dobrych praktyk w zakresie wdrażania aplikacji MongoDB i rozwiązywania problemów. Zamieszczoną tu wiedzę ilustrują liczne przykłady kodu napisanego w JavaScript czy Ruby.

Zagadnienia omówione w książce:

- podstawowe informacje na temat bazy danych MongoDB, jej budowy, przeznaczenia i funkcjonowania
- tworzenie aplikacji wykorzystujących MongoDB
- indeksowanie i optymalizacja zapytań
- silnik magazynu danych WiredTiger i obsługa wtyczek
- zapewnienie wysokiej dostępności danych i skalowalność systemu
- najlepsze praktyki wdrażania instalacji MongoDB, administrowania nimi i rozwiązywania problemów

Kyle Banker brał udział w rozwijaniu MongoDB. Obecnie pracuje w startupie. **Peter Bakkum** jest programistą o dużym doświadczeniu w pracy z MongoDB. **Shaun Verch** był członkiem zespołu, który przygotował podstawowy serwer dla MongoDB. Inżynier firmy Genentech **Doug Garrett** jest jednym ze zwycięzców MongoDB Innovation Award for Analytics. **Tim Hawkins** jest architektem oprogramowania. Kierował zespołem, który rozwijał funkcję wyszukiwania w Yahoo! Europe.