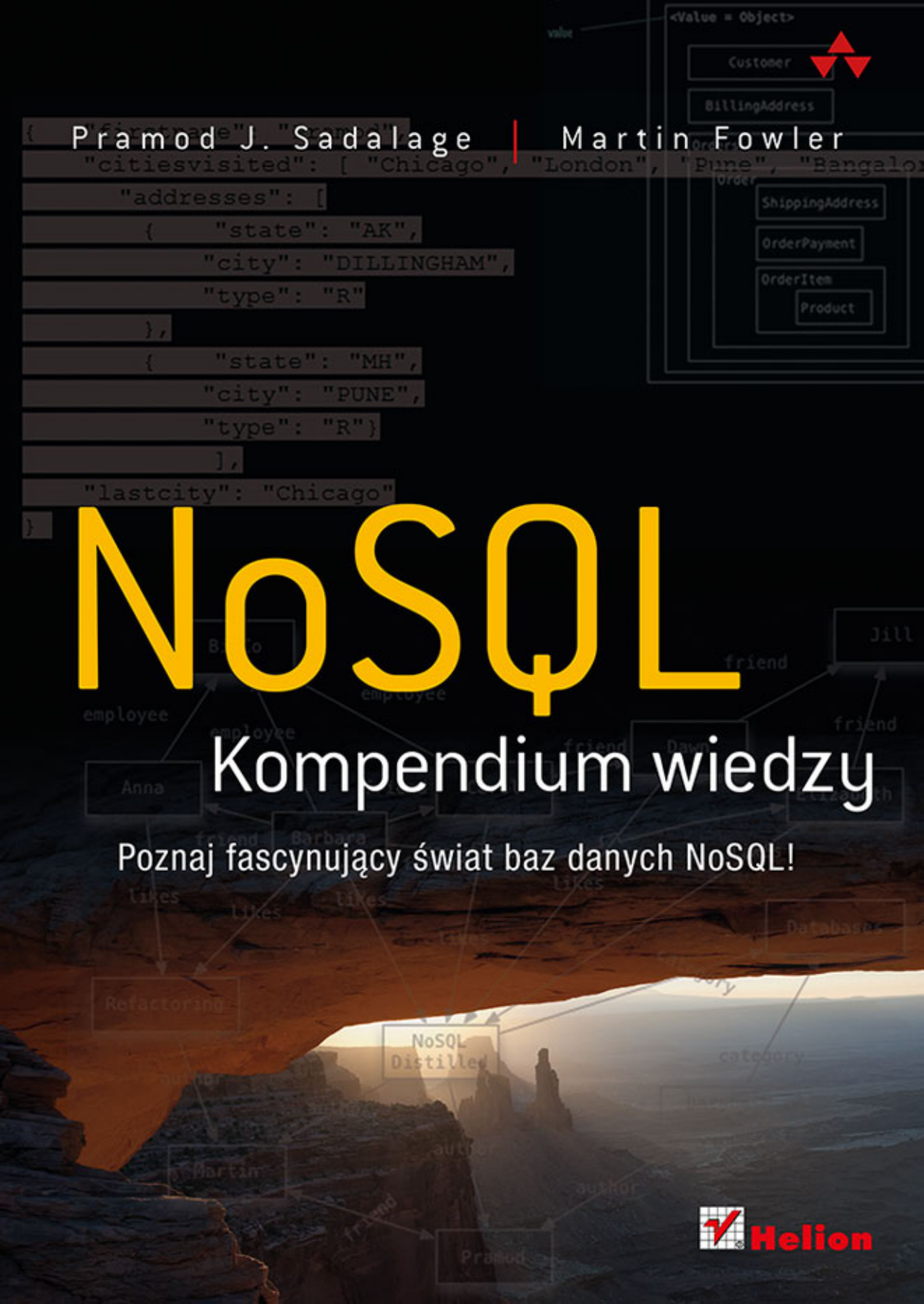

Pramod J. Sadalage
Martin Fowler
NoSQL
Kompendium wiedzy
Poznaj fascynujący świat baz danych NoSQL!
Helion

Tytuł oryginału: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence

Tłumaczenie: Jakub Hubisz

ISBN: 978-83-246-9905-6

Authorized translation from the English language edition, entitled: NOSQL DISTILLED:
A BRIEF GUIDE TO THE EMERGING WORLD OF POLYGLOT PERSISTENCE;
ISBN 0321826620; by Pramodkumar J. Sadalage; and by Martin Fowler; published
by Pearson Education, Inc, publishing as Addison Wesley.

