

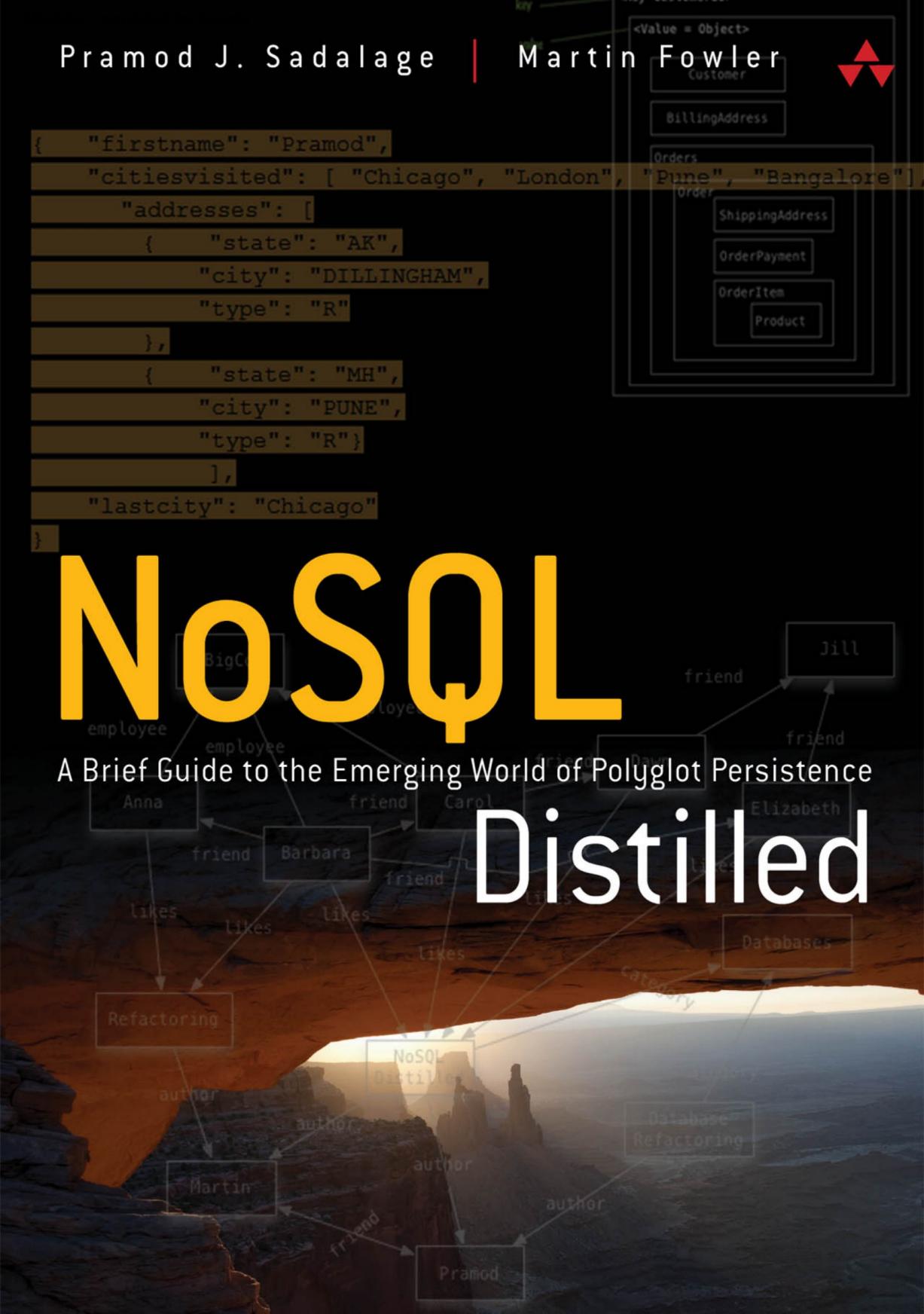


```
{ "firstname": "Pramod",
  "citiesvisited": [ "Chicago", "London", "Pune", "Bangalore"],
  "addresses": [
    { "state": "AK",
      "city": "DILLINGHAM",
      "type": "R"
    },
    { "state": "MH",
      "city": "PUNE",
      "type": "R"
    }
  ],
  "lastcity": "Chicago"
}
```

NoSQL

A Brief Guide to the Emerging World of Polyglot Persistence

Distilled





NoSQL destilado

Esta página foi deixada em branco intencionalmente

NoSQL destilado

**Um breve guia para o emergente
Mundo da Persistência Poliglota**

**Pramod J. Sadalage
Martin Fowler**

 Addison-Wesley

Upper Saddle River, NJ • Boston • Indianápolis • São Francisco
Nova York • Toronto • Montreal • Londres • Munique • Paris • Madri
Cidade do Cabo • Sydney • Tóquio • Cingapura • Cidade do México

Muitas das designações usadas por fabricantes e vendedores para distinguir seus produtos são reivindicadas como marcas registradas. Onde essas designações aparecem neste livro, e o editor estava ciente de uma reivindicação de marca registrada, as designações foram impressas com letras iniciais maiúsculas ou em todas as letras maiúsculas.

Os autores e a editora tomaram cuidado na preparação deste livro, mas não oferecem nenhuma garantia expressa ou implícita de qualquer tipo e não assumem nenhuma responsabilidade por erros ou omissões. Nenhuma responsabilidade é assumida por danos incidentais ou consequenciais em conexão com ou decorrentes do uso das informações ou programas aqui contidos.

Para obter informações sobre como comprar este título em grandes quantidades ou para oportunidades especiais de vendas (que podem incluir versões eletrônicas; designs de capa personalizados; e conteúdo específico para seu negócio, objetivos de treinamento, foco de marketing ou interesses de marca), entre em contato com nosso departamento de vendas corporativas em corpsales@pearsoned.com ou (800) 382-3419.

Para consultas de vendas governamentais, entre em contato com governmentsales@pearsoned.com.

Para dúvidas sobre vendas fora dos EUA, entre em contato com international@pearsoned.com.

Visite-nos na Web: informit.com/aw

Dados de catalogação na publicação da Biblioteca do Congresso

Sadalage, Pramod J.

NoSQL destilado: um breve guia para o mundo emergente da persistência poliglota / Pramod J Sadalage, Martin Fowler.

p.cm.

Inclui referências bibliográficas e índice.

ISBN 978-0-321-82662-6 (pbk. : alk. paper) -- ISBN 0-321-82662-0 (pbk. : alk. paper) 1. Bancos de dados--Inovações tecnológicas. 2. Sistemas de armazenamento e recuperação de informações. I. Fowler, Martin, 1963- II. Título.

QA76.9.D32S228 2013

005.74--dc23

Direitos autorais © 2013 Pearson Education, Inc.

Todos os direitos reservados. Impresso nos Estados Unidos da América. Esta publicação é protegida por direitos autorais, e a permissão deve ser obtida do editor antes de qualquer reprodução proibida, armazenamento em um sistema de recuperação ou transmissão de qualquer forma ou por qualquer meio, eletrônico, mecânico, fotocópia, gravação ou similar. Para obter permissão para usar o material deste trabalho, envie uma solicitação por escrito para Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, ou você pode enviar sua solicitação por fax para (201) 236-3290.

ISBN-13: 978-0-321-82662-6 ISBN-10:

0-321-82662-0 Texto impresso

nos Estados Unidos em papel reciclado na RR Donnelley em Crawfordsville, Indiana.

Quarta impressão, novembro de 2014

*Para meus professores Gajanan Chinchwadkar,
Dattatraya Mhaskar e Arvind Parchure. Vocês me
inspiraram muito, obrigado.*

—Pramod

*Para Cindy
—Martinho*

Esta página foi deixada em branco intencionalmente

Conteúdo

Prefácio	xiii
Parte I: Entenda	1
Capítulo 1: Por que NoSQL?	3
1.1 O valor dos bancos de dados relacionais	3
1.1.1 Obtendo dados persistentes	3
1.1.2 Concorrência	4
1.1.3 Integração	4
1.1.4 Um modelo (principalmente) padrão	4
1.2 Incompatibilidade de impedância	5
1.3 Bancos de Dados de Aplicação e Integração	6
1.4 Ataque dos Clusters	8
1.5 O Surgimento do NoSQL	9
1.6 Pontos-chave	12
Capítulo 2: Modelos de Dados Agregados	13
2.1 Agregados	14
2.1.1 Exemplo de relações e agregados	14
2.1.2 Consequências da Orientação Agregada	19...
2.2 Modelos de dados de chave-valor e documento	20
2.3 Lojas da família de colunas	21
2.4 Resumindo bancos de dados orientados a agregados	23
2.5 Leituras Adicionais	24
2.6 Pontos-chave	24
Capítulo 3: Mais detalhes sobre modelos de dados	25
3.1 Relacionamentos	25
3.2 Bancos de dados de grafos	26

3.3 Bancos de dados sem esquemas	28 3.4
Visualizações materializadas	30 3.5 Modelagem para
acesso a dados	31 3.6 Pontos-
chave	36 Capítulo 4: Modelos de
distribuição	37 4.1 Servidor
único	37 4.2
Fragmentação	38 4.3 Replicação
mestre-escravo	40 4.4 Replicação ponto a
ponto	42 4.5 Combinando fragmentação e
replicação	43 4.6 Pontos-
chave	44
Capítulo 5: Consistência	47 5.1
Consistência de atualização	47
5.2 Consistência de leitura	49
5.3 Relaxando a consistência	52
5.3.1 <i>O Teorema CAP</i>	53
5.4 Relaxando a Durabilidade	56
5.5 Quóruns	57 5.6
Leitura Adicional	59 5.7 Pontos
Principais	59 Capítulo
6: Carimbos de Versão	61 6.1
Transações Comerciais e de Sistema	61 6.2
Carimbos de Versão em Vários Nós	63 6.3
Pontos Principais	65
Capítulo 7: Map-Reduce	67
7.1 Map-Reduce Básico	68 7.2
Particionamento e Combinação	69
7.3 Compondo Cálculos de Map-Reduce	72
7.3.1 <i>Um Exemplo de Map-Reduce em Dois</i>	
Estágios	73 7.3.2 <i>Incremental Map-</i>
<i>Reduce</i>	76 7.4 Leituras
Adicionais	77 7.5 Pontos
Principais	77 Parte II:
Implementar	79
Capítulo 8: Bancos de Dados de Chave-Valor	81
8.1 O que é um Armazenamento de Chave-	

8.2.1 Consistência	83
8.2.2 Transações	84
8.2.3 Recursos de consulta	84
8.2.4 Estrutura de Dados	86
8.2.5 Escala	86
8.3 Casos de uso adequados	87
8.3.1 Armazenando informações da sessão	87
8.3.2 Perfis de usuário, preferências	87
8.3.3 Dados do carrinho de compras	87
8.4 Quando não usar	87
8.4.1 Relacionamentos entre Dados	87
8.4.2 Transações multioperacionais	88
8.4.3 Consulta por Dados	88
8.4.4 Operações por Conjuntos	88
Capítulo 9: Bancos de dados de documentos	89
9.1 O que é um banco de dados de documentos?	90
9.2 Características	91
9.2.1 Consistência	91
9.2.2 Transações	92
9.2.3 Disponibilidade	93
9.2.4 Recursos de consulta	94
9.2.5 Escala	95
9.3 Casos de uso adequados	97
9.3.1 Registro de eventos	97
9.3.2 Sistemas de gerenciamento de conteúdo, plataformas de blogs	98
9.3.3 Web Analytics ou Análise em Tempo Real	98
9.3.4 Aplicações de comércio eletrônico	98
9.4 Quando não usar	98
9.4.1 Transações complexas abrangendo diferentes operações	98
9.4.2 Consultas em Estrutura Agregada Variável	98
Capítulo 10: Lojas da família de colunas	99
10.1 O que é um armazenamento de dados de família de colunas?	99
10.2 Características	100
10.2.1 Consistência	103
10.2.2 Transações	104
10.2.3 Disponibilidade	104

10.2.4 Recursos de consulta	105	10.2.5
<i>Escalonamento</i>	107	10.3 Casos de uso
adequados	107	
10.3.1 Registro de eventos	107	10.3.2 Sistemas de
gerenciamento de conteúdo, plataformas de blogs	108	
10.3.3 Contadores	108	
10.3.4 Uso expirado	108	10.4 Quando não
usar	109	
Capítulo 11: Bancos de Dados de Grafos	111	11.1 O
que é um Banco de Dados de Grafos?	111	11.2
Recursos	113	
11.2.1 Consistência	114	
11.2.2 Transações	114	
11.2.3 Disponibilidade	115	11.2.4 Recursos de
consulta	115	
11.2.5		
<i>Escalonamento</i>	119	11.3 Casos de uso
adequados	120	
11.3.1 Dados conectados	120	
11.3.2 Roteamento, despacho e serviços baseados em localização	120	11.3.3
<i>Mecanismos de recomendação</i>	121	11.4 Quando não
usar	121	
Capítulo 12: Migrações de esquema	123	12.1 Alterações de
esquema	123	12.2 Alterações de esquema em
RDBMS	123	12.2.1 Migrações para projetos Green
Field	124	12.2.2 Migrações em projetos legados
126	12.3 Alterações de esquema em um armazenamento de dados NoSQL	128
12.3.1 <i>Migração incremental</i>	130	12.3.2 Migrações em bancos de
dados de gráfico	131	
agregada	132	12.3.3 Alterando a estrutura
12.4 Leitura Adicional	132	
Principais	132	Capítulo 13: Persistência
Poliglota	133	13.1 Necessidades Disparas de
Armazenamento de Dados	133	13.2 Uso do Armazenamento
de Dados Poliglota	134	13.3 Uso do Serviço em detrimento
do Uso Direto do Armazenamento de Dados	136	

13.4 Expansão para melhor funcionalidade	136	13.5 Escolha da tecnologia certa	138
13.6 Preocupações empresariais com persistência poliglota	138	13.7 Complexidade de implantação	139
13.8 Pontos-chave	140	Capítulo 14: Além do NoSQL	141
14.1 Sistemas de arquivos	141	14.2 Origem de eventos	142
14.3 Imagem de memória	144	14.4 Controle de versão	145
14.5 Bancos de dados XML	145	14.6 Bancos de dados de objetos	146
14.7 Pontos-chave	146	Capítulo 15: Escolha do seu banco de dados	147
15.1 Produtividade do programador	147	15.2 Desempenho de acesso a dados	149
15.3 Mantendo o padrão	150	15.4 Protegendo suas apostas	150
15.5 Pontos-chave	151	15.6 Considerações finais	152
Bibliografia	153		
Índice	157		

Esta página foi deixada em branco intencionalmente

Prefácio

Passamos cerca de vinte anos no mundo da computação empresarial. Vimos muitas coisas mudarem em linguagens, arquiteturas, plataformas e processos. Mas durante todo esse tempo uma coisa permaneceu constante — bancos de dados relacionais armazenam os dados. Houve desafiadores, alguns dos quais tiveram sucesso em alguns nichos, mas, no geral, a questão do armazenamento de dados para arquitetos tem sido a questão de qual banco de dados relacional usar.

Há muito valor na estabilidade desse reinado. Os dados de uma organização duram muito mais do que seus programas (pelo menos é o que as pessoas nos dizem — já vimos muitos programas muito antigos por aí). É valioso ter um armazenamento de dados estável que seja bem compreendido e acessível a partir de muitas plataformas de programação de aplicativos.

Agora, no entanto, há um novo desafiante no pedaço sob a etiqueta de confronto de NoSQL. Ele nasceu de uma necessidade de lidar com volumes maiores de dados, o que forçou uma mudança fundamental para a construção de grandes plataformas de hardware por meio de clusters de servidores de commodities. Essa necessidade também levantou preocupações de longa data sobre as dificuldades de fazer o código do aplicativo funcionar bem com o modelo de dados relacional.

O termo “NoSQL” é muito mal definido. Ele é geralmente aplicado a uma série de bancos de dados não relacionais recentes, como Cassandra, Mongo, Neo4J e Riak.

Eles adotam dados sem esquema, rodam em clusters e têm a capacidade de negociar a consistência tradicional por outras propriedades úteis. Os defensores dos bancos de dados NoSQL alegam que eles podem construir sistemas que são mais performáticos, escalam muito melhor e são mais fáceis de programar.

Este é o primeiro toque de finados para bancos de dados relacionais ou mais um pretendente ao trono? Nossa resposta para isso é “nenhum dos dois”. Bancos de dados relacionais são uma ferramenta poderosa que esperamos usar por muitas décadas, mas vemos uma mudança profunda, pois os bancos de dados relacionais não serão os únicos bancos de dados em uso. Nossa visão é que estamos entrando em um mundo de Persistência Poliglota, onde empresas, e até mesmo aplicativos individuais, usam várias tecnologias para gerenciamento de dados. Como resultado, os arquitetos precisarão estar familiarizados com essas tecnologias e ser capazes de avaliar quais usar para diferentes necessidades.

Se não tivéssemos pensado nisso, não teríamos investido tempo e esforço escrevendo este livro.

Este livro busca dar a você informações suficientes para responder à questão de se os bancos de dados NoSQL valem a pena ser considerados seriamente para seus projetos futuros. Cada projeto é diferente, e não há como escrever uma árvore de decisão simples para escolher o armazenamento de dados certo. Em vez disso, o que estamos tentando aqui é fornecer a você informações suficientes sobre como os bancos de dados NoSQL funcionam, para que você possa fazer esses julgamentos sozinho, sem ter que vasculhar toda a web. Nós deliberadamente fizemos deste um livro pequeno, para que você possa obter esta visão geral bem rápido. Ele não responderá às suas perguntas definitivamente, mas deve restringir o intervalo de opções que você tem que considerar e ajudá-lo a entender quais perguntas você precisa fazer.

Por que os bancos de dados NoSQL são interessantes?

Vemos duas razões principais pelas quais as pessoas consideram usar um banco de dados NoSQL.

- **Produtividade no desenvolvimento de aplicativos.** Muito esforço no desenvolvimento de aplicativos é gasto no mapeamento de dados entre estruturas de dados na memória e um banco de dados relacional. Um banco de dados NoSQL pode fornecer um modelo de dados que se ajuste melhor às necessidades do aplicativo, simplificando assim essa interação e resultando em menos código para escrever, depurar e evoluir.
- **Dados em larga escala.** As organizações estão descobrindo que é valioso capturar mais dados e processá-los mais rapidamente. Elas estão descobrindo que é caro, se é que é possível, fazer isso com bancos de dados relacionais. O principal motivo é que um banco de dados relacional é projetado para rodar em uma única máquina, mas geralmente é mais econômico rodar grandes cargas de dados e computação em clusters de muitas máquinas menores e mais baratas. Muitos bancos de dados NoSQL são projetados explicitamente para rodar em clusters, então eles se encaixam melhor em cenários de big data.

