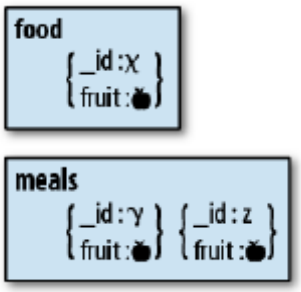
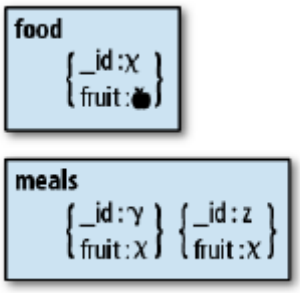
**Sztuczka 1: Powiel dane dla szybkości, dane referencyjne dla integralności**

Dane używane w wielu dokumentach mogą być osadzane (zdenormalizowane) lub przywoływane (znormalizowane). Denormalizacja nie jest lepsza niż normalizacja i na odwrót: każda z nich ma swoje własne kompromisy i powinieneś zrobić wszystko, co będzie najlepiej działać z twoją aplikacją. Denormalizacja może prowadzić do niespójnych danych: załóżmy, że chcesz zmienić jabłko w gruszkę jak na rysunku 1. Jeśli zmienisz wartość w jednym dokumencie, ale aplikacja ulegnie awarii, zanim będziesz mógł zaktualizować inne dokumenty, Twoja baza danych będzie miała dwie różne wartości dla pływających fruit.




Znormalizowany schemat. Pole owocowe jest przechowywane w kolekcji żywności i do którego odnoszą się dokumenty w kolekcji posiłków.

Niespójność nie jest wielka, ale poziom „nie-wielkości” zależy od tego, co przechowujesz. W przypadku wielu aplikacji krótkie okresy niespójności są w porządku: jeśli ktoś zmieni swoją nazwę użytkownika, może nie mieć znaczenia, że stare posty będą wyświetlane z jego starą nazwą użytkownika przez kilka godzin. Jeśli nawet chwilowe niespójne wartości nie są w porządku, należy przejść do normalizacji. Jeśli jednak normalizujesz, Twoja aplikacja musi wykonać dodatkowe zapytanie za każdym razem, gdy chce dowiedzieć się, co to jest owoc. Jeśli Twoja aplikacja nie może sobie pozwolić na to uderzenie w wydajność, a późniejsze uzgodnienie niespójności będzie w porządku, należy zdenormalizować




Schemat zdenormalizowany. Wartość owoców jest przechowywana zarówno w zbiorach żywności, jak i posiłków

To kompromis: nie możesz mieć zarówno najszybszej wydajności, jak i gwarantowanej natychmiastowej spójności. Musisz zdecydować, co jest ważniejsze dla twojego wniosku.

**Przykład: zamówienie w koszyku**

Załóżmy, że projektujemy schemat aplikacji koszyka. Nasza aplikacja przechowuje zamówienia w MongoDB, ale jakie informacje powinno zawierać zamówienie?

**Schemat znormalizowany**

Produkt:

{

"_id" : productId,

"name" : name,

"price" : price,

"desc" : description

}

Zamówienie:

{

"_id" : orderId,

"user" : userInfo,

"items" : [

productId1,

productId2,

productId3

]

}

Przechowujemy _id każdego produktu w dokumencie zamówienia. Następnie, kiedy wyświetlamy zawartość zamówienia, sprawdzamy kolekcję zamówień, aby uzyskać prawidłowe zamówienie, a następnie wysyłamy zapytanie do kolekcji produktów, aby uzyskać produkty powiązane z naszą listą _id. Nie ma możliwości uzyskania pełnego zamówienia w pojedynczym zapytaniu z tym schematem. Jeśli informacje o produkcie zostaną zaktualizowane, wszystkie dokumenty odnoszące się do tego produktu ulegną „zmianie”, ponieważ dokumenty te wskazują jedynie na dokument ostateczny. Normalizacja zapewnia wolniejsze odczyty i spójny obraz wszystkich zamówień; wiele dokumentów może się zmienić atomowo (ponieważ zmienia się tylko dokument referencyjny).

Schemat zdenormalizowany

Produkt (taki sam jak poprzedni):

{

"_id" : productId,

"name" : name,

"price" : price,

"desc" : description

}

Zamówienie:

{

"_id" : orderId,

"user" : userInfo,

"items" : [

{

"_id" : productId1,

"name" : name1,

"price" : price1

},

{

"_id" : productId2,

"name" : name2,

"price" : price2

},

{

"_id" : productId3,

"name" : name3,

"price" : price3

}

]

}

Informacje o produkcie przechowujemy jako osadzony dokument w zamówieniu. Następnie, kiedy wyświetlamy zamówienie, wystarczy wykonać jedno zapytanie. Jeśli informacje o produkcie są aktualizowane i chcemy, aby zmiana była propagowana w zamówieniach, musimy aktualizować każdy koszyk osobno. Denormalizacja zapewnia nam szybsze odczyty i mniej spójny obraz wszystkich zamówień; Szczegóły produktu nie mogą być zmieniane atomowo w wielu dokumentach. Biorąc więc pod uwagę te opcje, jak zdecydujesz, czy normalizować, czy denormalizować?

**Czynniki decyzyjne**

Należy wziąć pod uwagę trzy główne czynniki:

• Czy płacisz cenę za każdy odczyt z powodu bardzo rzadkich zmian danych? Możesz odczytać produkt 10 000 razy za każdym razem, gdy zmieniają się jego szczegóły. Czy chcesz zapłacić karę za każde z 10 000 odczytów, aby zapis był nieco szybszy lub gwarantował spójność? Większość aplikacji wymaga dużo więcej odczytu niż zapisu: sprawdź, jakie masz proporcje. Jak często dane, o których myślisz, faktycznie się zmieniają? Im mniej się zmienia, tym silniejszy argument za denormalizacją. Prawie nigdy nie warto odwoływać się do rzadko zmieniających się danych, takich jak imiona i nazwiska, daty urodzenia, symbole giełdowe i adresy.

• Jak ważna jest konsekwencja? Jeśli spójność jest ważna, powinieneś przejść do normalizacji. Na przykład załóżmy, że wiele dokumentów musi atomowo zobaczyć zmianę. Gdybyśmy projektowali aplikację handlową, w której niektórymi papierami wartościowymi można było handlować tylko w określonych porach, chcielibyśmy natychmiast „zablokować" je wszystkie, gdy nie można byłoby nimi handlować. Wtedy moglibyśmy wykorzystać pojedynczy dokument zamka jako odniesienie dla odpowiedniej grupy dokumentów papierów wartościowych. Jednak tego rodzaju rzeczy mogą być lepsze na poziomie aplikacji, ponieważ aplikacja będzie musiała znać zasady określające, kiedy i tak należy blokować i odblokowywać. Inną ważną kwestią jest spójność czasowa w aplikacjach, w których niespójności są trudne do pogodzenia. W przykładzie zamówień mamy ścisłą hierarchię: zamówienia pobierają informacje z produktów, produkty nigdy nie uzyskują informacji z zamówień. Gdyby istniało wiele dokumentów „źródłowych", trudno byłoby zdecydować, który powinien wygrać. Jednak w tym (nieco wymyślonym) zastosowaniu kolejności, spójność może w rzeczywistości być szkodliwa. Załóżmy, że chcemy wystawić produkt na sprzedaż z 20% rabatem. Nie chcemy zmieniać żadnych informacji w istniejących zamówieniach, chcemy tylko zaktualizować opis produktu. Tak więc w tym przypadku chcemy uzyskać migawkę tego, jak dane wyglądały w określonym momencie

• Czy odczyt musi być szybki? Jeśli odczyty muszą być tak szybkie, jak to możliwe, należy zdenormalizować. W tej aplikacji tak się nie dzieje, więc nie ma to znaczenia. Aplikacje działające w czasie rzeczywistym powinny zwykle denormalizować się w jak największym stopniu. Istnieje dobry argument za denormalizowaniem dokumentu zamówienia: informacje niewiele się zmieniają, a nawet jeśli tak się dzieje, nie chcemy, aby zamówienia odzwierciedlały zmiany. Normalizacja nie daje nam tutaj żadnej szczególnej przewagi. W takim przypadku najlepszym wyborem jest zdenormalizowanie schematu zamówień

**Sztuczka 2 : Dokonaj normalizacji, jeśli potrzebujesz danych zabezpieczonych w przyszłości**

Normalizacja „zabezpiecza dane w przyszłości": powinieneś być w stanie używać znormalizowanych danych dla różnych aplikacji, które będą w przyszłości przesyłać zapytania do danych na różne sposoby. Zakłada się, że masz zbiór danych, z których aplikacja po aplikacji będzie musiała korzystać przez wiele lat. Istnieją takie zestawy danych, ale dane większości ludzi stale się rozwijają, a stare dane są aktualizowane lub znikają. Większość ludzi chce, aby ich baza danych działała jak najszybciej w przypadku zapytań, które robią teraz, a jeśli zmienią te zapytania w przyszłości, zoptymalizują swoją bazę danych pod kątem nowych zapytań. Ponadto, jeśli aplikacja się powiedzie, jej zestaw danych często staje się bardzo specyficzny dla aplikacji. Nie oznacza to, że nie można go używać do więcej niż jednej aplikacji; często będziesz chciał przynajmniej przeprowadzić na nim metaanalizę. Ale to nie to samo, co „zabezpieczanie przyszłości", aby sprostać wszelkim zapytaniom, które ludzie chcą odpowiedzieć za 10 lat.

**Sztuczka 3: Spróbuj pobrać dane w jednym zapytaniu**

W tej sekcji jednostka aplikacji jest używana jako termin ogólny dla niektórych prac związanych z aplikacjami. Jeśli masz aplikację internetową lub mobilną, możesz myśleć o jednostce aplikacji jako o żądaniu do zaplecza. Kilka innych przykładów:

• W przypadku aplikacji komputerowej może to być interakcja użytkownika.

• W przypadku systemu analitycznego może to być jeden załadowany wykres.

Zasadniczo jest to dyskretna jednostka pracy, którą wykonuje aplikacja, która może obejmować dostęp do bazy danych. Schematy MongoDB powinny być zaprojektowane do wykonywania zapytań na jednostkę aplikacji.

**Przykład: blog**

Gdybyśmy projektowali aplikację blogową, prośba o wpis na blogu mogłaby być jedną jednostką aplikacji. Kiedy wyświetlamy post, zależy nam na treści, tagach, niektórych informacjach o autorze (chociaż prawdopodobnie nie o całym profilu) i komentarzach do posta. W ten sposób osadzilibyśmy wszystkie te informacje w dokumencie wiadomości i moglibyśmy pobrać wszystko, co jest potrzebne do tego widoku w jednym zapytaniu. Należy pamiętać, że celem jest jedno zapytanie, a nie jeden dokument na stronę: czasami możemy zwrócić wiele dokumentów lub części dokumentów (nie każde pole). Na przykład strona główna może zawierać ostatnie dziesięć postów z kolekcji postów, ale tylko ich tytuł, autora i podsumowanie:

```
> db.posts.find({}, {"title" : 1, "author" : 1, "slug" : 1, "_id" : 0}).sort(
... {"date" : -1}).limit(10)
```

Dla każdego tagu może istnieć strona, która zawierałaby listę ostatnich 20 postów z danym tagiem:

```
> db.posts.find({"tag" : someTag}, {"title" : 1, "author" : 1,
... "slug" : 1, "_id" : 0}).sort({"date" : -1}).limit(20)
```

Powstałaby oddzielna kolekcja autorów, która zawierałaby pełny profil każdego autora. Strona autorska jest prosta, byłby to po prostu dokument z kolekcji autorów:

```
> db.authors.findOne ({"name": authorName})
```

Dokumenty w kolekcji postów mogą zawierać podzbiór informacji, które pojawiają się w dokumencie autora, np. Nazwisko autora i miniaturowe zdjęcie profilowe. Zwróć uwagę, że jednostka aplikacji nie musi odpowiadać pojedynczemu dokumentowi, chociaż zdarza się to w niektórych z wcześniej opisanych przypadków (post na blogu i strona autora są zawarte w jednym dokumencie). Jednak istnieje wiele przypadków, w których jednostka aplikacji składałaby się z wielu dokumentów, ale dostępnych za pośrednictwem jednego zapytania.

**Przykład: tablica graficzna**

Załóżmy, że mamy tablicę z obrazkami, na której użytkownicy publikują wiadomości składające się z obrazu i tekstu w nowym lub istniejącym wątku. Następnie jednostka aplikacji przegląda 20 wiadomości w wątku, więc post każdej osoby będzie oddzielnym dokumentem w kolekcji postów. Kiedy chcemy wyświetlić stronę, wykonamy zapytanie:

```
> db.posts.find ({"threadId": id}). sort ({"date": 1}). limit (20)
```

Następnie, gdy chcemy uzyskać następną stronę wiadomości, będziemy wyszukiwać następnych 20 wiadomości w tym wątku, a następnie 20 kolejnych itd .:

> db.posts.find ({"threadId": id, "date": {"$ gt": latestDateSeen}}). sort (

... {"date": 1}). limit (20)

Następnie moglibyśmy umieścić indeks na {threadId: 1, date: 1}, aby uzyskać dobrą wydajność tych zapytań.

Nie używamy skip (20), ponieważ zakresy działają lepiej w przypadku paginacji.

