Nathan Marz, James Warren

BIG DATA

NAJLEPSZE PRAKTYKI BUDOWY SKALOWALNYCH SYSTEMÓW OBSŁUGI DANYCH W CZASIE RZECZYWISTYM

Tytuł oryginału: Big Data: Principles and best practices of scalable realtime data systems

Tłumaczenie: Lech Lachowski
Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

ISBN: 978-83-283-1892-2

Original edition copyright © 2015 by Manning Publications Co.
All rights reserved

Polish edition copyright © 2016 by HELION SA.
All rights reserved.

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

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

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

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

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

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.
Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/bigdat.zip

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

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Przedmowa 11
Podziękowania 13
O książce 17

Rozdział 1. Nowy paradygmat dla Big Data 19

1.1. Zawartość książki 20
1.2. Skalowanie tradycyjnej bazy danych 21
1.2.1. Skalowanie za pomocą kolejki 22
1.2.2. Skalowanie przez sharding bazy danych 22
1.2.3. Rozpoczynają się problemy z odpornością na błędy 23
1.2.4. Problemy z uszkodzeniem danych 24
1.2.5. Co poszło nie tak? 24
1.2.6. W jaki sposób techniki Big Data mogą pomóc? 24
1.3. NoSQL nie jest panaceum 25
1.4. Pierwsze zasady 25
1.5. Wymagane właściwości systemu Big Data 26
1.5.1. Niezawodność i odporność na błędy 26
1.5.2. Odczytywanie i aktualizowanie z niską latencją 27
1.5.3. Skalowalność 27
1.5.4. Uogólnienie 27
1.5.5. Rozszerzalność 27
1.5.6. Zapytania ad hoc 28
1.5.7. Minimalna konserwacja 28
1.5.8. Debugowalność 28
1.6. Problemy z architekturami w pełni przyrostowymi 29
1.6.1. Złożoność operacyjna 29
1.6.2. Ekstremalna złożoność osiągania spójności ostatecznej 30
1.6.3. Brak odporności na ludzkie błędy 32
1.6.4. Rozwiązanie w pełni przyrostowe w porównaniu z architekturą lambda 32
1.7. Architektura lambda 34
1.7.1. Warstwa przetwarzania wsadowego 36
1.7.2. Warstwa obsługująca 37
1.7.3. Warstwy przetwarzania wsadowego i obsługująca zapewniają niemal wszystkie właściwości 37
1.7.4. Warstwa przetwarzania czasu rzeczywistego 39
1.8. Najnowsze trendy w technologii 41
1.8.1. Procesory nie stają się coraz szybsze 42
1.8.2. Elastyczne chmury 42
1.8.3. Dynamiczny ekosystem open source dla Big Data 42

1.9. Przykładowa aplikacja: SuperWebAnalytics.com 44
1.10. Podsumowanie 44

CZĘŚĆ I. WARSTWA PRZETWARZANIA WSADOWEGO 47

Rozdział 2. Model danych dla Big Data 49

2.1. Właściwości danych 51
2.1.1. Dane są surowe 53
2.1.2. Dane są niemutowalne 56
2.1.3. Dane są wiecznie prawdziwe 59
2.2. Reprezentacja danych za pomocą modelu opartego na faktach 60
2.2.1. Przykładowe fakty i ich właściwości 60
2.2.2. Korzyści ze stosowania modelu opartego na faktach 62
2.3. Schematy graficzne 66
2.3.1. Elementy schematu graficznego 66
2.3.2. Potrzeba zapewnienia egzekwowalności schematu 67
2.4. Kompletny model danych dla aplikacji SuperWebAnalytics.com 68
2.5. Podsumowanie 70

Rozdział 3. Model danych dla Big Data: ilustracja 71

3.1. Dlaczego framework serializacji? 72
3.2. Apache Thrift 72
3.2.1. Węzły 73
3.2.2. Krawędzie 73
3.2.3. Właściwości 74
3.2.4. Połączenie wszystkich elementów w obiekty danych 75
3.2.5. Ewolucja schematu 75
3.3. Ograniczenia frameworku serializacji 76
3.4. Podsumowanie 78

Rozdział 4. Przechowywanie danych w warstwie przetwarzania wsadowego 79

4.1. Wymagania dotyczące przechowywania głównego zbioru danych 80
4.2. Wybór rozwiązania pamięci masowej dla warstwy przetwarzania wsadowego 81
4.2.1. Użycie magazynu danych klucz-wartość dla głównego zbioru danych 82
4.2.2. Rozproszone systemy plików 82
4.3. Sposób działania rozproszonych systemów plików 83
4.4. Przechowywanie głównego zbioru danych z wykorzystaniem rozproszonego systemu plików 85
4.5. Partycjonowanie pionowe 86
4.6. Niskopoziomowy charakter rozproszonych systemów plików 87
4.7. Przechowywanie głównego zbioru danych aplikacji SuperWebAnalytics.com w rozproszonym systemie plików 89
4.8. Podsumowanie 90

Kup ksi k Pole ksi k

Rozdział 5. Przechowywanie danych w warstwie przetwarzania wsadowego: ilustracja 91

5.1. Korzystanie z Hadoop Distributed File System 92
5.1.1. Problem małych plików 93
5.1.2. Dążenie do wyższego poziomu abstrakcji 93
5.2. Przechowywanie danych w warstwie przetwarzania wsadowego z wykorzystaniem biblioteki Pail 94
5.2.1. Podstawowe operacje biblioteki Pail 95
5.2.2. Serializacja i umieszczanie obiektów w wiaderkach 96
5.2.3. Operacje przetwarzania wsadowego z wykorzystaniem biblioteki Pail 98
5.2.4. Partycjonowanie pionowe z wykorzystaniem biblioteki Pail 99
5.2.5. Formaty plików i kompresja biblioteki Pail 100
5.2.6. Podsumowanie zalet biblioteki Pail 101
5.3. Przechowywanie głównego zbioru danych dla aplikacji SuperWebAnalytics.com 102
5.3.1. Ustrukturyzowane wiaderko dla obiektów Thrift 103
5.3.2. Podstawowe wiaderko dla aplikacji SuperWebAnalytics.com 104
5.3.3. Podział wiaderka w celu pionowego partycjonowania zbioru danych 104
5.4. Podsumowanie 107

Rozdział 6. Warstwa przetwarzania wsadowego 109

6.1. Przykłady do rozważenia 110
6.1.1. Liczba odsłon w czasie 110
6.1.2. Inferencja płci 111
6.1.3. Punkty wpływu 111
6.2. Obliczenia w warstwie przetwarzania wsadowego 112
6.3. Porównanie algorytmów ponownego obliczania z algorytmami przyrostowymi 114
6.3.1. Wydajność 116
6.3.2. Odporność na ludzkie błędy 117
6.3.3. Ogólność algorytmów 117
6.3.4. Wybór stylu algorytmu 118
6.4. Skalowalność w warstwie przetwarzania wsadowego 119
6.5. MapReduce: paradygmat dla obliczeń Big Data 119
6.5.1. Skalowalność 121
6.5.2. Odporność na błędy 123
6.5.3. Ogólność MapReduce 123
6.6. Niskopoziomowy charakter MapReduce 125
6.6.1. Wieloetapowe obliczenia są nienaturalne 125
6.6.2. Operacje łączenia są bardzo skomplikowane do ręcznej implementacji 126
6.6.3. Wykonywanie logiczne jest ściśle powiązane z fizycznym 128
6.7. Diagramy potokowe: wyższy poziom sposobu myślenia na temat obliczeń wsadowych 129
6.7.1. Koncepcje diagramów potokowych 129
6.7.2. Wykonywanie diagramów potokowych poprzez MapReduce 134
6.7.3. Agregator łączący 134
6.7.4. Przykłady diagramów potokowych 136
6.8. Podsumowanie 136

Kup książkę Poleć książkę

Rozdział 7. Warstwa przetwarzania wsadowego: ilustracja 139

7.1. Przykład ilustracyjny 140
7.2. Typowe pułapki narzędzi do przetwarzania danych 142
7.2.1. Języki niestandardowe 142
7.2.2. Słabo komponowalne abstrakcje 143
7.3. Wprowadzenie do JCascalog 144
7.3.1. Model danych JCascalog 144
7.3.2. Struktura zapytania JCascalog 145
7.3.3. Kwerendowanie wielu zbiorów danych 147
7.3.4. Grupowanie i agregatory 150
7.3.5. Analiza przykładowego zapytania 150
7.3.6. Niestandardowe operacje predykatów 153
7.4. Kompozycja 158
7.4.1. Łączenie podzapytań 158
7.4.2. Podzapytania tworzone dynamicznie 159
7.4.3. Makra predykatów 162
7.4.4. Makra predykatów tworzone dynamicznie 164
7.5. Podsumowanie 166

Rozdział 8. Przykładowa warstwa przetwarzania wsadowego: architektura i algorytmy 167

8.1. Projekt warstwy przetwarzania wsadowego aplikacji SuperWebAnalytics.com 168
8.1.1. Obsługiwane zapytania 168
8.1.2. Obrazy wsadowe 169
8.2. Przegląd przepływu pracy 172
8.3. Przyjmowanie nowych danych 174
8.4. Normalizacja adresów URL 174
8.5. Normalizacja identyfikatorów użytkowników 175
8.6. Usuwanie zduplikowanych odsłon 180
8.7. Obliczanie obrazów wsadowych 180
8.7.1. Liczba odsłon w czasie 180
8.7.2. Liczba unikatowych użytkowników w czasie 181
8.7.3. Analiza współczynnika odrzuceń 182
8.8. Podsumowanie 183

Rozdział 9. Przykładowa warstwa przetwarzania wsadowego: implementacja 185

9.1. Punkt startowy 186
9.2. Przygotowanie przepływu pracy 187
9.3. Przyjmowanie nowych danych 187
9.4. Normalizacja adresów URL 191
9.5. Normalizacja identyfikatorów użytkowników 192
9.6. Usuwanie zduplikowanych odsłon 197
9.7. Obliczanie obrazów wsadowych 197
9.7.1. Liczba odsłon w czasie 197

9.7.2. Liczba unikatowych użytkowników w czasie 200
9.7.3. Analiza współczynnika odrzuceń 201
9.8. Podsumowanie 204

CZĘŚĆ II. WARSTWA OBSŁUGUJĄCA 205

Rozdział 10. Warstwa obsługująca 207

10.1. Metryki wydajności dla warstwy obsługującej 209
10.2. Rozwiązanie warstwy obsługującej dotyczące problemu wyboru między normalizacją a denormalizacją 211
10.3. Wymagania względem bazy danych warstwy obsługującej 213
10.4. Projektowanie warstwy obsługującej dla aplikacji SuperWebAnalytics.com 215
10.4.1. Liczba odsłon w czasie 215
10.4.2. Liczba użytkowników w czasie 216
10.4.3. Analiza współczynnika odrzuceń 217
10.5. Porównanie z rozwiązaniem w pełni przyrostowym 217
10.5.1. W pełni przyrostowe rozwiązanie problemu liczby unikatowych użytkowników w czasie 218
10.5.2. Porównanie z rozwiązaniem opartym na architekturze lambda 224
10.6. Podsumowanie 224

Rozdział 11. Warstwa obsługująca: ilustracja 227

11.1. Podstawy ElephantDB 228
11.1.1. Tworzenie obrazu w ElephantDB 228
11.1.2. Serwowanie obrazu w ElephantDB 229
11.1.3. Korzystanie z ElephantDB 229
11.2. Budowanie warstwy obsługującej dla aplikacji SuperWebAnalytics.com 231
11.2.1. Liczba odsłon w czasie 231
11.2.2. Liczba unikatowych użytkowników w czasie 234
11.2.3. Analiza współczynnika odrzuceń 235
11.3. Podsumowanie 236

CZĘŚĆ III. WARSTWA PRZETWARZANIA CZASU RZECZYWISTEGO 237

Rozdział 12. Obrazy czasu rzeczywistego 239

12.1. Obliczanie obrazów czasu rzeczywistego 241
12.2. Przechowywanie obrazów czasu rzeczywistego 242
12.2.1. Dokładność ostateczna 243
12.2.2. Ilość stanu przechowywanego w warstwie przetwarzania czasu rzeczywistego 244
12.3. Wyzwania obliczeń przyrostowych 245
12.3.1. Słuszność twierdzenia CAP 245
12.3.2. Kompleksowa interakcja między twierdzeniem CAP a algorytmami przyrostowymi 247
12.4. Porównanie aktualizacji asynchronicznych z synchronicznymi 249
12.5. Wygaszanie obrazów czasu rzeczywistego 250
12.6. Podsumowanie 253

Rozdział 13. Obrazy czasu rzeczywistego: ilustracja 255

13.1. Model danych Cassandry 256
13.2. Korzystanie z bazy danych Cassandra 257
13.2.1. Zaawansowane funkcje Cassandry 259
13.3. Podsumowanie 259

Rozdział 14. Kolejkowanie i przetwarzanie strumieniowe 261

14.1. Kolejkowanie 262
14.1.1. Serwery kolejek pojedynczego konsumenta 263
14.1.2. Kolejki wielu konsumentów 264
14.2. Przetwarzanie strumieniowe 265
14.2.1. Kolejki i procesy robocze 266
14.2.2. Pułapki paradygmatu „kolejki i procesy robocze” 267
14.3. Pojedyncze przetwarzanie strumieniowe wyższego poziomu 268
14.3.1. Model Storm 268
14.3.2. Zapewnianie przetwarzania komunikatów 272
14.4. Warstwa przetwarzania czasu rzeczywistego dla aplikacji SuperWebAnalytics.com 274
14.4.1. Struktura topologii 277
14.5. Podsumowanie 278

Rozdział 15. Kolejkowanie i przetwarzanie strumieniowe: ilustracja 281

15.1. Definiowanie topologii za pomocą Apache Storm 281
15.2. Klastry Apache Storm i wdrażanie topologii 284
15.3. Gwarantowanie przetwarzania komunikatów 286
15.4. Implementacja warstwy przetwarzania czasu rzeczywistego aplikacji SuperWebAnalytics.com dla liczby unikatowych użytkowników w czasie 288
15.5. Podsumowanie 292

Rozdział 16. Mikrowsadowe przetwarzanie strumieniowe 293

16.1. Osiąganie semantyki „dokładnie raz” 294
16.1.1. Ściśle uporządkowane przetwarzanie 294
16.1.2. Mikrowsadowe przetwarzanie strumieniowe 295
16.1.3. Topologie przetwarzania mikrowsadowego 296
16.2. Podstawowe koncepcje mikrowsadowego przetwarzania strumieniowego 299
16.3. Rozszerzanie diagramów potokowych dla przetwarzania mikrowsadowego 300
16.4. Dokończenie warstwy przetwarzania czasu rzeczywistego dla aplikacji SuperWebAnalytics.com 302
16.4.1. Liczba odsłon w czasie 302
16.4.2. Analiza współczynnika odrzuceń 302
16.5. Inne spojrzenie na przykład analizy współczynnika odrzuceń 307
16.6. Podsumowanie 308

Kup ksi k Pole ksi k

Rozdział 17. Mikrowsadowe przetwarzanie strumieniowe: ilustracja 309

17.1. Korzystanie z interfejsu Trident 310

17.2. Dokończenie warstwy przetwarzania czasu rzeczywistego dla aplikacji SuperWebAnalytics.com 313

17.2.1. Liczba odsłon w czasie 314

17.2.2. Analiza współczynnika odrzuceń 316

17.3. W pełni odporne na błędy przetwarzanie mikrowsadowe z utrzymywaniem stanu w pamięci 322

17.4. Podsumowanie 323

Rozdział 18. Tajniki architektury lambda 325

18.1. Definiowanie systemów danych 325

18.2. Warstwa przetwarzania wsadowego i warstwa obsługująca 327

18.2.1. Przyrostowe przetwarzanie wsadowe 328

18.2.2. Pomiar i optymalizacja wykorzystania zasobów przez warstwę przetwarzania wsadowego 335

18.3. Warstwa przetwarzania czasu rzeczywistego 339

18.4. Warstwa zapytań 340

18.5. Podsumowanie 341

Skorowidz 343

Nowy paradygmat dla Big Data

W tym rozdziale omówione zostaną następujące zagadnienia:

- typowe problemy pojawiające się podczas skalowania tradycyjnej bazy danych;
- dlaczego NoSQL nie jest panaceum;
- myślenie o systemach Big Data od pierwszych zasad;
- krajobraz narzędzi Big Data;
- wprowadzenie przykładowej aplikacji SuperWebAnalytics.com.