O que há no livro

Dividimos este livro em duas partes. A primeira parte se concentra nos conceitos principais que achamos que você precisa saber para julgar se os bancos de dados NoSQL são relevantes para você e como eles diferem. Na segunda parte, nos concentraremos mais na implementação de sistemas com bancos de dados NoSQL.

O Capítulo 1 começa explicando por que o NoSQL teve uma ascensão tão rápida — a necessidade de processar volumes maiores de dados levou a uma mudança, em sistemas grandes, de escala vertical para escala horizontal em clusters. Isso explica um recurso importante do modelo de dados de muitos bancos de dados NoSQL — o armazenamento explícito de uma estrutura rica de dados intimamente relacionados que são acessados como uma unidade. Neste livro, chamamos esse tipo de estrutura de *agregado*.

O Capítulo 2 descreve como os agregados se manifestam em três dos principais modelos de dados no mundo NoSQL: bancos de dados de chave-valor (“Modelos de Dados de Chave-Valor e Documento”, p. 20), documentos (“Modelos de Dados de Chave-Valor e Documento”, p. 20) e famílias de colunas (“Armazenamentos de Famílias de Colunas”, p. 21). Os agregados fornecem uma unidade natural de interação para muitos tipos de aplicativos, o que melhora a execução em um cluster e facilita a programação do acesso aos dados. O Capítulo 3 muda para o lado negativo dos agregados — a dificuldade de lidar com relacionamentos (“Relacionamentos”, p. 25) entre entidades em diferentes agregados. Isso nos leva naturalmente aos bancos de dados gráficos (“Bancos de Dados Gráficos”, p. 26), um modelo de dados NoSQL que não se encaixa no campo orientado a agregados. Também analisamos a característica comum dos bancos de dados NoSQL que operam sem um esquema (“Bancos de Dados Sem Esquema”, p. 28) — um recurso que fornece alguma flexibilidade maior, mas não tanto quanto você pode pensar a princípio.

Tendo abordado o aspecto de modelagem de dados do NoSQL, passamos para a distribuição: o Capítulo 4 descreve como os bancos de dados distribuem dados para serem executados em clusters. Isso se divide em sharding (“Sharding,” p. 38) e replicação, sendo esta última replicação mestre-escravo (“Master-Slave Replication,” p. 40) ou ponto a ponto (“Peer-to-Peer Replication,” p. 42). Com os modelos de distribuição definidos, podemos então passar para a questão da consistência. Os bancos de dados NoSQL fornecem uma gama mais variada de opções de consistência do que os bancos de dados relacionais — o que é uma consequência de serem amigáveis a clusters. Então, o Capítulo 5 fala sobre como a consistência muda para atualizações (“Update Consistency,” p. 47) e leituras (“Read Consistency,” p. 49), o papel dos quóruns (“Quorums,” p. 57) e como até mesmo alguma durabilidade (“Relaxing Durability,” p. 56) pode ser negociada.

Se você já ouviu falar sobre NoSQL, certamente já ouviu falar do teorema CAP; a seção “O Teorema CAP” na p. 53 explica o que ele é e como ele se encaixa.

Embora esses capítulos se concentrem principalmente nos princípios de como os dados são distribuídos e mantidos consistentes, os próximos dois capítulos falam sobre algumas ferramentas importantes que fazem isso funcionar. O Capítulo 6 descreve carimbos de versão, que servem para manter o controle de alterações e detectar inconsistências. O Capítulo 7 descreve o map-reduce, que é uma maneira particular de organizar a computação paralela que se encaixa bem com clusters e, portanto, com sistemas NoSQL.

Depois de terminarmos com os conceitos, passamos para as questões de implementação, observando alguns bancos de dados de exemplo nas quatro categorias principais: O Capítulo 8 usa Riak

como um exemplo de bancos de dados de chave-valor, o Capítulo 9 toma o MongoDB como um exemplo para bancos de dados de documentos, o Capítulo 10 escolhe o Cassandra para explorar bancos de dados de família de colunas e, finalmente, o Capítulo 11 seleciona o Neo4J como um exemplo de bancos de dados de gráficos. Deveremos enfatizar que este não é um estudo abrangente — há muitos por aí para escrever, muito menos para tentarmos. Nem nossa escolha de exemplos implica em recomendações. Nossa objetivo aqui é dar a você uma ideia da variedade de armazenamentos que existem e de como diferentes tecnologias de banco de dados usam os conceitos que descrevemos anteriormente. Você verá que tipo de código precisa escrever para programar nesses sistemas e terá um vislumbre da mentalidade necessária para usá-los.

Uma declaração comum sobre bancos de dados NoSQL é que, como eles não têm esquema, não há dificuldade em alterar a estrutura dos dados durante a vida de um aplicativo. Nós discordamos — um banco de dados sem esquema ainda tem um esquema implícito que precisa de disciplina de mudança quando você o implementa, então o Capítulo 12 explica como fazer a migração de dados tanto para esquemas fortes quanto para sistemas sem esquema.

Tudo isso deve deixar claro que o NoSQL não é uma coisa única, nem é algo que substituirá bancos de dados relacionais. O Capítulo 13 analisa esse mundo futuro da Persistência Poliglota, onde vários mundos de armazenamento de dados coexistem, mesmo dentro do mesmo aplicativo. O Capítulo 14 expande nossos horizontes além deste livro, considerando outras tecnologias que não cobrimos e que também podem fazer parte desse mundo persistente poliglota.

Com todas essas informações, você finalmente chegou a um ponto em que pode fazer uma escolha de quais tecnologias de armazenamento de dados usar, então nosso capítulo final (Capítulo 15, “Escolhendo seu banco de dados”, p. 147) oferece alguns conselhos sobre como pensar sobre essas escolhas. Em nossa visão, há dois fatores principais: encontrar um modelo de programação produtivo em que o modelo de armazenamento de dados esteja bem alinhado ao seu aplicativo e garantir que você possa obter o desempenho de acesso a dados e a resiliência de que precisa. Como estamos nos primeiros dias da história de vida do NoSQL, tememos não ter um procedimento bem definido a seguir, e você precisará testar suas opções no contexto de suas necessidades.

Esta é uma breve visão geral — fomos muito deliberados em limitar o tamanho deste livro. Selecionamos as informações que achamos mais importantes — para que você não precise fazer isso. Se você for investigar seriamente essas tecnologias, precisará ir além do que cobrimos aqui, mas esperamos que este livro forneça um bom contexto para começar seu caminho.

Também precisamos enfatizar que este é um campo muito volátil da indústria de computadores. Aspectos importantes dessas lojas mudam a cada ano — novos recursos, novos bancos de dados. Fizemos um grande esforço para focar em conceitos, que achamos que serão valiosos para entender, mesmo quando a tecnologia subjacente mudar. Estamos bastante confiantes de que a maior parte do que dizemos terá essa longevidade, mas absolutamente certos de que nem tudo terá.

Quem deve ler este livro

Nosso público-alvo para este livro são pessoas que estão considerando usar alguma forma de banco de dados NoSQL. Isso pode ser para um novo projeto ou porque estão encontrando barreiras que sugerem uma mudança em um projeto existente.

Nosso objetivo é fornecer informações suficientes para você saber se a tecnologia NoSQL faz sentido para suas necessidades e, em caso afirmativo, qual ferramenta explorar com mais profundidade. Nosso público-alvo imaginado primário é um arquiteto ou líder técnico, mas achamos que este livro também é valioso para pessoas envolvidas em gerenciamento de software que querem ter uma visão geral desta nova tecnologia. Também achamos que se você é um desenvolvedor que quer uma visão geral desta tecnologia, este livro será um bom ponto de partida.

Não entramos em detalhes sobre programação e implantação de bancos de dados específicos aqui — deixamos isso para livros especializados. Também fomos muito firmes em um limite de páginas, para manter este livro como uma breve introdução. Este é o tipo de livro que achamos que você deveria ser capaz de ler em um voo de avião: ele não responderá a todas as suas perguntas, mas deve lhe dar um bom conjunto de perguntas a serem feitas.

Se você já se aprofundou no mundo do NoSQL, este livro provavelmente não adicionará nenhum item novo ao seu estoque de conhecimento. No entanto, ele ainda pode ser útil para ajudar você a explicar o que aprendeu para os outros. Entender as questões em torno do NoSQL é importante — principalmente se você estiver tentando persuadir alguém a considerar usar o NoSQL em um projeto.

O que são os bancos de dados

Neste livro, seguimos uma abordagem comum de categorizar bancos de dados NoSQL de acordo com seu modelo de dados. Aqui está uma tabela dos quatro modelos de dados e alguns dos bancos de dados que se encaixam em cada modelo. Esta não é uma lista abrangente — ela menciona apenas os bancos de dados mais comuns que encontramos. No momento da escrita, você pode encontrar listas mais abrangentes em <http://nosql-database.org> e <http://nosql.mypopescu.com/kb/nosql>. Para cada categoria, marcamos em itálico o banco de dados que usamos como exemplo no capítulo relevante.

Nosso objetivo é escolher uma ferramenta representativa de cada uma das categorias dos bancos de dados. Enquanto falamos sobre exemplos específicos, a maior parte da discussão deve se aplicar a toda a categoria, mesmo que esses produtos sejam únicos e não possam ser generalizados como tal. Escolheremos um banco de dados para cada um dos bancos de dados de chave-valor, documento, família de colunas e gráfico; quando apropriado, mencionaremos outros produtos que podem atender a uma necessidade de recurso específica.

Modelo de Dados	Bancos de dados de exemplo
Chave-Valor (“Bancos de Dados Chave-Valor”, p. 81)	BerkeleyDB NívelDB Memcached Projeto Voldemort Redis <i>Riak</i>
Documento (“Bancos de dados de documentos”, p. 89)	CouchDB <i>MongoDB</i> OrientDB RavenDB Terrastore
Família de colunas (“Armazenamentos de família de colunas”, p. 99)	Amazon SimpleDB <i>Cassandra</i> Base H Hipertable
Gráfico (“Bancos de dados de gráficos”, p. 111)	Banco de Dados do Robô HyperGraphDB Grafo Infinito <i>Neo4J</i> OrientDB

Essa classificação por modelo de dados é útil, mas grosseira. As linhas entre os diferentes modelos de dados, como a distinção entre bancos de dados de chave-valor e de documento (“Modelos de Dados de Chave-Valor e Documento”, p. 20), são frequentemente confusas. Muitos bancos de dados não se encaixam perfeitamente em categorias; por exemplo, o OrientDB se autodenomina um banco de dados de documentos e um banco de dados de gráficos.

Agradecimentos

Nossos primeiros agradecimentos vão para nossos colegas da ThoughtWorks, muitos dos quais têm aplicado o NoSQL em nossos projetos de entrega nos últimos dois anos. Suas experiências têm sido uma fonte primária tanto de nossa motivação para escrever este livro quanto de informações práticas sobre o valor desta tecnologia. O positivo

A experiência que tivemos até agora com armazenamentos de dados NoSQL é a base da nossa visão de que esta é uma tecnologia importante e uma mudança significativa no armazenamento de dados.

Gostaríamos também de agradecer a vários grupos que deram palestras públicas, publicaram artigos e blogs sobre o uso do NoSQL. Muito progresso no desenvolvimento de software fica escondido quando as pessoas não compartilham com seus pares o que aprenderam. Agradecimentos especiais aqui vão para o Google e a Amazon, cujos artigos sobre Bigtable e Dynamo foram muito influentes para dar início ao movimento NoSQL. Também agradecemos às empresas que patrocinaram e contribuíram para o desenvolvimento de código aberto de bancos de dados NoSQL. Uma diferença interessante com mudanças anteriores no armazenamento de dados é o grau em que o movimento NoSQL está enraizado no trabalho de código aberto.

Agradecimentos especiais à ThoughtWorks por nos dar tempo para trabalhar neste livro. Nós nos juntamos à ThoughtWorks mais ou menos na mesma época e estamos aqui há mais de uma década. A ThoughtWorks continua sendo um lar muito hospitalar para nós, uma fonte de conhecimento e prática, e um ambiente acolhedor para compartilhar abertamente o que aprendemos — tão diferente das organizações tradicionais de entrega de sistemas.

Bethany Anders-Beck, Ilias Bartolini, Tim Berglund, Duncan Craig, Paul Duvall, Oren Eini, Perryn Fowler, Michael Hunger, Eric Kascic, Joshua Kerievsky, Anand Krishnaswamy, Bobby Norton, Ade Oshineye, Thiyagu Palanisamy, Prasanna Pendse, Dan Pritchett, David Rice, Mike Roberts, Marko Rodriguez, Andrew Slocum, Toby Tripp, Steve Vinoski, Dean Wampler, Jim Webber e Wee Witthawaskul revisaram os primeiros rascunhos deste livro e nos ajudaram a melhorá-lo com seus conselhos.

Além disso, Pramod gostaria de agradecer à Biblioteca Schaumburg por fornecer um ótimo serviço e um espaço tranquilo para escrever; a Arhana e Arula, minhas lindas filhas, por compreenderem que o papai iria à biblioteca e não as levaria; a Rupali, minha amada esposa, por seu imenso apoio e ajuda em me manter focado.

Esta página foi deixada em branco intencionalmente

Parte I

Entender

Esta página foi deixada em branco intencionalmente

Capítulo 1

Por que NoSQL?

Por quase tanto tempo quanto estamos na profissão de software, bancos de dados relacionais têm sido a escolha padrão para armazenamento de dados sérios, especialmente no mundo de aplicativos corporativos. Se você é um arquiteto iniciando um novo projeto, sua única escolha provavelmente será qual banco de dados relacional usar. (E muitas vezes nem isso, se sua empresa tem um fornecedor dominante.) Houve momentos em que uma tecnologia de banco de dados ameaçou tomar uma parte da ação, como bancos de dados de objetos na década de 1990, mas essas alternativas nunca chegaram a lugar nenhum.

Após um período tão longo de dominância, a excitação atual sobre bancos de dados NoSQL é uma surpresa. Neste capítulo, exploraremos por que bancos de dados relacionais se tornaram tão dominantes e por que achamos que a ascensão atual de bancos de dados NoSQL não é um fenômeno passageiro.

1.1 O valor dos bancos de dados relacionais

Bancos de dados relacionais se tornaram uma parte tão incorporada da nossa cultura de computação que é fácil considerá-los garantidos. Portanto, é útil revisitar os benefícios que eles fornecem.

1.1.1 Obtendo dados persistentes

Provavelmente o valor mais óbvio de um banco de dados é manter grandes quantidades de dados persistentes. A maioria das arquiteturas de computadores tem a noção de duas áreas de memória: uma “memória principal” rápida e volátil e um “armazenamento de apoio” maior, porém mais lento. A memória principal é limitada em espaço e perde todos os dados quando você perde energia ou algo ruim acontece com o sistema operacional. Portanto, para manter os dados por perto, nós os gravamos em um armazenamento de apoio, comumente visto em um disco (embora hoje em dia esse disco possa ser memória persistente).

O armazenamento de apoio pode ser organizado de várias maneiras. Para muitos aplicativos de produtividade (como processadores de texto), é um arquivo no sistema de arquivos do sistema operacional

Capítulo 1 Por que NoSQL?

sistema. Para a maioria dos aplicativos empresariais, no entanto, o armazenamento de apoio é um banco de dados. O banco de dados permite mais flexibilidade do que um sistema de arquivos no armazenamento de grandes quantidades de dados, de forma que um programa aplicativo obtenha pequenas partes dessas informações de forma rápida e fácil.

1.1.2 Concorrência

Os aplicativos corporativos tendem a ter muitas pessoas olhando para o mesmo conjunto de dados ao mesmo tempo, possivelmente modificando esses dados. Na maioria das vezes, eles estão trabalhando em diferentes áreas desses dados, mas ocasionalmente operam no mesmo pedaço de dados. Como resultado, temos que nos preocupar em coordenar essas interações para evitar coisas como reservas duplas de quartos de hotel.

A simultaneidade é notoriamente difícil de acertar, com todos os tipos de erros que podem prender até os programadores mais cuidadosos. Como os aplicativos corporativos podem ter muitos usuários e outros sistemas trabalhando simultaneamente, há muito espaço para coisas ruins acontecerem. Os bancos de dados relacionais ajudam a lidar com isso controlando todo o acesso aos seus dados por meio de transações. Embora isso não seja uma cura para tudo (você ainda tem que lidar com um erro transacional quando tenta reservar um quarto que acabou de sair), o mecanismo transacional funcionou bem para conter a complexidade da simultaneidade.

As transações também desempenham um papel no tratamento de erros. Com transações, você pode fazer uma alteração e, se ocorrer um erro durante o processamento da alteração, você pode reverter a transação para limpar as coisas.

1.1.3 Integração

Os aplicativos corporativos vivem em um ecossistema rico que requer que vários aplicativos, escritos por equipes diferentes, colaborem para fazer as coisas. Esse tipo de colaboração entre aplicativos é estranho porque significa forçar os limites organizacionais humanos. Os aplicativos geralmente precisam usar os mesmos dados e as atualizações feitas por meio de um aplicativo precisam ser visíveis para os outros.

Uma maneira comum de fazer isso é a **integração de banco de dados compartilhado** [Hohpe e Woolf], onde vários aplicativos armazenam seus dados em um único banco de dados. Usar um único banco de dados permite que todos os aplicativos usem os dados uns dos outros facilmente, enquanto o controle de simultaneidade do banco de dados lida com vários aplicativos da mesma forma que lida com vários usuários em um único aplicativo.

1.1.4 Um modelo (principalmente) padrão

Os bancos de dados relacionais tiveram sucesso porque eles fornecem os principais benefícios que descrevemos anteriormente de uma forma (principalmente) padrão. Como resultado, desenvolvedores e profissionais de banco de dados podem aprender o modelo relacional básico e aplicá-lo em muitos projetos. Embora existam diferenças entre diferentes bancos de dados relacionais, o núcleo

os mecanismos permanecem os mesmos: os dialetos SQL de diferentes fornecedores são semelhantes, as transações operam basicamente da mesma maneira.

1.2 Incompatibilidade de Impedância

Bancos de dados relacionais oferecem muitas vantagens, mas não são de forma alguma perfeitos. Desde os primeiros dias, houve muitas frustrações com eles.

Para desenvolvedores de aplicativos, a maior frustração tem sido o que é comumente chamado de **incompatibilidade de impedância**: a diferença entre o modelo relacional e as estruturas de dados na memória. O modelo de dados relacional organiza os dados em uma estrutura de tabelas e linhas, ou mais propriamente, relações e tuplas. No modelo relacional, uma **tupla** é um conjunto de pares nome-valor e uma **relação** é um conjunto de tuplas.

(A definição relacional de uma tupla é ligeiramente diferente daquela em matemática e muitas linguagens de programação com um tipo de dados de tupla, onde uma tupla é uma sequência de valores.) Todas as operações em SQL consomem e retornam relações, o que leva à álgebra relacional matematicamente elegante.