Ponieważ aplikacja staje się bardziej skomplikowana, a użytkownicy i menedżerowie żądają więcej funkcji, nie martw się, jeśli musisz wykonać więcej niż jedno zapytanie na jednostkę aplikacji. Cel polegający na jednym zapytaniu na jednostkę jest dobrym punktem wyjścia i miernikiem do oceny początkowego schematu, ale rzeczywisty świat jest chaotyczny. W przypadku każdej wystarczająco złożonej aplikacji prawdopodobnie skończysz z wykonaniem więcej niż jednego zapytania dotyczącego jednej z bardziej absurdalnych funkcji aplikacji.

**Sztuczka 4 : Osadź pola zależne**

Rozważając osadzenie dokumentu lub odniesienie do niego, zadaj sobie pytanie, czy będziesz wyszukiwać informacje w tym polu samodzielnie, czy tylko w ramach większego dokumentu. Na przykład możesz chcieć wykonać zapytanie dotyczące tagu, ale tylko do linku z powrotem do postów z tym tagiem, a nie do samego tagu. Podobnie w przypadku komentarzy, możesz mieć listę ostatnich komentarzy, ale ludzie są zainteresowani przejściem do wpisu, który zainspirował komentarz (chyba że komentarze są obywatelami pierwszej kategorii w Twojej aplikacji). Jeśli korzystasz z relacyjnej bazy danych i przeprowadzasz migrację istniejącego schematu do MongoDB, tabele łączenia są doskonałymi kandydatami do osadzania. Tabele, które są w zasadzie kluczem i wartością - na przykład tagi, uprawnienia lub adresy - prawie zawsze działają lepiej osadzone w MongoDB. Wreszcie, jeśli tylko jeden dokument dotyczy pewnych informacji, umieść je w tym dokumencie. <br><br>

**Sztuczka 5 : Osadź dane „z punktu w czasie”**

Jak wspomniano w przykładzie zamówień w „Sztuczce 1, w rzeczywistości nie chcesz, aby informacje w zamówieniu uległy zmianie, jeśli produkt, powiedzmy, trafi do sprzedaży lub zostanie nowa miniatura. Wszelkie takie informacje, w których chcesz wykonać migawkę danych w określonym czasie, powinny zostać osadzone. Inny przykład z dokumentu zamówienia: pola adresowe również należą do kategorii danych „punkt w czasie”. Nie chcesz, aby wcześniejsze zamówienia użytkownika uległy zmianie, jeśli zaktualizuje on swój profil. <br><br>

**Sztuczka 6 : Nie osadzaj pól, które mają niezwiązany wzrostaa**

Ze względu na sposób, w jaki MongoDB przechowuje dane, ciągłe dołączanie informacji na końcu tablicy jest dość nieefektywne. Chcesz, aby tablice i obiekty miały dość stały rozmiar podczas normalnego użytkowania. Tak więc można osadzić 20 dokumentów podrzędnych lub 100 lub 1 000 000, ale zrób to z góry. Pozwolenie, aby dokument znacznie się rozrósł w miarę jak jest używany, prawdopodobnie będzie wolniejszy, niż byś chciał. Komentarze są często dziwnymi przypadkami, które różnią się w zależności od aplikacji. W przypadku większości aplikacji komentarze powinny być przechowywane jako osadzone w dokumencie nadrzędnym. Jednak w przypadku wniosków, w których komentarze stanowią odrębny podmiot lub często są ich setki lub więcej, należy je przechowywać jako oddzielne dokumenty. Jako inny przykład załóżmy, że tworzymy aplikację wyłącznie w celu komentowania. Przykład tablicy graficznej w „Sztuczce nr 3” wygląda następująco; główną treścią są komentarze. W tym przypadku chcielibyśmy, aby komentarze były oddzielnymi dokumentami.

**Sztuczka 7: Wypełnij wstępnie wszystko, co możesz**

Jeśli wiesz, że Twój dokument będzie potrzebował pewnych pól w przyszłości, bardziej efektywne jest ich wypełnienie przy pierwszym wstawieniu niż tworzenie pól na bieżąco. Na przykład załóżmy, że tworzysz aplikację do analizy witryny, aby zobaczyć, ilu użytkowników odwiedzało różne strony w ciągu minuty w ciągu dnia. Będziemy mieć kolekcję stron, w której każdy dokument reprezentuje 6-godzinny wycinek strony. Chcemy przechowywać informacje na minutę i na godzinę: {

"_id" : pageId,

"start" : time,

"visits" : {

"minutes" : [

[num0, num1, ..., num59],

[num0, num1, ..., num59],

[num0, num1, ..., num59],

[num0, num1, ..., num59],

[num0, num1, ..., num59],

[num0, num1, ..., num59]

],

"hours" : [num0, ..., num5]

}

}

Mamy tutaj ogromną przewagę: wiemy, jak te dokumenty będą wyglądać od teraz do końca czasu. Będzie taki, który rozpocznie się teraz, z wpisem co minutę przez następne sześć godzin. Wtedy będzie inny taki dokument i jeszcze jeden. W ten sposób moglibyśmy mieć zadanie wsadowe, które wstawia te „szablony" dokumentów w czasie wolnym od ruchu lub w sposób ciągły w ciągu dnia. Ten skrypt może wstawiać dokumenty wyglądające tak, zastępując someTime jakimkolwiek następnym 6-godzinnym interwałem:

{

"_id" : pageId,

"start" : someTime,

"visits" : {

"minutes" : [

[0, 0, ..., 0],

[0, 0, ..., 0],

[0, 0, ..., 0],

[0, 0, ..., 0],

[0, 0, ..., 0],

[0, 0, ..., 0]

],

"hours" : [0, 0, 0, 0, 0, 0]

}

}

Teraz, gdy zwiększasz lub ustawiasz te liczniki, MongoDB nie musi znajdować dla nich miejsca. Po prostu aktualizuje wartości, które już wprowadziłeś, co jest znacznie szybsze. Na przykład na początku godziny Twój program może wykonać coś takiego:

> db.pages.update ({"_ id": pageId, "start": th

isHour},

... {"$ inc": {"views.0.0": 3}})

Pomysł ten można rozszerzyć na inne typy danych, a nawet zbiory i same bazy danych. Jeśli codziennie korzystasz z nowej kolekcji, równie dobrze możesz utworzyć ją z wyprzedzeniem.

**Sztuczka 8: Jeśli to możliwe, przydziel wstępnie miejsce**

Jest to ściśle związane zarówno ze Sztuczką 6 jak i Sztuczką 7. Jest to optymalizacja, jeśli wiesz, że dokumenty zwykle osiągają określony rozmiar, ale zaczynają się od mniejszego rozmiaru. Kiedy po raz pierwszy wstawiasz dokument, dodaj pole śmieci, które zawiera ciąg o rozmiarze, jaki (ostatecznie) będzie miał dokument, a następnie natychmiast odznacz to pole:

> collection.insert({"_id" : 123, /* other fields */, "garbage" : someLongString})

> collection.update({"_id" : 123}, {"$unset" : {"garbage" : 1}})

W ten sposób MongoDB początkowo umieści dokument w miejscu, w którym będzie wystarczająco dużo miejsca na rozwój

**Sztuczka 9: Przechowuj osadzone informacje w tablicach, aby uzyskać anonimowy dostęp**

Często pojawia się pytanie, czy informacje należy osadzać w tablicy, czy w dokumencie podrzędnym. Dokumentów podrzędnych należy używać, gdy zawsze wiesz dokładnie, o co będziesz pytać. Jeśli jest jakakolwiek szansa, że nie wiesz dokładnie, o co pytasz, użyj tablicy. Tablice powinny być zwykle używane, gdy znasz jakieś kryteria dotyczące elementu, o który prosisz. Załóżmy, że programujemy grę, w której gracz podnosi różne przedmioty. Możemy modelować dokument odtwarzacza jako:

{

"_id" : "fred",

"items" : {

"slingshot" : {

"type" : "weapon",

"damage" : 23,

"ranged" : true

},

"jar" : {

"type" : "container",

"contains" : "fairy"

},

"sword" : {

"type" : "weapon",

"damage" : 50,

"ranged" : false

}

}

}

Teraz przypuśćmy, że chcemy znaleźć wszystkie bronie, których obrażenia są większe niż 20. Nie możemy! Dokumenty podrzędne nie pozwalają na sięgnięcie do przedmiotów i stwierdzenie: „Daj mi dowolny przedmiot z uszkodzeniami większymi niż 20”. Możesz zapytać tylko o konkretne pozycje: „Czy items.slingshot.dam wiek przekracza 20 lat? A co z items.sword.damage? ” i tak dalej. Jeśli chcesz mieć dostęp do dowolnego elementu bez znajomości jego identyfikatora, powinieneś ustawić schemat, aby przechowywać elementy w tablicy:

{

"_id" : "fred",

"items" : [

{

"id" : "slingshot",

"type" : "weapon",

"damage" : 23,

"ranged" : true

},

{

"id" : "jar",

"type" : "container",

"contains" : "fairy"

},

{

"id" : "sword",

"type" : "weapon",

"damage" : 50,

"ranged" : false

}

]

}

Teraz możesz użyć prostego zapytania, takiego jak {"items.damage": {"$ gt": 20}}. Jeśli potrzebujesz więcej niż jednego kryterium danego przedmiotu (na przykład obrażeń i dystansu), możesz użyć $ elemMatch. Kiedy więc należy użyć dokumentu podrzędnego zamiast tablicy? Kiedy znasz i zawsze będziesz znać nazwę pola, do którego masz dostęp. Załóżmy na przykład, że śledzimy umiejętności gracza: jego siłę, inteligencję, mądrość, zręczność, konstytucję i charyzmę. Zawsze będziemy wiedzieć, jakiej konkretnej umiejętności szukamy, więc możemy zapisać ją jako:

{

"_id" : "fred",

"race" : "gnome",

"class" : "illusionist",

"abilities" : {

"str" : 20,

"int" : 12,

"wis" : 18,

"dex" : 24,

"con" : 23,

"cha" : 22

}

}

Kiedy chcemy znaleźć określoną umiejętność, możemy wyszukać abilities.str lub abilities.con lub cokolwiek innego. Nigdy nie będziemy chcieli znaleźć umiejętności większej niż 20, zawsze będziemy wiedzieć, czego szukamy.

**Sztuczka 10: Zaprojektuj dokumenty tak, aby były samowystarczalne**

MongoDB ma być dużym, głupim magazynem danych. Oznacza to, że prawie nie przetwarza, po prostu przechowuje i pobiera dane. Powinieneś uszanować ten cel i starać się unikać zmuszania MongoDB do wykonywania jakichkolwiek obliczeń, które można wykonać na kliencie. Nawet „trywialne” zadania, takie jak znajdowanie średnich lub sumowanie pól, powinny być generalnie przekazywane klientowi. Jeśli chcesz zapytać o informacje, które muszą zostać obliczone i nie są wyraźnie obecne w dokumencie, masz dwie możliwości:

• Spowodować poważny spadek wydajności (zmuszając MongoDB do wykonywania obliczeń przy użyciu JavaScript)

• Wyraź informacje w swoim dokumencie

Ogólnie rzecz biorąc, powinieneś po prostu jasno określić informacje w swoim dokumencie. Załóżmy, że chcesz wyszukać dokumenty, w których całkowita liczba jabłek i pomarańczy wynosi 30. Oznacza to, że Twoje dokumenty wyglądają mniej więcej tak:

```
{
"_id" : 123,
"apples" : 10,
"oranges" : 5
}
```

Zapytanie o całość, biorąc pod uwagę powyższy dokument, będzie wymagało JavaScript, a zatem jest bardzo nieefektywne. Zamiast tego dodaj pole sumy do dokumentu:

```
{
"_id" : 123,
"apples" : 10,
"oranges" : 5,
"total" : 15
}
```

Następnie to pole ogółem można zwiększyć, gdy zmieni się jabłka lub pomarańcze:

```
> db.food.update(criteria,
... {"$inc" : {"apples" : 10, "oranges" : -2, "total" : 8}})
> db.food.findOne()
{
"_id" : 123,
"apples" : 20,
"oranges" : 3,
"total" : 23
```

}

Staje się to trudniejsze, jeśli nie masz pewności, czy aktualizacja cokolwiek zmieni. Na przykład załóżmy, że chcesz mieć możliwość zapytania o liczbę rodzajów owoców, ale nie wiesz, czy aktualizacja doda nowy typ. Załóżmy więc, że Twoje dokumenty wyglądały mniej więcej tak:

{

"_id" : 123,

"apples" : 20,

"oranges : 3,

"total" : 2

}

Teraz, jeśli wykonasz aktualizację, która może, ale nie musi, utworzyć nowe pole, czy zwiększasz sumę, czy nie? Jeśli aktualizacja zakończy się tworzeniem nowego pola, należy zaktualizować sumę:

> db.food.update({"_id" : 123}, {"$inc" : {"banana" : 3, "total" : 1}})

I odwrotnie, jeśli pole bananów już istnieje, nie powinniśmy zwiększać sumy. Ale od strony klienta nie wiemy, czy istnieje! Są dwa sposoby radzenia sobie z tym, które prawdopodobnie stają się znajome: szybki, niekonsekwentny i powolny, konsekwentny. Szybkim sposobem jest wybranie dodania lub nie dodania 1 do sumy i poinformowanie naszej aplikacji, że będzie musiała sprawdzić rzeczywistą sumę po stronie klienta. Możemy mieć ciągłe zadanie wsadowe, które koryguje wszelkie niespójności, które otrzymamy. Jeśli nasza aplikacja może zająć dodatkowy czas natychmiast, możemy wykonać findAndModify, która „blokuje” dokument (ustawiając „zablokowane” pole, które inni zapisują ręcznie sprawdzą), zwrócić dokument, a następnie wydać aktualizację, odblokowując dokument i aktualizując pola i zsumuj poprawnie:

> var result = db.runCommand({"findAndModify" : "food",

... "query" : {/* other criteria */, "locked" : false},

... "update" : {"$set" : {"locked" : true}}})

>

> if ("banana" in result.value) {

... db.fruit.update(criteria, {"$set" : {"locked" : false},

... "$inc" : {"banana" : 3}})

... } else {

... // increment total if banana field doesn't exist yet

... db.fruit.update(criteria, {"$set" : {"locked" : false},

... "$inc" : {"banana" : 3, "total" : 1}})

... }

Właściwy wybór zależy od aplikacji.