W ciągu ostatniej dekady znacznie wzrosła ilość tworzonych danych. *Co sekundę* generowanych jest ponad 30 000 gigabajtów danych, a tempo tworzenia danych cały czas rośnie.

Dane, z którymi mamy do czynienia, są zróżnicowane. Użytkownicy tworzą treści, takie jak posty na blogach, tweety, interakcje na portalach społecznościowych oraz zdjęcia. Serwery ciągle rejestrują komunikaty dotyczące przeprowadzanych operacji. Naukowcy tworzą szczegółowe pomiary otaczającego nas świata. Rozległość internetu jako ostatecznego źródła danych jest niemal niepojęta.

Ten zadziwiający wzrost ilości danych znacząco wpłynął na działanie przedsiębiorstw. Tradycyjne systemy baz danych, takie jak relacyjne bazy danych, zostały wykorzystane do granic. W rosnącej liczbie przypadków te systemy załamują się pod naciskiem

Kup ksi k Pole ksi k

„wielkich zbiorów danych" (ang. *Big Data*). Nie powiodło się skalowanie do Big Data tradycyjnych systemów i związanych z nimi technik zarządzania danymi.

W celu sprostania wyzwaniom Big Data opracowany został nowy rodzaj technologii. Wiele z tych nowych technologii zostało zgrupowanych pod pojęciem **NoSQL**. W niektórych aspektach są one bardziej skomplikowane niż tradycyjne bazy danych, a w innych są od nich prostsze. Systemy te można skalować do znacznie większych zbiorów danych, ale użycie wspomnianych technologii rzeczywiście wymaga zasadniczo nowego zestawu technik. Nie są one uniwersalnymi rozwiązaniami.

Wiele z tych systemów Big Data zostało zapoczątkowanych przez firmę Google — w tym rozproszone systemy plików, platforma MapReduce do przetwarzania równoległego oraz rozproszone usługi blokowania. Innym godnym uwagi pionierem w tej dziedzinie była firma Amazon, która stworzyła nowatorski, rozproszony magazyn danych typu klucz-wartość o nazwie Dynamo. W kolejnych latach społeczność open source odpowiedziała niezliczonymi projektami, takimi jak Hadoop, HBase, MongoDB, Cassandra czy RabbitMQ.

Ta książka koncentruje się w podobnym stopniu na złożoności, co na skalowalności. Aby sprostać wyzwaniom związanym z Big Data, przeanalizujemy na nowo systemy danych od podstaw. Dowiesz się, że niektóre z najbardziej podstawowych sposobów zarządzania danymi w tradycyjnych systemach (takie jak systemy zarządzania relacyjnymi bazami danych, ang. *Relational Database Management System* — RDBMS) są zbyt skomplikowane dla systemów Big Data. Prostszym, alternatywnym podejściem jest nowy paradygmat dla Big Data, którym się zajmiemy. Nazwaliśmy to podejście **architekturą lambda** (ang. *Lambda Architecture*).

W tym, zarazem pierwszym, rozdziale zbadamy „problem wielkich zbiorów danych" i zobaczymy, dlaczego potrzebny jest nowy paradygmat dla Big Data. Poznamy niebezpieczeństwa związane z niektórymi tradycyjnymi technikami skalowania i odkryjemy kilka poważnych wad tradycyjnego sposobu budowania systemów danych. Rozpoczynając od podstawowych zasad systemów danych, określimy inny sposób budowania tych systemów, który pozwoli uniknąć złożoności tradycyjnych technik. Zobaczymy, w jaki sposób najnowsze trendy w technologii zachęcają do korzystania z nowych rodzajów systemów, a na koniec przyjrzymy się przykładowemu systemowi Big Data, który będziemy budować w tej książce w celu zilustrowania kluczowych pojęć.

1.1. Zawartość książki

Tę książkę należy traktować przede wszystkim jako podręcznik teoretyczny, koncentrujący się na sposobach podejścia do tworzenia rozwiązań wszystkich problemów związanych z Big Data. Zasady, które poznasz, są prawdziwe niezależnie od stosowanych obecnie narzędzi i możesz je wykorzystać przy rygorystycznym podejmowaniu decyzji w kwestii narzędzi odpowiednich dla danej aplikacji.

Ta książka nie jest przeglądem baz danych, systemów obliczeniowych ani innych powiązanych z nimi technologii. Chociaż w trakcie lektury dowiesz się, jak korzystać z wielu narzędzi, takich jak Hadoop, Cassandra, Storm i Thrift, omawiana książka sama w sobie nie ma na celu prezentacji działania tych narzędzi. Są one raczej środkiem

Kup ksi k Pole ksi k

służącym do poznania podstawowych zasady tworzenia architektury dla solidnych i skalowalnych systemów danych. Zaprezentowanie szczegółowego zestawienia porównawczego tych narzędzi nie zdałoby egzaminu, ponieważ oderwałoby Cię po prostu od nauki podstawowych zasad. Innymi słowy, dowiesz się, jak łowić ryby, a nie tylko, jak używać konkretnej wędki.

W tym duchu przygotowana została struktura tej książki, bazująca na podziale na rozdziały **teoretyczne** i **ilustrujące** konkretne zagadnienia. Możesz przeczytać tylko rozdziały teoretyczne i dzięki temu w pełni zrozumieć, jak budować systemy wielkich zbiorów danych, ale uważamy, że proces mapowania tej teorii na konkretne narzędzia przedstawiony w rozdziałach ilustrujących daje bogatsze, bardziej wszechstronne zrozumienie materiału.

Nie pozwól się jednak zwieść tym nazwom — w rozdziałach teoretycznych wykorzystywanych jest wiele przykładów. Główny przykład w tej książce (aplikacja SuperWebAnalytics.com) jest wykorzystywany zarówno w rozdziałach teoretycznych, jak i ilustrujących. W rozdziałach teoretycznych znajdziesz algorytmy, projekty indeksów i architekturę dla aplikacji SuperWebAnalytics.com. W rozdziałach ilustrujących projekty te zostaną zmapowane na funkcjonujący kod za pomocą konkretnych narzędzi.

1.2. Skalowanie tradycyjnej bazy danych

Zacznijmy naszą eksplorację Big Data od miejsca, w którym zaczyna wielu programistów: od osiągnięcia granic tradycyjnych technologii baz danych.

Załóżmy, że Twój szef prosi Cię, żebyś zbudował prostą aplikację służącą do analityki internetowej. Aplikacja powinna śledzić liczbę odsłon strony dla dowolnego adresu URL wybranego przez klienta. Strona klienta ma pingować serwer WWW aplikacji ze swojego adresu URL za każdym razem, gdy otrzymywana jest informacja o odsłonie. Dodatkowo aplikacja powinna być w stanie w dowolnym momencie dostarczyć zestawienie 100 adresów URL z największą liczbą odsłon.

Zaczynasz od tradycyjnego schematu relacyjnego dla odsłon strony, który wygląda mniej więcej tak, jak przedstawiono na rysunku 1.1. Twój *back-end* składa się z systemu RDBMS z tabelą tego schematu oraz serwera WWW. Za każdym razem, gdy ktoś ładuje stronę śledzoną przez Twoją aplikację, strona ta pinguje Twój serwer WWW, informując o odsłonie, a serwer zwiększa wartość odpowiedniego wiersza w bazie danych.

Nazwa kolumny	Typ
id	integer
id_użytkownika	integer
url	varchar(255)
liczba odsłon strony	bigint

Rysunek 1.1. Schemat relacyjny prostej aplikacji analitycznej

Zobaczmy, jakie problemy będą się pojawiać wraz z rozwijaniem tej aplikacji. Jak sam się przekonasz, napotkamy problemy zarówno ze skalowalnością, jak i złożonością.

Kup ksi k Pole ksi k

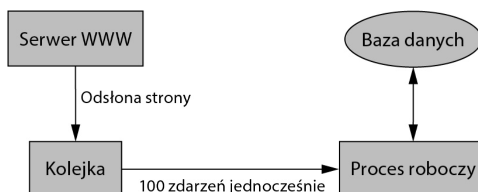
1.2.1. Skalowanie za pomocą kolejki

Przygotowany przez Ciebie produkt do analityki internetowej okazał się ogromnym sukcesem, a ruch kierowany do aplikacji rośnie jak niekontrolowany pożar. Twoja firma wydaje wielkie przyjęcie, ale w środku uroczystości zaczynasz otrzymywać mnóstwo e-maili z systemu monitorującego. We wszystkich e-mailach znajduje się ta sama informacja: „Błąd limitu czasu podczas wstawiania do bazy danych".

Przeglądasz wpisy dziennika i odkrywasz oczywisty problem. Baza danych nie może sprostać obciążeniu, przekraczany jest więc limit czasu dla żądań zapisania zwiększenia liczby odsłon stron.

Trzeba coś zrobić, aby rozwiązać ten problem, i należy to zrobić szybko. Zdajesz sobie sprawę, że marnotrawstwem jest wykonywanie za każdym razem tylko jednego zwiększenia liczby odsłon w bazie danych. Bardziej efektywne może być zgromadzenie wielu inkrementacji w jednym żądaniu. Aby to umożliwić, modyfikujesz architekturę *back-endu*.

Bezpośrednią komunikację serwera WWW z bazą danych zastępujesz kolejką między serwerem WWW a bazą danych. Gdy teraz otrzymasz informację o nowej odsłonie, zdarzenie zostanie dodane do kolejki. Następnie tworzysz proces roboczy, który odczytuje za jednym razem 100 zdarzeń z kolejki i łączy je w jedną aktualizację bazy danych. Przedstawiono to na rysunku 1.2.



Rysunek 1.2. Grupowanie aktualizacji za pomocą kolejki i procesu roboczego

Ten schemat dobrze się sprawdza i rozwiązuje problemy związane z przekraczaniem limitu czasu. Otrzymujesz nawet pewien bonus. Polega on na tym, że jeśli baza danych znowu zostanie przeciążona, kolejka po prostu będzie robić się coraz większa i nie będziesz miał do czynienia z przekroczeniem limitu czasu dla serwera WWW, co mogłoby potencjalnie powodować utratę danych.

1.2.2. Skalowanie przez sharding bazy danych

Niestety dodanie kolejki i przeprowadzanie zbiorczych aktualizacji okazuje się tylko plastrem opatrunkowym dla problemu skalowania. Twoja aplikacja nadal zyskuje na popularności, a baza danych ponownie jest przeciążana. Twój proces roboczy nie może nadążyć z przeprowadzaniem operacji zapisu, zatem próbujesz dodać więcej procesów roboczych, aby zapewnić równoległe aktualizacje. Niestety to nie pomaga. Baza danych jest najwyraźniej wąskim gardłem.

Szukasz w wyszukiwarce Google sposobów skalowania relacyjnej bazy danych mocno obciążonej operacjami zapisu. Dowiadujesz się, że najlepszym sposobem jest użycie wielu serwerów baz danych i rozłożenie tabeli na wszystkie te serwery. Każdy serwer będzie posiadał podzbiór danych dla tabeli. Nazywa się to **partycjonowaniem poziomym** lub **shardingiem**. Ta technika pozwala rozłożyć obciążenie związane z operacjami zapisu na wiele maszyn.

Kup ksi k Pole ksi k

Używana przez Ciebie technika *shardingu* polega na wybieraniu *sharda* dla każdego klucza za pomocą skrótu klucza zmodowanego przez liczbę shardów. Mapowanie kluczy na *shardy* z wykorzystaniem funkcji skrótu powoduje, że klucze są równomiernie dystrybuowane pomiędzy *shardami*. Piszesz skrypt do mapowania na wszystkie wiersze w Twojej pojedynczej instancji bazy danych i dzielisz dane na cztery *shardy*. Wprowadzenie tego rozwiązania wymaga nieco czasu, tak więc do momentu zakończenia operacji wyłączasz proces roboczy zwiększający liczniki odsłon. W przeciwnym razie straciłbyś inkrementacje w okresie przejściowym.

Na koniec cały kod aplikacji musi „wiedzieć", jak znaleźć *shard* dla każdego klucza. Opakowujesz więc kod do obsługi bazy danych za pomocą biblioteki, która odczytuje liczbę *shardów* z pliku konfiguracyjnego, a potem przeorganizowujesz cały kod aplikacji. Musisz zmodyfikować zapytanie o 100 najpopularniejszych adresów URL, aby pobierać 100 adresów z każdego *sharda* i na tej podstawie generować globalną listę 100 najpopularniejszych adresów URL.

Wraz z coraz większym wzrostem popularności aplikacji musisz cały czas dzielić bazę danych na coraz większą liczbę *shardów*, aby nadążyć za rosnącym obciążeniem związanym z operacjami zapisu. Za każdym razem staje się to coraz bardziej bolesne, ponieważ jest bardzo wiele pracy do skoordynowania. Nie możesz też po prostu uruchomić jednego skryptu, który przeprowadzi *resharding*, ponieważ byłoby to zbyt wolne. Wszystkie czynności związane z *reshardingiem* musisz wykonywać równolegle i zarządzać wieloma aktywnymi skryptami roboczymi naraz. Jeśli zapomnisz zaktualizować kod aplikacji nową liczbą *shardów*, spowoduje to, że wiele inkrementacji zostanie zapisanych w niewłaściwych *shardach*. Musisz więc napisać jednorazowy skrypt, który sprawdzi dane i przeniesie te, które zostały umieszczone w niewłaściwym miejscu.