Essa fundação em relações fornece uma certa elegância e simplicidade, mas também introduz limitações. Em particular, os valores em uma tupla relacional têm que ser simples — eles não podem conter nenhuma estrutura, como um registro aninhado ou uma lista.

Essa limitação não é verdadeira para estruturas de dados na memória, que podem assumir estruturas muito mais ricas do que relações. Como resultado, se você quiser usar uma estrutura de dados na memória mais rica, você tem que traduzi-la para uma representação relacional para armazená-la em disco. Daí a incompatibilidade de impedância — duas representações diferentes que exigem tradução (veja Figura 1.1).

A incompatibilidade de impedância é uma grande fonte de frustração para desenvolvedores de aplicativos e, na década de 1990, muitas pessoas acreditavam que isso levaria à substituição de bancos de dados relacionais por bancos de dados que replicam as estruturas de dados na memória para o disco. Essa década foi marcada pelo crescimento das linguagens de programação orientadas a objetos e, com elas, vieram os bancos de dados orientados a objetos — ambos parecendo ser o ambiente dominante para o desenvolvimento de software no novo milênio.

No entanto, enquanto as linguagens orientadas a objetos tiveram sucesso em se tornar a principal força na programação, os bancos de dados orientados a objetos caíram na obscuridade. Os bancos de dados relacionais superaram o desafio enfatizando seu papel como um mecanismo de integração, suportado por uma linguagem de manipulação de dados (SQL) principalmente padrão e uma crescente divisão profissional entre desenvolvedores de aplicativos e administradores de bancos de dados.

A incompatibilidade de impedância tornou-se muito mais fácil de lidar pela ampla disponibilidade de estruturas de mapeamento objeto-relacional, como Hibernate e iBATIS que implementam padrões de mapeamento bem conhecidos [Fowler PoEAA], mas o problema de mapeamento ainda é um problema. As estruturas de mapeamento objeto-relacional removem

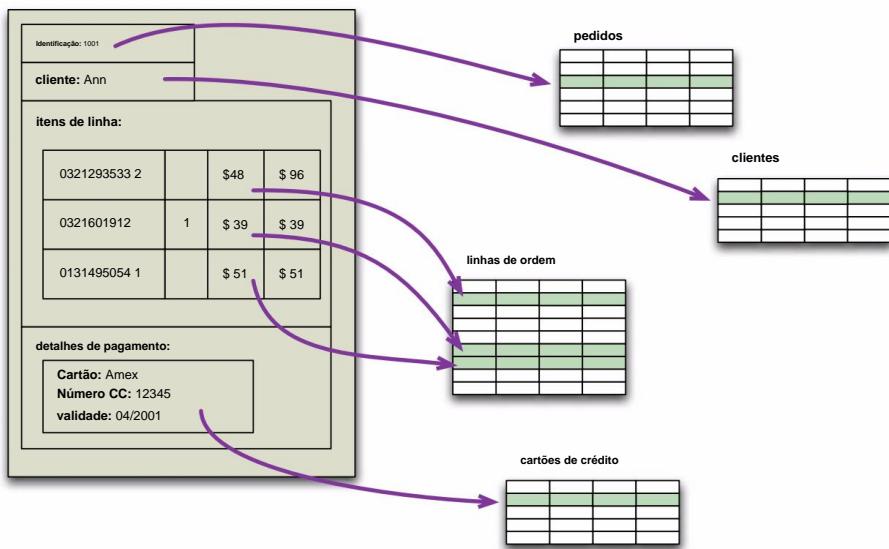


Figura 1.1 Um pedido, que se parece com uma única estrutura agregada na IU, é dividido em muitas linhas de muitas tabelas em um banco de dados relacional

muito trabalho pesado, mas pode se tornar um problema quando as pessoas tentam ignorar demais o banco de dados e o desempenho da consulta é prejudicado.

Os bancos de dados relacionais continuaram a dominar o mundo da computação empresarial na década de 2000, mas durante essa década começaram a surgir rachaduras em seu domínio.

1.3 Bancos de Dados de Aplicação e Integração

As razões exatas pelas quais os bancos de dados relacionais triunfaram sobre os bancos de dados OO ainda são o assunto de um debate ocasional de pub para desenvolvedores de uma certa idade. Mas, em nossa visão, o fator principal foi o papel do SQL como um mecanismo de integração entre aplicativos. Nesse cenário, o banco de dados atua como um **banco de dados de integração — com** vários aplicativos, geralmente desenvolvidos por equipes separadas, armazenando seus dados em um banco de dados comum. Isso melhora a comunicação porque todos os aplicativos estão operando em um conjunto consistente de dados persistentes.

Há desvantagens na integração de banco de dados compartilhado. Uma estrutura que é projetada para integrar muitos aplicativos acaba sendo mais complexa — na verdade, muitas vezes dramaticamente mais complexa — do que qualquer aplicativo individual precisa. Além disso, se um aplicativo quiser fazer alterações em seu armazenamento de dados, ele precisa coordenar com todos os outros aplicativos que usam o banco de dados. Diferentes aplicativos têm diferentes necessidades estruturais e de desempenho, então um índice exigido por um

1.3 Bancos de Dados de Aplicação e Integração

aplicativo pode causar um hit problemático em inserções para outro. O fato de que cada aplicativo é geralmente uma equipe separada também significa que o banco de dados geralmente não pode confiar em aplicativos para atualizar os dados de uma forma que preserve a integridade do banco de dados e, portanto, precisa assumir a responsabilidade por isso dentro do próprio banco de dados.

Uma abordagem diferente é tratar seu banco de dados como um **banco de dados de aplicativo** — que é acessado diretamente apenas por uma única base de código de aplicativo que é cuidada por uma única equipe. Com um banco de dados de aplicativo, apenas a equipe que usa o aplicativo precisa saber sobre a estrutura do banco de dados, o que torna muito mais fácil manter e evoluir o esquema. Como a equipe do aplicativo controla o banco de dados e o código do aplicativo, a responsabilidade pela integridade do banco de dados pode ser colocada no código do aplicativo.

As preocupações com a interoperabilidade agora podem mudar para as interfaces do aplicativo, permitindo melhores protocolos de interação e fornecendo suporte para alterá-los. Durante os anos 2000, vimos uma mudança distinta para serviços web [Daigneau], onde os aplicativos se comunicariam por HTTP. Os serviços web permitiram uma nova forma de um mecanismo de comunicação amplamente usado — um desafio ao uso do SQL com bancos de dados compartilhados. (Muito desse trabalho foi feito sob a bandeira da “Arquitetura Orientada a Serviços” — um termo mais notável por sua falta de um significado consistente.)

Um aspecto interessante dessa mudança para serviços web como um mecanismo de integração foi que isso resultou em mais flexibilidade para a estrutura dos dados que estavam sendo trocados. Se você se comunicar com SQL, os dados devem ser estruturados como relações. No entanto, com um serviço, você pode usar estruturas de dados mais ricas com registros e listas aninhados. Eles geralmente são representados como documentos em XML ou, mais recentemente, JSON. Em geral, com a comunicação remota, você deseja reduzir o número de viagens de ida e volta envolvidas na interação, então é útil poder colocar uma estrutura rica de informações em uma única solicitação ou resposta.

Se você for usar serviços para integração, na maioria das vezes serviços web — usando texto sobre HTTP — são o caminho a seguir. No entanto, se você estiver lidando com interações altamente sensíveis ao desempenho, pode precisar de um protocolo binário.

Faça isso somente se tiver certeza de que é necessário, pois os protocolos de texto são mais fáceis de trabalhar — considere o exemplo da Internet.

Depois de tomar a decisão de usar um banco de dados de aplicativo, você tem mais liberdade para escolher um banco de dados. Como há um desacoplamento entre seu banco de dados interno e os serviços com os quais você fala com o mundo externo, o mundo externo não precisa se importar com a forma como você armazena seus dados, permitindo que você considere opções não relacionais. Além disso, há muitos recursos de bancos de dados relacionais, como segurança, que são menos úteis para um banco de dados de aplicativo porque podem ser feitos pelo aplicativo envolvente.

Apesar dessa liberdade, no entanto, não ficou claro que os bancos de dados de aplicativos levaram a uma grande corrida para armazenamentos de dados alternativos. A maioria das equipes que adotaram a abordagem de banco de dados de aplicativos ficou com bancos de dados relacionais.

Afinal, usar um banco de dados de aplicativos produz muitas vantagens, mesmo ignorando a flexibilidade do banco de dados (é por isso que geralmente o recomendamos). Os bancos de dados relacionais são familiares e geralmente funcionam muito bem ou, pelo menos, bem o suficiente. Talvez, com o tempo, pos-

pode ter visto a mudança para bancos de dados de aplicativos para abrir uma rachadura real na hegemonia relacional — mas essas rachaduras vieram de outra fonte.

1.4 Ataque dos Clusters

No início do novo milênio, o mundo da tecnologia foi atingido pelo estouro da bolha pontocom dos anos 1990. Enquanto isso fez com que muitas pessoas questionassem o futuro econômico da Internet, os anos 2000 viram várias grandes propriedades da web aumentarem drasticamente em escala.

Esse aumento de escala estava acontecendo em muitas dimensões. Os sites começaram a rastrear a atividade e a estrutura de forma muito detalhada. Grandes conjuntos de dados apareceram: links, redes sociais, atividade em logs, dados de mapeamento. Com esse crescimento de dados, veio um crescimento de usuários — à medida que os maiores sites cresceram para se tornarem vastas propriedades que atendem regularmente a um grande número de visitantes.

Lidar com o aumento de dados e tráfego exigiu mais recursos de computação.

Para lidar com esse tipo de aumento, você tem duas escolhas: para cima ou para fora. Escalar para cima implica em máquinas maiores, mais processadores, armazenamento em disco e memória. Mas máquinas maiores ficam cada vez mais caras, sem mencionar que há limites reais conforme seu tamanho aumenta. A alternativa é usar muitas máquinas pequenas em um cluster.

Um cluster de pequenas máquinas pode usar hardware de commodity e acaba sendo mais barato nesses tipos de escalas. Ele também pode ser mais resiliente — embora falhas individuais de máquinas sejam comuns, o cluster geral pode ser construído para continuar funcionando apesar dessas falhas, fornecendo alta confiabilidade.

À medida que grandes propriedades migraram para clusters, isso revelou um novo problema: bancos de dados relacionais não são projetados para serem executados em clusters. Bancos de dados relacionais em cluster, como o Oracle RAC Server, funcionam no conceito de um subsistema de disco compartilhado. Eles usam um sistema de arquivos com reconhecimento de cluster que grava em um subsistema de disco altamente disponível, mas isso significa que o cluster ainda tem o subsistema de disco como um único ponto de falha. Bancos de dados relacionais também podem ser executados como servidores separados para diferentes conjuntos de dados, efetivamente fragmentando (“Sharding”, p. 38) o banco de dados. Embora isso separe a carga, todo o fragmentamento tem que ser controlado pelo aplicativo, que tem que manter o controle de qual servidor de banco de dados conversar para cada bit de dados.

Além disso, perdemos qualquer consulta, integridade referencial, transações ou controles de consistência que cruzam shards. Uma frase que ouvimos frequentemente neste contexto de pessoas que fizeram isso é “atos não naturais”.

Esses problemas técnicos são exacerbados pelos custos de licenciamento. Bancos de dados relacionais comerciais geralmente são especificados com base em uma suposição de servidor único, então executar em um cluster aumentou os preços e levou a negociações frustrantes com os departamentos de compras.

Essa incompatibilidade entre bancos de dados relacionais e clusters levou algumas organizações a considerar uma rota alternativa para armazenamento de dados. Duas empresas em particular — Google e Amazon — foram muito influentes. Ambas estavam na vanguarda da execução de grandes clusters desse tipo; além disso, estavam capturando

enormes quantidades de dados. Essas coisas deram a eles o motivo. Ambas eram empresas bem-sucedidas e em crescimento, com fortes componentes técnicos, o que lhes deu os meios e a oportunidade. Não era de se admirar que tivessem assassinato em mente para seus bancos de dados relacionais. À medida que os anos 2000 se aproximavam, ambas as empresas produziram artigos breves, mas altamente influentes, sobre seus esforços: BigTable do Google e Dynamo da Amazon.

Costuma-se dizer que a Amazon e o Google operam em escalas muito distantes da maioria das organizações, então as soluções que elas precisavam podem não ser relevantes para uma organização média. Embora seja verdade que a maioria dos projetos de software não precisa desse nível de escala, também é verdade que mais e mais organizações estão começando a explorar o que podem fazer capturando e processando mais dados — e a se deparar com os mesmos problemas. Então, conforme mais informações vazavam sobre o que o Google e a Amazon tinham feito, as pessoas começaram a explorar a criação de bancos de dados em linhas semelhantes — explicitamente projetados para viver em um mundo de clusters. Enquanto as ameaças anteriores ao domínio relacional acabaram sendo fantasmas, a ameaça dos clusters era séria.

1.5 O Surgimento do NoSQL

É uma ironia maravilhosa que o termo “NoSQL” tenha aparecido pela primeira vez no final dos anos 90 como o nome de um banco de dados relacional de código aberto [Strozzi NoSQL]. Liderado por Carlo Strozzi, esse banco de dados armazena suas tabelas como arquivos ASCII, cada tupla representada por uma linha com campos separados por tabulações. O nome vem do fato de que o banco de dados não usa SQL como uma linguagem de consulta. Em vez disso, o banco de dados é manipulado por meio de scripts de shell que podem ser combinados nos pipelines UNIX usuais. Além da coincidência terminológica, o NoSQL de Strozzi não teve influência nos bancos de dados que descrevemos neste livro.

O uso de “NoSQL” que reconhecemos hoje remonta a um encontro em 11 de junho de 2009 em São Francisco organizado por Johan Oskarsson, um desenvolvedor de software baseado em Londres. O exemplo do BigTable e do Dynamo inspirou vários projetos experimentando armazenamento de dados alternativo, e as discussões sobre eles se tornaram uma característica das conferências de melhor software naquela época. Johan estava interessado em descobrir mais sobre alguns desses novos bancos de dados enquanto estava em São Francisco para uma cúpula do Hadoop. Como tinha pouco tempo lá, ele sentiu que não seria viável visitar todos eles, então decidiu sediar um encontro onde todos pudessem se reunir e apresentar seu trabalho para quem estivesse interessado.

Johan queria um nome para o meetup — algo que daria uma boa hashtag no Twitter: curto, memorável e sem muitos resultados no Google para que uma busca pelo nome encontrasse o meetup rapidamente. Ele pediu sugestões no canal IRC #cassandra e recebeu algumas, selecionando a sugestão de “NoSQL” de Eric Evans (um desenvolvedor na Rackspace, sem conexão com o DDD Eric

Evans). Embora tivesse a desvantagem de ser negativo e não descrever realmente esses sistemas, ele se encaixava nos critérios da hashtag. Na época, eles estavam pensando em nomear apenas uma única reunião e não esperavam que pegasse para nomear toda essa tendência tecnológica [Oskarsson].

O termo “NoSQL” pegou como fogo, mas nunca foi um termo que teve muito em termos de uma definição forte. A chamada original [NoSQL Meetup] para o meetup pedia por “bancos de dados não relacionais, distribuídos e de código aberto”. As palestras lá [NoSQL Debrief] foram de Voldemort, Cassandra, Dynomite, HBase, Hypertable, CouchDB e MongoDB — mas o termo nunca foi confinado àquele septeto original. Não há uma definição geralmente aceita, nem uma autoridade para fornecer uma, então tudo o que podemos fazer é discutir algumas características comuns dos bancos de dados que tendem a ser chamados de “NoSQL”.

Para começar, há o ponto óbvio de que os bancos de dados NoSQL não usam SQL. Alguns deles têm linguagens de consulta, e faz sentido que sejam semelhantes ao SQL para torná-los mais fáceis de aprender. O CQL de Cassandra é assim — “exatamente como SQL (exceto onde não é)” [CQL]. Mas até agora nenhum implementou nada que se encaixasse nem mesmo na noção bastante flexível de SQL padrão. Será interessante ver o que acontece se um banco de dados NoSQL estabelecido decidir implementar um SQL razoavelmente padrão; o único resultado previsível para tal eventualidade é muita argumentação.

Outra característica importante desses bancos de dados é que eles são geralmente projetos de código aberto. Embora o termo NoSQL seja frequentemente aplicado a sistemas de código fechado, há uma noção de que NoSQL é um fenômeno de código aberto.

A maioria dos bancos de dados NoSQL é movida pela necessidade de rodar em clusters, e isso certamente é verdade para aqueles que foram discutidos durante o meetup inicial. Isso tem um efeito em seu modelo de dados, bem como em sua abordagem de consistência. Bancos de dados relacionais usam transações ACID (p. 19) para lidar com a consistência em todo o banco de dados. Isso entra em conflito inerentemente com um ambiente de cluster, então os bancos de dados NoSQL oferecem uma gama de opções para consistência e distribuição.

No entanto, nem todos os bancos de dados NoSQL são fortemente orientados para execução em clusters. Bancos de dados de grafos são um estilo de bancos de dados NoSQL que usa um modelo de distribuição semelhante a bancos de dados relacionais, mas oferece um modelo de dados diferente que o torna melhor no manuseio de dados com relacionamentos complexos.

Os bancos de dados NoSQL geralmente são baseados nas necessidades dos domínios da web do início do século XXI, então, normalmente, apenas os sistemas desenvolvidos durante esse período são chamados de NoSQL, descartando assim as hordas de bancos de dados criados antes do novo milênio, e muito menos antes de Cristo (antes de Codd).

Os bancos de dados NoSQL operam sem um esquema, permitindo que você adicione campos livremente aos registros do banco de dados sem precisar definir nenhuma alteração na estrutura primeiro. Isso é particularmente útil ao lidar com dados não uniformes e campos personalizados que forçam bancos de dados relacionais a usar nomes como customField6 ou tabelas de campos personalizados que são difíceis de processar e entender.

Todas as opções acima são características comuns de coisas que vemos descritas como bancos de dados NoSQL. Nenhuma delas é definicional e, de fato, é provável que haja

nunca será uma definição coerente de “NoSQL” (suspiro). No entanto, esse conjunto bruto de características tem sido nosso guia na escrita deste livro. Nossa principal entusiasmo com esse assunto é que a ascensão do NoSQL abriu o leque de opções para armazenamento de dados. Consequentemente, essa abertura não deve ser confinada ao que é geralmente classificado como um armazenamento NoSQL. Esperamos que outras opções de armazenamento de dados se tornem mais aceitáveis, incluindo muitas que são anteriores ao movimento NoSQL.

No entanto, há um limite para o que podemos discutir de forma útil neste livro, então decidimos nos concentrar nesta noDefinition.