**Sztuczka 11: Preferuj operatory $ zamiast JavaScript**

Niektórych operacji nie można wykonać za pomocą $ -operatorów. W przypadku większości aplikacji uczynienie dokumentu samowystarczalnym zminimalizuje złożoność zapytań, które musisz wykonać. Jednak czasami będziesz musiał zapytać o coś, czego nie możesz wyrazić za pomocą $ -operatorów. W takim przypadku na ratunek może przyjść JavaScript: możesz użyć klauzuli $ where, aby wykonać dowolny JavaScript jako część zapytania. Aby użyć znaku $ gdzie w zapytaniu, napisz funkcję JavaScript, która zwraca prawdę lub fałsz (niezależnie od tego, czy dokument pasuje do znaku $ gdzie, czy nie). Załóżmy więc, że chcemy zwrócić tylko rekordy, w których wartości elementu [0] .age i member [1] .age są równe. Moglibyśmy to zrobić za pomocą:

```
> db.members.find({"$where" : function() {
... return this.member[0].age == this.member[1].age;
... }})
```

Jak możesz sobie wyobrazić, $ where daje Twoim zapytaniom całkiem sporo mocy. Jednak jest również powolny.

**Za kulisami**

$ gdzie zajmuje dużo czasu z powodu tego, co MongoDB robi za kulisami: kiedy wykonujesz normalne zapytanie (inne niż $ gdzie), twój klient zamienia to zapytanie na BSON i wysyła je do bazy danych. MongoDB również przechowuje dane w BSON, więc może zasadniczo porównać zapytanie bezpośrednio z danymi. Jest to bardzo szybkie i wydajne. Teraz załóżmy, że masz klauzulę $ where, która musi zostać wykonana jako część zapytania. MongoDB będzie musiało utworzyć obiekt JavaScript dla każdego dokumentu w kolekcji, analizując BSON dokumentów i dodając wszystkie ich pola do obiektów JavaScript. Następnie wykonuje JavaScript, który wysłałeś do dokumentów, a następnie zrywa wszystko ponownie. Jest to niezwykle czasochłonne i wymagające dużej ilości zasobów.

**Uzyskanie lepszej wydajności**

$ where to dobry hack, gdy jest to konieczne, ale należy go unikać, gdy tylko jest to możliwe. W rzeczywistości, jeśli zauważysz, że twoje zapytania wymagają dużo $ gdzie, jest to dobra wskazówka, że powinieneś ponownie przemyśleć swój schemat. Jeśli zapytanie $ where jest potrzebne, możesz zmniejszyć wydajność, minimalizując liczbę dokumentów, które trafiają do $ where. Spróbuj wymyślić inne kryteria, które można sprawdzić bez znaku $ where i najpierw je wymień; im mniej dokumentów jest „uruchomionych” do czasu, gdy zapytanie dotrze do $ gdzie, tym mniej czasu zajmie $ gdzie. Na przykład załóżmy, że mamy przykład $ gdzie podany powyżej i zdajemy sobie sprawę, że podczas sprawdzania wieku dwóch członków jesteśmy tylko członkami z co najmniej wspólnym członkostwem, być może członkami rodziny:

```
> db.members.find({'type' : {$in : ['joint', 'family']},
... "$where" : function() {
... return this.member[0].age == this.member[1].age;
... }})
```

Teraz wszystkie dokumenty dotyczące pojedynczego członkostwa zostaną wykluczone do czasu, gdy zapytanie dotrze do $ where.

**Sztuczka 12: Oblicz agregacje na bieżąco**

O ile to możliwe, obliczaj agregacje w czasie za pomocą $ inc. Na przykład w „Porada nr 7: Wypełnij wstępnie wszystko, co możesz" na stronie 8, mamy aplikację analityczną ze statystykami według minuty i godziny. Możemy zwiększać statystyki godzinowe w tym samym czasie, co minutowe. Jeśli Twoje agregacje wymagają więcej zmian (na przykład znalezienia średniej liczby zapytań w ciągu godziny), przechowuj dane w polu minut, a następnie miej ciągły proces wsadowy, który oblicza średnie z ostatnich minut. Ponieważ wszystkie informacje niezbędne do obliczenia agregacji są przechowywane w jednym dokumencie, przetwarzanie to może nawet zostać przekazane klientowi w przypadku nowszych (niezagregowanych) dokumentów. Starsze dokumenty zostałyby już zliczone przez zadanie wsadowe.

**Sztuczka 13: Napisz kod, aby rozwiązać problemy z integralnością danych**

Biorąc pod uwagę bezschematyczny charakter MongoDB i zalety denormalizacji, musisz zachować spójność danych w swojej aplikacji. Wiele ODM ma sposoby na egzekwowanie spójnych schematów na różnych poziomach ścisłości. Istnieją jednak również problemy ze spójnością poruszone powyżej: niespójności danych spowodowane awariami systemu („Wskazówka nr 1: zduplikowane dane dotyczące szybkości, dane referencyjne dotyczące integralności" na stronie 1) oraz ograniczenia aktualizacji MongoDB („Wskazówka nr 10: Projekt dokumentów, aby były samowystarczalne "na stronie 12). W przypadku tego typu niespójności musisz faktycznie napisać skrypt, który sprawdzi Twoje dane. Jeśli zastosujesz się do wskazówek zawartych w tym rozdziale, możesz skończyć z kilkoma zadaniami cron, w zależności od aplikacji. Na przykład możesz mieć:

Naprawa spójności: Sprawdź obliczenia i zduplikowane dane, aby upewnić się, że wszyscy mają spójne wartości.

Pre-populator: Twórz dokumenty, które będą potrzebne w przyszłości.

Agregator: aktualizuj agregacje wbudowane.

Inne przydatne skrypty (niezwiązane ściśle z tym rozdziałem) to:

Narzędzie do sprawdzania schematu: upewnij się, że wszystkie aktualnie używane dokumenty mają określony zestaw pól, albo automatycznie je poprawiając, albo powiadamiając o błędnych.

Zadanie kopii zapasowej: fsync, blokowanie i zrzucanie bazy danych w regularnych odstępach czasu.

Wykonywanie zadań w tle, które sprawdzają i chronią Twoje dane, daje Ci więcej zmęczenia podczas zabawy z nimi.

**Sztuczka 14: Używaj właściwych typów**

Przechowywanie danych przy użyciu odpowiednich typów ułatwi Ci życie. Typ danych wpływa na sposób odpytywania danych, kolejność, w jakiej MongoDB będzie je sortować i ile bajtów zajmuje.

Liczby: każde pole, którego będziesz używać jako liczby, powinno zostać zapisane jako liczba. Oznacza to, że chcesz zwiększyć wartość lub posortować ją w porządku numerycznym. Jednak jaka liczba? Cóż, często nie ma to znaczenia - czasami ma. Sortowanie porównuje jednakowo wszystkie typy liczbowe: jeśli masz 32-bitową liczbę całkowitą, 64-bitową liczbę całkowitą i podwójną o wartościach 2, 1 i 1,5, zostaną one posortowane we właściwej kolejności. Jednak niektóre operacje wymagają pewnych typów: operacje bitowe (AND i OR) działają tylko na polach całkowitych (nie podwajają). Baza danych automatycznie zamieni 32-bitowe liczby całkowite na 64-bitowe liczby całkowite, jeśli będą przepełnione (na przykład z powodu $ inc), więc nie musisz się tym martwić.

Daty: Podobnie jak w przypadku liczb, dokładne daty należy zapisywać przy użyciu typu daty. Jednak daty takie jak urodziny nie są dokładne; kto zna ich czas urodzenia z dokładnością do milisekundy? W przypadku takich dat często równie dobrze sprawdza się użycie dat w formacie ISO: ciąg znaków w formacie rrrr-mm-dd. Pozwoli to poprawnie posortować urodziny i dopasować je bardziej elastycznie, niż gdybyś używał dat, co wymusza dopasowywanie urodzin do milisekund.

Ciągi znaków: wszystkie ciągi w MongoDB muszą być zakodowane w formacie UTF-8, więc ciągi w innych kodowaniach muszą zostać przekonwertowane na UTF-8 lub zapisane jako dane binarne.

ObjectIds: Zawsze zapisuj ObjectIds jako ObjectIds, a nie jako ciągi. Jest to ważne z kilku powodów. Po pierwsze, możliwość zapytań: ciągi znaków nie pasują do identyfikatorów obiektów, a identyfikatory obiektów nie są zgodne z ciągami. Po drugie, ObjectIds są przydatne: większość sterowników ma metody, które mogą automatycznie wyodrębnić datę utworzenia dokumentu z jego ObjectId. Wreszcie ciąg reprezentujący ObjectId jest ponad dwukrotnie większy na dysku niż ObjectId.

**Sztuczka 15: Zastąp _id, jeśli masz własny prosty, niepowtarzalny identyfikator**

Jeśli Twoje dane nie mają naturalnie występującego unikalnego pola (często tak jest), użyj domyślnego ObjectId dla _ids. Jeśli jednak Twoje dane mają unikalne pole i nie potrzebujesz właściwości ObjectId, to przejdź dalej i zastąp domyślny _id - użyj własnej unikalnej wartości. Oszczędza to trochę miejsca i jest szczególnie przydatne, jeśli zamierzasz indeksować swój unikalny identyfikator, ponieważ pozwoli to zaoszczędzić cały indeks w miejscu i zasobach (bardzo znaczące oszczędności). Jest kilka powodów, dla których nie powinieneś używać własnego identyfikatora _id, które powinieneś wziąć pod uwagę: po pierwsze, musisz być bardzo pewien, że jest on unikalny lub chcesz obsługiwać zduplikowane wyjątki klucza. Po drugie, należy pamiętać o strukturze drzewa indeksu (zobacz „Wskazówka nr 22: Użyj indeksów, aby zrobić więcej przy mniejszej ilości pamięci” na stronie 24) oraz o tym, jak losowe lub nielosowe będzie Twoje zamówienie reklamowe. ObjectIds mają doskonałą kolejność wstawiania, jeśli chodzi o drzewo indeksowe: zawsze rosną, co oznacza, że są zawsze wstawiane na prawej krawędzi B-drzewa. To z kolei oznacza, że MongoDB musi zachować w pamięci tylko prawą krawędź drzewa B. I odwrotnie, losowa wartość w polu _id oznacza, że _id zostanie wstawione w całym drzewie. Następnie maszyna musi przenieść stronę indeksu do pamięci, zaktualizować jej niewielki fragment, a następnie prawdopodobnie zignorować ją, aż ponownie wyślizgnie się z pamięci. Jest to mniej wydajne.

**Sztuczka 16: Unikaj używania dokumentu dla _id**

Prawie nigdy nie powinieneś używać dokumentu jako wartości _id, chociaż może to być nieuniknione w pewnych sytuacjach (takich jak dane wyjściowe MapReduce). Problem z używaniem dokumentu jako _id polega na tym, że indeksowanie dokumentu bardzo różni się od indeksowania pól w dokumencie. Tak więc, jeśli nie planujesz za każdym razem zapytać o cały subdokument, możesz i tak skończyć z wieloma indeksami na _id, _id.foo, _id.bar itp. Nie możesz również zmienić _id bez nadpisania całego dokumentu, więc niepraktyczne jest jego użycie, jeśli pola dokumentu podrzędnego mogą się zmienić.

**Sztuczka 17: Nie używaj odwołań do baz danych**

Ta wskazówka dotyczy w szczególności specjalnego typu dokumentu referencyjnego bazy danych, a nie ogólnie odwołań (jak omówiono w poprzednim rozdziale). Odniesienia do baz danych są zwykłymi dokumentami podrzędnymi w postaci {$ id: identyfikator, $ ref: nazwa_kolekcji} (opcjonalnie mogą mieć również pole $ db na nazwę bazy danych). Czują się trochę relacyjni: odnosisz się do dokumentu z innej kolekcji. Jednak tak naprawdę nie odwołujesz się do innej kolekcji, to tylko zwykły dokument podrzędny. Nie robi absolutnie nic magicznego. MongoDB nie może wyłuskiwać odwołań do bazy danych w locie; nie są sposobem łączenia w MongoDB. To tylko dokumenty podrzędne zawierające

identyfikator _id i nazwę kolekcji. Oznacza to, że aby je wyłuskać, należy ponownie wysłać zapytanie do bazy danych. Jeśli odwołujesz się do dokumentu, ale znasz już kolekcję, równie dobrze możesz zaoszczędzić miejsce i zapisać tylko _id, a nie _id i nazwę kolekcji. Odwołanie do bazy danych to strata miejsca, chyba że nie wiesz, w jakiej kolekcji będzie znajdował się dokument, do którego się odwołuje. Słyszałem tylko, że odniesienie do bazy danych zostało użyte z dobrym skutkiem, dotyczył systemu, który pozwalał użytkownikom na komentowanie czegokolwiek w system. Mieli kolekcję komentarzy i przechowywali w niej komentarze z odniesieniami do prawie każdej innej kolekcji i bazy danych w systemie.