1.2.3. Rozpoczynają się problemy z odpornością na błędy

W końcu masz tyle *shardów*, że nierzadkim przypadkiem jest awaria dysku w jednej z maszyn bazodanowych. Gdy maszyna jest wyłączona, ta część danych jest niedostępna. Aby temu zaradzić, robisz kilka rzeczy:

- Aktualizujesz swój system „kolejka-proces roboczy" w taki sposób, aby inkrementacje dla niedostępnych *shardów* były umieszczane w osobnej kolejce, którą próbujesz opróżniać raz na pięć minut, jako „oczekujące".
- Wykorzystujesz możliwości replikacji bazy danych, aby dla każdego *sharda* nadrzędnego (ang. *master shard*) dodać *shard* podrzędny (ang. *slave shard*). W ten sposób w wypadku awarii *sharda* nadrzędnego będziesz miał kopię zapasową. W *shardzie* podrzędnym nie będą przeprowadzane operacje zapisu, ale przynajmniej klienci będą mogli nadal przeglądać statystyki w aplikacji.

Myślisz sobie: „Na początku zajmowałem się tworzeniem nowych funkcji dla klientów. Teraz wydaje się, że cały swój czas poświęcam jedynie na rozwiązywanie problemów z operacjami odczytu i zapisu danych".

Kup ksi k Pole ksi k

1.2.4. Problemy z uszkodzeniem danych

Podczas pracy nad kodem „kolejka-proces roboczy” przypadkowo wdrażasz w środowisku produkcyjnym błąd, który dla każdego adresu URL zwiększa liczbę odsłon strony o dwie zamiast o jedną. Zauważasz to 24 godziny później, ale szkoda już została wyrządzona. Twoje cotygodniowe kopie zapasowe nie pomagają, ponieważ nie ma możliwości dowiedzieć się, które dane zostały uszkodzone. Włożyłeś wiele pracy w to, aby uczynić swój system skalowalnym i odpornym na awarie maszyn, ale nie jest on odporny na błędy popełniane przez człowieka. A jeśli w zakresie oprogramowania istnieje jakaś gwarancja, to taka, że błędy nieuchronnie przenikną do środowiska produkcyjnego, bez względu na to, jak bardzo będziesz się starał temu zapobiec.

1.2.5. Co poszło nie tak?

Wraz z ewolucją prostej aplikacji do analizy internetowej system stawał się coraz bardziej złożony: kolejki, *shardy*, replikacje, skrypty *reshardingujące* itd. Rozwijanie aplikacji opierającej się na danych wymaga znacznie więcej niż tylko znajomości schematu bazy danych. Twój kod musi „wiedzieć”, jak komunikować się z właściwymi *shardami*, a jeśli popełnisz błąd, nic nie zapobiegnie sytuacjom odczytu z niewłaściwego *sharda* lub zapisu na nim.

Jednym z problemów jest to, że baza danych nie jest świadoma swojego rozproszonego charakteru, nie może więc Ci pomóc uporać się z *shardami*, replikacjami i rozproszonymi zapytaniami. Cała ta złożoność została na Tobie wymuszona zarówno w kwestii obsługi bazy danych, jak i rozwijania kodu aplikacji.

Ale największym problemem jest to, że system nie został zaprojektowany z uwzględnieniem ludzkich błędów. Wręcz przeciwnie: system staje się coraz bardziej złożony, co zwiększa prawdopodobieństwo popełniania błędów. Błędy w oprogramowaniu są nieuniknione, jeśli więc nie uwzględnisz ich w projekcie, możesz równie dobrze pisać skrypty, które losowo uszkadzają dane. Wykonywanie kopii zapasowych nie wystarczy. System musi być dokładnie przemyślany pod względem ograniczenia szkód, jakie może spowodować ludzka pomyłka. Odporność na błędy człowieka nie jest opcjonalna. Jest zasadniczą kwestią, zwłaszcza, gdy wielkie zbiory danych w tak dużym stopniu zwiększają złożoność budowania aplikacji.

1.2.6. W jaki sposób techniki Big Data mogą pomóc?

Techniki Big Data, których się nauczysz, rozwiązują te problemy skalowalności i złożoności w radykalny sposób. Przede wszystkim bazy danych oraz systemy obliczeniowe używane dla wielkich zbiorów danych są świadome swojego rozproszonego charakteru. Dlatego elementy takie jak *sharding* i replikacje są obsługiwane za Ciebie. Nigdy nie znajdziesz się w sytuacji, w której przypadkowo będziesz kwerendować niewłaściwy *shard*, ponieważ ta logika jest zinternalizowana w bazie danych. Kiedy dojdzie do skalowania, będziesz po prostu dodawać węzły, a systemy automatycznie zrównoważą obciążenie, rozkładając je na nowe węzły.

Kolejna podstawowa technika, którą poznasz, sprawia, żeby dane były niemutowalne. Zamiast przechowywania liczby odsłon stron jako podstawowego zbioru danych, stale

Kup ksi k Pole ksi k

podlegającego mutacji wraz z pojawianiem się nowych odsłon, przechowywane są surowe informacje o odsłonach stron. Te surowe informacje nigdy nie są modyfikowane. Kiedy więc się pomylisz, możesz zapisać złe dane, ale przynajmniej nie zniszczysz dobrych danych. Jest to o wiele silniejsza gwarancja odporności na ludzkie błędy niż w tradycyjnym systemie opartym na mutacji. W tradycyjnych bazach danych należy ostrożnie używać niemutowalnych danych, gdyż takie zbiory danych będą szybko się powiększać. Ponieważ jednak techniki Big Data pozwalają skalować do tak dużej ilości danych, możesz inaczej projektować systemy.

1.3. NoSQL nie jest panaceum

W ciągu ostatnich 10 lat pojawiła się ogromna liczba innowacji w zakresie skalowalnych systemów danych. Należą do nich systemy obliczeń wielkoskalowych, takie jak Hadoop, oraz baz danych, takie jak Cassandra i Riak. Te systemy mogą obsługiwać bardzo duże ilości danych, ale wymuszają też poważne kompromisy.

Hadoop może na przykład czynić paralelnymi wielkoskalowe obliczenia wsadowe przeprowadzane na bardzo dużej ilości danych, ale obliczenia te mają spore opóźnienia. Nie należy używać Hadoopa w sytuacjach, w których wymagane są wyniki z niską latencją.

Bazy danych NoSQL, takie jak Cassandra, osiągają swoją skalowalność, oferując znacznie bardziej ograniczony model danych niż ten, do którego przywykłeś w wypadku SQL. Dostosowanie aplikacji do tych ograniczonych modeli danych może być bardzo złożone. A ponieważ te bazy danych są mutowalne, nie zapewniają odporności na ludzkie błędy.

Te narzędzia same w sobie nie są panaceum. Jednak inteligentnie ze sobą połączone mogą tworzyć skalowalne systemy do obsługi dowolnych danych, charakteryzujące się odpornością na ludzkie błędy i minimalną złożonością. To jest właśnie architektura lambda, którą poznasz w tej książce.

1.4. Pierwsze zasady

Aby dowiedzieć się, jak prawidłowo budować systemy danych, należy cofnąć się do pierwszych zasad. Co robią systemy danych na najbardziej podstawowym poziomie?

Zacznijmy od intuicyjnej definicji: **system danych odpowiada na pytania na podstawie informacji pozyskanych do momentu zadania pytania**. Tak więc profil na portalu społecznościowym odpowiada na pytania typu: „Jakie jest imię i nazwisko danej osoby?” oraz „Ilu znajomych ma dana osoba?”. Strona internetowa rachunku bankowego odpowiada na pytania takie jak: „Jakie jest moje bieżące saldo?” oraz „Jakie transakcje były ostatnio przeprowadzane na moim koncie?”.

Systemy danych nie tylko zapamiętują i zwracają informacje. Łączą ze sobą części i kawałki informacji, aby wygenerować odpowiedzi. Saldo rachunku bankowego jest na przykład oparte na połączeniu informacji o wszystkich transakcjach przeprowadzonych na danym koncie.

Kup ksi k Pole ksi k

Kolejnym ważnym spostrzeżeniem jest to, że nie wszystkie części informacji są sobie równe. Niektóre informacje pochodzą z innych fragmentów informacji. Saldo rachunku bankowego opiera się na historii transakcji. Liczba znajomych jest obliczana na podstawie listy znajomych, a lista znajomych jest z kolei wypadkową wszystkich operacji dodawania i usuwania znajomych przez użytkownika na jego profilu.

Kiedy spróbujesz prześledzić pochodzenie jakiejś informacji, w końcu dotrzesz do takiej, która nie pochodzi od żadnej innej. Jest to najbardziej surowa informacja, jaką masz: taka, którą uznajesz za prawdziwą tylko dlatego, że istnieje. Nazwijmy tę informację **danymi**.

Możesz mieć inną koncepcję dotyczącą znaczenia słowa **dane**. Jest ono często stosowane zamiennie ze słowem **informacja**. W pozostałej części tej książki będziemy jednak używać słowa **dane**, odwołując się do tej wyjątkowej informacji, od której pochodzą wszystkie pozostałe.

O ile system danych odpowiada na pytania na podstawie danych z przeszłości, o tyle system danych najbardziej ogólnego zastosowania odpowiada na pytania na podstawie *całego* zbioru danych. Dlatego definicja najbardziej ogólnego zastosowania, jaką możemy sformułować dla systemu danych, będzie następująca:

```
zapytanie = funkcja(wszystkie dane)
```

Wszystko, co kiedykolwiek mógłbyś zechcieć zrobić z danymi, może być wyrażone w postaci funkcji przyjmującej jako informacje wejściowe wszystkie posiadane przez Ciebie dane. Zapamiętaj to równanie, ponieważ stanowi ono sedno wszystkiego, czego się nauczysz. Będziemy odwoływać się do tego równania wielokrotnie.

Architektura lambda zapewnia mające ogólne zastosowanie podejście do implementacji dowolnej funkcji na dowolnym zbiorze danych i zwracanie przez tę funkcję wyników z niską latencją. Nie oznacza to, że będziesz stosować te same technologie za każdym razem, gdy będziesz implementować system danych. Konkretne, zastosowane technologie mogą się zmieniać w zależności od wymagań. Architektura lambda definiuje jednak spójne podejście do wyboru tych technologii i połączenia ich ze sobą w celu spełnienia istniejących wymagań.

Przejdźmy teraz do omówienia właściwości, którymi musi się charakteryzować system danych.

1.5. Wymagane właściwości systemu Big Data

Właściwości, do których należy dążyć w systemach Big Data, są związane w równie dużym stopniu ze złożonością, co ze skalowalnością. System Big Data musi być nie tylko wydajny oraz efektywny w kwestii wykorzystywania zasobów, ale powinien być także łatwy do zrozumienia. Przyjrzyjmy się po kolei wszystkim właściwościom.

1.5.1. Niezawodność i odporność na błędy

Budowanie systemów, „które robią to, co trzeba", jest trudne w obliczu wyzwań systemów rozproszonych. Systemy muszą zachowywać się właściwie bez względu na losowe awarie maszyn, złożoną semantykę spójności w rozproszonych bazach danych, zdupli-

Kup ksi k Pole ksi k

kowane dane, współbieżność itd. Wyzwania te utrudniają nawet zrozumienie tego, co system robi. Uzyskanie niezawodności systemu Big Data po części polega na unikaniu tych zawiłości, tak aby można było łatwo zrozumieć działanie systemu.

Jak omówiono wcześniej, imperatywem dla systemów jest odporność na ludzkie błędy. Jest to często pomijana właściwość systemów, której nie będziemy ignorować. W systemie produkcyjnym nieuniknione jest, że ktoś kiedyś popełni błąd, na przykład poprzez wdrożenie nieprawidłowego kodu uszkadzającego wartości w bazie danych. Jeśli wbudujesz w rdzeń systemu Big Data niemutowalność i zasadę ponownego przeliczania, system będzie z natury odporny na błędy człowieka w wyniku zapewnienia jasnego i prostego mechanizmu odzyskiwania. Zostanie to opisane szczegółowo w rozdziałach od 2. do 7.

1.5.2. Odczytywanie i aktualizowanie z niską latencją

Zdecydowana większość aplikacji wymaga, aby operacje odczytu charakteryzowały się bardzo niskim opóźnieniem, zwykle w przedziale od kilku do kilkuset milisekund. Z drugiej strony wymagania dotyczące opóźnień aktualizacji bardzo się różnią między aplikacjami. Niektóre aplikacje wymagają natychmiastowej propagacji aktualizacji, ale w innych aplikacjach dopuszczalna jest latencja rzędu kilku godzin. Niezależnie od tego powinieneś być w stanie osiągnąć niskie opóźnienia aktualizacji, *gdy będziesz ich potrzebować* w swoich systemach Big Data. Co ważniejsze, powinieneś móc osiągnąć niskie opóźnienia odczytów i aktualizacji bez zagrażania niezawodności systemu. O tym, jak osiągnąć niskie opóźnienia aktualizacji, dowiesz się podczas omawiania warstwy przetwarzania czasu rzeczywistego w rozdziale 12.

1.5.3. Skalowalność

Skalowalność to zdolność do utrzymywania wydajności w obliczu rosnącej ilości danych lub wzrastającego obciążenia — osiąga się ją poprzez dodawanie zasobów do systemu. Architektura lambda jest skalowalna poziomo we wszystkich warstwach stosu systemowego: skalowanie odbywa się poprzez dodawanie kolejnych maszyn.

1.5.4. Uogólnienie

Ogólny system może obsługiwać szeroką gamę aplikacji. Ta książka nie byłaby w rzeczywistości zbyt użyteczna, jeśli nie uwzględniałaby systemów dla dużego zakresu zastosowań! Ponieważ architektura lambda oparta jest na funkcji wszystkich danych, można uogólnić ją do wszystkich aplikacji, bez względu na to, czy będą to systemy zarządzania finansami, analizy mediów społecznościowych, aplikacje naukowe, serwisy społecznościowe, czy cokolwiek innego.

1.5.5. Rozszerzalność

Nie chciałbyś być zmuszony do wymyślania koła na nowo za każdym razem, gdy dodajesz jakąś powiązaną funkcję lub wprowadzasz zmiany w sposobie działania systemu. Systemy rozszerzalne pozwalają na dodawanie funkcjonalności przy minimalnym koszcie programistycznym.

Kup ksi k Pole ksi k

Często nowa funkcja lub zmiana w istniejącej funkcji wymaga migracji starych danych do nowego formatu. Uzyskanie rozszerzalności systemu polega po części na ułatwieniu przeprowadzania migracji na dużą skalę. Podstawą podejścia, którego się nauczysz, jest możliwość szybkiego i łatwego przeprowadzania wielkich migracji.

1.5.6. Zapytania ad hoc

Możliwość wykonywania zapytań *ad hoc* o dane jest niezwykle istotna. Prawie każdy duży zbiór danych zawiera jakąś niespodziewaną wartość. Możliwość dowolnej eksploatacji jakiegoś zbioru danych pozwala na optymalizacje biznesowe i tworzenie nowych aplikacji. Ostatecznie nie będziesz w stanie odkryć ciekawych zastosowań dla swoich danych, jeśli nie będziesz mógł zadawać dowolnych pytań o te dane. O tym, jak wykonywać zapytania *ad hoc*, dowiesz się w rozdziałach 6. i 7. podczas omawiania przetwarzania wsadowego.