Quando você ouve “NoSQL” pela primeira vez, uma pergunta imediata é o que isso significa — um “não” ao SQL? A maioria das pessoas que falam sobre NoSQL diz que ele realmente significa “Não apenas SQL”, mas essa interpretação tem alguns problemas. A maioria das pessoas escreve “NoSQL”, enquanto “Não apenas SQL” seria escrito “NOSQL”. Além disso, não faria muito sentido chamar algo de banco de dados NoSQL sob o significado de “não apenas” — porque então, Oracle ou Postgres se encaixariam nessa definição, provariam que preto é igual a branco e todos seríamos atropelados nas faixas de pedestres.

Para resolver isso, sugerimos que você não se preocupe com o que o termo representa, mas sim com o que ele significa (o que é recomendado para a maioria das siglas).

Assim, quando “NoSQL” é aplicado a um banco de dados, ele se refere a um conjunto mal definido de bancos de dados, em sua maioria de código aberto, desenvolvidos principalmente no início do século XXI e que, em sua maioria, não usam SQL.

A interpretação “não-somente” tem seu valor, pois descreve o ecossistema que muitas pessoas pensam ser o futuro dos bancos de dados. Isso é, de fato, o que consideramos ser a contribuição mais importante dessa maneira de pensar — é melhor pensar no NoSQL como um movimento do que como uma tecnologia. Não achamos que os bancos de dados relacionais vão desaparecer — eles ainda serão a forma mais comum de banco de dados em uso. Embora tenhamos escrito este livro, ainda recomendamos bancos de dados relacionais. Sua familiaridade, estabilidade, conjunto de recursos e suporte disponível são argumentos convincentes para a maioria dos projetos.

A mudança é que agora vemos bancos de dados relacionais como uma opção para armazenamento de dados. Esse ponto de vista é frequentemente chamado de **persistência poliglota** — usar diferentes armazenamentos de dados em diferentes circunstâncias. Em vez de simplesmente escolher um banco de dados relacional porque todo mundo faz isso, precisamos entender a natureza dos dados que estamos armazenando e como queremos manipulá-los. O resultado é que a maioria das organizações terá uma mistura de tecnologias de armazenamento de dados para diferentes circunstâncias.

Para fazer esse mundo poliglota funcionar, nossa visão é que as organizações também precisam mudar de bancos de dados de integração para bancos de dados de aplicativos. De fato, assumimos neste livro que você usará um banco de dados NoSQL como um banco de dados de aplicativo; geralmente não consideramos bancos de dados NoSQL uma boa escolha para bancos de dados de integração. Não vemos isso como uma desvantagem, pois achamos que, mesmo que você não use NoSQL, mudar para encapsular dados em serviços é uma boa direção a seguir.

Em nosso relato da história do desenvolvimento do NoSQL, nos concentramos em big data rodando em clusters. Embora pensemos que essa seja a principal coisa que impulsionou a abertura do mundo do banco de dados, não é a única razão pela qual vemos equipes de projeto

considerando bancos de dados NoSQL. Um motivo igualmente importante é a velha frustração com o problema de incompatibilidade de impedância. As preocupações com big data criaram uma oportunidade para as pessoas pensarem de forma renovada sobre suas necessidades de armazenamento de dados, e algumas equipes de desenvolvimento veem que usar um banco de dados NoSQL pode ajudar sua produtividade simplificando seu acesso ao banco de dados, mesmo que não precisem escalar além de uma única máquina.

Então, ao ler o restante deste livro, lembre-se de que há duas razões principais para considerar o NoSQL. Uma é lidar com acesso a dados com tamanhos e desempenho que exigem um cluster; a outra é melhorar a produtividade do desenvolvimento de aplicativos usando um estilo de interação de dados mais conveniente.

1.6 Pontos-chave

- Bancos de dados relacionais são uma tecnologia de sucesso há vinte anos, fornecendo persistência, controle de simultaneidade e um mecanismo de integração.
- Os desenvolvedores de aplicativos ficaram frustrados com a incompatibilidade de impedância entre o modelo relacional e as estruturas de dados na memória.
- Há um movimento de afastamento do uso de bancos de dados como pontos de integração para o encapsulamento de bancos de dados em aplicativos e integração por meio de serviços.
- O fator vital para uma mudança no armazenamento de dados foi a necessidade de suportar grandes volumes de dados executando em clusters. Bancos de dados relacionais não são projetados para executar eficientemente em clusters.
- NoSQL é um neologismo acidental. Não há uma definição prescritiva — tudo o que você pode fazer é uma observação de características comuns.
- As características comuns dos bancos de dados NoSQL são
 - Não usar o modelo relacional
 - Funcionando bem em clusters
 - Código aberto
 - Construído para os web estates do século XXI
 - Sem esquema
- O resultado mais importante da ascensão do NoSQL é a Persistência Poliglota.

Capítulo 2

Modelos de Dados Agregados

Um modelo de dados é o modelo através do qual percebemos e manipulamos nossos dados. Para pessoas que usam um banco de dados, o modelo de dados descreve como interagimos com os dados no banco de dados. Isso é diferente de um modelo de armazenamento, que descreve como o banco de dados armazena e manipula os dados internamente. Em um mundo ideal, deveríamos ignorar o modelo de armazenamento, mas na prática precisamos de pelo menos alguma noção dele — principalmente para atingir um desempenho decente.

Em conversação, o termo “modelo de dados” geralmente significa o modelo dos dados específicos em um aplicativo. Um desenvolvedor pode apontar para um diagrama entidade-relacionamento de seu banco de dados e se referir a ele como seu modelo de dados contendo clientes, pedidos, produtos e similares. No entanto, neste livro, usaremos principalmente “modelo de dados” para nos referirmos ao modelo pelo qual o banco de dados organiza os dados — o que pode ser chamado mais formalmente de metamodelo.

O modelo de dados dominante das últimas duas décadas é o modelo de dados relacional, que é melhor visualizado como um conjunto de tabelas, mais como uma página de uma planilha. Cada tabela tem linhas, com cada linha representando alguma entidade de interesse. Descrevemos essa entidade por meio de colunas, cada uma com um único valor. Uma coluna pode se referir a outra linha na mesma tabela ou em tabelas diferentes, o que constitui um relacionamento entre essas entidades. (Estamos usando terminologia informal, mas comum, quando falamos de tabelas e linhas; os termos mais formais seriam relações e tuplas.)

Uma das mudanças mais óbvias com o NoSQL é um afastamento do modelo relacional. Cada solução NoSQL tem um modelo diferente que usa, que colocamos em quatro categorias amplamente usadas no ecossistema NoSQL: chave-valor, documento, família de colunas e gráfico. Destes, os três primeiros compartilham uma característica comum de seus modelos de dados que chamaremos de orientação agregada. Neste capítulo, explicaremos o que queremos dizer com orientação agregada e o que isso significa para modelos de dados.

2.1 Agregados

O modelo relacional pega as informações que queremos armazenar e as divide em tuplas (linhas).

Uma tupla é uma estrutura de dados limitada: ela captura um conjunto de valores, então você não pode aninhar uma tupla dentro de outra para obter registros aninhados, nem pode colocar uma lista de valores ou tuplas dentro de outra. Essa simplicidade sustenta o modelo relacional — ela nos permite pensar em todas as operações como operando em e retornando tuplas.

A orientação agregada adota uma abordagem diferente. Ela reconhece que, frequentemente, você deseja operar em dados em unidades que têm uma estrutura mais complexa do que um conjunto de tuplas. Pode ser útil pensar em termos de um registro complexo que permite que listas e outras estruturas de registro sejam aninhadas dentro dele. Como veremos, bancos de dados de chave-valor, documento e família de colunas fazem uso desse registro mais complexo.

No entanto, não há um termo comum para esse registro complexo; neste livro usamos o termo “agregado”.

Aggregado é um termo que vem do Domain-Driven Design [Evans]. No Domain-Driven Design, um **agregado** é uma coleção de objetos relacionados que desejamos tratar como uma unidade. Em particular, é uma unidade para manipulação de dados e gerenciamento de consistência. Normalmente, gostamos de atualizar agregados com operações atômicas e nos comunicar com nosso armazenamento de dados em termos de agregados. Esta definição combina muito bem com o funcionamento dos bancos de dados de chave-valor, documento e família de colunas. Lidar com agregados torna muito mais fácil para esses bancos de dados lidar com a operação em um cluster, uma vez que o agregado constitui uma unidade natural para replicação e fragmentação. Os agregados também costumam ser mais fáceis para os programadores de aplicativos trabalharem, uma vez que eles geralmente manipulam dados por meio de agregados.

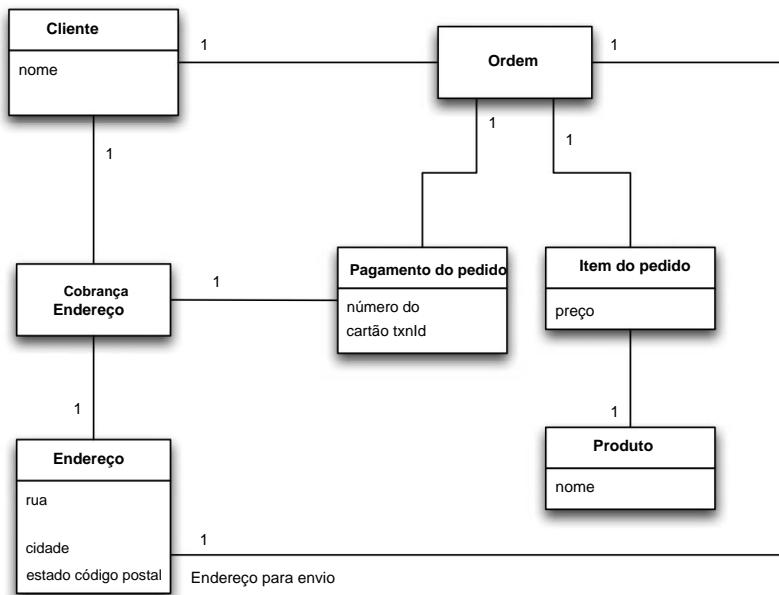
2.1.1 Exemplo de relações e agregados

Neste ponto, um exemplo pode ajudar a explicar o que estamos falando. Vamos supor que temos que construir um site de e-commerce; vamos vender itens diretamente aos clientes pela web, e teremos que armazenar informações sobre usuários, nosso catálogo de produtos, pedidos, endereços de entrega, endereços de cobrança e dados de pagamento. Podemos usar este cenário para modelar os dados usando um repositório de dados relacionais, bem como repositórios de dados NoSQL e falar sobre seus prós e contras. Para um banco de dados relacional, podemos começar com um modelo de dados mostrado na Figura 2.1.

A Figura 2.2 apresenta alguns dados amostra para este modelo.

Como somos bons soldados relacionais, tudo é adequadamente normalizado, de modo que nenhum dado é repetido em várias tabelas. Também temos integridade referencial. Um sistema de ordem realista seria naturalmente mais envolvido do que isso, mas esse é o benefício do ar rarefeito de um livro.

Agora vamos ver como esse modelo pode parecer quando pensamos mais termos orientados para agregados (Figura 2.3).



Customer	
ID	Name
1	Martin

Order		
ID	CustomerId	ShippingAddressId
99	1	77

Product	
ID	Name
27	NoSQL Distilled

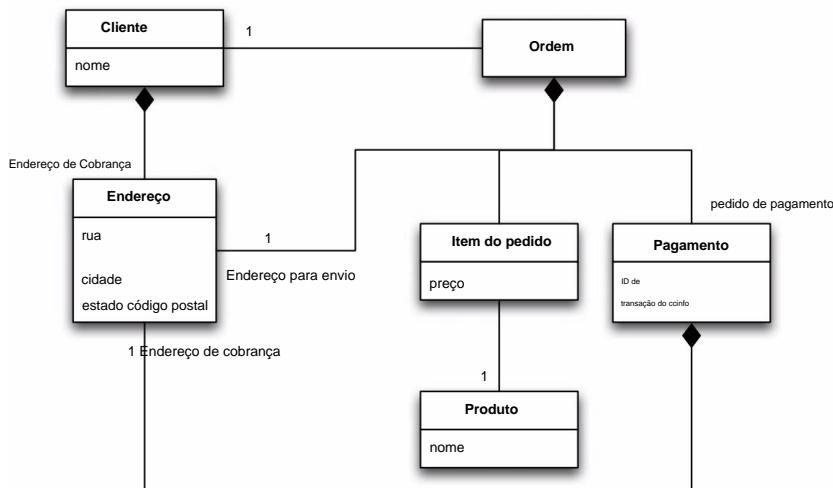
BillingAddress		
ID	CustomerId	AddressId
55	1	77

OrderItem			
ID	OrderId	ProductId	Price
100	99	27	32.45

Address	
ID	City
77	Chicago

OrderPayment				
ID	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

Figura 2.2 Dados típicos usando o modelo de dados RDBMS

**Figura 2.3** Um modelo de dados agregados

Novamente, temos alguns dados de amostra, que mostraremos no formato JSON, pois é isso uma representação comum para dados no mundo NoSQL.

```

// em clientes { "id":1,
    "name":"Martin",
    "billingAddress":[{"city":"Chicago"}] }

// em pedidos
{ "id":99,
  "customerId":1,
  "orderItems":
  [
    { "productId":27,
      "price": 32.45,
      "productName": "NoSQL Distilled"
    },
    ...
  ],
  "shippingAddress":{"city":"Chicago"} "orderPayment":
  [
    {
      "ccinfo":"1000-1000-1000-1000",
      "txnid":"abelif879rtf",
      "billingAddress": {"cidade": "Chicago"}
    }
  ],
}
  
```

Neste modelo, temos dois agregados principais: cliente e pedido. Usamos o marcador de composição black-diamond em UML para mostrar como os dados se encaixam na estrutura de agregação. O cliente contém uma lista de endereços de cobrança; o pedido contém uma lista de itens do pedido, um endereço de entrega e pagamentos. O pagamento em si contém um endereço de cobrança para esse pagamento.

Um único registro de endereço lógico aparece três vezes nos dados de exemplo, mas em vez de usar IDs, ele é tratado como um valor e copiado a cada vez. Isso se encaixa no domínio em que não queremos que o endereço de entrega, nem o endereço de cobrança do pagamento, sejam alterados. Em um banco de dados relacional, garantiríamos que as linhas de endereço não fossem atualizadas para esse caso, criando uma nova linha. Com agregados, podemos copiar toda a estrutura de endereço para o agregado conforme necessário.

O link entre o cliente e o pedido não está dentro de nenhum dos agregados — é um relacionamento entre agregados. Da mesma forma, o link de um item de pedido cruzaria para uma estrutura agregada separada para produtos, que não abordamos. Mostramos o nome do produto como parte do item do pedido aqui — esse tipo de desnormalização é semelhante às compensações com bancos de dados relacionais, mas é mais comum com agregados porque queremos minimizar o número de agregados que acessamos durante uma interação de dados.

O importante a ser notado aqui não é a maneira particular como desenhamos o limite agregado, mas sim o fato de que você tem que pensar sobre acessar esses dados — e fazer disso parte do seu pensamento ao desenvolver o modelo de dados do aplicativo. De fato, poderíamos desenhar nossos limites agregados de forma diferente, colocando todos os pedidos de um cliente no agregado de clientes (Figura 2.4).

Usando o modelo de dados acima, um exemplo de Cliente e Pedido ficaria assim:

```
// em clientes
{
  "cliente": {
    "id": 1,
    "nome": "Martin",
    "endereçoDeCobrança": [{"cidade": "Chicago"}],
    "pedidos": [
      {
        "id": 99,
        "customerId": 1,
        "orderItems": [
          {
            "productId": 27,
            "price": 32.45,
            "productName": "NoSQL Destilled"
          }
        ],
        "shippingAddress": {"city": "Chicago"}
      }
    ]
  }
}
```

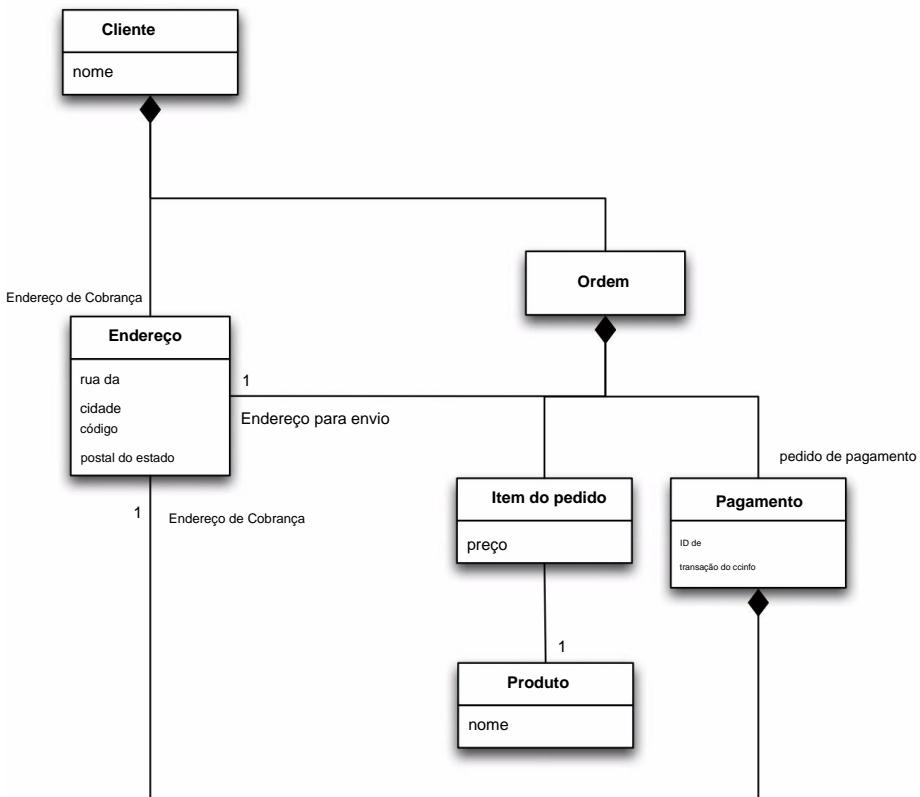


Figura 2.4 Incorpore todos os objetos para o cliente e os pedidos do cliente

"orderPayment":

```

[ { "ccinfo":"1000-1000-1000-1000",
  "txnId":"abelif879rft",
  "billingAddress": {"cidade": "Chicago"} }, ] ]
  
```

} }

Como a maioria das coisas na modelagem, não há uma resposta universal sobre como desenhar seus limites agregados. Depende inteiramente de como você tende a manipular seus dados. Se você tende a acessar um cliente junto com todos os pedidos desse cliente de uma vez, então você preferiria um único agregado. No entanto, se você tende a se concentrar em acessar um único pedido por vez, então você deve preferir ter agregados separados para cada pedido. Naturalmente, isso é muito específico do contexto; alguns

os aplicativos preferirão um ou outro, mesmo dentro de um único sistema, e é exatamente por isso que muitas pessoas preferem a ignorância agregada.