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)

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

Printed in Poland.

Spis treści

Rozdział 4

Modele dystrybucyjne

Główną cechą napędzającą zainteresowanie bazami NoSQL jest ich zdolność do działania w dużych klastrach. W miarę jak ilość danych rośnie, skalowanie w górę (kupno większego serwera, na którym działa baza danych) staje się coraz trudniejsze i droższe. Wygodniejszą opcją jest skalowanie w bok — uruchomienie bazy na klastrze serwerów. Orientacja na agregacje daje dobre podstawy do skalowania w bok, ponieważ agregacja jest idealną jednostką do dystrybucji.

Zależnie od Twojego modelu dystrybucyjnego możesz wykorzystać magazyn danych, który da Ci możliwość obsługi większych ilości danych, możliwość przetworzenia większego ruchu odczytu i zapisu lub większą niezawodność w dostępie do danych. Zazwyczaj są to ważne cechy, jednak wszystko ma swoją cenę. Praca w klastrze wprowadza złożoność — nie jest to więc coś, co powinieneś rozważać, jeżeli nie ma prawdziwej konieczności.

Są dwie główne ścieżki dystrybucji: replikacja i współdzielenie. W przypadku replikacji te same dane kopiowane są na wiele serwerów. Współdzielenie to umieszczenie różnych danych na różnych serwerach. Replikacja i współdzielenie to techniki ortogonalne — możesz korzystać tylko z jednej lub z obu. Replikacja przyjmuje dwie formy: *master-slave* lub *peer-to-peer*. Omówimy powyższe techniki, rozpoczynając od najprostszej, a następnie przechodząc do bardziej złożonych: najpierw pojedynczy serwer, potem replikacja *master-slave*, następnie współdzielenie i w końcu replikacja *peer-to-peer*.

4.1. Pojedynczy serwer

Pierwszą i najprostszą opcją dystrybucji jest ta, którą przeważnie polecam — brak dystrybucji, czyli uruchomienie bazy danych na jednej maszynie, która obsługuje wszystkie odczyty i zapisy do magazynu danych. Polecamy tę opcję, ponieważ pozwala wyeliminować problemy istniejące przy innych formach dystrybucji. Taki model nie sprawia problemów ani osobom zajmującym się serwerem, ani programistom korzystającym z bazy.

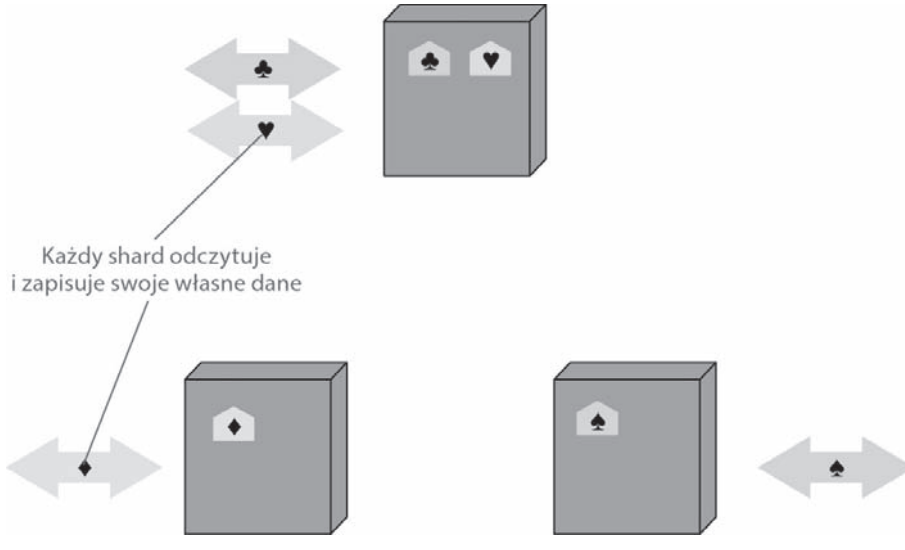
Mimo że wiele baz NoSQL zaprojektowanych jest z myślą o pracy w klastrach, sensowne jest też używanie NoSQL na pojedynczym serwerze, jeżeli model danych oferowany przez NoSQL jest bardziej przystosowany do danych zastosowań. Bazy grafowe są tutaj oczywistym przykładem — działają najlepiej na jednym serwerze. Jeżeli Twoje potrzeby skupiają się na

przetwarzaniu agregacji, jednoserwerowy magazyn dokumentów lub baza klucz – wartość mogą być warte rozważenia, ponieważ ułatwią pracę programistom.