1.5.7. Minimalna konserwacja

Konserwacja jest podatkiem nałożonym na deweloperów. Jest to praca wymagana do utrzymania płynnego działania systemu. Konserwacja obejmuje przewidywanie, kiedy dodawać maszyny w celu skalowania, utrzymywanie działających procesów oraz debugowanie wszelkich problemów występujących w środowisku produkcyjnym.

Ważnym elementem związanym z minimalizowaniem czynności konserwacyjnych jest wybór takich elementów, które mają możliwie najmniejszą **złożoność implementacyjną**. Najlepiej jest polegać na komponentach mających proste mechanizmy bazowe. W szczególności rozproszone bazy danych mają zwykle bardzo skomplikowane mechanizmy wewnętrzne. Im bardziej złożony system, tym większe prawdopodobieństwo, że coś pójdzie nie tak oraz tym większa wymagana wiedza o systemie w celu jego zdebugowania i dostrojenia.

Złożoność implementacji ogranicza się, polegając na prostych algorytmach i komponentach. Sztuczką wykorzystywaną w architekturze lambda jest przesunięcie złożoności z komponentów rdzenia na te elementy systemu, w których dane wyjściowe można porzucić po kilku godzinach. Najbardziej złożone z wykorzystywanych komponentów, takie jak rozproszone bazy danych odczytu/zapisu, znajdują się w tej warstwie, w której dane wyjściowe są ostatecznie porzucane. Omówimy tę technikę szczegółowo podczas opisywania warstwy przetwarzania czasu rzeczywistego w rozdziale 12.

1.5.8. Debugowalność

System Big Data musi zapewnić informacje niezbędne do debugowania systemu, gdy coś pójdzie nie tak. Kluczowa jest możliwość śledzenia dla każdej wartości w systemie tego, co dokładnie spowodowało, że ta wartość jest właśnie taka.

W architekturze lambda „debugowalność” osiąga się dzięki funkcjonalnej naturze warstwy przetwarzania wsadowego oraz preferowaniu użycia algorytmów ponownego przeliczania, kiedy tylko jest to możliwe.

Kup ksi k Pole ksi k

Osiągnięcie wszystkich tych właściwości w jednym systemie może wydawać się zniechęcającym wyzwaniem. Jeśli jednak rozpoczniemy od pierwszych zasad, tak jak czyni to architektura lambda, właściwości te wyłonią się w sposób naturalny z powstałego projektu systemu.

Zanim zagłębimy się w architekturę lambda, rzućmy okiem na bardziej tradycyjne architektury (charakteryzujące się obliczeniami przyrostowymi) i zobaczmy, dlaczego nie są one w stanie zapewnić wielu z tych właściwości.

1.6. Problemy z architekturami w pełni przyrostowymi

Na najwyższym poziomie tradycyjne architektury wyglądają tak, jak zostało to pokazane na rysunku 1.3. Architektury te charakteryzuje stosowanie baz danych odczytu/zapisu i utrzymywanie w nich stanu w sposób przyrostowy wraz z pojawianiem się nowych danych.

Rysunek 1.3. Architektura w pełni przyrostowa

Podejście przyrostowe do zliczania na przykład odsłon stron będzie polegało na przetwarzaniu nowej odsłony strony poprzez dodanie wartości jeden do licznika dla jej adresu URL. Ta charakterystyka architektur jest o wiele bardziej fundamentalna niż tylko porównanie relacyjności z nierelacyjnością — w rzeczywistości ogromna większość wdrożeń zarówno relacyjnych, jak i nierelacyjnych baz danych odbywa się z wykorzystaniem w pełni przyrostowych architektur. Tak było przez wiele dziesięcioleci.

Warto podkreślić, że w pełni przyrostowe architektury są tak powszechne, że wiele osób nie zdaje sobie sprawy, że można zapobiec związanym z nimi problemom przez zastosowanie innej architektury. Są to świetne przykłady **złożoności znajomej**, czyli takiej, która jest tak mocno zakorzeniona, że nawet nie próbujemy znaleźć sposobu na jej uniknięcie.

Problemy z w pełni przyrostowymi architekturami są znaczące. Zaczniemy nasze badanie tego tematu od przyjrzenia się ogólnym złożonościom generowanym przez dowolną, w pełni przyrostową architekturę. Następnie przeanalizujemy dwa odmienne rozwiązania tego samego problemu: jedno wykorzystujące możliwie najlepszą architekturę w pełni przyrostową oraz drugie wykorzystujące architekturę lambda. Zobaczysz, że w pełni przyrostowa wersja jest znacznie gorsza pod każdym względem.

1.6.1. Złożoność operacyjna

Istnieje wiele złożoności związanych z w pełni przyrostowymi architekturami, wywołujących trudności w operowaniu infrastrukturą produkcyjną. Skoncentrujemy się na jednej z nich, czyli na konieczności przeprowadzania kompaktowania online przez bazy danych odczytu/zapisu, oraz na tym, co trzeba zrobić, aby wszystko działało płynnie.

W wypadku bazy danych odczytu/zapisu, gdy indeks dysku jest przyrostowo dodawany i modyfikowany, części tego indeksu są niewykorzystywane. Te niewykorzystane części zajmują miejsce, które ostatecznie musi zostać odzyskane, aby zapobiec

zapełnieniu dysku. Odzyskiwanie przestrzeni dysku od razu, gdy staje się ona niewykorzystywana, jest zbyt kosztowne, więc od czasu do czasu przestrzeń jest odzyskiwana hurtowo w procesie zwanym **kompaktowaniem**.

Kompaktowanie to intensywna operacja. Podczas kompaktowania serwer znacznie zwiększa zapotrzebowanie na wykorzystanie CPU oraz dysków, co w istotny sposób obniża w tym czasie wydajność danej maszyny. Bazy danych takie jak HBase i Cassandra są dobrze znane z tego, że wymagają starannych konfiguracji i zarządzania, aby uniknąć problemów lub zawieszania się serwera podczas kompaktowania. Utrata wydajności w trakcie kompaktowania to złożoność, która może nawet spowodować kaskadowe awarie — jeśli zbyt wiele maszyn kompaktuje w tej samej chwili, obsługiwane przez nich obciążenie będzie musiało zostać obsłużone przez inne maszyny w klastrze. Może to potencjalnie przeciążyć resztę klastra, powodując całkowitą awarię. Widzieliśmy wiele razy, jak przydarza się ten rodzaj awarii.

Aby prawidłowo zarządzać kompaktowaniem, należy zaplanować kompaktowania na każdym węźle w taki sposób, aby nie przeprowadzało ich jednocześnie zbyt wiele węzłów. Musisz mieć świadomość tego, jak długo trwa kompaktowanie (oraz jak bardzo może być zmienny czas jego trwania), aby uniknąć sytuacji, w której wykonuje je więcej węzłów, niż zamierzałeś. Musisz upewnić się, że masz wystarczająco dużą pojemność dysku na węzłach, aby utrzymać je między kompaktowaniami. Ponadto musisz się upewnić, że masz wystarczającą pojemność w klastrze, aby nie został on przeciążony, gdy podczas kompaktowań zostaną utracone zasoby.

Wszystkim tym może zarządzać kompetentny personel operacyjny, ale uważamy, że najlepszym sposobem radzenia sobie z każdego rodzaju złożonością jest pozbycie się jej całkowicie. Im mniej masz trybów awarii w systemie, tym mniejsze prawdopodobieństwo, że doświadczysz nieoczekiwanych przestojów. Konieczność radzenia sobie z kompaktowaniem online jest złożonością związaną z w pełni przyrostowymi architekturami, ale w architekturze lambda podstawowe bazy danych nie wymagają kompaktowania online.

1.6.2. Ekstremalna złożoność osiągania spójności ostatecznej

Kolejna złożoność architektur przyrostowych pojawia się, gdy próbujemy zapewnić wysoką dostępność systemów. Wysoce dostępne systemy umożliwiają wykonywanie zapytań i aktualizacji nawet w wypadku awarii maszyny lub częściowej awarii sieci.

Okazuje się, że osiągnięcie wysokiej dostępności konkuruje bezpośrednio z inną ważną właściwością, zwaną **spójnością**. Spójny system zwraca wyniki, które uwzględniają wszystkie poprzednie operacje zapisu. Twierdzenie CAP (ang. *Consistency*, *Availability*, *Partition tolerance* — spójność, dostępność, odporność na podziały) wykazało, że niemożliwe jest osiągnięcie w tym samym systemie jednocześnie wysokiej dostępności i spójności w wypadku występowania podziałów sieci (ang. *network partitions*). Dlatego wysoce dostępny system zwraca czasem przestarzałe wyniki w trakcie występowania stanu podziału sieci.

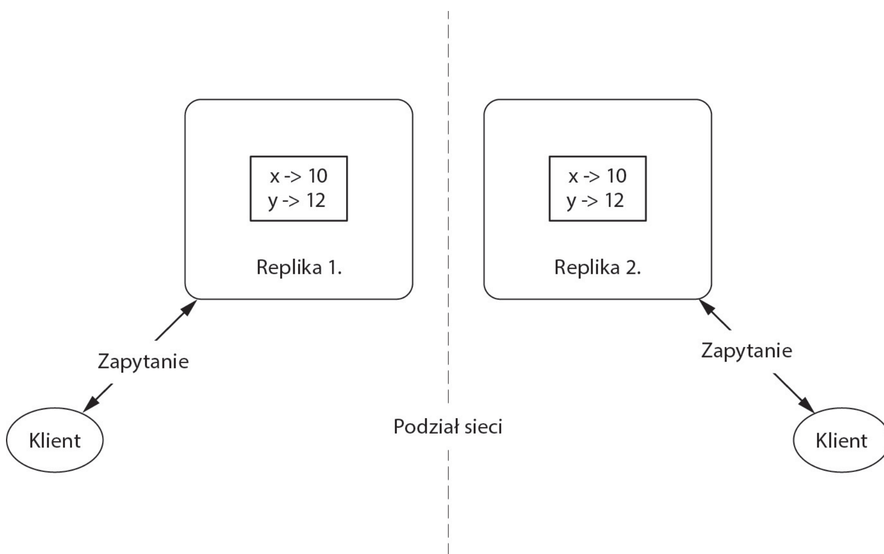
Twierdzenie CAP zostanie omówione szczegółowo w rozdziale 12. Tu chcemy skupić się na tym, w jaki sposób niemożność utrzymania przez cały czas pełnej spójności i wysokiej dostępności wpływa na zdolność do budowania systemów. Okazuje się, że

Kup ksi k Pole ksi k

jeśli wymagania biznesowe przedkładają wysoką dostępność nad pełną spójność, będziesz musiał poradzić sobie z wysokim poziomem złożoności.

Aby wysoce dostępny system powracał do stanu spójności po usunięciu podziału sieci (co jest znane jako **spójność ostateczna**, ang. *eventual consistency*), wymagane jest duże wsparcie ze strony aplikacji. Weźmy na przykład podstawowy przypadek użycia, jakim jest utrzymywanie zliczania w bazie danych. Oczywistym sposobem rozwiązania tej kwestii jest zapisanie w bazie danych pewnej liczby i zwiększanie jej wartości za każdym razem, gdy otrzymywane jest zdarzenie wymagające zwiększenia wskazania licznika. Możesz być zaskoczony tym, że jeśli przyjmiesz takie podejście, będziesz doświadczał masowej utraty danych w trakcie występowania stanu podziału sieci.

Wynika to ze sposobu, w jaki rozproszone bazy danych osiągają wysoką dostępność — poprzez utrzymywanie wielu replik wszystkich przechowywanych informacji. Kiedy przechowujesz wiele kopii tej samej informacji, pozostaje ona dostępna nawet w wypadku awarii maszyny lub wystąpienia podziału sieci, tak jak pokazano na rysunku 1.4. Podczas podziału sieci klienty systemu, dla którego wybrano wysoką dostępność, aktualizują jakiekolwiek dostępne dla nich repliki. To powoduje, że repliki zaczynają się różnić i odbierać odmienne zestawy aktualizacji. Dopiero po usunięciu podziału sieci repliki mogą zostać scalone i otrzymać wspólną wartość.



Rysunek 1.4. Wykorzystanie replikacji w celu zwiększenia dostępności

Załóżmy, że gdy rozpoczyna się podział sieci, masz dwie repliki z liczbą 10,. Przyjmijmy, że pierwsza replika otrzymuje dwa zwiększenia wartości, a druga replika otrzymuje jedno zwiększenie wartości. Jaka powinna być łączna wartość, kiedy przyjdzie czas scalenia tych replik z wartościami 12 i 11? Mimo że prawidłowa odpowiedź to 13, nie można tego stwierdzić, patrząc po prostu na liczby 12 i 11. Repliki mogły się rozejść przy wartości 11 (wówczas odpowiedzią będzie 12) lub mogły się rozejść przy wartości 0 (wówczas odpowiedzią będzie 23).

Kup ksi k Pole ksi k

Aby przeprowadzać prawidłowe zliczanie w wypadku wysokiej dostępności, nie wystarczy tylko przechowywać wartość licznika. Potrzebna jest struktura danych, która jest podatna na scalanie, gdy wartości różnią się od siebie. Należy także zaimplementować kod, który będzie naprawiać wartości po usunięciu podziałów sieci. To niesamowicie wysoki poziom złożoności, z jakim trzeba sobie poradzić, aby utrzymać jedynie zwykłe zliczanie.

Zasadniczo obsługa spójności ostatecznej w przyrostowych systemach o wysokiej dostępności jest nieintuicyjna i podatna na błędy. Jest to wrodzona złożoność w wypadku wysoce dostępnych, w pełni przyrostowych systemów. Zobaczysz później, jak architektura lambda organizuje się sama w inny sposób, który znacznie zmniejsza obciążenia związane z osiąganiem wysokiej dostępności ostatecznie spójnych systemów.

1.6.3. Brak odporności na ludzkie błędy

Ostatnim problemem z w pełni przyrostowymi architekturami, na który chcemy zwrócić uwagę, jest ich nieodłączny brak odporności na ludzkie błędy. System przyrostowy stale modyfikuje stan, który utrzymuje w bazie danych, co oznacza, że błąd może również zmodyfikować stan w bazie danych. Ponieważ błędy są nieuniknione, baza danych we w pełni przyrostowej architekturze na pewno zostanie uszkodzona.