**Sztuczka 18: Nie używaj GridFS do małych danych binarnych**

GridFS wymaga dwóch zapytań: jednego do pobrania metadanych pliku, a drugiego do pobrania jego zawartości. Tak więc, jeśli używasz GridFS do przechowywania małych plików, podwajasz liczbę zapytań, które musi wykonać Twoja aplikacja. GridFS to w zasadzie sposób na dzielenie dużych obiektów binarnych do przechowywania w bazie danych. GridFS służy do przechowywania dużych zbiorów danych - większych niż zmieści się w jednym dokumencie. Z reguły wszystko, co jest zbyt duże, aby załadować je wszystkie naraz na kliencie, prawdopodobnie nie jest czymś, co chcesz załadować na serwerze. Dlatego wszystko, co zamierzasz przesyłać strumieniowo do klienta, jest dobrym kandydatem do GridFS. Rzeczy, które zostaną załadowane w całości na kliencie, takie jak obrazy, dźwięki, a nawet małe klipy wideo, powinny być po prostu osadzone w głównym dokumencie.

**Sztuczka 19: Obsługuj „płynne" przełączanie awaryjne**

Często ludzie słyszeli, że MongoDB bezproblemowo obsługuje przełączanie awaryjne i są zaskoczeni, gdy zaczynają otrzymywać wyjątki. MongoDB próbuje odzyskać sprawność po awariach bez interwencji, ale automatyczna obsługa niektórych błędów jest niemożliwa. Załóżmy, że wysyłasz żądanie do serwera i otrzymujesz błąd sieciowy. Twój kierowca ma teraz kilka opcji. Jeśli sterownik nie może ponownie połączyć się z bazą danych, to oczywiście nie może automatycznie ponowić próby wysłania żądania do tego serwera. Załóżmy jednak, że masz inny serwer, o którym wie sterownik, czy może on automatycznie wysłać żądanie do tego serwera? To zależy od tego, jaka była prośba. Jeśli zamierzałeś wysłać list do szkoły podstawowej, prawdopodobnie nie ma jeszcze innej podstawowej. Jeśli masz zamiar czytać, to może być coś jak długo działające MapReduce do slave'a, który jest teraz wyłączony, a sterownik nie powinien wysyłać tego do innego losowego serwera (podstawowego?). Dlatego nie może automatycznie ponowić próby na innym serwerze. Jeśli błąd był chwilową awarią sieci i sterownik natychmiast ponownie łączy się z serwerem, nadal nie powinien próbować ponownie wysyłać żądania. Co się stanie, jeśli sterownik wysłał oryginalny komunikat, a następnie napotkał błąd sieci lub błąd w odpowiedzi bazy danych? Wtedy żądanie może być już przetwarzane przez bazę danych, więc nie chcesz wysyłać go po raz drugi. Jest to trudny problem, który często zależy od aplikacji, więc kierowcy zgłaszają ten problem. Musisz wychwycić każdy wyjątek, który jest zgłaszany przy błędach sieci (informacje o tym, jak to zrobić, powinieneś znaleźć w dokumentacji swojego sterownika). Zajmij się wyjątkiem, a następnie dowiedz się na podstawie żądania po żądaniu: czy chcesz ponownie wysłać wiadomość? Czy musisz najpierw sprawdzić stan w bazie danych? Możesz się po prostu poddać, czy musisz ponawiać próby?

**Sztuczka 20: Obsługa awarii zestawu replik i przełączania awaryjnego**

Twoja aplikacja powinna być w stanie obsłużyć wszystkie ekscytujące scenariusze awarii, które mogą wystąpić w przypadku zestawu replik. Załóżmy, że Twoja aplikacja zgłasza błąd „nie nadrzędny". Istnieje kilka możliwych przyczyn tego błędu: Twój zestaw może mieć awarię do nowej prawybory i musisz z wdziękiem poradzić sobie z czasem podczas prawyborów. Czas potrzebny na wybory jest różny: zwykle jest to kilka sekund, ale jeśli nie masz szczęścia, może to zająć 30 lub więcej sekund. Jeśli

jesteś po niewłaściwej stronie partycji sieciowej, możesz nie być w stanie zobaczyć mastera przez wiele godzin. Brak możliwości zobaczenia wzorca w ogóle jest ważnym przypadkiem do obsłużenia: czy aplikacja może przejść w tryb tylko do odczytu, jeśli tak się stanie? Twoja aplikacja powinna być w stanie obsługiwać tylko do odczytu przez krótkie okresy (podczas wyborów głównych) i długie (gdy większość jest nieobecna lub podzielona na partycje). Niezależnie od tego, czy istnieje mistrz, powinieneś być w stanie kontynuować wysyłanie odczytów do dowolnego członka zestawu, do którego możesz dotrzeć. Członkowie mogą na krótko przejść przez niemożliwą do odczytania fazę „odzyskiwania" podczas wyborów: członkowie w tym stanie będą zgłaszać błędy o tym, że nie są mistrzami lub drugorzędnymi, jeśli kierowca spróbuje je odczytać, i mogą znajdować się w tym stanie tak przelotnie, że błędy te prześlizgną się pomiędzy sterowniki ping wysyłane do bazy danych.

**Sztuczka 21: Zminimalizuj dostęp do dysku**

Dostęp do danych z pamięci RAM jest szybki, a dostęp do danych z dysku jest wolny. Dlatego większość technik optymalizacji to w zasadzie wymyślne sposoby minimalizowania ilości dostępów do dysku.

**Fuzzy Math**

Czytanie z dysku jest (około) milion razy wolniejsze niż czytanie z pamięci. Większość wirujących dysków może uzyskać dostęp do danych w, powiedzmy, 10 milisekund, podczas gdy pamięć zwraca dane w ciągu 10 nanosekund. (Zależy to w dużej mierze od rodzaju dysku twardego i rodzaju pamięci RAM, ale dokonamy bardzo szerokiego uogólnienia, które jest z grubsza dokładne dla większości ludzi). Oznacza to, że stosunek czasu dysku do czasu pamięci RAM wynosi od 1 milisekundy do 1 nanosekundy. Jedna milisekunda odpowiada milionowi nanosekund, więc dostęp do dysku trwa (w przybliżeniu) milion razy dłużej niż dostęp do pamięci RAM. W związku z tym odczytanie zawartości dysku zajmuje naprawdę dużo czasu w terminach obliczeniowych.

W systemie Linux możesz zmierzyć sekwencyjny dostęp do dysku na swoim komputerze, uruchamiając sudo hdparm -t / dev / hdcokolwiek. Nie daje to dokładnej miary, ponieważ MongoDB będzie wykonywać niesekwencyjne odczyty i zapisy, ale warto zobaczyć, co potrafi Twoja maszyna. Więc co można z tym zrobić? Istnieje kilka „łatwych" rozwiązań:

Używaj dysków SSD: dyski SSD (dyski półprzewodnikowe) są znacznie szybsze niż obracające się dyski twarde w wielu przypadkach, ale często są mniejsze, droższe, trudno je bezpiecznie wymazać i nadal nie zbliżają się do prędkości, z jaką można czytać pamięć. Nie zniechęca Cię to do ich używania: zwykle działają fantastycznie z MongoDB, ale nie są magicznym lekarstwem na wszystko.

Dodaj więcej pamięci RAM: Dodanie większej pamięci RAM oznacza, że musisz mniej uderzać w dysk. Jednak dodanie pamięci RAM tylko doprowadzi Cię na tyle daleko - w pewnym momencie Twoje dane nie zmieszczą się już w pamięci RAM.

Powstaje więc pytanie: w jaki sposób przechowujemy terabajty (petabajty?) Danych na dysku, ale programujemy aplikację, która w większości będzie miała dostęp do danych znajdujących się już w pamięci i przenosi dane z dysku do pamięci tak rzadko, jak to możliwe? Jeśli dosłownie uzyskujesz losowy dostęp do wszystkich danych w czasie rzeczywistym, będziesz potrzebować po prostu dużo pamięci RAM. Jednak większość aplikacji tego nie robi: dostęp do najnowszych danych jest większy niż do starszych danych, niektórzy użytkownicy są bardziej aktywni niż inni, niektóre regiony mają więcej klientów niż inne. Aplikacje takie jak te można zaprojektować tak, aby przechowywały określone dokumenty w pamięci i bardzo rzadko trafiały na dysk.

**Sztuczka 22: Użyj indeksów, aby zrobić więcej przy mniejszej ilości pamięci**

Po pierwsze, abyśmy wszyscy byli na tej samej stronie, przyjmiemy, że strona pamięci ma 4 KB, chociaż nie jest to uniwersalna prawda. Powiedzmy, że masz maszynę z 256 GB danych i 16 GB pamięci. Powiedzmy, że większość tych danych znajduje się w jednym zbiorze i wyszukujesz ten zbiór. Co robi MongoDB? MongoDB ładuje pierwszą stronę dokumentów z dysku do pamięci i porównuje je z zapytaniem. Następnie ładuje następną stronę i porównuje je. Następnie ładuje następną stronę. I tak dalej, poprzez 256 GB danych. Nie może iść na skróty: nie może wiedzieć, czy dokument pasuje bez patrzenia na dokument, więc musi spojrzeć na każdy dokument. Dlatego będzie musiał załadować wszystkie 256 GB do pamięci (system operacyjny dba o wymianę najstarszych stron z pamięci, ponieważ potrzebuje miejsca na nowe). To zajmie dużo czasu. Jak możemy uniknąć ładowania wszystkich 256 GB do pamięci za każdym razem, gdy wykonujemy zapytanie? Możemy powiedzieć MongoDB, aby utworzył indeks na danym polu, x, a MongoDB utworzy drzewo wartości kolekcji dla tego pola. MongoDB zasadniczo przetwarza dane, dodając każdą wartość x w kolekcji do uporządkowanego drzewa. Każdy wpis indeksu w drzewie zawiera wartość x i wskaźnik do dokumentu z tą wartością x. Drzewo zawiera po prostu wskaźnik do dokumentu, a nie sam dokument, co oznacza, że indeks jest zwykle znacznie mniejszy niż cała kolekcja. Gdy zapytanie zawiera x jako część kryteriów, MongoDB zauważy, że ma indeks na x i przejrzy uporządkowane drzewo wartości. Teraz, zamiast przeglądać każdy dokument, MongoDB może powiedzieć: „Czy szukana przeze mnie wartość jest większa czy mniejsza niż wartości tego węzła drzewa? Jeśli większy, idź w prawo, jeśli mniej, idź w lewo ”. Kontynuuje w ten sposób, dopóki nie znajdzie wartości, której szuka, albo nie zobaczy, że wartość, której szuka, nie istnieje. Jeśli znajdzie wartość, podąża za wskaźnikiem do właściwego dokumentu, ładując stronę tego dokumentu do pamięci, a następnie zwracając ją. Więc załóżmy, że wykonamy zapytanie, które zakończy się dopasowaniem dokumentu lub dwóch w naszej kolekcji. Jeśli nie korzystamy z indeksu, musimy załadować do pamięci 64 miliony stron z dysku:

Strony danych: 256 GB / (4 KB / strona) = 64 miliony stron

Załóżmy, że nasz indeks ma rozmiar około 80 GB. Wtedy indeks ma rozmiar około 20 milionów stron:

Liczba stron w naszym indeksie: 80 GB / (4 KB / strona) = 20 milionów stron

Jednak indeks jest uporządkowany, co oznacza, że nie musimy przeglądać każdego wpisu: wystarczy załadować określone węzły. Ile?

Liczba stron indeksu, które należy załadować do pamięci: ln (20 000 000) = 17 stron

Od 64 000 000 do 17!

OK, więc to nie jest dokładnie 17: kiedy już znaleźliśmy wynik w indeksie, musimy załadować dokument z pamięci, więc to jest załadowany inny rozmiar stron dokumentu, plus węzły w drzewie mogą być więcej niż jeden rozmiar strony. Mimo to jest to niewielka liczba stron w porównaniu z przeglądaniem całej kolekcji! Mamy nadzieję, że teraz możesz sobie wyobrazić, jak indeksy przyspieszają wykonywanie zapytań.