2.1.2 Consequências da Orientação Agregada

Embora o mapeamento relacional capture os vários elementos de dados e seus relacionamentos razoavelmente bem, ele o faz sem qualquer noção de uma entidade agregada.

Em nossa linguagem de domínio, podemos dizer que um pedido consiste em itens de pedido, um endereço de entrega e um pagamento. Isso pode ser expresso no modelo relacional em termos de relacionamentos de chave estrangeira — mas não há nada que distinga relacionamentos que representam agregações daqueles que não representam. Como resultado, o banco de dados não pode usar um conhecimento de estrutura agregada para ajudá-lo a armazenar e distribuir os dados.

Várias técnicas de modelagem de dados forneceram maneiras de marcar estruturas agregadas ou compostas. O problema, no entanto, é que os modeladores raramente fornecem qualquer semântica para o que torna um relacionamento agregado diferente de qualquer outro; onde há semântica, elas variam. Ao trabalhar com bancos de dados orientados a agregados, temos uma semântica mais clara a considerar ao focar na unidade de interação com o armazenamento de dados. No entanto, não é uma propriedade lógica de dados: é tudo sobre como os dados estão sendo usados pelos aplicativos — uma preocupação que geralmente está fora dos limites da modelagem de dados.

Bancos de dados relacionais não têm conceito de agregado dentro de seu modelo de dados, então os chamamos de **agregado-ignorante**. No mundo NoSQL, bancos de dados de grafos também são agregado-ignorante. Ser agregado-ignorante não é uma coisa ruim. Muitas vezes é difícil traçar bem os limites agregados, particularmente se os mesmos dados são usados em muitos contextos diferentes. Um pedido faz um bom agregado quando um cliente está fazendo e revisando pedidos, e quando o varejista está processando pedidos.

No entanto, se um varejista quiser analisar suas vendas de produtos nos últimos meses, então um agregado de pedidos se torna um problema. Para chegar ao histórico de vendas de produtos, você terá que cavar em cada agregado no banco de dados. Então, uma estrutura agregada pode ajudar com algumas interações de dados, mas ser um obstáculo para outras. Um modelo agregado-ignorante permite que você olhe facilmente para os dados de diferentes maneiras, então é uma escolha melhor quando você não tem uma estrutura primária para manipular seus dados.

O motivo decisivo para a orientação agregada é que ela ajuda muito na execução em um cluster, o que, como você deve se lembrar, é o argumento decisivo para a ascensão do NoSQL. Se estivermos executando em um cluster, precisamos minimizar quantos nós precisamos consultar quando estamos coletando dados. Ao incluir explicitamente agregados, damos ao banco de dados informações importantes sobre quais bits de dados serão manipulados juntos e, portanto, devem viver no mesmo nó.

Agregados têm uma consequência importante para transações. Bancos de dados relacionais permitem que você manipule qualquer combinação de linhas de qualquer tabela em uma única transação. Essas transações são chamadas de **transações ACID**: Atômica, Consistente, Isolada e Durável. ACID é uma sigla um tanto artificial; o ponto real é a atomicidade: Muitas linhas abrangendo muitas tabelas são atualizadas como um

operação única. Esta operação é bem-sucedida ou falha em sua totalidade, e as operações concorrentes são isoladas umas das outras, de modo que não podem ver uma atualização parcial.

É frequentemente dito que bancos de dados NoSQL não suportam transações ACID e, portanto, sacrificam a consistência. Esta é uma simplificação bastante abrangente. Em geral, é verdade que bancos de dados orientados a agregados não têm transações ACID que abrangem vários agregados. Em vez disso, eles suportam manipulação atômica de um único agregado por vez. Isso significa que, se precisarmos manipular vários agregados de forma atômica, temos que gerenciar isso nós mesmos no código do aplicativo.

Na prática, descobrimos que na maioria das vezes conseguimos manter nossas necessidades de atomicidade dentro de um único agregado; de fato, isso faz parte da consideração para decidir como dividir nossos dados em agregados. Também devemos lembrar que grafos e outros bancos de dados que ignoram agregados geralmente suportam transações ACID semelhantes a bancos de dados relacionais. Acima de tudo, o tópico de consistência é muito mais envolvente do que se um banco de dados é ACID ou não, como exploraremos no Capítulo 5.

2.2 Modelos de dados de chave-valor e documento

Dissemos anteriormente que os bancos de dados de chave-valor e documentos eram fortemente orientados a agregados. O que queríamos dizer com isso era que pensamos nesses bancos de dados como construídos principalmente por meio de agregados. Ambos os tipos de bancos de dados consistem em muitos agregados, com cada agregado tendo uma chave ou ID que é usada para obter os dados.

Os dois modelos diferem porque, em um banco de dados de chave-valor, o agregado é opaco para o banco de dados — apenas uma grande mancha de bits sem sentido. Em contraste, um banco de dados de documentos é capaz de ver uma estrutura no agregado. A vantagem da opacidade é que podemos armazenar o que quisermos no agregado. O banco de dados pode impor algum limite geral de tamanho, mas, além disso, temos liberdade completa.

Um banco de dados de documentos impõe limites sobre o que podemos colocar nele, definindo estruturas e tipos permitidos. Em troca, no entanto, obtemos mais flexibilidade no acesso.

Com um armazenamento de chave-valor, podemos acessar um agregado somente por lookup com base em sua chave. Com um banco de dados de documentos, podemos enviar consultas ao banco de dados com base nos campos no agregado, podemos recuperar parte do agregado em vez de tudo, e o banco de dados pode criar índices com base no conteúdo do agregado.

Na prática, a linha entre chave-valor e documento fica um pouco tênue. As pessoas geralmente colocam um campo de ID em um banco de dados de documentos para fazer uma pesquisa no estilo chave-valor. Bancos de dados classificados como bancos de dados de chave-valor podem permitir estruturas para dados além de apenas um agregado opaco. Por exemplo, o Riak permite que você adicione metadados a agregados para indexação e links interagregados, o Redis permite que você divida o agregado em listas ou conjuntos. Você pode dar suporte à consulta integrando ferramentas de pesquisa como o Solr. Como exemplo, o Riak inclui um recurso de pesquisa que usa pesquisa semelhante ao Solr em quaisquer agregados armazenados como estruturas JSON ou XML.

Apesar dessa imprecisão, a distinção geral ainda se mantém. Com bancos de dados de chave-valor, esperamos principalmente procurar agregados usando uma chave. Com bancos de dados de documentos, esperamos principalmente enviar alguma forma de consulta com base na estrutura interna do documento; isso pode ser uma chave, mas é mais provável que seja outra coisa.

2.3 Lojas da família de colunas

Um dos primeiros e influentes bancos de dados NoSQL foi o BigTable [Chang etc.] do Google. Seu nome evocou uma estrutura tabular que ele realizou com colunas esparsas e nenhum esquema. Como você verá em breve, não ajuda pensar nessa estrutura como uma tabela; em vez disso, é um mapa de dois níveis. Mas, não importa como você pense sobre a estrutura, ela tem sido um modelo que influenciou bancos de dados posteriores, como HBase e Cassandra.

Esses bancos de dados com um modelo de dados no estilo bigtable são frequentemente chamados de armazenamentos de colunas, mas esse nome já existe há algum tempo para descrever um animal diferente. Os armazenamentos de colunas pré-NoSQL, como o C-Store [C-Store], estavam felizes com o SQL e o modelo relacional. O que os tornava diferentes era a maneira como eles armazenavam dados fisicamente. A maioria dos bancos de dados tem uma linha como unidade de armazenamento, o que, em particular, ajuda no desempenho da gravação. No entanto, há muitos cenários em que as gravações são raras, mas você geralmente precisa ler algumas colunas de muitas linhas de uma vez.

Nessa situação, é melhor armazenar grupos de colunas para todas as linhas como a unidade de armazenamento básica, e é por isso que esses bancos de dados são chamados de armazenamentos de colunas.

O Bigtable e seus descendentes seguem essa noção de armazenar grupos de colunas (famílias de colunas) juntos, mas se separam do C-Store e amigos ao abandonar o modelo relacional e o SQL. Neste livro, nos referimos a essa classe de bancos de dados como bancos de dados da família de colunas.

Talvez a melhor maneira de pensar no modelo de família de colunas seja como uma estrutura agregada de dois níveis. Assim como com armazenamentos de chave-valor, a primeira chave é frequentemente descrita como um identificador de linha, pegando o agregado de interesse. A diferença com estruturas de família de colunas é que esse agregado de linha é formado por um mapa de valores mais detalhados. Esses valores de segundo nível são chamados de colunas. Além de acessar a linha como um todo, as operações também permitem escolher uma coluna específica, então para obter o nome de um cliente específico da Figura 2.5 você pode fazer algo como `get('1234', 'name')`.

Bancos de dados de famílias de colunas organizam suas colunas em famílias de colunas. Cada coluna deve fazer parte de uma única família de colunas, e a coluna atua como uma unidade de acesso, partindo do pressuposto de que os dados de uma determinada família de colunas geralmente serão acessados juntos.

Isso também oferece algumas maneiras de pensar sobre como os dados são estruturados.

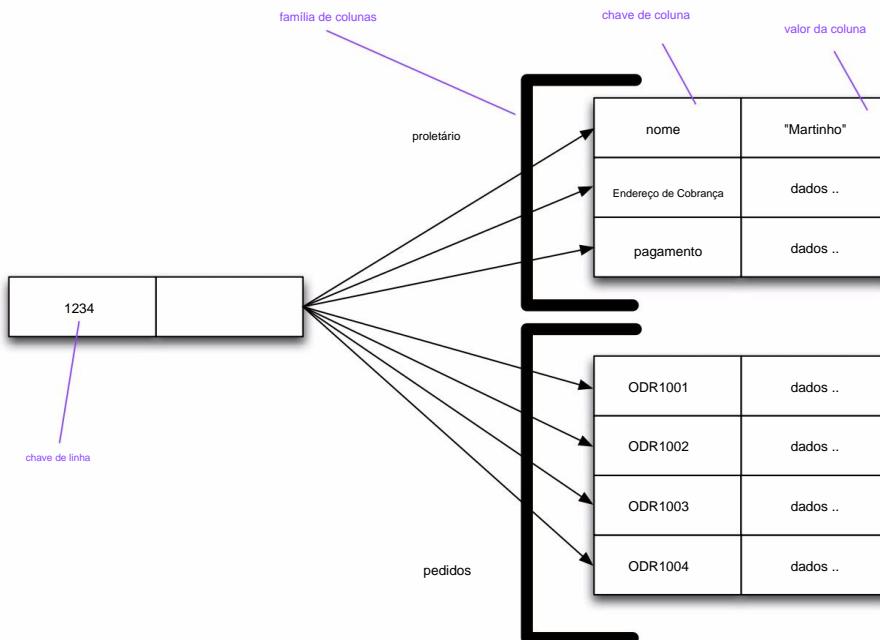


Figura 2.5 Representando informações do cliente em uma estrutura de família de colunas

- Orientado por linha: Cada linha é um agregado (por exemplo, cliente com o ID de 1234) com famílias de colunas representando blocos úteis de dados (perfil, histórico de pedidos) dentro desse agregado.
- Orientado a colunas: cada família de colunas define um tipo de registro (por exemplo, cliente perfis) com linhas para cada um dos registros. Você então pensa em uma linha como o junção de registros em todas as famílias de colunas.

Este último aspecto reflete a natureza colunar dos bancos de dados de famílias de colunas.

Como o banco de dados conhece esses agrupamentos comuns de dados, ele pode usar essas informações para seu comportamento de armazenamento e acesso. Mesmo que um documento banco de dados declare alguma estrutura para o banco de dados, cada documento ainda é visto como uma única unidade. As famílias de colunas conferem uma qualidade bidimensional à família de colunas bases de dados.

Esta terminologia é estabelecida pelo Google Bigtable e HBase, mas Cassandra olha as coisas de forma ligeiramente diferente. Uma linha em Cassandra só ocorre em uma coluna família, mas essa família de colunas pode conter supercolunas — colunas que contêm colunas aninhadas. As supercolunas em Cassandra são o melhor equivalente ao famílias de colunas clássicas do Bigtable.

Ainda pode ser confuso pensar em famílias de colunas como tabelas. Você pode adicionar qualquer coluna para qualquer linha, e as linhas podem ter chaves de coluna muito diferentes. Enquanto

2.4 Resumindo bancos de dados orientados a agregados

novas colunas são adicionadas às linhas durante o acesso regular ao banco de dados; definir novas famílias de colunas é muito mais raro e pode envolver a interrupção do banco de dados para que isso aconteça.

O exemplo da Figura 2.5 ilustra outro aspecto dos bancos de dados de família de colunas que pode ser desconhecido para pessoas acostumadas com tabelas relacionais: a família de colunas `orders`. Como as colunas podem ser adicionadas livremente, você pode modelar uma lista de itens tornando cada item uma coluna separada. Isso é muito estranho se você pensar em uma família de colunas como uma tabela, mas bastante natural se você pensar em uma linha de família de colunas como um agregado. Cassandra usa os termos "largo" e "fino". **Linhas finas** têm poucas colunas com as mesmas colunas usadas em muitas linhas diferentes. Nesse caso, a família de colunas define um tipo de registro, cada linha é um registro e cada coluna é um campo. Uma **linha larga** tem muitas colunas (talvez milhares), com linhas tendo colunas muito diferentes. Uma família de colunas larga modela uma lista, com cada coluna sendo um elemento nessa lista.

Uma consequência de famílias de colunas amplas é que uma família de colunas pode definir uma ordem de classificação para suas colunas. Dessa forma, podemos acessar pedidos por sua chave de pedido e acessar intervalos de pedidos por suas chaves. Embora isso possa não ser útil se codificarmos pedidos por seus IDs, seria se fizéssemos a chave a partir de uma concatenação de data e ID (por exemplo, 20111027-1001).

Embora seja útil distinguir famílias de colunas por sua natureza ampla ou estreita, não há nenhuma razão técnica pela qual uma família de colunas não possa conter colunas do tipo campo e colunas do tipo lista — embora isso possa confundir a ordem de classificação.

2.4 Resumindo bancos de dados orientados a agregados

Até aqui, cobrimos material suficiente para lhe dar uma visão geral razoável dos três estilos diferentes de modelos de dados orientados a agregados e como eles diferem.

O que todos eles compartilham é a noção de um agregado indexado por uma chave que você pode usar para pesquisa. Esse agregado é central para execução em um cluster, pois o banco de dados garantirá que todos os dados de um agregado sejam armazenados juntos em um nó. O agregado também atua como a unidade atômica para atualizações, fornecendo uma quantidade útil, embora limitada, de controle transacional.

Dentro dessa noção de agregado, temos algumas diferenças. O modelo de dados de valor-chave trata o agregado como um todo opaco, o que significa que você só pode fazer uma pesquisa de chave para todo o agregado — você não pode executar uma consulta nem recuperar uma parte do agregado.

O modelo de documento torna o agregado transparente para o banco de dados, permitindo que você faça consultas e recuperações parciais. No entanto, como o documento não tem esquema, o banco de dados não pode agir muito na estrutura do documento para otimizar o armazenamento e a recuperação de partes do agregado.

Os modelos de família de colunas dividem o agregado em famílias de colunas, permitindo que o banco de dados os trate como unidades de dados dentro do agregado de linhas. Isso impõe alguma estrutura ao agregado, mas permite que o banco de dados tire vantagem dessa estrutura para melhorar sua acessibilidade.

2.5 Leituras Adicionais

Para mais informações sobre o conceito geral de agregados, que são frequentemente usados com bancos de dados relacionais também, veja [Evans]. A comunidade Domain-Driven Design é a melhor fonte para mais informações sobre agregados — informações recentes geralmente aparecem em <http://domaindrivendesign.org>.

2.6 Pontos-chave

- Um agregado é uma coleção de dados com os quais interagimos como uma unidade.
Os agregados formam os limites das operações ACID com o banco de dados.
- Bancos de dados de chave-valor, documentos e famílias de colunas podem ser vistos como formas de banco de dados orientados a agregados.
- Os agregados facilitam o gerenciamento do armazenamento de dados pelo banco de dados aglomerados.
- Bancos de dados orientados a agregados funcionam melhor quando a maior parte da interação de dados é feita com o mesmo agregado; bancos de dados que ignoram agregados são melhores quando as interações usam dados organizados em muitas formações diferentes.

Capítulo 3

Mais detalhes sobre modelos de dados

Até agora, cobrimos o recurso principal na maioria dos bancos de dados NoSQL: seu uso de ag-agregados e como bancos de dados orientados a agregados modelam agregados de diferentes maneiras. Embora os agregados sejam uma parte central da história do NoSQL, há mais no lado da modelagem de dados do que isso, e exploraremos esses conceitos mais adiante neste capítulo.

3.1 Relacionamentos

Agregados são úteis porque reúnem dados que são comumente acessados juntos. Mas ainda há muitos casos em que dados relacionados são acessados de forma diferente. Considere o relacionamento entre um cliente e todos os seus pedidos. Alguns aplicativos vão querer acessar o histórico de pedidos sempre que acessarem o cliente; isso se encaixa bem com a combinação do cliente com seu histórico de pedidos em um único agregado. Outros aplicativos, no entanto, querem processar pedidos individualmente e, portanto, modelar pedidos como agregados independentes.

Neste caso, você vai querer agregados separados de pedidos e clientes, mas com algum tipo de relacionamento entre eles, para que qualquer trabalho em um pedido possa procurar dados do cliente. A maneira mais simples de fornecer esse link é incorporar o ID do cliente dentro dos dados agregados do pedido. Dessa forma, se você precisar de dados do registro do cliente, você lê o pedido, descobre o ID do cliente e faz outra chamada ao banco de dados para ler os dados do cliente. Isso funcionará e ficará bem em muitos cenários — mas o banco de dados ignorará o relacionamento nos dados. Isso pode ser importante porque há momentos em que é útil para o banco de dados saber sobre esses links.

Como resultado, muitos bancos de dados — até mesmo armazenamentos de chave-valor — fornecem maneiras de tornar esses relacionamentos visíveis para o banco de dados. Os armazenamentos de documentos tornam o conteúdo do agregado disponível para o banco de dados para formar índices e consultas. O Riak, um armazenamento de chave-valor, permite que você coloque informações de link em metadados, suportando recuperação parcial e capacidade de link-walking.

Um aspecto importante dos relacionamentos entre agregados é como eles lidam com atualizações. Bancos de dados orientados a agregados tratam o agregado como a unidade de recuperação de dados. Consequentemente, a atomicidade só é suportada dentro do conteúdo de um único agregado. Se você atualizar vários agregados de uma vez, terá que lidar com uma falha no meio do caminho. Bancos de dados relacionais ajudam você com isso permitindo que você modifique vários registros em uma única transação, fornecendo garantias ACID enquanto altera muitas linhas.