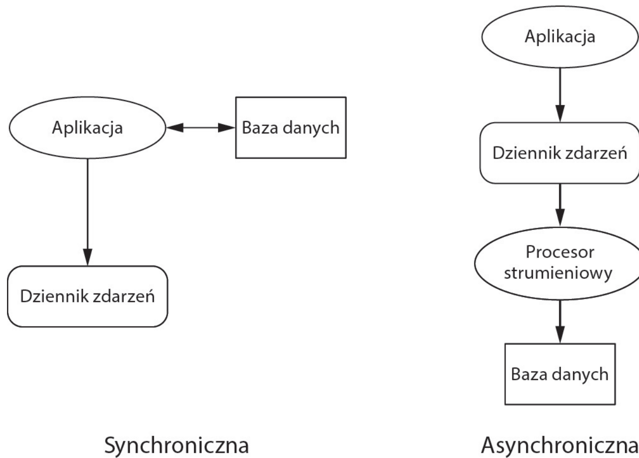
Należy zauważyć, że we w pełni przyrostowych architekturach jest to jedna z niewielu złożoności, które mogą zostać rozwiązane bez konieczności przemyślenia od nowa całej architektury. Rozważmy dwie architektury przedstawione na rysunku 1.5: synchroniczną, w której aplikacja wykonuje aktualizacje bezpośrednio w bazie danych, oraz asynchroniczną, w której zdarzenia są kolejkowane przed aktualizacją bazy danych w ramach procesu działającego w tle. W obu sytuacjach każde zdarzenie jest na stałe rejestrowane w magazynie danych zdarzeń. Dzięki przechowywaniu każdego zdarzenia można wrócić do magazynu zdarzeń i odtworzyć właściwy stan dla bazy danych, jeśli ludzki błąd spowoduje uszkodzenie bazy danych. Ponieważ magazyn zdarzeń jest niemutowalny i stale się powiększa, można zastosować kontrole nadmiarowe, takie jak uprawnienia, aby prawie do zera zminimalizować prawdopodobieństwo uszkodzenia magazynu zdarzeń w wyniku błędu. Ta technika jest również podstawowa dla architektury lambda i zostanie omówiona szczegółowo w rozdziałach 2. i 3.

Chociaż w pełni przyrostowe architektury z rejestrowaniem mogą przezwyciężyć niedoskonałość wynikającą z braku odporności na ludzkie błędy, charakterystyczną dla architektur bez rejestrowania, to rejestrowanie w żaden sposób nie pomaga uporać się z pozostałymi omówionymi złożonościami. Jak zobaczysz w następnym punkcie, każda architektura oparta wyłącznie na obliczeniach w pełni przyrostowych (w tym architektury z rejestrowaniem) będzie borykała się z wieloma problemami.

1.6.4. Rozwiązanie w pełni przyrostowe w porównaniu z architekturą lambda

Jedno z przykładowych zapytań wykorzystywanych w całej książce służy jako doskonały kontrast pomiędzy architekturą w pełni przyrostową a architekturą lambda. To zapytanie nie jest w żaden sposób sztucznie wymyślonym przykładem — jest oparte na

Kup ksi k Pole ksi k



Rysunek 1.5. Dodawanie rejestrowania do w pełni przyrostowych architektur

rzeczywistych problemach, z jakimi wiele razy borykaliśmy się w naszych karierach zawodowych. Jest ono związane z analizą odsłon stron i wykonywane względem dwóch rodzajów przychodzących danych, którymi są:

- **Odsłony stron**, które zawierają identyfikator użytkownika, adres URL i znacznik czasu.
- **Ekwiwalenty**, które zawierają dwa identyfikatory użytkownika. Ekwiwalent wskazuje, że dwa identyfikatory użytkownika odnoszą się do tej samej osoby. Możesz mieć na przykład ekwiwalent pomiędzy adresem e-mailowym ***sally@gmail.com*** i nazwą użytkownika ***sally***. Jeżeli *sally@gmail.com* zostanie zarejestrowany również z podaniem nazwy użytkownika ***sally2***, wówczas będziesz mieć ekwiwalent pomiędzy *sally@gmail.com* i *sally2*. Dzięki przechodniości wiesz, że nazwy użytkownika ***sally*** i ***sally2*** odnoszą się do tej samej osoby.

Celem tego zapytania jest obliczenie liczby unikatowych użytkowników odwiedzających dany adres URL w określonym przedziale czasu. Zapytania powinny być na bieżąco ze wszystkimi danymi i zwracać odpowiedzi z minimalnym opóźnieniem (mniejszym niż 100 milisekund). Oto interfejs dla tego zapytania:

```
long uniquesOverTime(String url, int startHour, int endHour)
```

Utrudnieniem w implementacji tego zapytania są ekwiwalenty. Jeśli w jakimś przedziale czasu pewna osoba odwiedza ten sam adres URL, wykorzystując dwa identyfikatory użytkownika podłączone poprzez ekwiwalenty (nawet w sposób przechodni), powinno to być liczone tylko jako jedna wizyta. Przychodzący nowy ekwiwalent może zmienić wyniki dla każdego zapytania w dowolnym przedziale czasu dla dowolnego adresu URL.

Na tym etapie powstrzymamy się od przedstawienia szczegółów tych rozwiązań, ponieważ do ich zrozumienia wymagane jest omówienie zbyt wielu koncepcji, takich jak indeksowanie, rozproszone bazy danych, przetwarzanie wsadowe czy HyperLogLog. Przytłaczanie Cię tymi wszystkimi koncepcjami w tym momencie byłoby niecelowe. Zamiast tego skupimy się na charakterystyce rozwiązań i uderzających różnicach między

Kup ksi k Pole ksi k

nimi. Najlepsze możliwe rozwiązanie w pełni przyrostowe zostanie przedstawione szczegółowo w rozdziale 10., a rozwiązanie oparte na architekturze lambda będzie tworzone w rozdziałach 8., 9., 14. i 15.

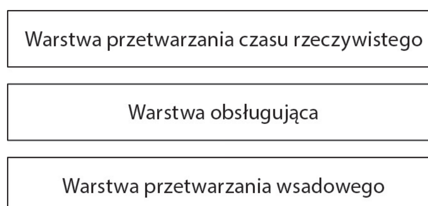
Te dwa rozwiązania mogą być porównywane na trzech płaszczyznach: dokładności, opóźnień i przepustowości. Rozwiązanie oparte na architekturze lambda jest znacznie lepsze pod każdym względem. Oba rozwiązania wymagają zastosowania przybliżeń, ale wersja w pełni przyrostowa wymusza wykorzystanie gorszej techniki aproksymacji z 3 – 5 razy wyższym współczynnikiem błędu. Wykonywanie zapytań jest znacznie bardziej kosztowne w wersji w pełni przyrostowej, co wpływa zarówno na opóźnienia, jak i na wydajność. Jednak najbardziej uderzającą różnicą między tymi dwoma podejściami jest konieczność użycia dla w pełni przyrostowego rozwiązania specjalnego sprzętu do osiągnięcia w miarę rozsądnej wydajności. Ponieważ w pełni przyrostowa wersja musi wykonywać wiele losowych wyszukiwań dostępu w celu rozwiązywania zapytań, praktycznie wymagane jest użycie napędów SSD, aby uniknąć powstawania wąskich gardeł podczas przeszukiwania dysku.

To, że architektura lambda pozwala tworzyć rozwiązania o wyższej wydajności pod każdym względem oraz unikać złożoności nękających architektury w pełni przyrostowe, pokazuje, iż dzieje się coś bardzo fundamentalnego. Kluczowe jest zrzucenie kajdan obliczeń w pełni przyrostowych i zaakceptowanie innych technik. Zobaczmy teraz, jak to zrobić.

1.7. Architektura lambda

Obliczanie dowolnych funkcji na dowolnym zbiorze danych w czasie rzeczywistym jest trudnym zadaniem. Nie istnieje pojedyncze narzędzie zapewniające całościowe rozwiązanie. Zamiast tego należy użyć różnorodnych technik i narzędzi, aby zbudować kompletny system Big Data.

Główną ideą architektury lambda jest budowanie systemów Big Data jako szeregu warstw, tak jak pokazano to na rysunku 1.6. Każda warstwa zapewnia podzbiór właściwości i wykorzystuje funkcjonalność dostarczaną przez warstwę znajdującą się pod nią. Sposobu projektowania, implementowania i wdrażania każdej warstwy będziesz uczyć się w całej książce, ale ogólne koncepcje budowy struktury całego systemu są dość łatwe do zrozumienia.



Rysunek 1.6. Architektura lambda

Wszystko zaczyna się od równania `zapytanie = funkcja(wszystkie dane)`. Najlepiej, gdyby można było uruchamiać te funkcje „w locie”, aby uzyskać wyniki. Niestety, nawet jeśli byłoby to możliwe, wymagałoby ogromnej ilości zasobów i byłoby niewspółmiernie kosztowne. Wyobraź sobie odczytywanie petabajtowego zbioru danych za każdym razem, gdy chcesz odpowiedzieć na zapytanie o czyjąś bieżącą lokalizację.

Najbardziej oczywistym rozwiązaniem alternatywnym jest wcześniejsze przeliczenie funkcji zapytania. Nazwijmy tę wstępnie przeliczoną funkcję zapytania **obrazem**

Kup ksi k Pole ksi k

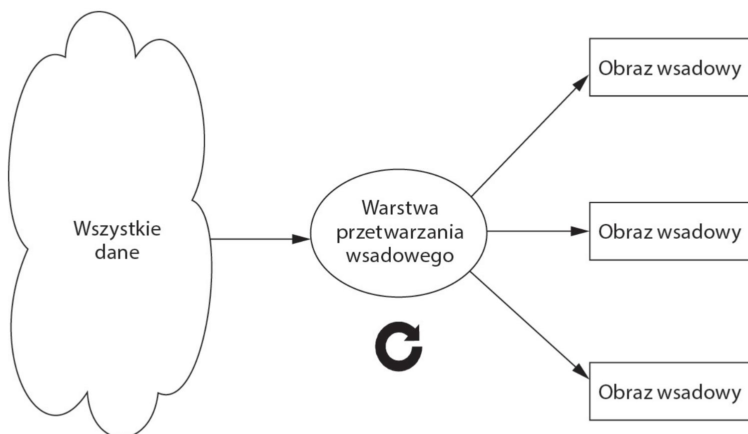
wsadowym (ang. *batch view*). Zamiast obliczać zapytanie w locie, można odczytać wyniki z wcześniej obliczonego obrazu. Wstępnie przeliczony obraz jest indeksowany, aby można było uzyskiwać do niego dostęp za pomocą losowych operacji odczytu. Ten system wygląda następująco:

```
obraz wsadowy = funkcja(wszystkie dane)
    zapytanie = funkcja(obraz wsadowy)
```

W tym systemie funkcja jest uruchamiana na wszystkich danych, aby uzyskać obraz wsadowy. Następnie, kiedy chcesz poznać wartość dla jakiegoś zapytania, uruchamiasz funkcję na tym obrazie wsadowym. Obraz wsadowy pozwala uzyskać bardzo szybko wartości bez konieczności skanowania wszystkiego, co się w nim znajduje.

Ponieważ ta dyskusja jest do pewnego stopnia abstrakcyjna, poprzyjmy ją przykładem. Załóżmy, że budujesz aplikację przeznaczoną do analityki internetowej (ponownie) i chcesz kwerendować liczbę odsłon dla jakiegoś adresu URL z dowolnej liczby dni. Jeśli obliczałbyś to zapytanie jako funkcję wszystkich danych, skanowałbyś zbiór danych pod kątem odsłon strony dla tego adresu URL w danym przedziale czasu i zwracał zliczenie tych wyników.

W podejściu wykorzystującym obraz wsadowy zamiast tego uruchamiana jest funkcja wszystkich odsłon strony, aby obliczyć wstępnie indeks z klucza *[url, dzień]* do liczby odsłon dla tego adresu URL w danym dniu. Następnie, w celu rozwiązania zapytania, pobierane są wszystkie wartości z tego obrazu dla wszystkich dni w tym przedziale czasowym i sumowane są liczby, aby uzyskać wynik. To podejście zostało przedstawione na rysunku 1.7.



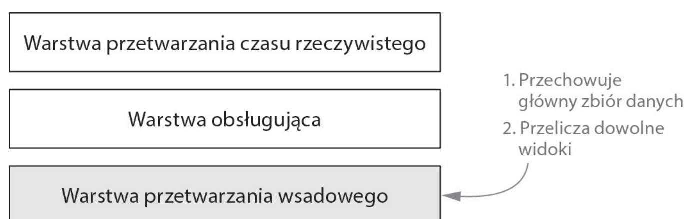
Rysunek 1.7. Architektura warstwy przetwarzania wsadowego

Powinno być jasne, że na podstawie tego, co zostało dotąd opisane, w tym podejściu czegoś brakuje. Wyraźnie widać, że tworzenie obrazu wsadowego będzie operacją o wysokiej latencji, ponieważ polega na uruchomieniu funkcji na wszystkich posiadanych danych. Do czasu zakończenia tej operacji zebranych zostanie wiele nowych danych, które nie będą reprezentowane w obrazach wsadowych, a zapytania będą uwzględniały dane, które nie zawierają wielu godzin ostatnich informacji. Zignorujmy jednak na razie ten problem, ponieważ będziemy w stanie to naprawić. Udajmy, że jest

w porządku, iż zapytania będą zdezaktualizowane o kilka godzin i kontynuujmy badanie koncepcji wstępnego obliczania obrazu wsadowego poprzez uruchamianie funkcji na pełnym zbiorze danych.

1.7.1. Warstwa przetwarzania wsadowego

Ta część architektury lambda, która implementuje równanie **obraz wsadowy = funkcja (wszystkie dane)** nazywa się **warstwą przetwarzania wsadowego** (ang. *batch layer*). Warstwa przetwarzania wsadowego przechowuje główną kopię zbioru danych (ang. *master dataset*) i na niej przelicza obrazy wsadowe (patrz rysunek 1.8). Główny zbiór danych może być traktowany jako bardzo obszerna lista rekordów.



Rysunek 1.8. Warstwa przetwarzania wsadowego

Warstwa przetwarzania wsadowego musi być w stanie robić dwie rzeczy: przechowywać niemutowalny, stale rosnący główny zbiór danych oraz przeliczać dowolne funkcje na tym zbiorze danych. Ten rodzaj przetwarzania najlepiej jest wykonywać, stosując systemy przetwarzania wsadowego. Kanonicznym przykładem systemu przetwarzania wsadowego jest platforma Hadoop, którą wykorzystamy w tej książce do zaprezentowania koncepcji warstwy wspomnianego przetwarzania.

Najprostsza forma warstwy przetwarzania wsadowego może być przedstawiona za pomocą następującego pseudokodu:

```
function runBatchLayer():
  while(true):
    recomputeBatchViews()
```

Warstwa przetwarzania wsadowego działa w pętli `while(true)` i stale przelicza od podstaw obrazy wsadowe. W rzeczywistości warstwa ta jest nieco bardziej zawiła, ale wrócimy do tego w dalszej części książki. Na tym etapie to jest najlepszy sposób wyobrażenia sobie warstwy przetwarzania wsadowego.

Zaletą omawianej warstwy jest prostota użycia. Obliczenia wsadowe są napisane jak programy jednowątkowe, a równoległość jest uzyskiwana za darmo. Łatwo jest napisać solidne, wysoce skalowalne obliczenia w warstwie przetwarzania wsadowego. Warstwę tę skaluje się poprzez dodawanie nowych maszyn.