Przez resztę tego rozdziału będziemy rozważać zalety i problemy związane z bardziej rozbudowanymi modelami dystrybucyjnymi. Nie myśl jednak, że ilość poświęconego miejsca oznacza, że preferujemy te opcje. Jeżeli możemy poradzić sobie bez konieczności dystrybuowania danych, zawsze wybieramy rozwiązanie z pojedynczym serwerem.

4.2. Współdzielenie

Zajęty magazyn danych jest często zajęty, ponieważ różne osoby uzyskują dostęp do różnych części danych. W takich przypadkach możemy wykonać skalowanie poziome, umieszczając różne części danych na różnych serwerach — technika ta nazywana jest **shardingiem** (rysunek 4.1).



Rysunek 4.1. *Sharding pozwala umieścić różne dane na osobnych serwerach; każdy z tych serwerów wykonuje własne odczyty i zapisy*

W idealnym przypadku różni użytkownicy komunikowaliby się z różnymi serwerami. Każdy użytkownik musiałby komunikować się z tylko jednym serwerem, dzięki czemu odpowiedzi na żądania byłyby bardzo szybkie. Ruch rozkładałby się na wszystkie serwery — na przykład jeżeli mamy dziesięć serwerów, każdy musiałby obsłużyć tylko 10% ruchu.

Oczywiście sytuacja idealna występuje bardzo rzadko. Aby się do niej zbliżyć, musimy zadbać, aby dane, które pobierane są wspólnie, były przechowywane razem oraz aby organizacja danych była maksymalnie zoptymalizowana pod względem dostępu do danych.

Pierwsze pytanie brzmi, jak zgrupować dane tak, aby użytkownik, uzyskując do nich dostęp, odwoływał się w większości do jednego serwera. Tutaj z pomocą przychodzi orientacja na agregacje. Cała idea agregacji polega na budowaniu ich w taki sposób, aby dane, które prze-

ważnie obsługiwane są wspólnie, były też wspólnie przechowywane — agregacje stają się więc naturalną jednostką dystrybucji.

Jeżeli chodzi o organizację danych na serwerze, jest kilka czynników, które mogą pomóc poprawić wydajność. Jeżeli wiesz, że większość dostępu do danych agregacji wychodzi z jednego fizycznego miejsca, możesz umieścić te dane blisko odbiorców. Jeżeli realizujesz zamówienia dla kogoś mieszkającego w Londynie, możesz umieścić dane w centrum danych na Wyspach Brytyjskich.

Kolejnym czynnikiem jest rozłożenie obciążenia. Oznacza to, że powinieneś tak organizować agregacje, aby były rozłożone na serwerach równomiernie, dzięki czemu poszczególne serwery będą obciążone w podobnym stopniu. Ten czynnik może być zmienny w czasie, na przykład jeżeli niektóre dane używane są tylko w niektóre dni tygodnia — w tym przypadku mogą więc występować reguły zależne od zastosowania bazy.

W niektórych przypadkach warto umieszczać agregacje razem, jeżeli myślisz, że mogą być pobierane w sekwencjach. Dokument o BigTable [Chang i inni] opisał przechowywanie wierszy w porządku leksykograficznym i sortowanie adresów na podstawie odwróconych domen (np. `com.martinfowler`). Dzięki temu dane z różnych stron mogły być odczytywane razem, co pozwoliło podnieść wydajność przetwarzania.