**Sztuczka 23: Nie zawsze używaj indeksu**

Teraz, gdy już zastanawiam się nad użytecznością indeksów, pozwolę sobie ostrzec, że nie należy ich używać do wszystkich zapytań. Załóżmy, że w powyższym przykładzie zamiast pobrać kilka rekordów, zwracaliśmy około 90% dokumentu w kolekcji. Jeśli użyjemy indeksu do tego typu zapytania, w końcu przejrzymy większość drzewa indeksu, ładując, powiedzmy, 60 GB indeksu do pamięci. Następnie musielibyśmy postępować zgodnie ze wszystkimi wskazówkami w indeksie, ładując 230 GB danych z kolekcji. Skończyło się na tym, że wczytujemy 230 GB + 60 GB = 290 GB - więcej niż gdybyśmy w ogóle nie używali indeksu! W związku z tym indeksy są zazwyczaj najbardziej przydatne, gdy masz niewielki

podzbiór wszystkich danych, które chcesz zwrócić. Dobrą zasadą jest to, że przestają być przydatne po zwróceniu około połowy danych w kolekcji. Jeśli masz indeks w polu, ale wykonujesz duże zapytanie, które byłoby mniej wydajne przy użyciu tego indeksu, możesz zmusić MongoDB do nieużywania indeksu, sortując według {"$ natural": 1}. To sortowanie oznacza „zwracanie danych w kolejności, w jakiej pojawiają się na dysku”, co wymusza

MongoDB, aby nie używać indeksu:

> db.foo.find (). sort ({"$ natural": 1})

Jeśli zapytanie nie korzysta z indeksu, MongoDB skanuje tabelę, co oznacza, że przeszukuje wszystkie dokumenty w kolekcji w celu znalezienia wyników.

**Szybkość pisania**

Za każdym razem, gdy dodaje się, usuwa lub aktualizuje nowy rekord, każdy indeks, na który ma wpływ zmiana, musi zostać zaktualizowany. Załóżmy, że wstawiasz dokument. Dla każdego indeksu MongoDB musi znaleźć miejsce, w którym wartość nowego dokumentu przypada na drzewo indeksu, a następnie wstawić ją tam. Aby usunąć, musi znaleźć i usunąć wpis z drzewa. W przypadku aktualizacji może dodać nowy wpis indeksu, taki jak wstawka, usunąć wpis, taki jak usuń, lub będzie musiał zrobić jedno i drugie, jeśli wartość się zmieni. W związku z tym indeksy mogą znacznie zwiększyć nakłady związane z zapisami.

**Sztuczka 24: Twórz indeksy, które obejmują Twoje zapytania**

Jeśli chcemy, aby zwracane były tylko niektóre pola i możemy uwzględnić wszystkie te pola w indeksie, MongoDB może wykonać zapytanie dotyczące indeksu, w którym nigdy nie musi podążać za wskaźnikami do dokumentów i po prostu zwraca dane indeksu do klienta. Załóżmy na przykład, że mamy indeks dla jakiegoś zbioru pól:

> db.foo.ensureIndex ({"x": 1, "y": 1, "z": 1})

Następnie, jeśli zapytamy o indeksowane pola i zażądamy tylko zwróconych pól zindeksowanych, MongoDB nie ma powodu, aby załadować cały dokument:

> db.foo.find({"x" : criteria, "y" : criteria},

... {"x" : 1, "y" : 1, "z" : 1, "_id" : 0})

Teraz to zapytanie dotknie tylko danych w indeksie, nigdy nie musi dotykać właściwego zbioru. Zauważ, że dołączamy klauzulę „_id”: 0 w argumencie pola do zwrócenia. Domyślnie zawsze zwracany jest _id, ale nie jest on częścią naszego indeksu, więc MongoDB musiałby przejść do dokumentu, aby pobrać _id. Usunięcie go z pól do zwrotu oznacza, że MongoDB może po prostu zwrócić wartości z indeksu. Jeśli niektóre zapytania zwracają tylko kilka pól, rozważ wrzucenie tych pól do indeksu, aby można było wykonywać zapytania dotyczące indeksów pokrytych, nawet jeśli nie będą one przeszukiwane. Na przykład z nie jest używane w powyższym zapytaniu, ale jest to pole w polach do zwrotu, a tym samym w indeksie.

Teraz to zapytanie dotknie tylko danych w indeksie, nigdy nie musi dotykać właściwego zbioru. Zauważ, że dołączamy klauzulę „_id”: 0 w argumencie pola do zwrócenia. Domyślnie zawsze zwracany jest _id, ale nie jest on częścią naszego indeksu, więc MongoDB musiałby przejść do dokumentu, aby pobrać _id. Usunięcie go z pól do zwrotu oznacza, że MongoDB może po prostu zwrócić wartości z indeksu. Jeśli niektóre zapytania zwracają tylko kilka pól, rozważ wrzucenie tych pól do indeksu, aby można było wykonywać zapytania dotyczące indeksów pokrytych, nawet jeśli nie będą one przeszukiwane. Na

przykład z nie jest używane w powyższym zapytaniu, ale jest to pole w polach do zwrotu, a tym samym w indeksie.

**Sztuczka 25: Użyj indeksów złożonych, aby szybko wykonać wiele zapytań**

Jeśli to możliwe, utwórz indeks złożony, który może być używany przez wiele zapytań. Nie zawsze jest to możliwe, ale jeśli wykonujesz wiele zapytań z podobnymi argumentami, może tak być. Każde zapytanie pasujące do przedrostka indeksu może używać indeksu. Dlatego chcesz utworzyć indeksy z największą liczbą kryteriów wspólnych dla zapytań. Załóżmy, że Twoja aplikacja uruchamia te zapytania:

collection.find({"x" : criteria, "y" : criteria, "z" : criteria})

collection.find({"z" : criteria, "y" : criteria, "w" : criteria})

collection.find({"y" : criteria, "w" : criteria})

Jak widać, y jest jedynym polem, które pojawia się w każdym zapytaniu, więc jest to bardzo dobry kandydat do umieszczenia w indeksie. z pojawia się w pierwszych dwóch, a w w drugich dwóch, więc każda z nich będzie działać jako następna opcja (zobacz więcej o porządkowaniu indeksów w „Wskazówka nr 27: I-zapytania powinny pasować jak najmniej tak szybko, jak to możliwe ”

na stronie 30 i „Wskazówka nr 28: zapytania OR powinny odpowiadać tak szybko, jak to możliwe” na stronie 31). Chcemy trafiać do tego indeksu jak najczęściej i jak najczęściej. Jeśli określone zapytanie powyżej jest ważniejsze niż inne lub będzie uruchamiane znacznie częściej, nasz indeks powinien faworyzować to jedno. Na przykład załóżmy, że pierwsze zapytanie zostanie uruchomione tysiące razy częściej niż następne dwa. Następnie chcemy faworyzować ten w naszym indeksie:

collection.ensureIndex ({"y": 1, "z": 1, "x": 1})

Pierwsze zapytanie będzie maksymalnie zoptymalizowane, a kolejne dwa będą używać indeksu dla części zapytania. Jeśli wszystkie trzy zapytania zostaną uruchomione w przybliżeniu na tej samej wysokości, dobrym indeksem może być:

collection.ensureIndex ({"y": 1, "w": 1, "z": 1})

Wtedy wszyscy trzej będą mogli użyć indeksu dla kryteriów y, dwaj pozostali będą mogli go użyć jako w, a środkowy będzie mógł w pełni wykorzystać indeks. Możesz użyć wyjaśnienia, aby zobaczyć, jak indeks jest używany w zapytaniu:

collection.find(criteria).explain()

**Sztuczka 26: Twórz hierarchiczne dokumenty, aby przyspieszyć skanowanie**

Utrzymanie hierarchicznej organizacji danych nie tylko zapewnia ich organizację, ale MongoDB może również szybciej wyszukiwać je bez indeksu (w niektórych przypadkach). Na przykład załóżmy, że masz zapytanie, które nie używa indeksu. Jak wspomniano wcześniej, MongoDB musi przejrzeć każdy dokument w kolekcji, aby sprawdzić, czy cokolwiek pasuje do kryteriów zapytania. Może to zająć różny czas, w zależności od struktury dokumentów. Załóżmy, że masz dokumenty użytkownika o płaskiej strukturze, takiej jak ta:

{

"_id" : id,

"name" : username,

"email" : email,

"twitter" : username,

"screenname" : username,

"facebook" : username,

"linkedin" : username,

"phone" : number,

"street" : street

"city" : city,

"state" : state,

"zip" : zip,

"fax" : number

}

Teraz załóżmy, że zapytamy:

> db.users.find ({"zip": "10003"})

Co robi MongoDB? Musi przejrzeć każde pole każdego dokumentu, szukając pola zip. Korzystając z osadzonych dokumentów, możemy stworzyć własne „drzewo” i pozwolić MongoDB zrobić to szybciej. Załóżmy, że zmienimy nasz schemat, aby wyglądał następująco:

{

"_id" : id,

"name" : username,

"online" : {

"email" : email,

"twitter" : username,

"screenname" : username,

"facebook" : username,

"linkedin" : username,

},

"address" : {

"street" : street,

"city" : city,

"state" : state,

"zip" : zip

}

"tele" : {

"phone" : number,

"fax" : number,

}

}

Teraz nasze zapytanie wyglądałoby tak:

> db.users.find ({"address.zip": "10003"})

A MongoDB musiałby tylko spojrzeć na _id, nazwę i online, zanim zobaczyłby, że adres jest pożądanym prefiksem, a następnie szukałby w nim zip. Korzystanie z rozsądnej hierarchii pozwala MongoDB nie sprawdzać każdego pola w poszukiwaniu dopasowania.

**Sztuczka 27: Zapytania AND powinny pasować jak najmniej tak szybko, jak to możliwe**

Załóżmy, że pytamy o dokumenty spełniające kryteria A, B i C. Załóżmy teraz, że kryterium A pasuje do 40 000 dokumentów, B do 9 000, a C do 200. Jeśli zapytamy MongoDB o kryteria w podanej kolejności, nie być bardzo wydajnym. Jeśli umieścimy C na pierwszym miejscu, następnie B, a następnie A, wystarczy spojrzeć na 200 dokumentów (najwyżej) pod kątem kryteriów B i C. Jak widać, znacznie zmniejsza to ilość pracy. Jeśli wiesz, że określone kryteria będą pasować do mniejszej liczby dokumentów, upewnij się, że kryteria są pierwsze (zwłaszcza jeśli są indeksowane).

**Sztuczka 28: zapytania OR powinny być zgodne tak szybko, jak to możliwe**

Zapytania w stylu OR są dokładnym przeciwieństwem zapytań AND: staraj się umieścić na początku najbardziej włączające klauzule, ponieważ MongoDB musi nadal sprawdzać dokumenty, które nie są jeszcze częścią zestawu wyników dla każdego dopasowania. Jeśli zrobimy to w tej samej kolejności, w jakiej ustrukturyzowaliśmy zapytanie AND, musimy sprawdzić dokumenty pod kątem każdej klauzuli. Zamiast tego dopasowujemy jak najwięcej kolekcji tak szybko, jak to możliwe.

**Sztuczka 29: Zapisuj do dziennika dla pojedynczego serwera, repliki dla wielu serwerów**

W idealnym świecie wszystkie zapisy byłyby natychmiast, trwale zapisywane na dyskach i można je było natychmiast odzyskać z dowolnego miejsca. Niestety, w prawdziwym świecie jest to niemożliwe, możesz poświęcić więcej czasu na upewnienie się, że dane są bezpieczne, lub zapisać je szybciej i przy mniejszym bezpieczeństwie. MongoDB zapewnia więcej pokręteł do obracania w tym obszarze niż ma pokrętła do wszystkiego innego razem, dlatego ważne jest, aby zrozumieć dostępne opcje. Replikacja i dziennikowanie to dwa podejścia do bezpieczeństwa danych, które można zastosować w MongoDB. Ogólnie rzecz biorąc, powinieneś uruchomić replikację i mieć księgowany co najmniej jeden z serwerów. Na blogu MongoDB znajduje się dobry post wyjaśniający, dlaczego nie należy uruchamiać MongoDB (ani żadnej bazy danych) na jednym serwerze. Replikacja MongoDB automatycznie kopiuje wszystkie Twoje zapisy na inne serwery. Jeśli twój obecny serwer główny ulegnie awarii, możesz ustawić inny serwer jako nowy serwer główny (dzieje się to automatycznie w przypadku zestawów replik). Jeśli członek zestawu replik wyłączy się nieczysto i nie był uruchomiony z - dziennikiem, MongoDB nie daje żadnych gwarancji co do swoich danych (mogą być uszkodzone). Musisz uzyskać czystą kopię danych: albo wyczyść dane i zsynchronizuj je ponownie lub załaduj kopię zapasową i

szybką synchronizację. Kronikowanie zapewnia bezpieczeństwo danych na jednym serwerze. Wszystkie operacje są zapisywane w dzienniku (dzienniku), który jest regularnie opróżniany na dysk. Jeśli twój komputer ulegnie awarii, ale sprzęt jest w porządku, możesz zrestartować serwer, a dane naprawią się samoczynnie, czytając dziennik. Pamiętaj, że MongoDB nie może uchronić Cię przed problemami sprzętowymi: jeśli dysk zostanie uszkodzony, prawdopodobnie nie będzie można odzyskać bazy danych. Replikacja i kronikowanie mogą być używane w tym samym czasie, ale musisz to zrobić strategicznie, aby zminimalizować spadek wydajności. Obie metody w zasadzie tworzą kopię wszystkich zapisów, więc masz:

Brak bezpieczeństwa: zapis jednego serwera na żądanie zapisu

Replikacja: dwa zapisy na serwerze na żądanie zapisu

Kronikowanie: dwa zapisy na serwerze na żądanie zapisu

Replikacja + dziennikowanie: trzy zapisy na serwerze na żądanie zapisu

Pisanie każdej informacji trzy razy to dużo, ale jeśli Twoja aplikacja nie wymaga wysokiej wydajności, a bezpieczeństwo danych jest bardzo ważne, możesz rozważyć użycie obu. Istnieje kilka bardzo bezpiecznych alternatywnych wdrożeń, które są bardziej wydajne niż kolejny.