Tudo isso significa que bancos de dados orientados a agregados se tornam mais estranhos à medida que você precisa operar em vários agregados. Há várias maneiras de lidar com isso, que exploraremos mais adiante neste capítulo, mas a estranheza fundamental permanece.

Isso pode implicar que, se você tiver dados baseados em muitos relacionamentos, você deve preferir um banco de dados relacional em vez de um armazenamento NoSQL. Embora isso seja verdade para bancos de dados orientados a agregados, vale lembrar que bancos de dados relacionais também não são tão estelares com relacionamentos complexos. Embora você possa expressar consultas envolvendo junções em SQL, as coisas rapidamente ficam muito cabeludas — tanto com a escrita em SQL quanto com o desempenho resultante — à medida que o número de junções aumenta.

Este é um bom momento para introduzir outra categoria de bancos de dados que é frequentemente agrupados na pilha NoSQL.

3.2 Bancos de dados de grafos

Bancos de dados de grafos são um peixe estranho no lago NoSQL. A maioria dos bancos de dados NoSQL foi inspirada pela necessidade de rodar em clusters, o que levou a modelos de dados orientados a agregados de grandes registros com conexões simples. Bancos de dados de grafos são motivados por uma frustração diferente com bancos de dados relacionais e, portanto, têm um modelo oposto — pequenos registros com interconexões complexas, algo como a Figura 3.1.

Neste contexto, um gráfico não é um gráfico de barras ou histograma; em vez disso, nos referimos a um estrutura de dados de gráfico de nós conectados por arestas.

Na Figura 3.1 temos uma rede de informações cujos nós são muito pequenos (nada mais que um nome), mas há uma rica estrutura de interconexões entre eles. Com essa estrutura, podemos fazer perguntas como “encontre os livros na categoria Databases que foram escritos por alguém de quem um amigo meu gosta”.

Os bancos de dados de gráficos são especializados em capturar esse tipo de informação — mas em uma escala muito maior do que um diagrama legível poderia capturar. Isso é ideal para capturar quaisquer dados que consistam em relacionamentos complexos, como redes sociais, preferências de produtos ou regras de elegibilidade.

O modelo de dados fundamental de um banco de dados de grafos é muito simples: nós conectados por arestas (também chamados de arcos). Além dessa característica essencial, há muita variação nos modelos de dados — em particular, quais mecanismos você tem

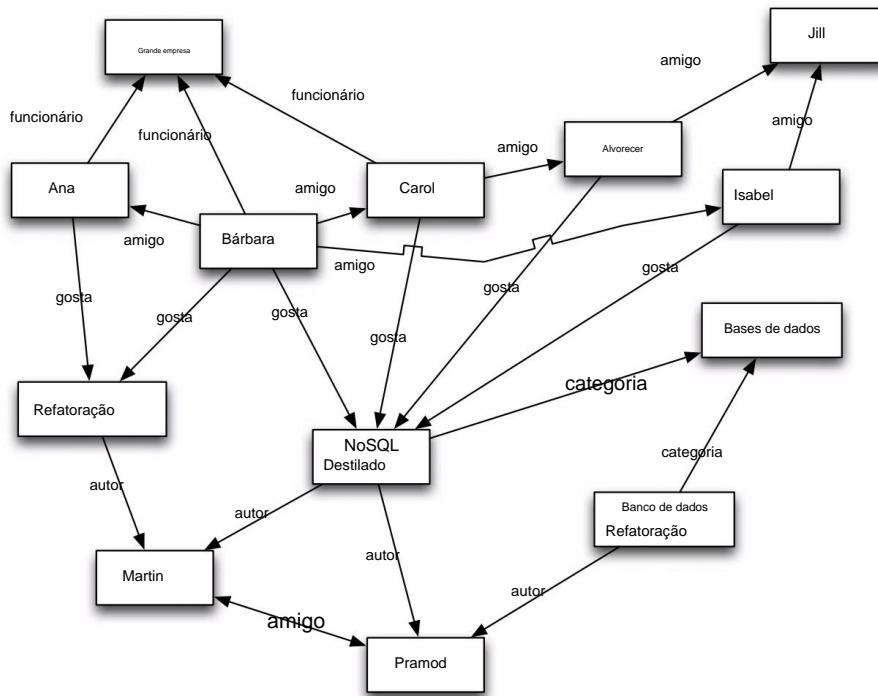


Figura 3.1 Um exemplo de estrutura de gráfico

para armazenar dados em seus nós e arestas. Uma amostra rápida de alguns recursos atuais ilustra essa variedade de possibilidades: FlockDB é simplesmente nós e arestas sem mecanismo para atributos adicionais; Neo4J permite que você anexe objetos Java como propriedades a nós e arestas de forma sem esquema ("Features," p. 113); Infinite Graph armazena seus objetos Java, que são subclasses de seus tipos internos, como nós e arestas.

Depois de criar um gráfico de nós e arestas, um banco de dados de gráficos permite que você consulte essa rede com operações de consulta projetadas com esse tipo de gráfico em mente. É aqui que entram as diferenças importantes entre bancos de dados de gráficos e relacionais. Embora os bancos de dados relacionais possam implementar relacionamentos usando chaves estrangeiras, as junções necessárias para navegar podem ficar bem caras — o que significa que o desempenho geralmente é ruim para modelos de dados altamente conectados. Os bancos de dados de gráficos tornam a travessia ao longo dos relacionamentos muito barata. Uma grande parte disso ocorre porque os bancos de dados de gráficos transferem a maior parte do trabalho de navegação de relacionamentos do tempo de consulta para o tempo de inserção. Isso naturalmente compensa em situações em que o desempenho da consulta é mais importante do que a velocidade de inserção.

Na maioria das vezes, você encontra dados navegando pela rede de arestas, com consultas como "diga-me todas as coisas que Anna e Barbara gostam".

No entanto, você precisa de um ponto de partida, então geralmente alguns nós podem ser indexados por um atributo como ID. Então você pode começar com uma pesquisa de ID (ou seja, pesquisar as pessoas chamadas "Anna" e "Barbara") e então começar a usar as arestas. Ainda assim, bancos de dados de grafos esperam que a maior parte do seu trabalho de consulta seja navegar por relacionamentos.

A ênfase em relacionamentos torna os bancos de dados de gráficos muito diferentes dos bancos de dados orientados a agregados. Essa diferença de modelo de dados tem consequências em outros aspectos também; você descobrirá que esses bancos de dados têm mais probabilidade de rodar em um único servidor do que distribuídos em clusters. As transações ACID precisam cobrir vários nós e arestas para manter a consistência.

A única coisa que eles têm em comum com os bancos de dados orientados a agregados é sua rejeição do modelo relacional e um aumento na atenção que receberam quase na mesma época que o resto do campo NoSQL.

3.3 Bancos de dados sem esquema

Um tema comum em todas as formas de bancos de dados NoSQL é que eles não têm esquema. Quando você quer armazenar dados em um banco de dados relacional, primeiro você tem que definir um esquema — uma estrutura definida para o banco de dados que diz quais tabelas existem, quais colunas existem e quais tipos de dados cada coluna pode conter. Antes de armazenar alguns dados, você tem que ter o esquema definido para eles.

Com bancos de dados NoSQL, armazenar dados é muito mais casual. Um armazenamento de chave-valor permite que você armazene quaisquer dados que você quiser sob uma chave. Um banco de dados de documentos efetivamente faz a mesma coisa, já que não faz restrições sobre a estrutura dos documentos que você armazena. Bancos de dados de família de colunas permitem que você armazene quaisquer dados sob qualquer coluna que você quiser. Bancos de dados de grafos permitem que você adicione livremente novas arestas e adicione livremente propriedades a nós e arestas como você desejar.

Os defensores da ausência de esquemas se alegram com essa liberdade e flexibilidade. Com um esquema, você tem que descobrir com antecedência o que precisa armazenar, mas isso pode ser difícil de fazer. Sem um esquema vinculando você, você pode facilmente armazenar o que precisa. Isso permite que você altere facilmente seu armazenamento de dados conforme aprende mais sobre seu projeto. Você pode facilmente adicionar coisas novas conforme as descobre. Além disso, se você descobrir que não precisa mais de algumas coisas, pode simplesmente parar de armazená-las, sem se preocupar em perder dados antigos como aconteceria se você excluisse colunas em um esquema relacional.

Além de manipular alterações, um armazenamento sem esquema também facilita o tratamento de **dados não uniformes**: dados em que cada registro tem um conjunto diferente de campos. Um esquema coloca todas as linhas de uma tabela em uma camisa de força, o que se torna estranho se você tiver diferentes tipos de dados em diferentes linhas. Ou você acaba com muitas colunas que geralmente são nulas (uma tabela esparsa), ou acaba com colunas sem sentido, como a coluna personalizada 4. A ausência de esquema evita isso, permitindo que cada registro contenha apenas o que precisa — nem mais, nem menos.

A ausência de esquema é atraente e certamente evita muitos problemas que existem com bancos de dados de esquema fixo, mas traz alguns problemas próprios. Se tudo o que você está fazendo é armazenar alguns dados e exibi-los em um relatório como uma lista simples de linhas `fieldName: value`, então um esquema só vai atrapalhar. Mas normalmente fazemos com nossos dados mais do que isso, e fazemos isso com programas que precisam saber que o endereço de cobrança é chamado `billingAddress` e não `addressForBilling` e que o campo `quantify` será um inteiro 5 e não cinco.

O fato vital, embora às vezes inconveniente, é que sempre que escrevemos um programa que acessa dados, esse programa quase sempre depende de alguma forma de esquema implícito. A menos que ele apenas diga algo como

```
//pseudocódigo
foreach (Registro r em registros) {
    foreach (Campo f em r.fields) { print
        (f.name, f.value)
    }
}
```

ele assumirá que certos nomes de campo estão presentes e carregam dados com um certo significado, e assumirá algo sobre o tipo de dados armazenados naquele campo.

Programas não são humanos; eles não podem ler “qty” e inferir que isso deve ser o mesmo que “quantity” — pelo menos não a menos que os programemos especificamente para isso. Então, por mais sem esquema que nosso banco de dados seja, geralmente há um esquema implícito presente. Esse **esquema implícito** é um conjunto de suposições sobre a estrutura dos dados no código que os manipula.

Ter o esquema implícito no código do aplicativo resulta em alguns problemas.

Isso significa que, para entender quais dados estão presentes, você precisa cavar fundo no código do aplicativo. Se esse código for bem estruturado, você deve ser capaz de encontrar um lugar claro do qual deduzir o esquema. Mas não há garantias; tudo depende de quanto claro o código do aplicativo é. Além disso, o banco de dados permanece ignorante do esquema — ele não pode usar o esquema para ajudá-lo a decidir como armazenar e recuperar dados de forma eficiente. Ele não pode aplicar suas próprias validações sobre esses dados para garantir que diferentes aplicativos não manipulem dados de forma inconsistente.

Essas são as razões pelas quais bancos de dados relacionais têm um esquema fixo e, de fato, as razões pelas quais a maioria dos bancos de dados teve esquemas fixos no passado. Esquemas têm valor, e a rejeição de esquemas por bancos de dados NoSQL é realmente bastante surpreendente.

Essencialmente, um banco de dados sem esquema desloca o esquema para o código do aplicativo que o acessa. Isso se torna problemático se vários aplicativos, desenvolvidos por pessoas diferentes, acessam o mesmo banco de dados. Esses problemas podem ser reduzidos com algumas abordagens. Uma é encapsular toda a interação do banco de dados em um único aplicativo e integrá-lo a outros aplicativos usando serviços da web.

Isso se encaixa bem com a preferência atual de muitas pessoas por usar serviços da web para integração. Outra abordagem é delinear claramente diferentes áreas de um agregado

para acesso por diferentes aplicativos. Podem ser diferentes seções em um banco de dados de documentos ou diferentes famílias de colunas em um banco de dados de famílias de colunas.

Embora os fãs do NoSQL frequentemente critiquem os esquemas relacionais por terem que ser definidos antecipadamente e serem inflexíveis, isso não é realmente verdade. Os esquemas relacionais podem ser alterados a qualquer momento com comandos SQL padrão. Se necessário, você pode criar novas colunas de forma ad-hoc para armazenar dados não uniformes. Raramente vimos isso ser feito, mas funcionou razoavelmente bem onde vimos. Na maioria das vezes, no entanto, a não uniformidade em seus dados é um bom motivo para favorecer um banco de dados sem esquema.

A ausência de esquema tem um grande impacto nas mudanças da estrutura de um banco de dados ao longo do tempo, particularmente para dados mais uniformes. Embora não seja praticado tão amplamente quanto deveria ser, alterar o esquema de um banco de dados relacional pode ser feito de forma controlada. Da mesma forma, você tem que exercer controle ao alterar como você armazena dados em um banco de dados sem esquema para que você possa acessar facilmente dados抗igos e novos. Além disso, a flexibilidade que a ausência de esquema lhe dá se aplica somente dentro de um agregado — se você precisar alterar seus limites de agregado, a migração é tão complexa quanto no caso relacional. Falaremos mais sobre migração de banco de dados mais tarde (“Migrações de esquema”, p. 123).

3.4 Visualizações Materializadas

Quando falamos sobre modelos de dados orientados a agregados, enfatizamos suas vantagens. Se você quiser acessar pedidos, é útil ter todos os dados de um pedido contidos em um único agregado que pode ser armazenado e acessado como uma unidade. Mas a orientação agregada tem uma desvantagem correspondente: o que acontece se um gerente de produto quiser saber quanto um item específico vendeu nas últimas semanas? Agora, a orientação agregada trabalha contra você, forçando-o a potencialmente ler cada pedido no banco de dados para responder à pergunta. Você pode reduzir esse fardo construindo um índice sobre o produto, mas ainda está trabalhando contra a estrutura agregada.

Bancos de dados relacionais têm uma vantagem aqui porque sua falta de estrutura agregada permite que eles suportem o acesso a dados de diferentes maneiras. Além disso, eles fornecem um mecanismo conveniente que permite que você olhe para os dados de forma diferente da forma como eles são armazenados — visualizações. Uma visualização é como uma tabela relacional (é uma relação), mas é definida pela computação sobre as tabelas base. Quando você acessa uma visualização, o banco de dados computa os dados na visualização — uma forma útil de encapsulamento.

As visualizações fornecem um mecanismo para ocultar do cliente se os dados são dados derivados ou dados base — mas não podem evitar o fato de que algumas visualizações são caras para computar. Para lidar com isso, **as visualizações materializadas** foram inventadas, que são visualizações computadas antecipadamente e armazenadas em cache no disco. As visualizações materializadas são eficazes para dados que são lidos intensamente, mas podem ficar um pouco obsoletos.

Embora os bancos de dados NoSQL não tenham visualizações, eles podem ter consultas pré-computadas e armazenadas em cache, e reutilizam o termo “visualização materializada” para descrevê-los. Também é um aspecto muito mais central para bancos de dados orientados a agregados do que para sistemas relacionais, já que a maioria dos aplicativos terá que lidar com algumas consultas que não se encaixam bem com a estrutura agregada. (Frequentemente, os bancos de dados NoSQL criam visualizações materializadas usando uma computação map-reduce, sobre a qual falaremos no Capítulo 7.)

Existem duas estratégias básicas para construir uma visualização materializada. A primeira é a abordagem ansiosa, na qual você atualiza a visualização materializada ao mesmo tempo em que atualiza os dados base para ela. Nesse caso, adicionar um pedido também atualizaria os agregados do histórico de compras para cada produto. Essa abordagem é boa quando você tem leituras mais frequentes da visualização materializada do que gravações e deseja que as visualizações materializadas sejam o mais atualizadas possível. A abordagem do banco de dados do aplicativo (p. 7) é valiosa aqui, pois torna mais fácil garantir que quaisquer atualizações nos dados base também atualizem as visualizações materializadas.

Se você não quiser pagar essa sobrecarga em cada atualização, você pode executar trabalhos em lote para atualizar as visualizações materializadas em intervalos regulares. Você precisará entender seus requisitos de negócios para avaliar o quanto obsoletas suas visualizações materializadas podem ser.

Você pode construir visualizações materializadas fora do banco de dados lendo os dados, computando a visualização e salvando-a de volta no banco de dados. Mais frequentemente, os bancos de dados suportam a construção de visualizações materializadas por si mesmos. Neste caso, você fornece a computação que precisa ser feita, e o banco de dados executa a computação quando necessário de acordo com alguns parâmetros que você configura. Isto é particularmente útil para atualizações rápidas de visualizações com redução de mapa incremental (“Redução de mapa incremental,” p. 76).

Visualizações materializadas podem ser usadas dentro do mesmo agregado. Um documento de pedido pode incluir um elemento de resumo do pedido que fornece informações resumidas sobre o pedido para que uma consulta para um resumo do pedido não tenha que transferir o documento do pedido inteiro. Usar diferentes famílias de colunas para visualizações materializadas é um recurso comum de bancos de dados de famílias de colunas. Uma vantagem de fazer isso é que permite que você atualize a visualização materializada dentro da mesma operação atômica.

3.5 Modelagem para acesso a dados

Conforme mencionado anteriormente, ao modelar agregados de dados, precisamos considerar como os dados serão lidos, bem como quais são os efeitos colaterais nos dados relacionados a esses agregados.

Vamos começar com o modelo onde todos os dados do cliente são incorporados usando um armazenamento de chave-valor (veja Figura 3.2).

Neste cenário, o aplicativo pode ler as informações do cliente e todos os dados relacionados usando a chave. Se os requisitos forem ler os pedidos ou o

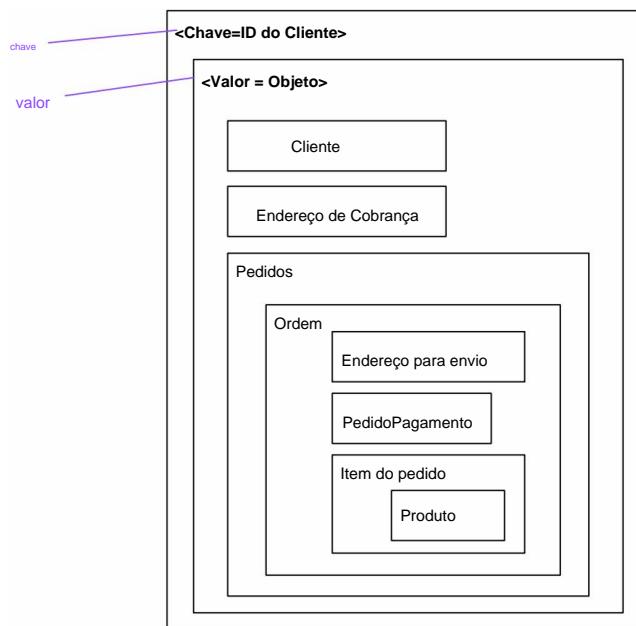


Figura 3.2 Incorpore todos os objetos para o cliente e seus pedidos.

produtos vendidos em cada pedido, o objeto inteiro tem que ser lido e então analisado no lado do cliente para construir os resultados. Quando referências são necessárias, podemos mudar para armazenamentos de documentos e então consultar dentro dos documentos, ou até mesmo alterar os dados para o armazenamento de chave-valor para dividir o objeto de valor em objetos Cliente e Pedido e então manter as referências desses objetos entre si.