Większość projektantów uwzględniała lub uwzględnia sharding w warstwie logiki aplikacji. Możesz umieścić wszystkich klientów o nazwiskach zaczynających się od A do D na jednym serwerze, a tych, których nazwiska zaczynają się od E do G, na innym. To komplikuje model programistyczny, ponieważ kod aplikacji musi zadbać, aby zapytania były dystrybuowane na różne serwery. Co więcej, zmiana rozłożenia danych na serwerach wiąże się ze zmianami w kodzie aplikacji oraz migracją danych. Wiele baz NoSQL udostępnia opcję *auto-sharding*, która pozwala przenieść odpowiedzialność za rozdzielanie danych i przekierowywanie zapytań do odpowiedniego serwera na bazę. Takie rozwiązanie jest znacznie prostsze niż sharding za pośrednictwem aplikacji.

Sharding jest bardzo pomocny w poprawianiu wydajności, ponieważ zwiększa zarówno szybkość odczytu, jak i szybkość zapisu danych. Korzystając z replikacji, szczególnie przy wykorzystywaniu buforowania, można znacznie poprawić wydajność operacji odczytu, jednak replikacja nie poprawi wydajności w przypadku bazy, do której wykonywane jest dużo zapisów.

Sam sharding poprawia niezawodność w bardzo niewielkim stopniu. Mimo że dane są na różnych serwerach, awaria serwera spowoduje, że dane te będą niedostępne, tak samo jak w przypadku rozwiązania z pojedynczym serwerem. Jedyna poprawa polega na tym, że w przypadku awarii serwera jej skutki odczują tylko użytkownicy tych danych; mimo wszystko jednak niedobrze jest mieć bazę danych posiadającą tylko część danych. Mając pojedynczy serwer, łatwiej jest poświęcić uwagę i pieniądze w celu zapewnienia jego działania; w klastrach pracują przeważnie bardziej zawodne maszyny i awaria serwera w klastrze jest bardziej prawdopodobna. To wszystko powoduje, że w praktyce sam sharding wpływa na osłabienie niezawodności.

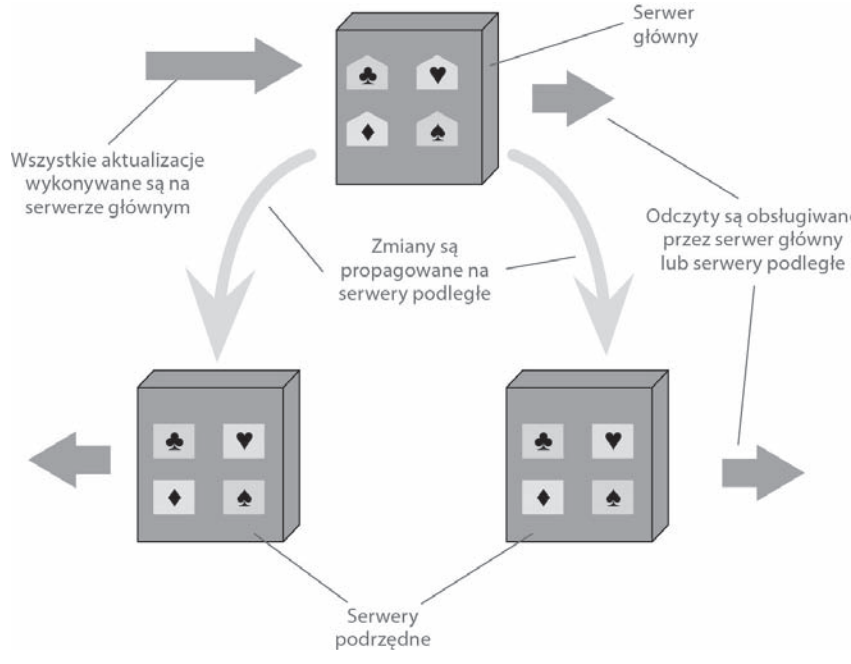
Mimo że sharding jest łatwiejszy do zaimplementowania w przypadku agregacji, nie jest to krok, który powinniśmy wykonywać pochopnie. Niektóre bazy danych są przeznaczone do wykorzystania shardingu od samego początku; w takim przypadku warto uruchamiać je na klastrze już w czasie prac nad programem i koniecznie od początku działania aplikacji produkcyjnej. Inne bazy pozwalają na wykorzystanie shardingu w formie rozbudowy

rozwiązania jednoserwerowego; wtedy klaster powinien być zakładany dopiero wtedy, gdy wydajność pojedynczego serwera przestanie być wystarczająca.