**Sztuczka 30: Zawsze używaj replikacji, kronikowania lub obu**

Jeśli używasz jednego serwera, użyj opcji --journal. W rozwoju nie ma powodu, by nie używać - dziennik przez cały czas.

Dodaj dzienniki do lokalnej konfiguracji MongoDB, aby mieć pewność, że nie stracisz danych podczas programowania. Biorąc pod uwagę kary za wydajność związane z używaniem kronikowania, możesz chcieć mieszać serwery księgowane i nieobjęte dziennikiem, jeśli masz wiele komputerów. Kopie zapasowe podrzędne mogą być kronikowane, podczas gdy główne i pomocnicze (szczególnie te równoważące obciążenie odczytu) mogą nie być rejestrowane. Solidną, małą konfigurację pokazano na rysunku 4-1. Podstawowa i dodatkowa nie są uruchamiane z kronikowaniem, dzięki czemu są szybkie do odczytu i zapisu. W przypadku normalnej awarii serwera można w razie potrzeby przełączyć się awaryjnie na serwer pomocniczy i ponownie uruchomić komputer, który uległ awarii. Jeśli którekolwiek z centrów danych ulegnie całkowitej awarii, nadal masz bezpieczną kopię swoich danych. Jeśli DC2 przestanie działać, po ponownym uruchomieniu możesz ponownie uruchomić serwer zapasowy. Jeśli DC1 ulegnie awarii, możesz ustawić maszynę zapasową jako główną lub użyć jej danych do ponownego zaszczepienia maszyn w DC1. Jeśli oba centra danych ulegną awarii, przynajmniej masz kopię zapasową w DC2, z której możesz załadować wszystko. Inną bezpieczną konfigurację dla pięciu serwerów przedstawia Rysunek 4-2. Jest to konfiguracja nieco bardziej niezawodna niż powyżej: w obu centrach danych są elementy pomocnicze, członek opóźniony chroniący przed błędami użytkownika oraz członek kronikowany do tworzenia kopii zapasowych.

**Sztuczka 31: Nie polegaj na naprawie, aby odzyskać dane**

Jeśli baza danych ulegnie awarii i nie korzystałeś z --journal, nie używaj danych serwera w obecnej postaci. Może się to wydawać w porządku przez kilka tygodni, aż nagle uzyskasz dostęp do uszkodzonego dokumentu, który powoduje, że aplikacja przestaje działać. Lub indeksy mogą być pomieszane, więc z bazy danych zwracane są tylko częściowe wyniki. Albo sto innych rzeczy; korupcja jest zła, podstępna i często niewykrywalna ... przez jakiś czas. Masz kilka opcji. Możesz uruchomić naprawę. To kusząca opcja, ale to naprawdę ostatnia deska ratunku. Po pierwsze, naprawa sprawdza każdy znaleziony dokument i tworzy jego czystą kopię. Zajmuje to dużo czasu, zajmuje dużo miejsca na

dysku (tyle samo co aktualnie używanego miejsca) i powoduje pominięcie uszkodzonych rekordów. Oznacza to, że jeśli nie może znaleźć milionów dokumentów z powodu korupcji, nie skopiuje ich i zostaną utracone. Twoja baza danych może już nie być uszkodzona, ale może być również znacznie mniejsza. Ponadto naprawa nie powoduje introspekcji dokumentów: może wystąpić uszkodzenie, które uniemożliwia przeanalizowanie niektórych pól, których naprawa nie znajdzie ani nie naprawi. Preferowaną opcją jest szybka synchronizacja z kopii zapasowej lub ponowna synchronizacja od zera. Pamiętaj, że przed ponowną synchronizacją musisz wyczyścić potencjalnie uszkodzone dane; Replikacja MongoDB nie może „naprawić" uszkodzonych danych.

**Sztuczka 32: Zrozum getlasterror**

Domyślnie zapisy nie zwracają żadnej odpowiedzi z bazy danych. Jeśli wyślesz do bazy danych aktualizację, wstawisz lub usuniesz, przetworzy ją i nie zwróci niczego użytkownikowi. Dlatego kierowcy nie spodziewają się żadnej reakcji w przypadku sukcesu lub porażki. Jednak oczywiście jest wiele sytuacji, w których chcesz otrzymać odpowiedź z bazy danych. Aby sobie z tym poradzić, MongoDB udostępnia polecenie „o tej ostatniej operacji ..." o nazwie getlasterror. Pierwotnie opisywał tylko wszelkie błędy, które wystąpiły w ostatniej operacji, ale rozszerzył się na podawanie wszelkiego rodzaju informacji o zapisie i zapewnianie różnych opcji związanych z bezpieczeństwem. Aby uniknąć niezamierzonych błędów odczytu i ostatniego zapisu, getlasterror utknie na samym końcu żądania zapisu, zasadniczo zmuszając bazę danych do traktowania zapisu i getlasterror jako pojedynczego żądania. Są wysyłane razem i gwarantowane, że będą przetwarzane jeden po drugim, bez żadnych innych operacji pomiędzy nimi. Sterowniki zawierają tę funkcjonalność, więc nie musisz sam się tym zająć, ogólnie nazywając to „bezpiecznym" zapisem.

**Sztuczka 33: Zawsze używaj bezpiecznych zapisów podczas programowania**

W programowaniu chcesz mieć pewność, że aplikacja działa zgodnie z oczekiwaniami, a bezpieczne zapisy mogą Ci w tym pomóc. Jakie rzeczy mogą się nie udać podczas pisania? Zapis może próbować wypchnąć coś do pola innego niż tablica, spowodować wyjątek zduplikowanego klucza (próba przechowywania dwóch dokumentów o tej samej wartości w unikatowo indeksowanym polu), usunąć pole _id lub milion innych błędów użytkownika. Będziesz chciał wiedzieć, że zapis nie jest ważny przed wdrożeniem. Jednym podstępnym błędem jest brak miejsca na dysku: wszystkie nagłe zapytania w tajemniczy sposób zwracają mniej danych. To jest trudne, jeśli nie używasz bezpiecznych zapisów, ponieważ wolne miejsce na dysku nie jest czymś, co zwykle sprawdzasz. Często przypadkowo ustawiam-dbpath na niewłaściwą partycję, przez co MongoDB zabraknie miejsca znacznie wcześniej niż planowano. Podczas programowania istnieje wiele powodów, dla których napis może nie przejść z powodu błędu programisty i warto o nich wiedzieć.

**Sztuczka 34: Użyj w z replikacją**

W przypadku ważnych operacji należy upewnić się, że zapisy zostały zreplikowane do większości zestawu. Zapis nie jest „zatwierdzany", dopóki nie znajdzie się na większości serwerów w zestawie. Jeśli zapis nie został zatwierdzony, a partycje sieciowe lub awarie serwera izolują go od większości zestawu, zapis może zostać wycofany. (To trochę wykracza poza zakres tej wskazówki, ale jeśli obawiasz się wycofywania zmian, napisałem post opisujący, jak sobie z tym poradzić). W kontroluje liczbę serwerów, na które należy napisać odpowiedź, zanim zwróci sukces. Działa to tak, że wysyłasz getlasterror do serwera (zwykle po prostu ustawiając w dla danej operacji zapisu). Serwer odnotowuje, gdzie jest w swoim oplog („Jestem w operacji 123"), a następnie czeka, aż slave'y w-1 zastosują operację 123 do swojego zestawu danych. Gdy każdy slave zapisuje daną operację, wartość w jest dekrementowana na urządzeniu głównym. Gdy w wynosi 0, getlasterror zwraca sukces. Należy pamiętać, że ponieważ replikacja zawsze zapisuje operacje w kolejności, różne serwery w zestawie

mogą znajdować się w różnych „punktach historii”, ale nigdy nie będą miały niespójnego zestawu danych. Będą identyczne jak master minutę temu, kilka sekund temu, tydzień temu itd. Nie zabraknie w nich przypadkowych operacji. Oznacza to, że zawsze możesz upewnić się, że niewolnicy num-1 są zsynchronizowani z serwerem głównym, uruchamiając:

> db.runCommand ({"getlasterror": 1, "w": num})

Zatem pytanie z punktu widzenia programisty aplikacji brzmi: co mam ustawić? Jak wspomniano powyżej, aby napis był naprawdę „bezpieczny”, potrzebujesz większości zestawu. Jednak pisanie do mniejszości zestawu może mieć również swoje zastosowania. Jeśli w jest ustawione na mniejszość serwerów, jest to łatwiejsze do wykonania i może być „wystarczająco dobre”. Jeśli ta mniejszość zostanie oddzielona od zestawu przez awarię partycji sieciowej lub serwera, większość zestawu może wybrać nową podstawową i nie zobaczyć operacji, która została wiernie zreplikowana na serwerach w. Jeśli jednak nawet jeden z członków, który otrzymał zapis, nie został oddzielony, pozostali członkowie zestawu zsynchronizowaliby się z tym zapisem przed wybraniem nowego mistrza. Jeśli w jest ustawione na większość serwerów i wystąpi jakaś partycja sieciowa lub niektóre serwery ulegną awarii, nowy master nie będzie mógł zostać wybrany bez tego zapisu. To potężna gwarancja, ale odbywa się to kosztem mniejszego prawdopodobieństwa sukcesu: im więcej serwerów jest potrzebnych do sukcesu, tym mniejsze jest prawdopodobieństwo sukcesu.

**Sztuczka 35: Zawsze używaj wtimeout z w**

Załóżmy, że masz zestaw replik składający się z trzech elementów (jeden podstawowy i dwa dodatkowe) i chcesz się upewnić, że twoi dwaj niewolnicy są na bieżąco z mistrzem, więc uruchamiasz:

> db.runCommand ({"getlasterror": 1, "w": 2})

Ale co, jeśli któryś z twoich drugorzędnych jest uszkodzony? MongoDB nie sprawdza poprawności liczby włożonych elementów pomocniczych: z przyjemnością zaczeka, aż będzie mógł zreplikować się do 2, 20 lub 200 niewolników (jeśli tak było). Dlatego zawsze powinieneś uruchamiać getlasterror z opcją wtimeout ustawioną na rozsądną wartość dla twojej aplikacji. wtimeout podaje liczbę milisekund oczekiwania na raport podrzędny, a następnie kończy się niepowodzeniem. Ten przykład będzie czekał 100 milisekund:

> db.runCommand ({"getlasterror": 1, "w": 2, "wtimeout": 100})

Zauważ, że MongoDB stosuje replikowane operacje w kolejności: jeśli napiszesz A, B i C na master, zostaną one zreplikowane do slave'a jako A, następnie B, a następnie C. Załóżmy, że masz sytuację pokazaną na rysunku 4-3 . Jeśli napiszesz N na master i wywołasz getlasterror, slave musi powtórzyć zapisy E-N, zanim getlasterror będzie mógł zgłosić sukces. Dlatego getlasterror może znacznie spowolnić twoją aplikację, jeśli masz za sobą niewolników. Inną kwestią jest to, jak zaprogramować aplikację do obsługi przekroczenia limitu czasu getlasterror, co jest tylko pytaniem, na które tylko Ty możesz odpowiedzieć. Oczywiście, jeśli gwarantujesz replikację na inny serwer, ten zapis jest dość ważny: co zrobisz, jeśli zapis powiedzie się lokalnie, ale nie powiedzie się na wystarczającej liczbie maszyn?

**Sztuczka 36: Nie używaj fsync przy każdym zapisie**

Jeśli masz ważne dane, które chcesz mieć pewność, że trafią do dziennika, musisz użyć opcji fsync podczas zapisu. fsync czeka na następny opróżnienie (to jest do 100 ms), aby dane zostały pomyślnie zapisane w dzienniku, zanim zwróci powodzenie. Należy zauważyć, że fsync nie usuwa natychmiast danych na dysk, po prostu wstrzymuje działanie programu, dopóki dane nie zostaną opróżnione na

dysk. Tak więc, jeśli uruchomisz fsync na każdej wstawce, będziesz mógł wykonać tylko jedną wstawkę na 100 ms. Jest to około miliard razy wolniejsze niż zwykle wstawia MongoDB, więc używaj fsync oszczędnie. Zasadniczo fsync powinien być używany tylko z kronikowaniem. Nie używaj go, gdy dziennik nie jest włączony, chyba że masz pewność, że wiesz, co robisz. Możesz łatwo sprecyzować swoje osiągi bez żadnych korzyści.

**Sztuczka 37: Uruchom normalnie po awarii**

Jeśli korzystałeś z kronikowania, a system ulega awarii w sposób możliwy do odzyskania (tj. Dysk nie jest zniszczony, maszyna nie jest pod wodą itp.), Możesz normalnie zrestartować bazę danych. Upewnij się, że używasz wszystkich zwykłych opcji, zwłaszcza - dbpath (aby mógł znaleźć pliki dziennika) i oczywiście -journal. MongoDB zajmie

dbaj o automatyczne naprawianie danych, zanim zaczną one akceptować połączenia. W przypadku dużych zestawów danych może to zająć kilka minut, ale nie powinno być zbliżone do czasów, które są znane osobom, które przeprowadzały naprawy dużych zestawów danych (prawdopodobnie około pięciu minut). Pliki kroniki są przechowywane w katalogu dziennika. Nie usuwaj tych plików.