Poniżej zamieszczony został przykład obliczeń warstwy przetwarzania wsadowego. Nie przejmuj się, jeśli nie rozumiesz tego kodu — chodzi o to, aby pokazać, jak wygląda program z natury równoległy:

```
Api.execute(Api.hfsSeqfile("/tmp/pageview-counts"),
     new Subquery("?url", "?count")
         .predicate(Api.hfsSeqfile("/data/pageviews"),
```

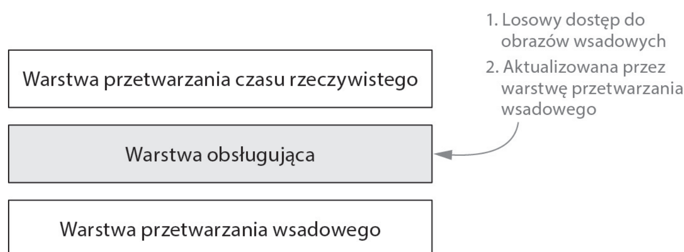
Kup ksi k Pole ksi k

```
        "?url", "?user", "?timestamp")
    .predicate(new Count(), "?count");
```

Ten kod oblicza liczbę odsłon strony dla każdego adresu URL dla danego zbioru surowych danych wejściowych odsłon strony. W tym kodzie ciekawe jest to, że wszystkie wyzwania współbieżności związane z planowaniem prac i scalaniem wyników są obsługiwane za Ciebie. Ponieważ algorytm został napisany w ten sposób, może być dowolnie rozmieszczany w klastrze MapReduce, co pozwala skalować na tyle węzłów, ile jest dostępnych. Po zakończeniu obliczeń katalog wyjściowy będzie zawierał pewną liczbę plików z wynikami. Sposobu pisania programów takich jak ten nauczysz się w rozdziale 7.

1.7.2. Warstwa obsługująca

Warstwa przetwarzania wsadowego emituje obrazy wsadowe jako wyniki swoich funkcji. Następnym krokiem jest załadowanie gdzieś tych obrazów, aby mogły być kwerendowane. Tu do gry wkracza warstwa obsługująca (ang. *serving layer*). Jest ona wyspecjalizowaną, rozproszoną bazą danych, która ładuje obrazy wsadowe i umożliwia wykonywanie na nich losowych operacji odczytu (patrz rysunek 1.9). Gdy dostępne stają się nowe obrazy wsadowe, warstwa obsługująca automatycznie zastępuje nimi starsze obrazy, aby można było uzyskać bardziej aktualne wyniki.



Rysunek 1.9. Warstwa obsługująca

Baza danych warstwy obsługującej obsługuje aktualizacje i losowe operacje odczytu. Co ważniejsze, nie musi obsługiwać losowych operacji zapisu. Jest to bardzo ważna uwaga, ponieważ losowe operacje zapisu są przyczyną większości złożoności w bazach danych. Dzięki brakowi obsługi losowych operacji zapisu te bazy danych są bardzo proste. Ta prostota sprawia, że są one niezawodne, przewidywalne, łatwe w konfiguracji i obsłudze. Baza danych warstwy obsługującej, ElephantDB, z której nauczysz się korzystać z tej książki, ma tylko kilka tysięcy linii kodu.

1.7.3. Warstwy przetwarzania wsadowego i obsługująca zapewniają niemal wszystkie właściwości

Warstwy przetwarzania wsadowego i obsługująca obsługują dowolne zapytania dla jakiegokolwiek zbioru danych przy przyjęciu takiego kompromisu, że zapytania będą uwzględniały dane, które nie zawierają wielu godzin ostatnich informacji. Nowa porcja danych potrzebuje kilku godzin na propagację w warstwie przetwarzania wsadowego do warstwy obsługującej, w której może być kwerendowana. Ważną rzeczą jest to, że

Kup ksi k Pole ksi k

poza niskimi opóźnieniami aktualizacji warstwy przetwarzania wsadowego i obsługująca zapewniają wszystkie właściwości pożądane w systemie Big Data, opisane w podrozdziale 1.5. Przyjrzyjmy się im po kolei:

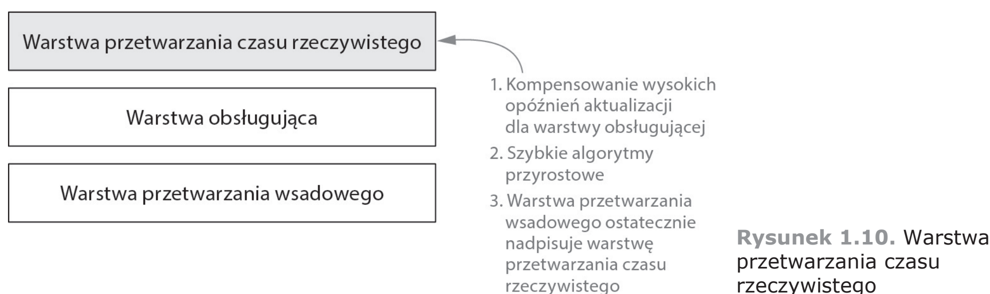
- **Niezawodność i odporność na błędy**. Hadoop obsługuje przełączanie awaryjne, gdy maszyny ulegają awarii. Pod maską warstwa obsługująca wykorzystuje replikację w celu zapewnienia dostępności w wypadku awarii serwerów. Warstwy przetwarzania wsadowego i obsługująca są również odporne na ludzkie błędy, ponieważ, gdy popełniony zostanie błąd, można naprawić algorytm lub usunąć uszkodzone dane i przeliczyć obrazy od nowa.
- **Skalowalność**. Obie warstwy, przetwarzania wsadowego i obsługująca, są łatwo skalowalne. Obie są systemami w pełni rozproszonymi i ich skalowanie polega po prostu na dodawaniu nowych maszyn.
- **Uogólnienie**. Opisana architektura jest tak ogólna, jak to tylko możliwe. Można obliczać i aktualizować dowolne obrazy wybranego zbioru danych.
- **Rozszerzalność**. Dodanie nowego obrazu jest tak proste jak dodanie nowej funkcji głównego zbioru danych. Ponieważ główny zbiór danych może zawierać dowolne dane, łatwo można dodawać nowe typy danych. Jeśli chcesz ulepszyć obraz, nie musisz się martwić o obsługę wielu wersji tego obrazu w aplikacji. Możesz po prostu przeliczyć cały obraz od nowa.
- **Zapytania *ad hoc***. Warstwa przetwarzania wsadowego obsługuje zapytania *ad hoc* z natury. Wygodne jest to, że wszystkie dane są dostępne w jednym miejscu.
- **Minimalna konserwacja**. Głównym komponentem wymagającym konserwacji w tym systemie jest Hadoop, który choć wymaga pewnej wiedzy administracyjnej, jest dość prosty w obsłudze. Jak wyjaśniono wcześniej, bazy danych warstwy obsługującej są nieskomplikowane, gdyż nie wykonują losowych operacji zapisu. Ponieważ baza danych warstwy obsługującej ma tak mało ruchomych części, zawieść może o wiele mniej elementów. W konsekwencji w bazie danych warstwy obsługującej istnieje znacznie mniejsze prawdopodobieństwo, że cokolwiek *pójdzie* nie tak, a więc te bazy danych są łatwiejsze w utrzymaniu.
- **Debugowalność**. W warstwie przetwarzania wsadowego zawsze znajdują się dane wejściowe i wyjściowe obliczeń. W tradycyjnej bazie danych dane wyjściowe mogą zastąpić oryginalne dane wejściowe, na przykład w wypadku zwiększania wartości. W warstwach przetwarzania wsadowego i obsługującej dane wejściowe stanowi główny zbiór danych, a danymi wyjściowymi są obrazy. Podobnie istnieją dane wejściowe i wyjściowe dla wszystkich etapów pośrednich. Posiadanie danych wejściowych i wyjściowych zapewnia wszystkie informacje potrzebne do debugowania, gdy coś pójdzie nie tak.

Piękno warstw przetwarzania wsadowego i obsługującej polega na tym, że zapewniają one prawie wszystkie wymagane właściwości w prostym i łatwym do zrozumienia podejściu. Nie ma kwestii współbieżności, z którymi trzeba by sobie radzić, a skalowanie jest trywialne. Jedyną brakującą właściwością są aktualizacje o niskiej latencji. Ten problem rozwiązuje ostatnia warstwa, czyli warstwa przetwarzania czasu rzeczywistego.

Kup ksi k Pole ksi k

1.7.4. Warstwa przetwarzania czasu rzeczywistego

Warstwa obsługująca przeprowadza aktualizacje za każdym razem, kiedy warstwa przetwarzania wsadowego zakończy wstępne przeliczanie obrazu wsadowego. Oznacza to, że jedynymi danymi niereprezentowanymi w obrazie wsadowym są dane, które nadeszły w trakcie procesu wstępnego przeliczania. Aby otrzymać system danych, który jest w pełni systemem czasu rzeczywistego (czyli pozwala obliczać dowolne funkcje na dowolnych danych w czasie rzeczywistym), pozostało jedynie skompensować te dane z ostatnich kilku godzin. Jest to zadanie warstwy przetwarzania czasu rzeczywistego (ang. *speed layer*). Jej celem jest zapewnienie, aby nowe dane były reprezentowane w funkcjach zapytań tak szybko, jak określają to wymagania aplikacji (patrz rysunek 1.10).



Rysunek 1.10. Warstwa przetwarzania czasu rzeczywistego

Możesz potraktować warstwę przetwarzania czasu rzeczywistego jako podobną do warstwy przetwarzania wsadowego pod tym względem, że produkuje obrazy na podstawie otrzymywanych danych. Jedna z wyraźnych różnic polega na tym, że warstwa przetwarzania czasu rzeczywistego bierze pod uwagę tylko najnowsze dane, natomiast warstwa przetwarzania wsadowego uwzględnia wszystkie dane naraz. Kolejną dużą różnicą jest to, że w celu uzyskania najmniejszych możliwych opóźnień warstwa przetwarzania czasu rzeczywistego nie uwzględnia wszystkich nowych danych jednocześnie. Aktualizuje obrazy czasu rzeczywistego, gdy otrzymuje nowe dane, zamiast ponownie obliczać obrazy, jak robi to warstwa przetwarzania wsadowego. Warstwa przetwarzania czasu rzeczywistego przeprowadza obliczenia przyrostowe, a nie ponowne obliczenia, które są wykonywane w warstwie przetwarzania wsadowego.

Można sformalizować przepływ danych w warstwie przetwarzania czasu rzeczywistego za pomocą następującego równania:

```
obraz czasu rzeczywistego = funkcja(obraz czasu rzeczywistego, nowe dane)
```

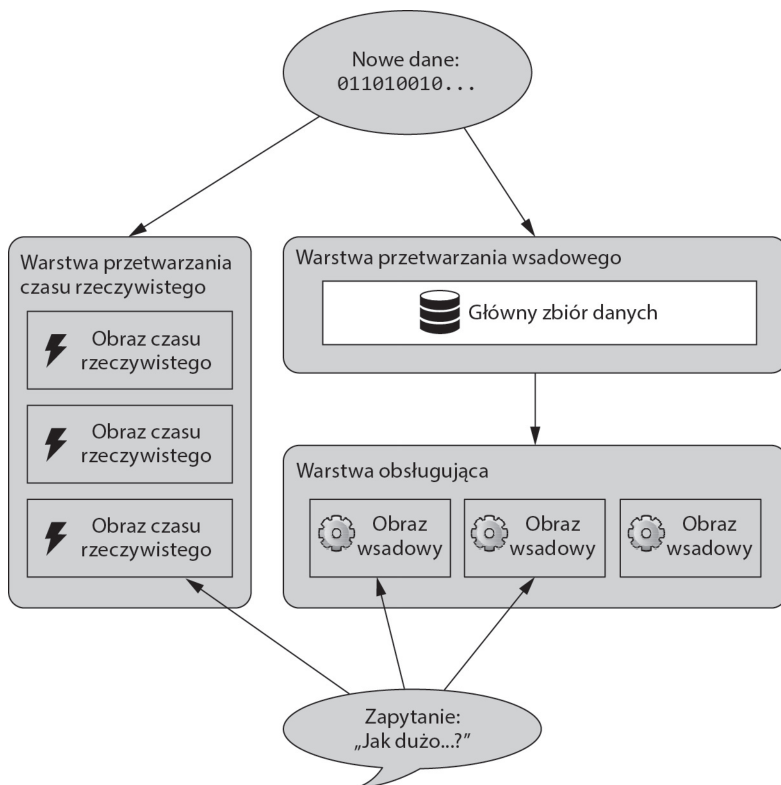
Obraz czasu rzeczywistego jest aktualizowany na podstawie nowych danych oraz istniejącego obrazu czasu rzeczywistego.

Architekturę lambda można w całości podsumować za pomocą tych trzech równań:

```
            obraz wsadowy = funkcja(wszystkie dane)
obraz czasu rzeczywistego = funkcja(obraz czasu rzeczywistego, nowe dane)
                 zapytanie = funkcja(obraz wsadowy, obraz czasu rzeczywistego)
```

Kup ksi k Pole ksi k

Graficzna reprezentacja tych koncepcji została przedstawiona na rysunku 1.11. Zamiast obsługi zapytań tylko poprzez wykonanie funkcji na obrazie wsadowym zapytania są rozwiązywane poprzez uwzględnienie zarówno obrazu wsadowego, jak i obrazu czasu rzeczywistego, a następnie scalenie wyników.



Rysunek 1.11. Schemat architektury lambda

Warstwa przetwarzania czasu rzeczywistego wykorzystuje bazy danych obsługujące losowe operacje odczytu i zapisu. Ponieważ takie bazy danych obsługują losowe operacje zapisu, są znacznie bardziej skomplikowane pod względem implementacji i eksploatacji niż bazy danych używane w warstwie obsługującej.

Piękno architektury lambda polega na tym, że gdy dane raz dotrą poprzez warstwę przetwarzania wsadowego do warstwy obsługującej, odpowiadające im wyniki w obrazach czasu rzeczywistego *nie są już potrzebne*. Oznacza to, że gdy fragmenty obrazów czasu rzeczywistego stają się zbyteczne, można z nich zrezygnować. To wspaniały rezultat, ponieważ warstwa przetwarzania czasu rzeczywistego jest o wiele bardziej skomplikowana niż warstwy przetwarzania wsadowego i obsługująca. Ta właściwość architektury lambda nazywa się **izolacją złożoności** (ang. *complexity isolation*), co oznacza, że złożoność jest przesuwana do warstwy, której wyniki są tylko tymczasowe. Jeśli coś nie wyjdzie, zawsze można porzucić stan dla całej warstwy przetwarzania czasu rzeczywistego i wszystko wróci do normy w ciągu kilku godzin.

Kontynuujmy przykład budowania aplikacji do analizy internetowej, która obsługuje zapytania o liczbę odsłon strony w ciągu pewnej liczby dni. Przypomnijmy, że warstwa przetwarzania wsadowego produkuje obrazy wsadowe z klucza *[url, dzień]* do liczby odsłon strony.

Warstwa przetwarzania czasu rzeczywistego utrzymuje swoje własne osobne obrazy klucza *[url, dzień]* do liczby odsłon strony. Podczas gdy warstwa przetwarzania wsadowego ponownie przelicza swoje obrazy poprzez dosłowne liczenie odsłon, warstwa przetwarzania czasu rzeczywistego aktualizuje swoje obrazy, zwiększając wartość licznika w obrazach za każdym razem, kiedy otrzymuje nowe dane. Aby odpowiedzieć na zapytanie, kwerendowane są zarówno obrazy wsadowe, jak i obrazy czasu rzeczywistego, niezbędne do zagwarantowania danych dla określonego przedziału czasu, a następnie wyniki są sumowane w celu uzyskania ostatecznej liczby odsłon. Nieco pracy wymaga zapewnienie poprawnej synchronizacji wyników, ale zajmiemy się tym w późniejszym rozdziale.