W obu przypadkach przejście z jednego serwera do klastra będzie wymagało uwagi. Znamy opowieści o zespołach, które napotkały problemy, ponieważ zbyt długo odkładały sharding, a kiedy włączyły go na środowisku produkcyjnym, baza przestała być dostępna, ponieważ wsparcie dla shardingu pochłonęło wszystkie dostępne zasoby na potrzeby przenoszenia danych na nowe serwery. Morał z tego taki, że powinieneś implementować sharding, zanim zaczniesz go potrzebować, kiedy jeszcze masz zasoby do przenoszenia danych.

4.3. Replikacja master-slave

Podczas dystrybucji *master-slave* replikujesz dane na wiele serwerów. Jeden z serwerów jest serwerem głównym (*master*). Serwer główny jest autorytatywnym źródłem danych i jest z reguły odpowiedzialny za przetwarzanie aktualizacji tych danych. Pozostałe serwery to serwery podległe (*slave*). Proces replikacji synchronizuje serwery podrzędne z serwerem głównym (rysunek 4.2).



Rysunek 4.2. *Dane są replikowane z serwera głównego do serwerów podległych. Główny serwer obsługuje zapis danych, a odczyt może być obsłużony przez serwer główny lub któryś z serwerów podległych*

Replikacja *master-slave* jest pomocna w przypadku systemów, w których dane są głównie odczytywane. Można skalować w bok w celu poprawienia wydajności odczytu danych poprzez dodanie większej liczby serwerów i zapewnienie rozprowadzenia po nich ruchu.

W przypadku zapisu jesteś jednak nadal ograniczany przez możliwości głównego serwera. W konsekwencji nie jest to rozwiązanie dobre dla baz obsługujących duży ruch, chociaż rozłożenie odczytów pozwoli też nieznacznie poprawić wydajność zapisu.

Drugą zaletą replikacji *master-slave* jest **niezawodność odczytu**: jeżeli główny serwer zawiedzie, serwery podrzędne nadal będą odpowiadały na żądania odczytu. Jest to oczywiście użyteczne, jeżeli większość ruchu stanowi odczyt. Awaria serwera głównego eliminuje możliwość zapisu danych, dopóki serwer główny nie zostanie przywrócony lub nie zostanie wyznaczony nowy serwer główny. Mając serwery podległe, na które replikowane są dane z serwera głównego, możemy szybko przywrócić system do działania, ponieważ serwer podległy może być szybko wyznaczony na serwer główny.

Możliwość wyznaczenia serwera podległego na nowy serwer główny oznacza, że ten typ replikacji może być przydatny nawet wtedy, kiedy nie musisz skalować w bok. Cały ruch może być kierowany na serwer główny, a serwer poboczny może działać jako wciąż aktualizowana kopia zapasowa. W takim przypadku najłatwiej myśleć o systemie jako o systemie z pojedynczym serwerem z automatycznie aktualizowaną kopią zapasową. Wygoda systemu jednoserwerowego łączy się z poprawą niezawodności — jest to szczególnie wygodne, jeżeli chcesz z łatwością radzić sobie z awariami systemu.

Serwery główne mogą być wyznaczane ręcznie lub automatycznie. Ręczne wyznaczanie serwera głównego oznacza zazwyczaj, że podczas konfiguracji klastra określasz jeden z serwerów jako główny. W wyznaczaniu automatycznym tworzysz klaster serwerów, a serwery same wybierają jeden z nich na serwer główny. Poza łatwiejszą konfiguracją wyznaczanie automatyczne oznacza, że klaster może automatycznie wyznaczyć nowy serwer główny w przypadku awarii poprzednika.