**Sztuczka 38: Wykonuj błyskawiczne kopie zapasowe trwałych serwerów**

Aby wykonać kopię zapasową bazy danych z włączonym kronikowaniem, możesz albo wykonać migawkę systemu plików, albo wykonać normalną blokadę fsync +, a następnie zrzucić. Pamiętaj, że nie możesz po prostu skopiować wszystkich plików bez fsync i blokowania, ponieważ kopiowanie nie jest operacją natychmiastową. Możesz skopiować dziennik w innym momencie niż bazy danych, a wtedy kopia zapasowa będzie gorsza niż bezużyteczna (pliki dziennika mogą uszkodzić pliki danych po ich zastosowaniu).

**Sztuczka 39: Ręcznie wyczyść kolekcje porcji**

GridFS przechowuje zawartość pliku w kolekcji porcji, domyślnie nazywanej fs.chunks. Każdy dokument w kolekcji files wskazuje na co najmniej jeden dokument w kolekcji porcji. Dobrze jest co jakiś czas sprawdzać i upewnić się, że nie ma żadnych „osieroconych" fragmentów - fragmentów krążących po okolicy bez linku do pliku. Taka sytuacja może wystąpić, jeśli baza danych została zamknięta w trakcie zapisywania pliku (dokument fs.files jest zapisywany po fragmentach). Aby sprawdzić swoją kolekcję porcji, wybierz czas, w którym jest mały ruch (ponieważ będziesz ładować dużo danych do pamięci) i uruchom coś takiego:

```
> var cursor = db.fs.chunks.find({}, {"_id" : 1, "files_id" : 1});

> while (cursor.hasNext()) {

... var chunk = cursor.next();

... if (db.fs.files.findOne({_id : chunk.files_id}) == null) {

... print("orphaned chunk: " + chunk._id);

... }
```

Spowoduje to wydrukowanie identyfikatorów _id dla wszystkich osieroconych fragmentów. Teraz, zanim przejdziesz i usuniesz wszystkie osierocone fragmenty, upewnij się, że nie są one częściami plików, które są obecnie zapisywane! Powinieneś sprawdzić db.curren tOp () i kolekcję fs.files pod kątem ostatnich dat przesłania.

**Sztuczka 40: Kompaktowe bazy danych z naprawą**

W „Sztuczka 31” , wyjaśniamy, dlaczego zazwyczaj nie powinieneś używać naprawy do faktycznej naprawy danych (chyba że jesteś w poważnych tarapatach). Jednak naprawy można użyć do kompaktowania baz danych. repair w zasadzie wykonuje mongodump, a następnie mongorestore, robiąc czystą kopię danych i, w trakcie tego procesu, usuwając wszelkie puste „dziury” w plikach danych. (Gdy wykonujesz wiele operacji usuwania lub aktualizacji, które przenoszą elementy, duże części Twojej kolekcji mogą być puste). Repair wstawia wszystko ponownie w zwartej formie. Pamiętaj o zastrzeżeniach dotyczących korzystania z naprawy:

• Blokuje operacje, więc nie chcesz go uruchamiać na wzorcu. Zamiast tego uruchom go najpierw na każdym serwerze pomocniczym, a następnie na końcu obniż podstawowy i uruchom go na tym serwerze.

• Zajmie to dwa razy więcej miejsca na dysku, z którego obecnie korzysta Twoja baza danych (np. Jeśli masz 200 GB danych, na dysku musi być co najmniej 200 GB wolnego miejsca, aby można było przeprowadzić naprawę).

Jednym z problemów wielu ludzi jest to, że mają zbyt dużo danych, aby przeprowadzić naprawę: mogą mieć bazę danych o pojemności 500 GB na serwerze z 700 GB dysku. Jeśli jesteś w takiej sytuacji, możesz wykonać „ręczną” naprawę, wykonując mongodump, a następnie sklep mongorestore. Na przykład załóżmy, że mamy serwer, który zapełnia się głównie pustą przestrzenią pod adresem ny1. Baza danych ma 300 GB, a serwer, na którym działa, ma tylko dysk 400 GB. Mamy jednak również ny2, który jest identyczną maszyną o pojemności 400 GB, na której jeszcze nic nie ma. Najpierw rezygnujemy z ny1, jeśli jest to master, i fsync i blokujemy go, aby uzyskać spójny widok danych na dysku:

> rs.stepDown ()

> db.runCommand ({fsync: 1, lock: 1})

Możemy zalogować się do ny2 i uruchomić:

ny2 $ mongodump --host ny1

Spowoduje to zrzucenie bazy danych do katalogu o nazwie dump na ny2. mongodump będzie prawdopodobnie ograniczony przez prędkość sieci w powyższej operacji. Jeśli masz fizyczny dostęp do maszyny, podłącz zewnętrzny dysk twardy i wykonaj lokalny mongodump. Gdy masz zrzut, musisz go przywrócić do ny1:

1. Wyłącz mongod działający na ny1.

2. Utwórz kopię zapasową plików danych na ny1 (np. Wykonaj migawkę EBS), na wszelki wypadek.

3. Usuń pliki danych w witrynie ny1.

4. Zrestartuj (teraz pusty) ny1. Jeśli był częścią zestawu replik, uruchom go na innym porcie i bez opcji --replSet, aby nie pomylić go (i reszty zestawu). Wreszcie, uruchom mongorestore z ny2

ny2 $ mongorestore --host ny1 --port 10000 # określ port, jeśli nie jest to 27017

Teraz ny1 będzie miał skompaktowaną postać plików bazy danych i możesz go zrestartować z normalnymi opcjami.

**Sztuczka 41: nie zmieniaj liczby głosów oddanych na członków zestawu replik**

Jeśli szukasz sposobu na wskazanie preferencji dla mistrzostwa, szukasz priorytetu. W 1.9.0 możesz ustawić priorytet członka tak, aby był wyższy niż priorytety innych członków i zawsze będzie to preferowane, gdy zostaniesz głównym członkiem. W wersjach wcześniejszych niż 1.9.0 można używać tylko priorytetu 1 (może zostać nadrzędny) i priorytet 0 (nie może zostać nadrzędny). Jeśli chcesz mieć pewność, że jeden serwer zawsze staje się głównym, nie możesz (przed 1.9.0) bez nadania wszystkim pozostałym serwerom priorytetu 0. Ludzie często antropomorfizują bazę danych i zakładają, że zwiększenie liczby głosów na serwerze ma sprawi, że wygra wybory. Jednak serwery nie są „samolubne" i niekoniecznie głosują na siebie! Członek zestawu replik jest niesamolubny i równie chętnie głosuje na swojego sąsiada, jak sam.

**Sztuczka 42: Zestawy replik mogą być rekonfigurowane bez mastera**

Jeśli masz skonfigurowaną mniejszość repliki, ale inne serwery zniknęły na dobre, oficjalnym protokołem jest zniszczenie lokalnej bazy danych i rekonfiguracja zestawu od podstaw. W wielu przypadkach jest to w porządku, ale oznacza to, że będziesz mieć trochę przestoju podczas odbudowywania zestawu i ponownego przydzielania oplogów. Jeśli chcesz, aby Twoja aplikacja działała (chociaż będzie tylko do odczytu, ponieważ nie ma nadrzędnego), możesz to zrobić, o ile nadal masz więcej niż jednego niewolnika.

Wybierz niewolnika do pracy. Wyłącz to slave i uruchom ponownie na innym porcie bez opcji --replSet. Na przykład, jeśli zacząłeś go z tymi opcjami:

$ mongod --repl Ustaw foo --port 5555

Możesz go ponownie uruchomić za pomocą:

$ mongod - port 5556

Teraz nie zostanie rozpoznany jako członek zestawu przez innych członków (ponieważ będą go szukać na innym porcie) i nie będzie próbował użyć swojej konfiguracji zestawu replik (ponieważ nie powiedziałeś to, że był członkiem zestawu replik). W tej chwili jest to zwykły serwer mongod. Teraz zamierzamy zmienić konfigurację zestawu replik, więc połącz się z tym serwerem za pomocą powłoki. Przełącz się do lokalnej bazy danych i zapisz konfigurację zestawu replik w zmiennej JavaScript. Na przykład, gdybyśmy mieli zestaw replik czterowęzłowych, mógłby wyglądać mniej więcej tak:

> use local

> config = db.system.replset.findOne()

{

"_id" : "foo",

"version" : 2,

"members" : [

{

"_id" : 0,

"host" : "rs1:5555"

},

{

"_id" : 1,

"host" : "rs2:5555",

"arbiterOnly" : true

},

{

"_id" : 2,

"host" : "rs3:5555"

},

{

"_id" : 3,

"host" : "rs4:5555"

}

]

}

Aby zmienić naszą konfigurację, musimy zmienić obiekt konfiguracyjny na żądaną konfigurację i oznaczyć go jako „nowszy” niż konfiguracja, którą mają inne serwery, aby mogły przyjąć zmianę. Powyższa konfiguracja dotyczy zestawu replik czteroelementowych, ale przypuśćmy, że chcieliśmy zmienić to na zestaw replik trzyelementowych, składający się z hostów rs1, rs2 i rs4. Aby to osiągnąć, musimy usunąć element rs3 tablicy, co można zrobić za pomocą funkcji plasterek JavaScript:

> config.slice(2, 1)

> config{

"_id" : "foo",

"version" : 2,

"members" : [

{

"_id" : 0,

"host" : "rs1:5555"

},

{

"_id" : 1,

"host" : "rs2:5555",

"arbiterOnly" : true

},

{

"_id" : 3,

"host" : "rs4:5555"

}

]

}

Upewnij się, że nie zmienisz parametru rs4 _id na 2. Spowoduje to zamieszanie w zestawie replik. Jeśli dodajesz nowe węzły do zestawu, użyj funkcji push JavaScript, aby dodać elementy z _id 4, 5 itd. Jeśli jednocześnie dodajesz i usuwasz węzły, możesz zagłębić się w zamieszaniu, jakim jest funkcja łączenia JavaScript (lub możesz wystarczy nacisnąć i pokroić). Teraz zwiększ numer wersji (config.version). To informuje inne serwery, że jest to nowa konfiguracja i powinny się odpowiednio zaktualizować. Teraz potrójnie sprawdź swój dokument konfiguracyjny. Jeśli zepsujesz konfigurację, możesz całkowicie zmienić konfigurację zestawu replik. Dla jasności: nic złego się nie stanie z Twoimi danymi, ale być może będziesz musiał zamknąć wszystko i zdmuchnąć lokalną bazę danych na wszystkich serwerach. Upewnij się więc, że ta konfiguracja odwołuje się do właściwych serwerów, nikt nie zmienił się spod nich _id i nie utworzyłeś żadnych arbitrów niebędących arbitrażami ani na odwrót. Gdy jesteś już absolutnie pewien, że jest to odpowiednia konfiguracja, zamknij serwer. Następnie uruchom go ponownie z jego zwykłymi opcjami (--replSet i jego standardowy port). Za kilka sekund pozostali członkowie połączą się z nim, zaktualizują swoją konfigurację i wybiorą nowego administratora.

**Sztuczka 43: --shardsvr i --configsvr nie są wymagane**

Dokumentacja wydaje się sugerować, że są one wymagane podczas konfigurowania fragmentacji, ale tak nie jest. Po prostu zmieniają port (co może poważnie zepsuć istniejący zestaw replik): --shardsvr zmienia port na 27018, a --configsvr zmienia go na 27019. Jeśli konfigurujesz wiele serwerów na wielu komputerach, pomóc połączyć ze sobą właściwe rzeczy: wszystkie procesy mongos na 27017, wszystkie fragmenty na 27018, wszystkie serwery konfiguracyjne na 27019. Ta konfiguracja znacznie ułatwia śledzenie wszystkiego, jeśli budujesz klaster od zera, ale nie rób tego Nie przejmuj się tym zbytnio, jeśli masz istniejący zestaw replik, który zamieniasz w odłamek. --configsvr nie tylko zmienia domyślny port, ale na wszelki wypadek włącza diaglog, dziennik, który przechowuje każdą akcję wykonywaną przez bazę danych konfiguracji w formacie możliwym do odtworzenia. Jeśli używasz wersji 1.6, użyj tylko opcji --port 27019 i --diaglog, jako --configsvr

włącza diaglog w wersji 1.6.5+. Jeśli używasz wersji 1.8, użyj --port 27019 i --journal (zamiast --diaglog). Kronikowanie daje taki sam efekt, jak diaglog, ale mniejszy wpływ na wydajność.

**Sztuczka 44: Używaj tylko --notablescan w fazie rozwoju**

MongoDB ma opcję --notablescan, która zwraca błąd, gdy zapytanie musiałoby wykonać skanowanie tabeli (zapytania korzystające z indeksów są przetwarzane normalnie). Może to być przydatne w programowaniu, jeśli chcesz mieć pewność, że wszystkie zapytania trafiają do indeksów, ale nie używaj go w środowisku produkcyjnym. Problem polega na tym, że wiele prostych zadań administracyjnych wymaga skanowania tabel. Jeśli używasz MongoDB z --notablescan i chcesz zobaczyć listę kolekcji w swojej bazie danych, szkoda, że wymaga to skanowania tabeli. Chcesz wprowadzić aktualizacje administracyjne w oparciu o pola, które nie są indeksowane? Trudne, skanowanie tabel nie jest

dozwolone. --notablescan to dobre narzędzie do debugowania, ale zwykle wykonywanie tylko zapytań indeksowanych jest niezwykle niepraktyczne.