Com as referências (veja Figura 3.3), agora podemos encontrar os pedidos independentemente do Cliente, e com a referência orderId no Cliente podemos encontrar todos os Pedidos para o Cliente. Usar agregados dessa forma permite a otimização de leitura, mas temos que enviar a referência orderId para o Cliente toda vez com um novo Pedido.

```
# Objeto do cliente
```

```
{
  "customerId": 1,
  "customer": {
    "name": "Martin",
    "billingAddress": [{"city": "Chicago"}], "payment": [{"type": "debit", "ccinfo": "1000-1000-1000-1000"}], "orders": [{"orderId": 99}]}
}
```

3.5 Modelagem para acesso a dados

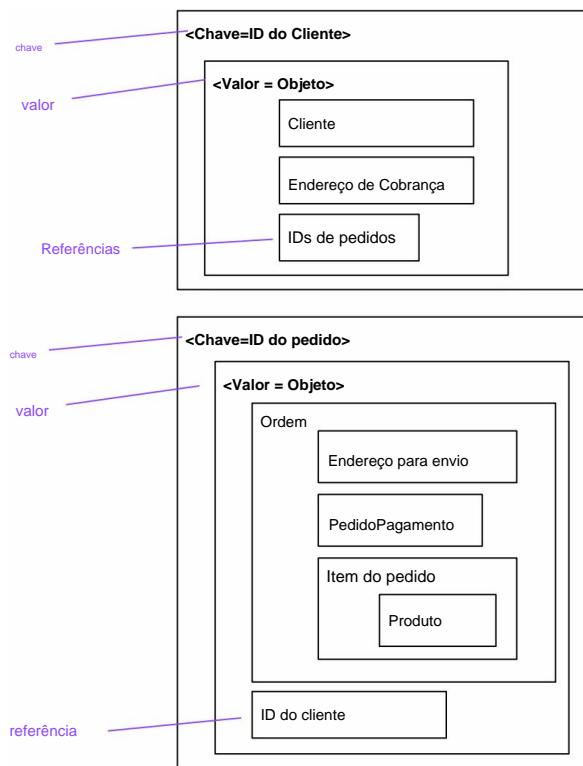


Figura 3.3 O cliente é armazenado separadamente do pedido.

Objeto do pedido

```
{
  "customerId": 1,
  "orderId": 99,
  "order": [
    {
      "orderDate": "20/11/2011",
      "orderItems": [{"productId": 27, "price": 32,45}],
      "orderPayment": [
        {"ccinfo": "1000-1000-1000-1000",
         "txnid": "abelif879rft"}],
      "enderecoenvio": {"cidade": "Chicago"} }
  ]
}
```

Agregados também podem ser usados para obter análises; por exemplo, uma atualização agregada pode preencher informações sobre quais Pedidos têm um determinado Produto . Essa desnормalização dos dados permite acesso rápido aos dados nos quais estamos interessados e é a base para **BI em tempo real** ou **análise em tempo real** , onde as empresas não precisam depender de execuções em lote no final do dia para preencher o data warehouse.

Capítulo 3 Mais detalhes sobre modelos de dados

tabelas e gerar análises; agora eles podem preencher esse tipo de dado, para vários tipos de requisitos, quando o pedido é feito pelo cliente.

```
{ "itemid":27,
"pedidos":{99.545.897.678} }
```

```
{ "itemid":29,
"pedidos":{199.545.704.819} }
```

Em armazenamentos de documentos, como podemos consultar dentro de documentos, remover referências a Orders do objeto Customer é possível. Essa alteração nos permite não atualizar o objeto Customer quando novos pedidos são feitos pelo Customer.

Objeto do cliente

```
{ "customerId": 1,
"name": "Martin",
"billingAddress": [{"city": "Chicago"}], "payment": [ {"type":
"debit", "ccinfo":
"1000-1000-1000-1000"} ]
```

```
}
```

Objeto do pedido

```
{ "orderId": 99,
"customerId": 1,
"orderDate": "20/11/2011",
"orderItems":[{"productId":27, "price": 32,45}], "orderPayment":
[{"ccinfo":"1000-1000-1000-1000",
"txnid":"abelif879rft"}, "endereçodeenvio":{"cidade":"Chicago"} } 
```

Como os armazenamentos de dados de documentos permitem que você consulte por atributos dentro do documento, pesquisas como "encontrar todos os pedidos que incluem o produto *Refactoring Databases*" são possíveis, mas a decisão de criar um agregado de itens e pedidos aos quais eles pertencem não se baseia na capacidade de consulta do banco de dados, mas na otimização de leitura desejada pelo aplicativo.

Ao modelar para armazenamentos de famílias de colunas, temos o benefício das colunas serem ordenadas, permitindo-nos nomear colunas que são usadas com frequência para que sejam buscadas primeiro. Ao usar as famílias de colunas para modelar os dados, é importante lembrar de fazê-lo de acordo com seus requisitos de consulta e não com o propósito de escrever; a regra geral é facilitar a consulta e desnormalizar os dados durante a escrita.

3.5 Modelagem para acesso a dados

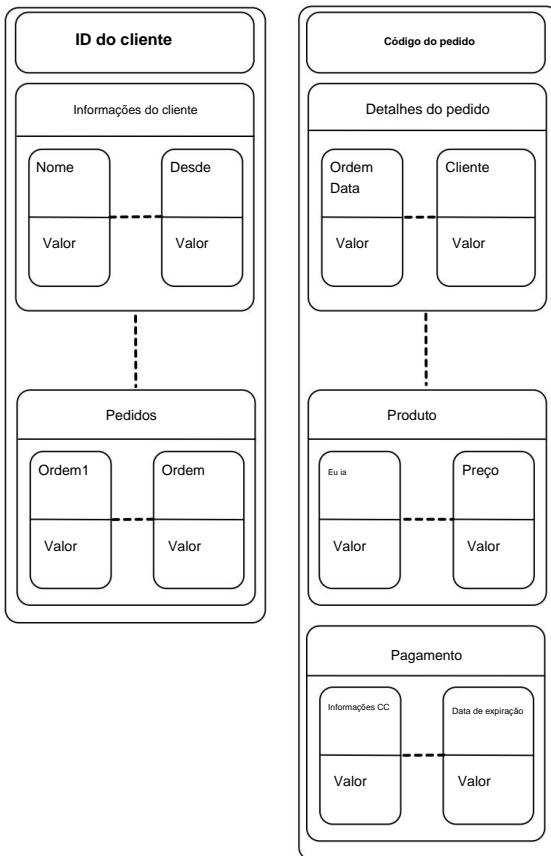


Figura 3.4 Visão conceitual de um armazenamento de dados de coluna

Como você pode imaginar, há várias maneiras de modelar os dados; uma maneira é armazenar o Cliente e o Pedido em diferentes famílias de *colunas* (veja a Figura 3.4).

Aqui, é importante notar que a referência a todos os pedidos feitos pelo cliente está na família de colunas Customer. Outras desnormalizações semelhantes são geralmente feitas para que o desempenho da consulta (leitura) seja melhorado.

Ao usar bancos de dados de gráficos para modelar os mesmos dados, modelamos todos os objetos como nós e as relações dentro deles como relacionamentos; esses relacionamentos têm tipos e significado direcional.

Cada nó tem relacionamentos independentes com outros nós. Esses relacionamentos têm nomes como PURCHASED, PAID_WITH ou BELONGS_TO (veja a Figura 3.5); esses nomes de relacionamento permitem que você percorra o gráfico. Digamos que você queira encontrar todos os Customers que PURCHASED um produto com o nome *Refactoring Databases*. Tudo o que precisamos fazer é consultar o nó do produto Refactoring Databases e procurar todos os Customers com o relacionamento PURCHASED de entrada.

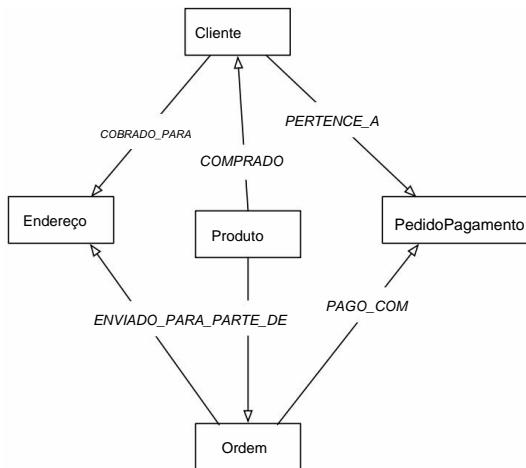


Figura 3.5 Modelo gráfico de dados de comércio eletrônico

Esse tipo de travessia de relacionamento é muito fácil com bancos de dados de grafos. É especialmente conveniente quando você precisa usar os dados para recomendar produtos aos usuários ou para encontrar padrões em ações tomadas pelos usuários.

3.6 Pontos-chave

- Os bancos de dados orientados a agregados tornam os relacionamentos entre agregados mais difícil de lidar do que relacionamentos intra-agregados.
- Os bancos de dados de gráficos organizam os dados em gráficos de nós e arestas; eles funcionam melhor para dados que possuem estruturas de relacionamento complexas.
- Os bancos de dados sem esquema permitem que você adicione campos livremente aos registros, mas há geralmente um esquema implícito esperado pelos usuários dos dados.
- Bancos de dados orientados a agregados frequentemente computam visualizações materializadas para fornecer dados organizados de forma diferente de seus agregados primários. Isso geralmente é feito com cálculos map-reduce.

Capítulo 4

Modelos de Distribuição

O principal motivador de interesse no NoSQL tem sido sua capacidade de executar bancos de dados em um grande cluster. À medida que os volumes de dados aumentam, fica mais difícil e caro escalar verticalmente — comprar um servidor maior para executar o banco de dados. Uma opção mais atraente é escalar horizontalmente — executar o banco de dados em um cluster de servidores. A orientação agregada se encaixa bem com a escala horizontal porque o agregado é uma unidade natural para uso na distribuição.

Dependendo do seu modelo de distribuição, você pode obter um armazenamento de dados que lhe dará a capacidade de lidar com grandes quantidades de dados, a capacidade de processar um tráfego maior de leitura ou gravação ou mais disponibilidade em face de lentidão ou quebras de rede. Esses são frequentemente benefícios importantes, mas têm um custo. Executar em um cluster introduz complexidade — então não é algo a ser feito a menos que os benefícios sejam convincentes.

Em termos gerais, há dois caminhos para a distribuição de dados: replicação e fragmentação. A replicação pega os mesmos dados e os copia em vários nós. O sharding coloca dados diferentes em nós diferentes. A replicação e o sharding são técnicas ortogonais: você pode usar uma ou ambas. A replicação vem em duas formas: mestre-escravo e ponto a ponto. Agora discutiremos essas técnicas começando pelas mais simples e trabalhando até as mais complexas: primeiro servidor único, depois sharding, depois replicação mestre-escravo e, finalmente, replicação ponto a ponto.

4.1 Servidor Único

A primeira e mais simples opção de distribuição é a que mais frequentemente recomendamos — nenhuma distribuição. Execute o banco de dados em uma única máquina que lida com todas as leituras e gravações no armazenamento de dados. Preferimos essa opção porque ela elimina todas as complexidades que as outras opções introduzem; é fácil para as pessoas de operações gerenciarem e fácil para os desenvolvedores de aplicativos raciocinarem sobre isso.

Embora muitos bancos de dados NoSQL sejam projetados com base na ideia de execução em um cluster, pode fazer sentido usar o NoSQL com uma distribuição de servidor único

modelo se o modelo de dados do armazenamento NoSQL é mais adequado à aplicação.

Bancos de dados de gráficos são a categoria óbvia aqui — eles funcionam melhor em uma configuração de servidor único. Se o uso de seus dados for principalmente sobre processamento de agregados, então um documento de servidor único ou um armazenamento de valor-chave pode valer a pena porque é mais fácil para desenvolvedores de aplicativos.

No restante deste capítulo, vamos explorar as vantagens e complicações de esquemas de distribuição mais sofisticados. Não deixe que o volume de palavras o engane, fazendo-o pensar que preferiríamos essas opções. Se pudermos nos safar sem distribuir nossos dados, sempre escolheremos uma abordagem de servidor único.

4.2 Fragmentação

Frequentemente, um armazenamento de dados movimentado está ocupado porque pessoas diferentes estão acessando partes diferentes do conjunto de dados. Nessas circunstâncias, podemos dar suporte à escalabilidade horizontal colocando diferentes partes dos dados em servidores diferentes — uma técnica chamada **sharding** (veja a Figura 4.1).

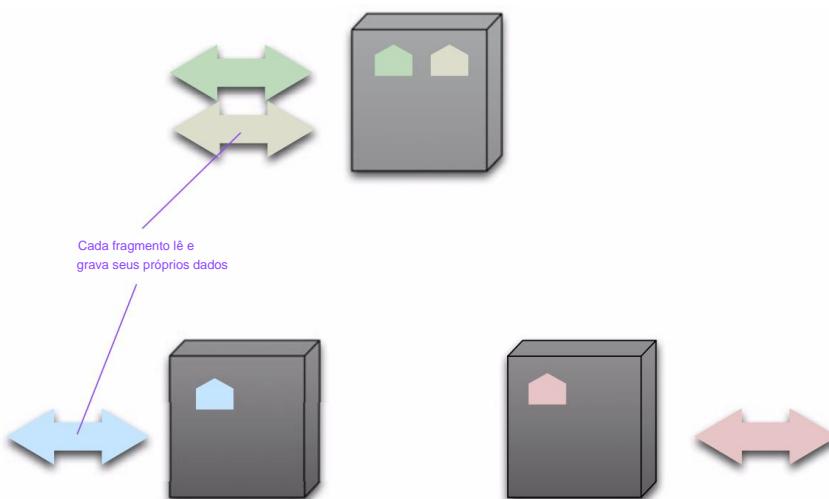


Figura 4.1 A fragmentação coloca dados diferentes em nós separados, cada um dos quais faz suas próprias leituras e gravações.

No caso ideal, temos diferentes usuários conversando com diferentes nós de servidor. Cada usuário só precisa falar com um servidor, então obtém respostas rápidas desse servidor.

A carga é bem equilibrada entre os servidores — por exemplo, se tivermos dez servidores, cada um precisa lidar com apenas 10% da carga.

Claro que o caso ideal é uma fera bem rara. Para chegar perto dele, temos que garantir que os dados que são acessados juntos sejam agrupados no mesmo nó e que esses aglomerados sejam organizados nos nós para fornecer os melhores dados acesso.

A primeira parte desta questão é como agrupar os dados para que um usuário obtenha seus dados principalmente de um único servidor. É aqui que a orientação agregada se torna realmente útil. O ponto principal dos agregados é que os projetamos para combinar dados que são comumente acessados juntos — então os agregados saltam como uma unidade óbvia de distribuição.

Quando se trata de organizar os dados nos nós, há vários fatores que podem ajudar a melhorar o desempenho. Se você sabe que a maioria dos acessos de certos agregados são baseados em um local físico, você pode colocar os dados perto de onde eles estão sendo acessados. Se você tem pedidos para alguém que mora em Boston, você pode colocar esses dados no seu data center no leste dos EUA.

Outro fator é tentar manter a carga uniforme. Isso significa que você deve tentar organizar os agregados para que eles sejam distribuídos uniformemente pelos nós, que recebem quantidades iguais de carga. Isso pode variar ao longo do tempo, por exemplo, se alguns dados tendem a ser acessados em certos dias da semana — então pode haver regras específicas de domínio que você gostaria de usar.

Em alguns casos, é útil juntar agregados se você acha que eles podem ser lidos em sequência. O artigo do Bigtable [Chang etc.] descreveu manter suas linhas em ordem lexicográfica e classificar endereços da web com base em nomes de domínio invertidos (por exemplo, com.martinfowler). Dessa forma, dados de várias páginas podem ser acessados juntos para melhorar a eficiência do processamento.

Historicamente, a maioria das pessoas fez sharding como parte da lógica do aplicativo. Você pode colocar todos os clientes com sobrenomes começando de A a D em um shard e de E a G em outro. Isso complica o modelo de programação, pois o código do aplicativo precisa garantir que as consultas sejam distribuídas entre os vários shards. Além disso, rebalancear o sharding significa alterar o código do aplicativo e migrar os dados. Muitos bancos de dados NoSQL oferecem **sharding automático**, onde o banco de dados assume a responsabilidade de alocar dados para shards e garantir que o acesso aos dados vá para o shard certo. Isso pode tornar muito mais fácil usar o sharding em um aplicativo.

O sharding é particularmente valioso para o desempenho porque pode melhorar o desempenho de leitura e gravação. Usar replicação, particularmente com cache, pode melhorar muito o desempenho de leitura, mas faz pouco para aplicativos que têm muitas gravações. O sharding fornece uma maneira de escalar gravações horizontalmente.

O sharding faz pouco para melhorar a resiliência quando usado sozinho. Embora os dados estejam em nós diferentes, uma falha de nó torna os dados desse shard indisponíveis tão certamente quanto para uma solução de servidor único. O benefício de resiliência que ele fornece é que apenas os usuários dos dados naquele shard sofrerão; no entanto, não é bom ter um banco de dados com parte de seus dados faltando. Com um único servidor, é mais fácil pagar o esforço e o custo para manter esse servidor ativo e funcionando; os clusters geralmente tentam

para usar máquinas menos confiáveis, e você tem mais probabilidade de ter uma falha de nó. Então, na prática, o sharding sozinho provavelmente diminuirá a resiliência.

Apesar do fato de que o sharding é muito mais fácil com agregados, ainda não é um passo a ser tomado levianamente. Alguns bancos de dados são destinados desde o início a usar sharding, nesse caso é sensato executá-los em um cluster desde o início do desenvolvimento e, certamente, na produção. Outros bancos de dados usam sharding como um passo deliberado acima de uma configuração de servidor único, nesse caso é melhor começar com servidor único e usar sharding somente quando suas projeções de carga indicarem claramente que você está ficando sem espaço livre.

Em qualquer caso, a etapa de um único nó para o sharding será complicada. Ouvimos histórias de equipes que tiveram problemas porque deixaram o sharding para muito tarde, então, quando o ativaram na produção, seu banco de dados ficou essencialmente indisponível porque o suporte ao sharding consumiu todos os recursos do banco de dados para mover os dados para novos shards. A lição aqui é usar o sharding bem antes de precisar — quando você tem espaço suficiente para executar o sharding.