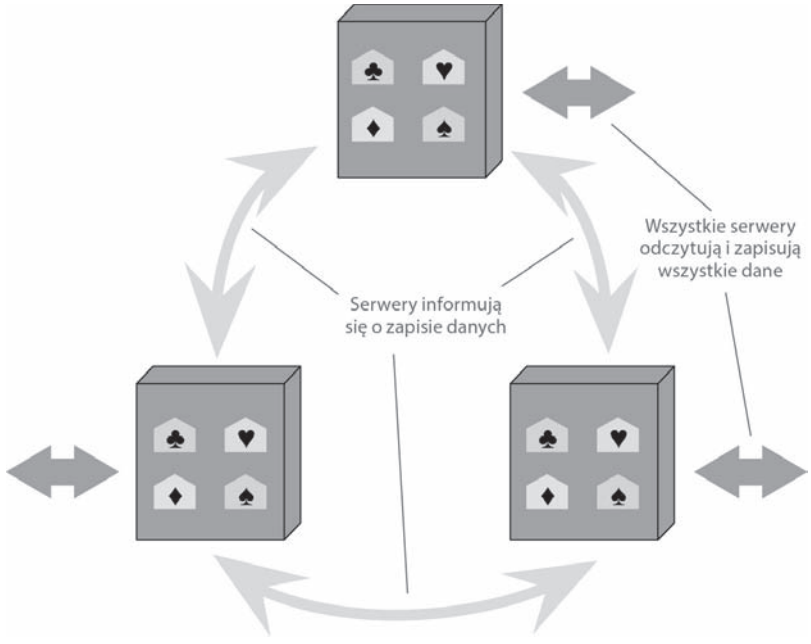
W celu uzyskania niezawodności odczytu musisz dopilnować, aby ścieżki odczytu i zapisu w Twojej aplikacji były różne, tak abyś mógł obsłużyć awarię ścieżki zapisu i móc nadal odczytywać dane. Łączy się to z koniecznością zapisywania i odczytywania danych za pomocą różnych połączeń — jest to funkcjonalność nieczęsto wspierana przez biblioteki do interakcji z bazami. Jak w przypadku wszystkich funkcjonalności, nie możesz mieć pewności, że odczyt danych jest niezawodny, bez dobrych testów wyłączających możliwość zapisu i sprawdzających odczyt.

Replikacja ma na pewno pociągające zalety, ma jednak także ciemną stronę — brak spójności. Istnieje niebezpieczeństwo, że różni klienci odczytujący różne serwery podrzędne zobaczą różne wartości, ponieważ wszystkie zmiany nie zostały rozpropagowane po wszystkich serwerach. W najgorszym przypadku może to oznaczać, że klient nie zobaczy danych, które właśnie zapisał. Nawet jeżeli korzystasz z replikacji *master-slave* wyłącznie do tworzenia kopii zapasowej danych, nadal możesz mieć problem, ponieważ jeżeli serwer główny ulegnie awarii, zmiany nieprzekazane do serwera podrzędnego zostaną utracone. Porozmawiamy o tym, jak radzić sobie z tym problemem, w rozdziale 5., „Spójność”.

4.4. Replikacja peer-to-peer

Replikacja *master-slave* pomaga w skalowaniu możliwości odczytu danych, nie pomaga jednak w przypadku zapisu. Poprawia niezawodność pod względem awarii serwerów podległych, ale nie serwera głównego. W gruncie rzeczy serwer główny jest nadal słabym ogniwem

systemu i miejscem podatnym na awarię. Replikacja *peer-to-peer* (rysunek 4.3) radzi sobie z tym problemem, eliminując serwer główny. Wszystkie repliki mają równe prawa i wszystkie mogą wykonywać zapis, a utrata którejś z nich nie ogranicza dostępu do magazynu danych.



Rysunek 4.3. *W replikacji peer-to-peer wszystkie serwery obsługują odczyt i zapis danych*

Perspektywa wygląda pięknie. Z klastrem replikującym się w trybie *peer-to-peer* możesz radzić sobie z awariami serwerów bez utraty dostępu do danych. Co więcej, możesz z łatwością dodawać nowe serwery, poprawiając wydajność. Taki system ma wiele zalet — powoduje też jednak pewne komplikacje.

Największym problemem jest, po raz kolejny, spójność. Jeżeli możesz zapisywać w dwóch różnych miejscach, występuje ryzyko, że dwie osoby spróbują jednocześnie zapisać te same dane: konflikt zapis – zapis. Niespójność przy odczycie prowadzi do problemów, ale niespójność taka jest chwilowa. Niespójność przy zapisie jest natomiast trwała.

W dalszej części wyjaśnimy, jak radzić sobie z niespójnością danych, w tej chwili jednak omówimy z grubsza kilka opcji. Z jednej strony możemy zapewnić, że kiedy dane zostaną zapisane, repliki skoordynują się w celu zapobieżenia konfliktom. Może nam to dać tak dobrą gwarancję jak baza główna, jednak kosztem ruchu sieciowego występującego podczas koordynacji. Nie potrzebujemy, aby wszystkie repliki zgadzały się na zapis, wystarczy większość, dzięki czemu będziemy mogli przeżyć, jeżeli stracimy mniejszość serwerów.