**Sztuczka 45: Naucz się JavaScript**

Nawet jeśli używasz języka z jego własną doskonałą powłoką (np. Python) lub ODM, który odciąga Twoją aplikację od bezpośredniego kontaktu z MongoDB (np. Mongoid), powinieneś znać powłokę JavaScript. Jest to najszybszy i najlepszy sposób na szybki dostęp do informacji i wspólny język wśród wszystkich programistów MongoDB. Aby wydobyć z powłoki wszystko, co możliwe, warto znać trochę JavaScript. Poniższe wskazówki omawiają niektóre funkcje języka, które są często pomocne, ale istnieje wiele innych, których możesz chcieć użyć. W Internecie jest mnóstwo darmowych zasobów, a jeśli lubisz książki (i myślę, że tak, jeśli czytasz ten), możesz chcieć pobrać JavaScript: The Good Parts (O'Reilly) który jest znacznie cieńszy i bardziej przystępny niż JavaScript: The Definitive Guide (również dobry, ale o 700 stron dłuższy). Nie mógłbym trafić w każdą użyteczną funkcję JavaScript, ale jest to bardzo elastyczny i potężny język.

**Sztuczka 46: Zarządzaj wszystkimi swoimi serwerami i bazami danych z jednej powłoki**

Domyślnie mongo łączy się z localhost: 27017. Możesz połączyć się z dowolnym serwerem podczas uruchamiania, uruchamiając mongo host: port / database. Możesz także łączyć się z wieloma serwerami lub bazami danych w powłoce.

Na przykład załóżmy, że mamy aplikację, która ma dwie bazy danych: jedną bazę danych klientów i jedną bazę danych gier. Jeśli pracujesz z oboma, możesz przełączać się między nimi za pomocą klientów użytecznych, używać gier, klientów i tak dalej. Możesz jednak po prostu użyć oddzielnych zmiennych dla oddzielnych baz danych:

> db

test

> customers = db.getSisterDB("customers")

customers

> game = db.getSisterDB("game")

game

Teraz możesz ich używać w ten sam sposób, w jaki używasz db:

game.players.find(), custom

ers.europe.update(), etc.

You can also connect db, or any other variable, to another server:

> db = connect("ny1a:27017/foo")

connecting to: ny1a:27017/foo

foo

> db

foo

Może to być szczególnie przydatne, jeśli używasz zestawu replik lub klastra podzielonego na fragmenty i chcesz połączyć się z więcej niż jednym węzłem. Możesz utrzymywać oddzielne połączenia z master i slave w swojej powłoce:

```
> master = connect("ny1a:27017/admin")

connecting to: ny1a:27017/admin

admin

> slave = connect("ny1b:27017/admin")

connecting to: ny1b:27017/admin

admin
```

Możesz także łączyć się bezpośrednio z shardami, serwerami konfiguracji lub dowolnym uruchomionym serwerem MongoDB. Niektóre funkcje powłoki, szczególnie pomoce rs, zakładają, że używasz bazy danych db. Jeśli db jest podłączony do slave'a lub arbitra, niektórzy pomocnicy nie będą działać. Jedną irytacją związaną z łączeniem się z wieloma serwerami przez powłokę jest to, że MongoDB śledzi wszystkie połączenia, które kiedykolwiek nawiązałeś, a jeśli jedno z nich się zepsuje, będzie narzekać na to bez przerwy, dopóki nie przywrócisz serwera lub nie zrestartujesz powłoki. Nawet niezdefiniowanie połączenia nie powoduje zresetowania tego! Zostanie to naprawione w wersji 1.9, ale obecnie jest to nieszkodliwe, ale hałaśliwe.

**Sztuczka 47: Uzyskaj „pomoc” dla dowolnej funkcji**

JavaScript pozwala zobaczyć kod źródłowy większości funkcji uruchamianych w powłoce. Jeśli jesteś ciekawy, jakie argumenty przyjmuje funkcja lub nie pamiętasz, jakie zwraca, możesz zobaczyć źródło, uruchamiając nazwę funkcji bez nawiasów. Na przykład załóżmy, że pamiętamy, że db.addUser dodaje użytkownika, ale nie pamiętamy dokładnie, jakie są argumenty:

```
> db.addUser

function (username, pass, readOnly) {

readOnly = readOnly || false;

var c = this.getCollection("system.users");

var u = c.findOne({user: username}) || {user: username};

u.readOnly = readOnly;

u.pwd = hex_md5(username + (":mongo:" + pass));

print(tojson(u));

c.save(u);

}
```

Od razu widzimy, że nadajemy mu nazwę użytkownika, hasło i że istnieje opcja tylko do odczytu (aby utworzyć użytkownika, który może czytać tylko z danej bazy danych). Pamiętaj również, że możesz zobaczyć JavaScript API online. „Dokumentacja” online nie jest właściwie udokumentowana, ale zawiera pełne odniesienie do dostępnych funkcji. Istnieje również sporo wbudowanej pomocy dla

poleceń. Jeśli nie pamiętasz polecenia, które chcesz uruchomić, możesz zobaczyć je wszystkie, o ile pamiętasz jedno: polecenie listCommands! To pokazuje nazwę każdego polecenia:

```
> db.runCommand({listCommands : 1})

{

"commands" : {

"_isSelf" : { ... },

...

}

"ok" : 1

}
```

Jeśli masz nazwę polecenia, możesz pobrać wbudowaną dokumentację z bazy danych, uruchamiając {commandName: 1, help: 1} (nawet jeśli polecenie zwykle nie ma 1 po nazwie). Spowoduje to wyświetlenie podstawowej dokumentacji dotyczącej każdego polecenia, którą baza danych zawiera, od bardzo pomocnych do ledwo angielskich:

```
> db.runCommand({collstats : 1, help : 1})

{

"help" : "help for: collStats { collStats: "blog.posts" , scale : 1 }

scale divides sizes e.g. for KB use 1024",

"lockType" : -1,

"ok" : 1

}
```

Powłoka ma również uzupełnianie kart, więc możesz otrzymywać sugestie dotyczące tego, co należy wpisać, w oparciu o funkcje, pola, a nawet istniejące kolekcje:

```
> db.c

db.cloneCollection( db.constructor db.currentOP(

db.cloneDatabase( db.copyDatabase( db.currentOp(

db.commandHelp( db.createCollection(

> db.copyDatabase()
```

Wskazówka # 48: Utwórz pliki startowe

Opcjonalnie można uruchomić plik (lub pliki) startowe podczas uruchamiania powłoki. Plik startowy to zwykle lista funkcji pomocniczych zdefiniowanych przez użytkownika, ale jest to po prostu program JavaScript. Aby to zrobić, utwórz plik z przyrostkiem .js (powiedzmy startup.js) i uruchom mongo z mongo startup.js. Na przykład, przypuśćmy, że robisz konserwację powłoki i nie chcesz przypadkowo upuścić bazy danych lub usunąć rekordów. Możesz usunąć niektóre mniej bezpieczne polecenia w powłoce (np. Usuwanie bazy danych i kolekcji, usuwanie dokumentów itp.):

```
// no-delete.js

usuń DBCollection.prototype.drop;

usuń DBCollection.prototype.remove;

usuń DB.prototype.dropDatabase;
```

Teraz, jeśli spróbujesz usunąć kolekcję, mongo nie rozpozna funkcji:

```
$ mongo no-delete.js

Wersja powłoki MongoDB: 1.8.0

łączenie z: test

> db.foo.drop ()
```

Wed Feb 16 14:24:16 TypeError: db.foo.drop nie jest funkcją (powłoka): 1 Ma to na celu tylko uniemożliwić użytkownikowi wyrzucenie danych: zapewnia zerową ochronę przed użytkownikiem, który wie, co robi i jest zdecydowany porzucić kolekcję. Usuwanie funkcji nie powinno być używane jako zabezpieczenie przed złośliwymi atakami (ponieważ nie daje żadnego), tylko po to, aby zapobiec błędom. Jeśli ktoś był naprawdę zdeterminowany, aby porzucić kolekcję i drop () zniknął, mógłby po prostu uruchomić db. $ Cmd.findOne ({drop: "foo"}). Nie można temu zapobiec bez usunięcia funkcji find (), co spowodowałoby, że powłoka byłaby zasadniczo bezużyteczna. Możesz utworzyć dość obszerną listę funkcji na czarnej liście, w zależności od tego, co chcesz zapobiec (tworzenie indeksu, uruchamianie poleceń bazy danych, itp.) Możesz określić dowolną liczbę tych plików podczas uruchamiania mongo, abyś mógł je modularyzować , także.

**Sztuczka 49: Dodaj własne funkcje**

Jeśli chcesz dodać własne funkcje, możesz je zdefiniować i używać, dodając je jako funkcje globalne, do instancji klasy lub do samej klasy (co oznacza, że każda instancja klasy będzie miała instancję funkcji) . Na przykład załóżmy, że używamy „Wskazówka # 46: Zarządzaj wszystkimi swoimi serwerami i bazami danych z jednej powłoki” na stronie 46, aby połączyć się z każdym elementem zestawu replik i chcemy dodać funkcję getOplogLength. Jeśli pomyślimy o tym, zanim zaczniemy, moglibyśmy dodać to do klasy bazy danych (DB):

```
DB.prototype.getOplogLength = function() {

var local = this.getSisterDB("local");

var first = local.oplog.rs.find().sort({$natural : 1}).limit(1).next();

var last = local.oplog.rs.find().sort({$natural : -1}).limit(1).next();

print("total time: " + (last.ts.t - first.ts.t) + " secs");

};
```

Następnie, gdy łączymy się z bazami danych rsA, rsB i rsC, każda z nich będzie miała metodę getOplogSize. Jeśli zaczęliśmy już używać rsA, rsB i rsC, to nie zauważą, że dodałeś nową metodę do klasy, z której pochodzą (klasy w JavaScript przypominają

szablony dla instancji klas: instancja nie jest zależna od klasy po zainicjowaniu). Jeśli połączenia zostały już zainicjowane, możesz dodać tę metodę do każdej instancji:

```
// store the function in a variable to save typing

var f = function() { ... }

rsA.getOplogSize = f;

rsB.getOplogSize = f;

rsC.getOplogSize = f;
```

You could also just alter it slightly to be a global function:

```
getOplogLength = function(db) {

var local = db.getSisterDB("local");

...

};
```

Możesz oczywiście zrobić to również dla pól obiektu (jak również dla jego metod).

Ładowanie JavaScript z plików

Możesz dodać biblioteki JavaScript do swojej powłoki w dowolnym momencie za pomocą funkcji load (). load () pobiera plik JavaScript i wykonuje go w kontekście powłoki (dzięki czemu będzie wiedział o wszystkich globalnych zmiennych w powłoce). Możesz także dodawać zmienne do globalnego zasięgu powłoki, definiując je w załadowanych plikach. Możesz również wydrukować wynik z tych plików do powłoki za pomocą funkcji print:

```
// hello.js

print("Hello, world!")
```

Then, in the shell:

```
> load("hello.js")

Hello, world!
```

Jedno z najczęstszych zapytań dotyczących zestawów replik i fragmentacji pochodzi od osób z ops, które chcą mieć możliwość ich skonfigurowania z pliku konfiguracyjnego. Musisz programowo skonfigurować zestawy replik i fragmenty, ale możesz zapisać funkcje konfiguracyjne w pliku JavaScript, który możesz wykonać, aby skonfigurować zestaw. Już prawie będzie można użyć pliku konfiguracyjnego.

**Sztuczka 50: Użyj jednego połączenia do czytania własnych zapisów**

Podczas tworzenia połączenia z serwerem MongoDB to połączenie zachowuje się jak kolejka żądań. Na przykład, jeśli wyślesz wiadomości A, B, a następnie C do bazy danych za pośrednictwem tego połączenia, MongoDB przetworzy wiadomość A, następnie wiadomość B, a następnie wiadomość C. To nie gwarantuje, że każda operacja zakończy się sukcesem: A może będzie komendą shutdownServer, a następnie B i C zwróciłyby błędy (jeśli byłyby to wiadomości, które w ogóle oczekiwały odpowiedzi). Masz jednak gwarancję, że zostaną wysłane i przetworzone w kolejności. Jest to przydatne: załóżmy, że zwiększasz liczbę pobrań produktu i wykonujesz findOne w tym produkcie: spodziewasz się wzrostu liczby pobrań. Jeśli jednak używasz więcej niż jednego połączenia (a większość sterowników automatycznie korzysta z puli połączeń), możesz tego nie robić. Załóżmy, że masz dwa połączenia z

bazą danych (z tego samego klienta). Każde z połączeń będzie wysyłać wiadomości, które będą przetwarzane szeregowo, ale nie ma gwarancji kolejności między połączeniami: jeśli pierwsze połączenie wysyła wiadomości A, B i C, a drugie wysyła wiadomości D, E i F, wiadomości mogą być przetwarzane jako A, D, B, E, C, F lub A, B, C, D, E, F lub dowolne inne połączenie dwóch sekwencji. Jeśli A wstawia nowy dokument, a D pyta o ten dokument, D może skończyć jako pierwszy (powiedzmy D, A, E, B, F, C) i tym samym nie znaleźć rekordu. Aby to naprawić, sterowniki z pulą połączeń zazwyczaj określają, że grupa żądań powinna być wysyłana przez to samo połączenie, aby zapobiec rozbieżności „odczytu własnego zapisu". Inne sterowniki zrobią to automatycznie (używając jednego połączenia z puli na „sesję"). Szczegółowe informacje znajdziesz w dokumentacji sterownika.