Niektóre algorytmy są trudne do obliczania przyrostowego. Podział na warstwy przetwarzania wsadowego i przetwarzania czasu rzeczywistego daje możliwość elastycznego wykorzystania dokładnego algorytmu w warstwie przetwarzania wsadowego i przybliżonego algorytmu w warstwie przetwarzania czasu rzeczywistego. Warstwa przetwarzania wsadowego raz za razem nadpisuje warstwę przetwarzania czasu rzeczywistego, więc aproksymacja jest korygowana, a system wykazuje właściwość **dokładności ostatecznej** (ang. *eventual accuracy*). Obliczanie na przykład unikatowych liczb może okazać się trudne, jeśli zbiory unikatowych elementów stają się obszerne. Łatwo jest wykonać zliczanie unikatowych wartości w warstwie przetwarzania wsadowego, ponieważ brane są pod uwagę wszystkie dane naraz, ale w warstwie przetwarzania czasu rzeczywistego można używać jedynie aproksymacyjnego algorytmu HyperLogLog.

Ostatecznie otrzymujesz to, co najlepsze z obu światów: wydajność i niezawodność. System, który przeprowadza dokładne obliczenia w warstwie przetwarzania wsadowego i przybliżone obliczenia w warstwie przetwarzania czasu rzeczywistego, wykazuje dokładność ostateczną, ponieważ warstwa przetwarzania wsadowego koryguje to, co jest obliczane w warstwie przetwarzania czasu rzeczywistego. Nadal otrzymujemy aktualizacje o niskiej latencji, ale ponieważ warstwa przetwarzania czasu rzeczywistego jest przejściowa, złożoność osiągnięcia tego stanu nie ma wpływu na niezawodność Twoich wyników. Przejściowy charakter warstwy przetwarzania czasu rzeczywistego pozwala Ci być elastycznym, dzięki czemu możesz być jednocześnie bardzo agresywnym, gdy przychodzi do podejmowania kompromisowych decyzji ze względu na wydajność. Oczywiście dla obliczeń, które mogą być wykonywane dokładnie w sposób przyrostowy, ten system jest w pełni dokładny.

1.8. Najnowsze trendy w technologii

Pomocne jest zrozumienia tła dla narzędzi, których będziemy używać w całej książce. Jest wiele trendów w technologii mających duży wpływ na sposoby budowania systemów Big Data.

Kup ksi k Pole ksi k

1.8.1. Procesory nie stają się coraz szybsze

Zaczęliśmy zbliżać się do fizycznych granic prędkości, jakie mogą być osiągane przez pojedyncze procesory. Oznacza to, że jeśli chcesz skalować systemy do większej ilości danych, musisz być w stanie zapewnić równoległość obliczeń. Doprowadziło to do powstania algorytmów równoległych typu *shared-nothing* i odpowiadających im systemów, takich jak MapReduce. Zamiast po prostu próbować skalować poprzez kupowanie lepszej maszyny, co określa się jako **skalowanie pionowe**, systemy skaluje się wskutek dodawania kolejnych maszyn, co jest znane jako **skalowanie poziome**.

1.8.2. Elastyczne chmury

Kolejnym trendem w technologii było powstanie elastycznych chmur, co jest określane również terminem **infrastruktura jako usługa** (ang. *Infrastructure as a Service* — IaaS). Najbardziej godną uwagi elastyczną chmurą jest Amazon Web Services (AWS). Elastyczne chmury umożliwiają wynajmowanie sprzętu na żądanie, dzięki czemu nie jest konieczne posiadanie swojego sprzętu we własnej lokalizacji. Pozwalają także zwiększać lub zmniejszać rozmiar klastra niemal natychmiastowo, jeśli więc masz jakieś duże zadanie, które chcesz uruchomić, możesz przydzielić dla niego sprzęt tymczasowo.

Elastyczne chmury radykalnie upraszczają administrację systemem. Zapewniają również dodatkowe opcje alokacji pamięci masowej i sprzętu, które mogą znacznie obniżyć koszty infrastruktury. AWS oferuje na przykład funkcję o nazwie ***spot instances***, w której można licytować instancje, zamiast płacić ustaloną cenę. Jeśli ktoś zaoferuje wyższą cenę od Ciebie, nie zdobędziesz instancji. Ponieważ licytowane instancje mogą zniknąć w każdej chwili, z reguły są znacznie tańsze niż normalne. W wypadku rozproszonych systemów obliczeniowych, takich jak MapReduce, elastyczne chmury są świetnym rozwiązaniem, ponieważ odporność na błędy jest wspierana w warstwie oprogramowania.

1.8.3. Dynamiczny ekosystem open source dla Big Data

W ciągu ostatnich kilku lat społeczność *open source* stworzyła mnóstwo technologii Big Data. Wszystkie technologie opisane w tej książce są dostarczane bezpłatnie na licencji *open source*.

Istnieje pięć kategorii projektów *open source*, które poznasz. Pamiętaj, że nie jest to książka przeglądowa — jej intencją nie jest po prostu przedstawienie kilku technologii. Celem jest nauczenie Cię podstawowych zasad, abyś był w stanie ocenić odpowiednie narzędzia i wybrać je do własnych potrzeb:

- **Systemy obliczeń wsadowych**. Są to systemy o wysokiej przepustowości i wysokich opóźnieniach. Systemy obliczeń wsadowych mogą wykonywać niemal dowolne obliczenia, ale te obliczenia mogą trwać wiele godzin lub dni. Jedynym systemem obliczeń wsadowych, jaki wykorzystamy, jest Hadoop. Projekt Hadoop ma dwa podprojekty: Hadoop Distributed File System (HDFS) i Hadoop MapReduce. HDFS jest rozproszonym, odpornym na błędy systemem pamięci masowej, który można skalować do petabajtów danych. MapReduce jest poziomo skalowalnym frameworkiem obliczeniowym integrującym się z HDFS.

- **Frameworki serializacji**. Zapewniają narzędzia i biblioteki umożliwiające używanie obiektów pomiędzy różnymi językami. Mogą serializować obiekt do postaci tablicy bajtów z dowolnego języka, a następnie deserializować tę tablicę bajtów do postaci obiektu w dowolnym języku. Frameworki serializacji zapewniają język SDL (ang. *Schema Definition Language*) do definiowania obiektów i ich pól oraz mechanizmy bezpiecznego wersjonowania obiektów, tak aby schemat mógł ewoluować bez unieważnienia istniejących obiektów. Trzy znaczące frameworki serializacji to: Thrift, Protocol Buffers i Avro.
- **Bazy danych NoSQL losowego dostępu**. W ciągu kilku ostatnich lat powstało mnóstwo baz danych NoSQL. Trudno być na bieżąco ze wszystkimi, ale należą do nich m.in.: Cassandra, HBase, MongoDB, Voldemort, Riak, CouchDB. Wszystkie te bazy danych mają jedną wspólną cechę: poświęcają pełną ekspresyjność języka SQL na rzecz specjalizacji w określonych rodzajach operacji. Mają one różną semantykę i są przeznaczone do konkretnych celów. Ich przeznaczeniem *nie jest* magazynowanie dowolnych danych. Pod wieloma względami podjęcie decyzji o użyciu bazy danych NoSQL jest jak wybór pomiędzy tablicą mieszającą, mapą posortowaną, listą powiązaną a wektorem, gdy masz określić, jaką strukturę danych wykorzystać w programie. Wiesz dokładnie wcześniej, co masz zamiar zrobić, i na tej podstawie dokonujesz odpowiedniego wyboru. Jako część przykładowej aplikacji, którą będziemy budować, wykorzystamy bazę danych Cassandra.
- **Systemy wymiany komunikatów/kolejkowania**. System wymiany komunikatów/kolejkowania umożliwia wysyłanie i wykorzystywanie komunikatów między procesami w sposób odporny na błędy i asynchroniczny. Kolejka komunikatów jest kluczowym elementem dla wykonywania przetwarzania w czasie rzeczywistym. W tej książce będziemy używać systemu Apache Kafka.
- **Systemy obliczeń w czasie rzeczywistym**. Charakteryzują się wysoką przepustowością, niskimi opóźnieniami oraz przetwarzaniem strumieniowym. Nie mogą one wykonywać takiego zakresu obliczeń, jaki są w stanie wykonywać systemy przetwarzania wsadowego, ale mogą bardzo szybko przetwarzać komunikaty. W tej książce będziemy używać systemu Storm. Topologie tego systemu są łatwe do napisania i skalowania.

Gdy te projekty *open source* dojrzały, wokół niektórych z nich uformowały się firmy zapewniające wsparcie dla zastosowań typu *enterprise*. Firma Cloudera zapewnia na przykład wsparcie dla platformy Hadoop, a DataStax — dla bazy danych Cassandra. Inne projekty są produktami firmowymi, na przykład Riak jest produktem Basho Technologies, MongoDB jest produktem 10gen, a RabbitMQ to z kolei produkt SpringSource, czyli oddziału firmy VMWare.

Kup ksi k Pole ksi k

1.9. Przykładowa aplikacja: SuperWebAnalytics.com

Zbudujemy w tej książce przykładową aplikację Big Data, która posłuży do zilustrowania przedstawianych koncepcji. Zbudujemy warstwę zarządzania danymi dla usługi typu Google Analytics. Usługa ta będzie w stanie śledzić miliardy odsłon dziennie.

Wspomniana usługa będzie wspierać wiele różnych metryk. Każda metryka będzie obsługiwana w czasie rzeczywistym. Będzie to szeroki zakres metryk: od prostych metryk zliczających po złożone analizy tego, jak użytkownicy nawigują po stronie internetowej.

Obsługiwane będą następujące metryki:

- **Liczba odsłon dla adresu URL w funkcji czasu**. Przykładowe zapytania to: „Jakie są wskaźniki odsłon strony dla każdego dnia w ciągu ostatniego roku?" oraz „Ile było odsłon strony w ciągu ostatnich 12 godzin?".
- **Liczba unikatowych użytkowników odwiedzających adres URL zobrazowana w czasie**. Przykładowe zapytania to: „Ile unikatowych osób odwiedziło tę domenę w roku 2010?" oraz „Ile unikatowych osób odwiedziło tę domenę w poszczególnych godzinach w ciągu ostatnich trzech dni?".
- **Analiza współczynnika odrzuceń** (ang. *bounce rate*). „Jaki procent ludzi odwiedza daną stronę i nie odwiedza innych stron tej witryny?".

Zbudujemy warstwy do przechowywania, przetwarzania i obsługi zapytań wysyłanych do aplikacji.

1.10. Podsumowanie

Zobaczyłeś, co może nie wyjść podczas skalowania systemu relacyjnego za pomocą tradycyjnych technik, takich jak *sharding*. Problemy, z którymi trzeba się zmierzyć, wykraczają poza skalowanie, gdy system staje się coraz bardziej złożony w zarządzaniu, a tym samym trudniejszy w rozszerzaniu, a nawet w zrozumieniu. Gdy w kolejnych rozdziałach będziesz się uczyć budować systemy Big Data, skupimy się zarówno na niezawodności, jak i na skalowalności. Jak się przekonasz, gdy budujesz komponenty we właściwy sposób, w tym samym systemie możesz osiągnąć jednocześnie niezawodność i skalowalność.

Korzyści płynące z systemów danych zbudowanych przy użyciu architektury lambda wykraczają poza samo skalowanie. Ponieważ system będzie w stanie obsłużyć znacznie większą ilość danych, będziesz w stanie zgromadzić jeszcze więcej danych i uzyskać z nich dalsze korzyści. Zwiększenie ilości i różnorodności przechowywanych danych da większe możliwości eksploatacji danych, tworzenia analiz i budowania nowych aplikacji.

Kolejną zaletą korzystania z architektury lambda jest wysoki stopień niezawodności tworzonych aplikacji. Składa się na to wiele czynników, na przykład będziesz miał możliwość uruchamiania obliczeń na całym zbiorze danych, aby przeprowadzić migracje lub naprawić błędy. Już nigdy nie będziesz mieć do czynienia z sytuacjami, w których istnieją różne aktywne wersje schematu w tym samym czasie. Podczas zmiany schematu będziesz miał możliwość aktualizowania wszystkich danych do nowego schematu. Podobnie, jeśli w środowisku produkcyjnym przypadkowo zostanie wdrożony

Kup ksi k Pole ksi k

nieprawidłowy algorytm, który uszkodzi serwowane dane, można łatwo to naprawić przez ponowne przeliczenie uszkodzonych wartości. Jak się przekonasz, istnieje wiele innych powodów, dla których aplikacje Big Data będą bardziej niezawodne.

Wreszcie wydajność będzie bardziej przewidywalna. Chociaż architektura lambda jako całość jest ogólna i elastyczna, poszczególne komponenty tworzące system są wyspecjalizowane. W porównaniu z czymś takim, jak planista zapytań SQL, za kulisami nie ma zbyt wiele „magii". Prowadzi to do uzyskania bardziej przewidywalnej wydajności.

Nie przejmuj się, jeśli duża część tego materiału nadal wydaje się nieco niejasna. Do omówienia pozostało jeszcze wiele kwestii i do każdego wprowadzonego w tym rozdziale tematu będziemy powracać w kolejnych partiach książki, by omówić go szczegółowo. W następnym rozdziale zaczniesz uczyć się sposobu budowania architektury lambda. Rozpoczniesz od samych podstaw stosu i sposobu modelowania oraz schematyzacji głównej kopii zbioru danych.

Kup ksi k Pole ksi k

Część I

Warstwa przetwarzania wsadowego

Część I koncentruje się na warstwie przetwarzania wsadowego architektury lambda. Rozdziały teoretyczne są przeplatane rozdziałami ilustracyjnymi.

Rozdział 2. omawia sposób modelowania i schematyzacji danych w głównym zbiorze danych. Rozdział 3. ilustruje te koncepcje za pomocą narzędzia Apache Thrift.

W rozdziale 4. przedstawiono wymagania względem przechowywania głównego zbioru danych. Przekonasz się, że wiele funkcji dostarczanych zazwyczaj przez rozwiązania bazodanowe nie jest wymaganych dla głównego zbioru danych i w rzeczywistości przeszkadza w optymalizacji przechowywania głównego zbioru danych. Wymagania te lepiej spełnia prostsze i oferujące mniej funkcji rozwiązanie dla przechowywania danych. Rozdział 5. ilustruje praktyczne przechowywanie głównego zbioru danych przy użyciu rozproszonego systemu plików Hadoop (ang. *Hadoop Distributed Filesystem* — HDFS).

W rozdziale 6. omówiono obliczanie dowolnych funkcji na głównym zbiorze danych z wykorzystaniem paradygmatu MapReduce. MapReduce jest na tyle ogólny, żeby

Kup ksi k   Pole ksi k

obliczyć dowolną skalowalną funkcję. Chociaż MapReduce jest wszechstronny, zobaczysz, że abstrakcje wyższego poziomu znacznie ułatwiają jego użycie. Rozdział 7. przedstawia wszechstronną wysokopoziomową abstrakcję MapReduce o nazwie JCascalog.