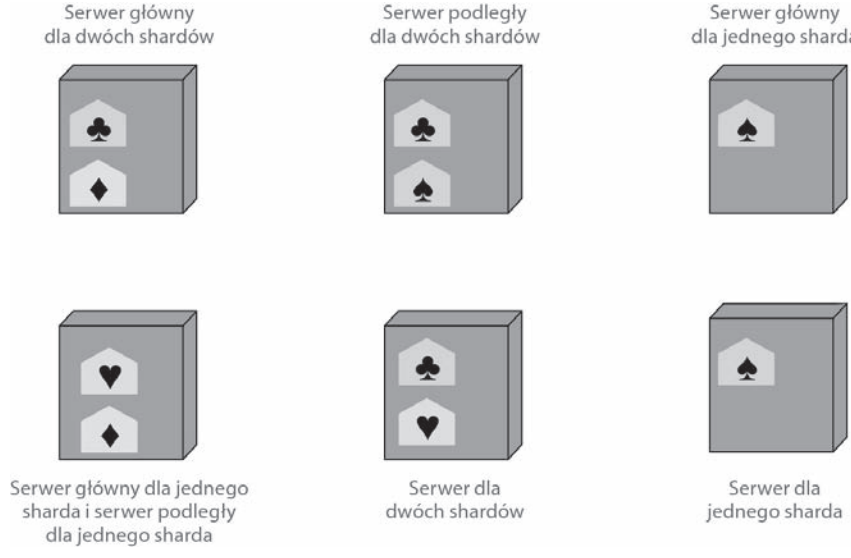
Z drugiej strony możemy zdecydować, aby zezwolić na niespójny zapis. W niektórych przypadkach możemy znaleźć sposób na scalenie niespójnych danych. W takim przypadku możemy korzystać z pełni wydajności zapewnianej przez repliki.

Te dwa sposoby są po przeciwnych stronach spektrum — zamieniamy spójność na dostępność.

4.5. Łączenie shardingu i replikacji

Replikacja i sharding to strategie, które możemy połączyć. Jeżeli wykorzystamy zarówno replikację *master-slave*, jak i sharding (rysunek 4.4), będziemy mieli wiele serwerów głównych, jednak każda część danych będzie miała tylko jeden serwer główny. Zależnie od konfiguracji możesz wybrać, aby serwer był serwerem głównym dla niektórych danych, a serwerem podległym dla innych, lub możesz wyznaczyć serwery dedykowane do bycia serwerami głównymi i pobocznymi.

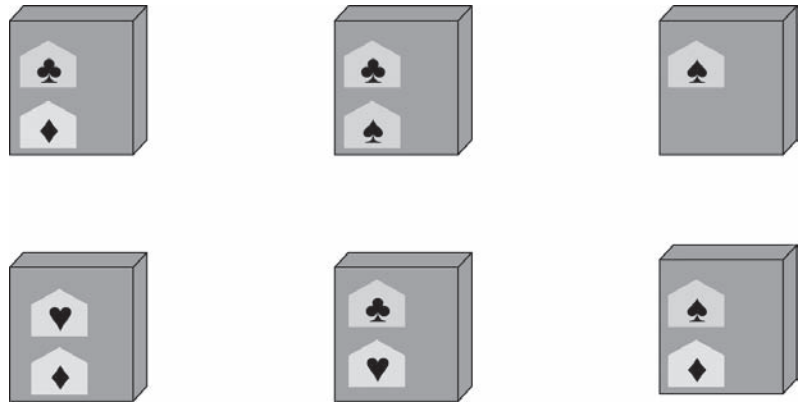
Wykorzystanie replikacji *peer-to-peer* i shardingu jest częste w przypadku baz rodziny kolumn. W takim przypadku możesz mieć dziesiątki lub setki serwerów w klastrze. Dobrym punktem startu w replikacji *peer-to-peer* jest współczynnik powtórzenia równy 3, czyli każda część danych reprezentowana jest na trzech serwerach. Jeżeli któryś z serwerów ulegnie awarii, shardy z tego serwera będą działały na pozostałych serwerach (rysunek 4.5).



Rysunek 4.4. *Wykorzystanie replikacji master-slave razem z shardingiem*

4.6. Najważniejsze kwestie

- Istnieją dwa sposoby obsługi danych rozproszonych:
  - Sharding rozdziela dane pomiędzy wiele serwerów, tak aby każdy serwer działał jako źródło dla podzbioru danych.
  - Replikacja kopiuje dane pomiędzy wieloma serwerami, tak aby każda część danych znajdowała się w wielu miejscach.



Rysunek 4.5. *Wykorzystanie replikacji peer-to-peer razem z shardingiem*

- Replikacja przyjmuje dwie postacie:
  - W replikacji *master-slave* jeden serwer odgrywa rolę serwera głównego obsługującego zapis danych i rozgłaszającego dane do serwerów podległych, które mogą obsługiwać odczyt danych.
  - W replikacji *peer-to-peer* zapis możliwy jest do dowolnego serwera; serwery koordynują synchronizację danych.

  Replikacja *master-slave* redukuje ryzyko wystąpienia konfliktu zapisu, ale replikacja *peer-to-peer* pomaga rozładować ruch związany z zapisem danych.

Skorowidz

F

G

H

I

K

L

Ł

M

N

O

P

Q

R

S

T

W

Z

Zmień sposób myślenia na nierelacyjny!

Bazy danych NoSQL są coraz popularniejsze. Pozwalają na przechowywanie gigantycznych ilości informacji, a przy tym zachowują cały czas najwyższą wydajność. Sprawdzają się doskonale wszędzie tam, gdzie konieczne są: wysoka skalowalność systemu, elastyczne przechowywanie często zmieniających się danych lub inne specyficzne zastosowania. Jeżeli jesteś zagorzałym użytkownikiem relacyjnych baz danych SQL, jeżeli słyszałeś o bazach NoSQL i chcesz je poznać, trafiłeś na doskonałą książkę!

Stanowi ona świetne wprowadzenie do świata baz danych NoSQL. Na własnej skórze przekonasz się, w jakich zastosowaniach sprawdzą się one dobrze, a w jakich lepiej ich nie używać. W kolejnych rozdziałach poznasz stosowane modele danych oraz dowiesz się, co to jest map-reduce. Część druga książki została poświęcona konkretnym implementacjom — zaznajomisz się z bazami klucz-wartość, bazami dokumentów oraz bazami grafowymi. Sprawdź, które najlepiej rozwiążą Twoje problemy! Sięgnij po tę książkę i śmiało wkrocz w świat baz danych NoSQL!

PRAMOD J. SADALAGE — pracownik firmy ThoughtWorks, rozpoznawanej przez specjalistów branży IT na całym świecie. Doradza klientom i pomaga rozwiązywać problemy z zakresu baz danych i programowania. Pomysłodawca ewolucyjnego sposobu przechowywania informacji.

MARTIN FOWLER — legenda branży IT. Pracuje dla cenionej firmy ThoughtWorks i zajmuje się najlepszymi sposobami projektowania systemów informatycznych. W obszarze jego zainteresowań znajdują się również zagadnienia związane z refaktoryzacją kodu oraz pracą nad produktywnością programistów.

Dzięki tej książce:

- poznasz nierelacyjne modele przechowywania danych
- przekonasz się, gdzie najlepiej stosować bazy danych NoSQL
- nauczysz się korzystać z mechanizmu map-reduce
- wybierzesz bazę, która zaspokoi Twoje potrzeby
- zrozumiesz nierelacyjny świat baz danych NoSQL

sięgnij po WIĘCEJ

KOD KORZYŚCI

helion.pl księgarnia internetowa

Nr katalogowy: 25880

Księgarnia internetowa: http://helion.pl

Zamówienia telefoniczne: 0 801 339900 / 0 601 339900

Addison-Wesley
Pearson Education

Helion

Sprawdź najnowsze promocje: http://helion.pl/promocje
Książki najchętniej czytane: http://helion.pl/bestsellery
Zamów informacje o nowościach: http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

ISBN 978-83-246-9905-6

cena: 39,00 zł

Informatyka w najlepszym wydaniu