Aby połączyć ze sobą wszystkie koncepcje, w rozdziałach 8. i 9. zaimplementowano kompletną warstwę przetwarzania wsadowego dla działającego przykładu aplikacji SuperWebAnalytics.com. W rozdziale 8. opisano ogólną architekturę i algorytmy, natomiast w rozdziale 9. szczegółowo przedstawiono działający kod.

Kup ksi k

Pole ksi k

Skorowidz

A

abstrakcja, 93
  słabo komponowalna, 143
adres URL, 174, 191
agregatory, 134, 150, 155
  łączące, 134, 157
  równoległe, 150, 157
aktualizowanie, 56
  asynchroniczne, 249
  synchroniczne, 249
aktualność, 326
algorytm HyperLogLog, 171
algorytmy
  naprawiania odczytu, 248
  ponownego obliczania, 114
  przyrostowe, 115, 172, 247
analiza
  współczynnika odrzuceń, 182, 201, 217, 235, 302, 307, 316
  zapytania, 150
Apache
  Storm, 281
  Thrift, 72
aplikacja SuperWebAnalytics.com, 44, 68
architektura
  Apache Storm, 285
  lambda, 12, 20, 34, 50, 65, 208, 325
  przyrostowa, 29
asynchroniczne aktualizacje, 250
AWS, Amazon Web Services, 42

B

back-end, 21
baza danych
  Cassandra, 257
  ElephantDB, 228
  NoSQL, 243
bazy danych klucz-wartość, 82
biblioteka
  JCascalog, 144
  Pail, 94, 98, 101
bieżąca lokalizacja, 57
bloki, 84
błędy, 23, 32, 338
brak odporności, 32
budowanie warstwy obsługującej, 231
bufory, 150

C

Cassandra, 257
CRDT, 248
częściowe
  informacje, 63
  obliczanie ponowne, 329

D

DAG, directed acyclic graph, 273
DAG-i krotek, 273
dane, 26, 51
  niemutowalne, 56, 326
  nieustrukturyzowane, 55
  prawdziwe, 59
  surowe, 53
  znormalizowane, 55
debugowalność, 28, 38
definiowanie
  systemów danych, 325
  topologii, 281
dekompozycja problemu, 128
denormalizacja, 211
diagram potokowy, 129, 133, 136, 202, 301, 303, 310, 314, 317
dodawanie próbkowania, 223
dokładność, 326
  ostateczna, 41, 243
dostępność, 31, 247

dynamiczne tworzenie
makra predykatów, 164
podzapytania, 159
działanie rozproszonych systemów plików, 83
dziennik transakcji, 322

E

efektywność, 278
egzekwowalność schematu, 67
ekwiwalenty, 33
elastyczne chmury, 42
elementy schematu graficznego, 66
ElephantDB, 228
konfigurowanie klastra, 230
kwerendowanie klastra, 231
serwowanie obrazu, 229
tworzenie obrazu, 228
tworzenie shardów, 230
ewolucja schematu, 75

F

fakty, 60
identyfikowalność, 61
niepodzielne, 60
opatrzone znacznikiem czasu, 60
filtr, 134, 153
Blooma, 330
formaty plików, 100
frameworki serializacji, 43, 68, 72
funkcja, 134, 154
Split, 311
funkcje
Cassandry, 259

G

generowanie mapy, 105
grom
aktualizacji bazy danych, 277
normalizacji adresów URL, 277
rozdzielacza, 272
grupowanie, 150
aktualizacji, 22
pól, 270
strumieni, 270, 271
tasujące, 270
według klucza, 134
gwarantowanie przetwarzania komunikatów, 286

H

Hadoop, 92
HDFS, 92

I

idempotentność, 318
implementacja
inferencji płci, 124
interfejsu PailStructure, 97
liczby odsłon, 123
obiektów data, 104
punktów wpływu, 124
warstwy przetwarzania
czasu rzeczywistego, 288
warstwy przyrostowej, 333
inferencja płci, 111
informacja, 26, 51
infrastruktura jako usługa, 42
inkrementalizacja, 329
interfejs
PailStructure, 97
Queue, 263
Trident, 310
iteracja
przepływu pracy, 336
algorytmu, 177
izolacja złożoności, 40

J

JCascalog, 144
języki niestandardowe, 142

K

klasa CassandraState, 313
klaster, 84, 230
Apache Storm, 284
klucze, 256
kolejki, 22, 266
pojedynczego konsumenta, 263
wielu konsumentów, 264
kolejkowanie, 262
kolumny, 256
kompozytowe, 259
kompaktowanie, 30
online, 244
kompozycja, 143, 158
kompresja biblioteki Pail, 100

komunikaty, 306
koncepcje diagramów potokowych, 129
konfigurowanie klastra, 230
kotwiczenie, 286
kranik, 141, 188, 230
krawędzie, 67, 73
krotka, 144
  tickowa, 291
kwerendowanie, 63
  klastra, 231
  wielu zbiorów danych, 147
kwerendy, 52

L

latencja, 27
liczba
  odsłon, 110, 169, 180, 197, 215, 302, 314
  unikatowych użytkowników, 171, 181, 200, 216, 234
licznik słów, 120, 312
logowanie, 96
losowe operacje
  odczytu, 213, 243
  operacje zapisu, 243

Ł

łańcuchowanie, 156
łączenie, 134, 331
  podzapytań, 158
  wewnętrzne, 126

M

magazyn danych klucz-wartość, 82
makra predykatów, 162
  tworzone dynamicznie, 164
MapReduce, 125, 137
metryki wydajności, 209
mikrowsadowe przetwarzanie strumieniowe, 292–295, 309
minimalna konserwacja, 28
model
  danych Cassandry, 256
  danych JCascalog, 144
  oparty na faktach, 60, 62
  Storm, 268
modowanie skrótu, 228

N

narzędzie Apache Thrift, 72
niemutowalność, 52
niezawodność, 26, 38
niskie opóźnienia aktualizacji, 278
normalizacja, 64, 211
  adresów URL, 174, 191
  identyfikatorów użytkowników, 175–179, 192–196
  semantyczna, 55
NoSQL, 20, 25

O

obciążenie, 119
obiekty
  danych, 75
  Thrift, 103
obliczanie
  obrazów czasu rzeczywistego, 241
  obrazów wsadowych, 180, 197
obliczenia
  lokalne dla wsadu, 299
  przyrostowe, 240, 245
  stanowe, 299
obrazy, 52
  czasu rzeczywistego, 239, 241, 250, 255
  warstwy obsługującej, 252
  wsadowe, 35, 112, 169, 180, 197
odporność na błędy, 23, 26, 38, 56, 117, 123, 243
odsłony stron, 33
odzyskiwanie pamięci, 59
ogólność
  algorytmów, 117
  MapReduce, 123
ograniczenia frameworku serializacji, 76
okienkowe przetwarzanie strumieniowe, 305
open source, 42
operacja
  łączenia, 331
  predykatu, 146
operacje
  biblioteki Pail, 95
  łączenia, 126, 132
  niestandardowe predykatów, 153
  przetwarzania wsadowego, 98
  scalania, 133
opóźnienie, 209, 326
optymalizacja
  obrazu wsadowego, 201
  wykorzystania zasobów, 335

P

pamięć masowa, 81
paradygmat MapReduce, 122, 137
partycjonowanie
pionowe, 86, 99
poziome, 22
platforma
MapReduce, 125
Spark, 125
podzapytania tworzone dynamicznie, 159
pomiar wykorzystania zasobów, 335
potwierdzanie, 286
poziom abstrakcji, 93
predykat
agregatora, 146
filtra, 146
funkcji, 146
generatora, 146
problem małych plików, 93
procesory, 42
procesy robocze, 22, 266
projektowanie warstwy obsługującej, 215
próbkowanie skrótów, 222
przechowywanie
danych
w warstwie przetwarzania wsadowego, 94
obrazów czasu rzeczywistego, 242
zbioru danych, 80, 89, 102
przepływ pracy, 172, 186
przepustowość, 209
przetwarzanie
czasu rzeczywistego, 244
danych, 142
komunikatów, 272, 286
strumieniowe
mikrowsadowe, 265, 293, 295
pojedyncze, 265, 268
ściśle uporządkowane, 294
wsadowe, 36, 79
wiele warstw, 334
przyrostowe, 328
przyjmowanie nowych danych, 187
przyrostowe przetwarzanie wsadowe, 308, 328
pułapki narzędzi, 142
punkt
stały, 176
startowy, 186
wpływu, 111

R

RDBMS, 20
rekordy reakcji, 111
relacje, 52
między węzłami, 67
replikacja, 31
repliki, 246
reprezentacja danych, 60
rodziny kolumn, 256
role warstw, 113
rozdrabnianie, 188
rozmiar klastra, 337
rozproszone systemy plików, 82, 87, 89, 92
rozszerzalność, 27, 38
rozszerzanie diagramów potokowych, 300
rozwiązanie oparte na architekturze lambda, 224
w pełni przyrostowe, 217

S

scalanie liczników G-Counter, 248
schemat, 144
graficzny, 66
nieznormalizowany, 64
partycjonowania pionowego, 99
relacyjny, 21
shardingu, 228
typu odpal i zapomnij, 262
semantyka transakcyjna, 314
serializacja, 43, 96
serwery kolejek, 263
sharding, 22
shardy, 24
skalowalność, 27, 38, 119–121, 213, 243
skalowanie
bazy danych, 21
przez sharding, 22
za pomocą kolejki, 22
skrypty reshardingujące, 24
SLA, Service Level Agreement, 265
sortowanie wtórne, 163
Spark, 323
Streaming, 323
spot instances, 42
spójność ostateczna, 31
struktura
topologii, 277
zapytania JCascalog, 145
struktury bezkonfliktowe, 248
Supervisor, 284

Kup ksi k
Pole ksi k

SuperWebAnalytics.com
normalizacja adresów URL, 174
normalizacja identyfikatorów użytkowników, 175
nowe dane, 174
obrazy wsadowe, 169, 180
usuwanie zduplikowanych odsłon, 180
warstwa obsługująca, 215
warstwa przetwarzania czasu rzeczywistego, 274, 313
zapytania, 168
surowość, 52
synchroniczne aktualizacje, 249
systemy
danych, 325
obliczeń w czasie rzeczywistym, 43
obliczeń wsadowych, 42
wymiany komunikatów/kolejkowania, 43
szablon JCascalog, 162

Ś

ściśle uporządkowane przetwarzanie, 294
śledzenie
DAG-ów krotek, 274
wizyty, 304

T

tabele znormalizowane, 65
tasowanie, 122
token, 61
tolerancja na błędy, 213
topologia, 268
dla liczby unikatowych użytkowników, 288
licznika słów, 282
przetwarzania mikrowsadowego, 296
twierdzenie CAP, 245, 247
tworzenie
obrazu, 228
shardów, 230
typ union, 192
typy węzłów, 69

U

uogólnienie, 27, 38
ustrukturyzowane wiaderko, 103
usuwanie zduplikowanych odsłon, 180, 197
uszkodzenie danych, 24
utrzymywanie stanu, 322

W

warstwa
obsługująca, 37, 205, 207–25, 327
przetwarzania czasu rzeczywistego, 39, 237, 240, 274, 288, 302, 313, 339
przetwarzania wsadowego, 36, 47, 91, 109, 139, 167, 327
zapytań, 340
wdrażanie topologii, 284
węzeł Nimbus, 284
węzły danych, 73, 84
wiaderko, 96, 104
wieczność, 52
właściwości, 74
danych, 51
systemu, 26
współbieżność, 244
współczynnik odrzuceń, 172, 182, 201, 217, 235, 302, 316
wybór stylu algorytmu, 118
wydajność, 116, 337, 338
wygaszanie obrazów, 250
wykonywanie
diagramów potokowych, 134
logiczne, 128
wylewka, 269
odsłon, 277
transakcyjna, 299, 300
wyłączne dopisywanie, 70
wymagania dla przechowywania danych, 81
wzrost poziomu błędów, 338

Z

zadania konserwacyjne HDFS, 94
zadanie, 270
zapisywalność obrazów wsadowych, 213
zapytania, 52, 150, 168
ad hoc, 28
JCascalog, 145, 148
zbiór danych, 49, 50
złożoność
ekstremalna, 30
niezbędna, 142
operacyjna, 29
przypadkowa, 142
znajoma, 29
Zookeeper, 284
zwiększanie dostępności, 31

Kup ksi k Pole ksi k

Kup ksi k

Pole ksi k

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

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

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA

BIG DATA

NAJLEPSZE PRAKTYKI BUDOWY SKALOWALNYCH SYSTEMÓW OBSŁUGI DANYCH W CZASIE RZECZYWISTYM

Obsługa aplikacji, które operują na ogromnych zbiorach danych, czyli na przykład portali społecznościowych, przekracza możliwości zwykłych relacyjnych baz. Praca ze złożonymi zbiorami danych wymaga architektury obejmującej wielomaszynowe klastry, dzięki którym możliwe jest przechowywanie i przesyłanie informacji praktycznie dowolnej wielkości. Architektura taka powinna dodatkowo być prosta w użyciu, niezawodna i skalowalna.

Dzięki tej książce nauczysz się budować tego rodzaju architekturę. Zapoznasz się z technologią wykorzystywania klastrów maszyn. Dowiesz się, jak działają narzędzia przeznaczone specjalnie do przechwytywania i analizy danych na wielką skalę. W książce zaprezentowano łatwe do zrozumienia podejście do obsługi systemów wielkich zbiorów danych, które mogą być budowane i uruchamiane przez niewielki zespół. Nie zabrakło też wyczerpującego opisu praktycznej implementacji systemu Big Data z wykorzystaniem rzeczywistego przykładu.

W tej książce znajdziesz:

- teoretyczne podstawy koncepcji systemów Big Data
- wskazówki umożliwiające optymalne wykorzystanie zasobów do obsługi danych
- wybór technik przetwarzania i obsługi wielkich ilości danych w czasie rzeczywistym
- zagadnienia dotyczące baz danych NoSQL, przetwarzania strumieniowego i zarządzania złożonością obliczeń przyrostowych
- informacje o praktycznym stosowaniu takich narzędzi jak Hadoop, Cassandra i Storm
- wskazówki umożliwiające poszerzenie wiedzy o zwykłych bazach danych

Nathan Marz — jest twórcą projektu Apache Storm i autorem architektury lambda dla systemów Big Data.

James Warren — jest architektem analityki z doświadczeniem w uczeniu maszynowym i obliczeniach naukowych.

Big Data — to skalowalność i prostota obsługi wielkich ilości danych!

sięgnij po WIĘCEJ

KOD KORZYŚCI

Helion

42001 numer katalogowy

księgarnia internetowa

http://helion.pl

zamówienia telefoniczne

0 801 339900

0 601 339900

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

Informatyka w najlepszym wydaniu

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

ISBN 978-83-283-1892-2

9 788328 318922

cena: 59,00 zł