4.3 Replicação Mestre-Escravo

Com a distribuição mestre-escravo, você replica dados em vários nós. Um nó é designado como mestre, ou primário. Este mestre é a fonte autoritativa para os dados e geralmente é responsável por processar quaisquer atualizações desses dados. Os outros nós são escravos, ou secundários. Um processo de replicação sincroniza os escravos com o mestre (veja a Figura 4.2).

A replicação mestre-escravo é mais útil para dimensionamento quando você tem um conjunto de dados com uso intensivo de leitura. Você pode dimensionar horizontalmente para lidar com mais solicitações de leitura adicionando mais nós escravos e garantindo que todas as solicitações de leitura sejam roteadas para os escravos. Você ainda está, no entanto, limitado pela capacidade do mestre de processar atualizações e sua capacidade de repassar essas atualizações. Consequentemente, não é um esquema tão bom para conjuntos de dados com tráfego de gravação pesado, embora descarregar o tráfego de leitura ajude um pouco a lidar com a carga de gravação.

Uma segunda vantagem da replicação mestre-escravo é a **resiliência de leitura**: se o mestre falhar, os escravos ainda podem lidar com solicitações de leitura. Novamente, isso é útil se a maior parte do seu acesso a dados for leituras. A falha do mestre elimina a capacidade de lidar com gravações até que o mestre seja restaurado ou um novo mestre seja nomeado.

Entretanto, ter escravos como réplicas do mestre acelera a recuperação após uma falha do mestre, pois um escravo pode ser nomeado um novo mestre muito rapidamente.

A capacidade de nomear um escravo para substituir um mestre com falha significa que a replicação mestre-escravo é útil mesmo se você não precisar escalar horizontalmente. Todo o tráfego de leitura e gravação pode ir para o mestre enquanto o escravo atua como um backup ativo. Neste caso, é

4.3 Replicação Mestre-Escravo

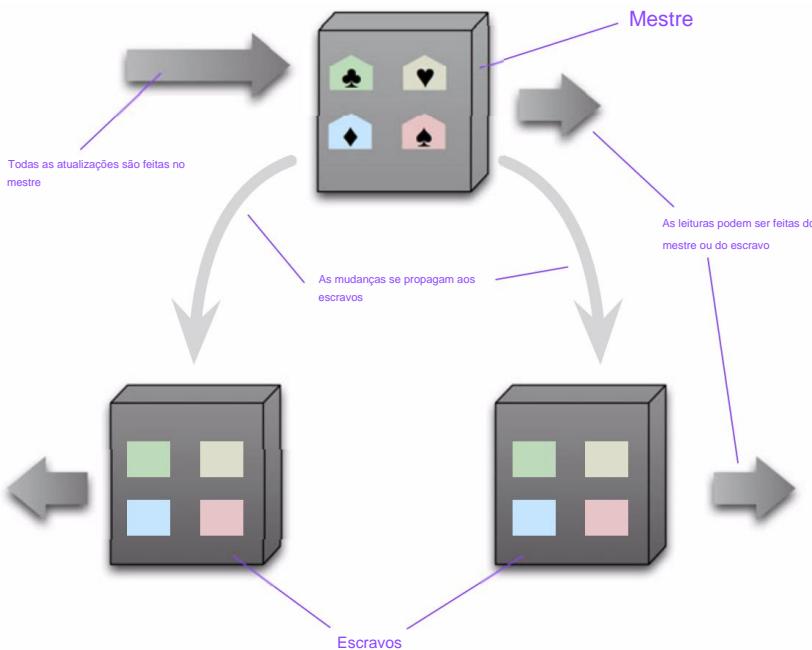


Figura 4.2 Os dados são replicados do master para os slaves. O master atende a todas as gravações; as leituras podem vir do master ou dos slaves.

é mais fácil pensar no sistema como um armazenamento de servidor único com um backup ativo. Você obtém a conveniência da configuração de servidor único, mas com maior resiliência, o que é particularmente útil se você quiser lidar com falhas de servidor com elegância.

Os mestres podem ser nomeados manualmente ou automaticamente. A nomeação manual normalmente significa que, quando você configura seu cluster, configura um nó como mestre. Com a nomeação automática, você cria um cluster de nós e eles elegem um deles para ser o mestre. Além da configuração mais simples, a nomeação automática significa que o cluster pode nomear automaticamente um novo mestre quando um mestre falha, reduzindo o tempo de inatividade.

Para obter resiliência de leitura, você precisa garantir que os caminhos de leitura e gravação em seu aplicativo sejam diferentes, para que você possa lidar com uma falha no caminho de gravação e ainda ler. Isso inclui coisas como colocar as leituras e gravações por meio de conexões de banco de dados separadas — um recurso que geralmente não é suportado por bibliotecas de interação de banco de dados. Como com qualquer recurso, você não pode ter certeza de que tem resiliência de leitura sem bons testes que desabilitem as gravações e verifiquem se as leituras ainda ocorrem.

A replicação traz alguns benefícios atraentes, mas também traz um lado negro inevitável: a inconsistência. Você corre o risco de que clientes diferentes, lendo escravos diferentes, verá valores diferentes porque as mudanças não foram todas propagadas para os escravos. No pior dos casos, isso pode significar que um cliente não pode ler e escrever que acabou de fazer. Mesmo se você usar replicação mestre-escravo apenas para hot backup isso pode ser uma preocupação, porque se o mestre falhar, quaisquer atualizações não passadas para o backup são perdidos. Falaremos sobre como lidar com esses problemas mais tarde ("Consistência", p. 47).

4.4 Replicação ponto a ponto

A replicação mestre-escravo ajuda na escalabilidade de leitura, mas não ajuda com escalabilidade de gravações. Ele fornece resiliência contra falhas de um escravo, mas não de um mestre. Essencialmente, o mestre ainda é um gargalo e um ponto único de falha. A replicação ponto a ponto (ver Figura 4.3) ataca esses problemas por não ter

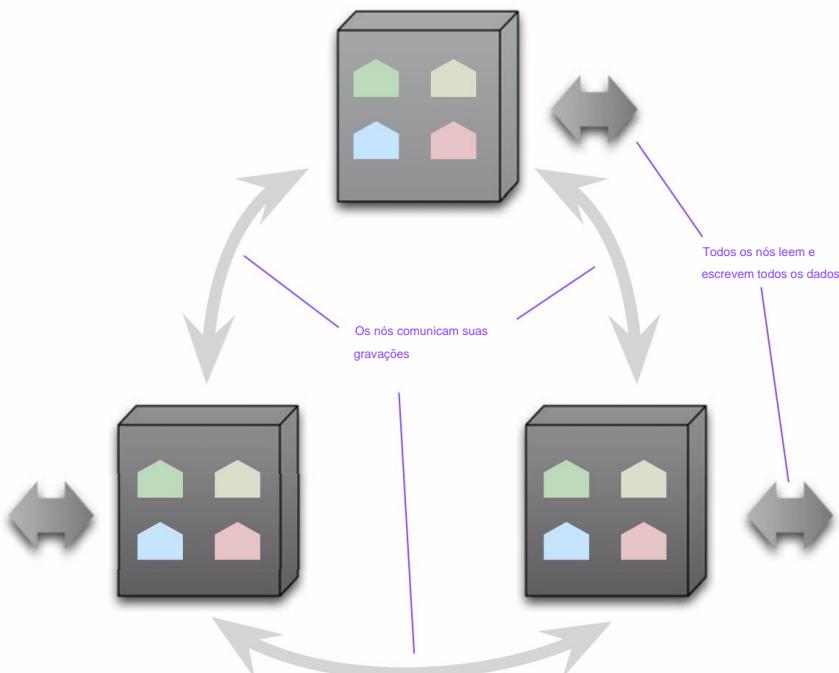


Figura 4.3 A replicação ponto a ponto tem todos os nós aplicando leituras e gravações em todos os dados.

4.5 Combinando Fragmentação e Replicação

um mestre. Todas as réplicas têm peso igual, todas podem aceitar gravações, e a perda de qualquer uma delas não impede o acesso ao armazenamento de dados.

A perspectiva aqui parece muito boa. Com um cluster de replicação ponto a ponto, você pode superar falhas de nó sem perder acesso aos dados. Além disso, você pode facilmente adicionar nós para melhorar seu desempenho. Há muito o que gostar aqui — mas há complicações.

A maior complicação é, novamente, a consistência. Quando você pode gravar em dois lugares diferentes, corre o risco de que duas pessoas tentem atualizar o mesmo registro ao mesmo tempo — um conflito de gravação-gravação. Inconsistências na leitura levam a problemas, mas pelo menos são relativamente transitórias. Gravações inconsistentes são para sempre.

Falaremos mais sobre como lidar com inconsistências de gravação mais tarde, mas por enquanto vamos observar algumas opções amplas. Em uma ponta, podemos garantir que sempre que gravarmos dados, as réplicas se coordenem para garantir que evitemos um conflito. Isso pode nos dar uma garantia tão forte quanto a de um master, embora ao custo do tráfego de rede para coordenar as gravações. Não precisamos que todas as réplicas concordem com a gravação, apenas uma maioria, então ainda podemos sobreviver perdendo uma minoria dos nós de réplica.

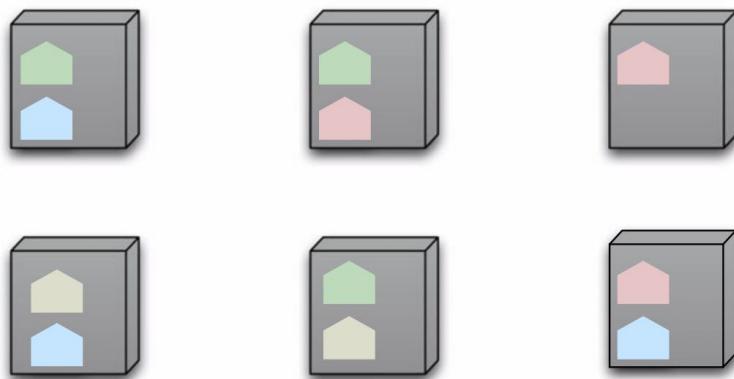
No outro extremo, podemos decidir lidar com uma gravação inconsistente. Há contextos em que podemos criar uma política para mesclar gravações inconsistentes. Nesse caso, podemos obter o benefício total de desempenho da gravação em qualquer réplica.

Esses pontos estão nas extremidades de um espectro onde trocamos consistência por disponibilidade.

4.5 Combinando Fragmentação e Replicação

Replicação e fragmentação são estratégias que podem ser combinadas. Se usarmos replicação mestre-escravo e fragmentação (veja Figura 4.4), isso significa que temos vários mestres, mas cada item de dados tem apenas um único mestre. Dependendo da sua configuração, você pode escolher um nó para ser um mestre para alguns dados e escravos para outros, ou pode dedicar nós para tarefas de mestre ou escravo.

Usar replicação peer-to-peer e sharding é uma estratégia comum para bancos de dados de família de colunas. Em um cenário como esse, você pode ter dezenas ou centenas de nós em um cluster com dados sharded sobre eles. Um bom ponto de partida para replicação peer-to-peer é ter um fator de replicação de 3, para que cada shard esteja presente em três nós. Se um nó falhar, os shards naquele nó serão construídos nos outros nós (veja a Figura 4.5).

**Figura 4.4** Usando replicação mestre-escravo junto com fragmentação**Figura 4.5** Usando replicação ponto a ponto junto com fragmentação

4.6 Pontos-chave

- Existem dois estilos de distribuição de dados:
 - O sharding distribui dados diferentes em vários servidores, de modo que cada o servidor atua como a única fonte para um subconjunto de dados.

- A replicação copia dados em vários servidores, para que cada bit de dados possa ser encontrado em vários lugares.

Um sistema pode usar uma ou ambas as técnicas.

- A replicação ocorre em duas formas:

- A replicação mestre-escravo torna um nó a cópia autoritativa que manipula gravações, enquanto os escravos sincronizam com o mestre e podem manipular leituras.
- A replicação ponto a ponto permite gravações em qualquer nó; os nós se coordenam para sincronizar suas cópias dos dados.

A replicação mestre-escravo reduz a chance de conflitos de atualização, mas a replicação ponto a ponto evita carregar todas as gravações em um único ponto de falha.

Esta página foi deixada em branco intencionalmente

Capítulo 5

Consistência

Uma das maiores mudanças de um banco de dados relacional centralizado para um banco de dados NoSQL orientado a cluster está em como você pensa sobre consistência. Bancos de dados relacionais tentam exibir **forte consistência** evitando todas as várias inconsistências que discutiremos em breve. Uma vez que você começa a olhar para o mundo NoSQL, frases como "teorema CAP" e "consistência eventual" aparecem, e assim que você começa a construir algo, você tem que pensar sobre que tipo de consistência você precisa para seu sistema.

A consistência vem em várias formas, e essa palavra abrange uma miríade de maneiras pelas quais os erros podem se infiltrar em sua vida. Então, vamos começar falando sobre as várias formas que a consistência pode assumir. Depois disso, discutiremos por que você pode querer relaxar a consistência (e sua irmã mais velha, a durabilidade).

5.1 Consistência de atualização

Começaremos considerando atualizar um número de telefone. Coincidentemente, Martin e Pramod estão olhando o site da empresa e notam que o número de telefone está desatualizado. Implausivelmente, ambos têm acesso de atualização, então ambos entram ao mesmo tempo para atualizar o número. Para tornar o exemplo interessante, assumiremos que eles o atualizam de forma ligeiramente diferente, porque cada um usa um formato ligeiramente diferente. Esse problema é chamado de **conflito de gravação-gravação**: duas pessoas atualizando o mesmo item de dados ao mesmo tempo.

Quando as gravações chegam ao servidor, o servidor as **serializa** — decide aplicar uma, depois a outra. Vamos supor que ele use ordem alfabética e escolha a atualização de Martin primeiro, depois a de Pramod. Sem nenhum controle de simultaneidade, a atualização de Martin seria aplicada e imediatamente substituída pela de Pramod. Neste caso, a de Martin é uma **atualização perdida**. Aqui, a atualização perdida não é um grande problema, mas frequentemente é. Vemos isso como uma falha de consistência porque a atualização de Pramod foi baseada no estado antes da atualização de Martin, mas foi aplicada depois dela.

Abordagens para manter a consistência diante da simultaneidade são frequentemente descritas como pessimistas ou otimistas. Uma abordagem **pessimista** funciona prevenindo a ocorrência de conflitos; uma abordagem **otimista** permite que os conflitos ocorram, mas os detecta e toma medidas para resolvê-los. Para conflitos de atualização, a abordagem pessimista mais comum é ter bloqueios de gravação, de modo que, para alterar um valor, você precisa adquirir um bloqueio, e o sistema garante que apenas um cliente possa obter um bloqueio por vez. Então, Martin e Pramod tentariam adquirir o bloqueio de gravação, mas apenas Martin (o primeiro) teria sucesso. Pramod então veria o resultado da gravação de Martin antes de decidir se faria sua própria atualização.

Uma abordagem otimista comum é uma **atualização condicional** onde qualquer cliente que faz uma atualização testa o valor logo antes de atualizá-lo para ver se ele mudou desde sua última leitura. Neste caso, a atualização de Martin teria sucesso, mas a de Pramod falharia. O erro deixaria Pramod saber que ele deveria olhar o valor novamente e decidir se tentaria uma atualização adicional.

Ambas as abordagens pessimista e otimista que acabamos de descrever dependem de uma serialização consistente das atualizações. Com um único servidor, isso é óbvio — ele tem que escolher um, depois o outro. Mas se houver mais de um servidor, como na replicação ponto a ponto, então dois nós podem aplicar as atualizações em uma ordem diferente, resultando em um valor diferente para o número de telefone em cada ponto.

Frequentemente, quando as pessoas falam sobre simultaneidade em sistemas distribuídos, elas falam sobre consistência sequencial, garantindo que todos os nós apliquem operações na mesma ordem.

Há outra maneira otimista de lidar com um conflito de gravação-gravação: salvar as duas atualizações e registrar que elas estão em conflito. Essa abordagem é familiar para muitos programadores de sistemas de controle de versão, particularmente sistemas de controle de versão distribuídos que, por sua natureza, frequentemente terão confirmações conflitantes. O próximo passo novamente segue do controle de versão: você tem que mesclar as duas atualizações de alguma forma. Talvez você mostre os dois valores ao usuário e peça para ele resolver — é isso que acontece se você atualizar o mesmo contato no seu telefone e no seu computador. Alternativamente, o computador pode ser capaz de executar a mesclagem sozinho; se for um problema de formatação do telefone, ele pode ser capaz de perceber isso e aplicar o novo número com o formato padrão. Qualquer mesclagem automatizada de conflitos de gravação-gravação é altamente específica do domínio e precisa ser programada para cada caso particular.

Frequentemente, quando as pessoas encontram esses problemas pela primeira vez, sua reação é preferir a simultaneidade pessimista porque estão determinadas a evitar conflitos. Embora em alguns casos essa seja a resposta certa, sempre há uma compensação. A programação simultânea envolve uma compensação fundamental entre segurança (evitar erros como conflitos de atualização) e vivacidade (responder rapidamente aos clientes). Abordagens pessimistas frequentemente degradam severamente a capacidade de resposta de um sistema a ponto de ele se tornar inadequado para seu propósito. Esse problema é agravado pelo perigo de erros — a simultaneidade pessimista frequentemente leva a deadlocks, que são difíceis de prevenir e depurar.

A replicação torna muito mais provável que ocorram conflitos de gravação-gravação. Se diferentes nós tiverem cópias diferentes de alguns dados que podem ser atualizados independentemente, você terá conflitos, a menos que tome medidas específicas para evitá-los.

Usar um único nó como alvo para todas as gravações de alguns dados torna isso muito mais fácil para manter a consistência da atualização. Dos modelos de distribuição que discutimos anteriormente, todos, exceto a replicação ponto a ponto, fazem isso.

5.2 Consistência de leitura

Ter um armazenamento de dados que mantém a consistência das atualizações é uma coisa, mas não garantir que os leitores desse armazenamento de dados sempre receberão respostas consistentes suas solicitações. Vamos imaginar que temos um pedido com itens de linha e um frete taxa. A taxa de envio é calculada com base nos itens de linha do pedido. Se adicionarmos um item de linha, portanto também precisamos recalcular e atualizar o frete cobrar. Em um banco de dados relacional, a taxa de envio e os itens de linha estarão em tabelas separadas. O perigo da inconsistência é que Martin adiciona um item de linha a seu pedido, Pramod então lê os itens de linha e o custo de envio e, em seguida, Martin atualiza a taxa de envio. Este é um **conflito de leitura inconsistente** ou de leitura-escrita:

Na Figura 5.1, Pramod fez uma leitura no meio da escrita de Martin.

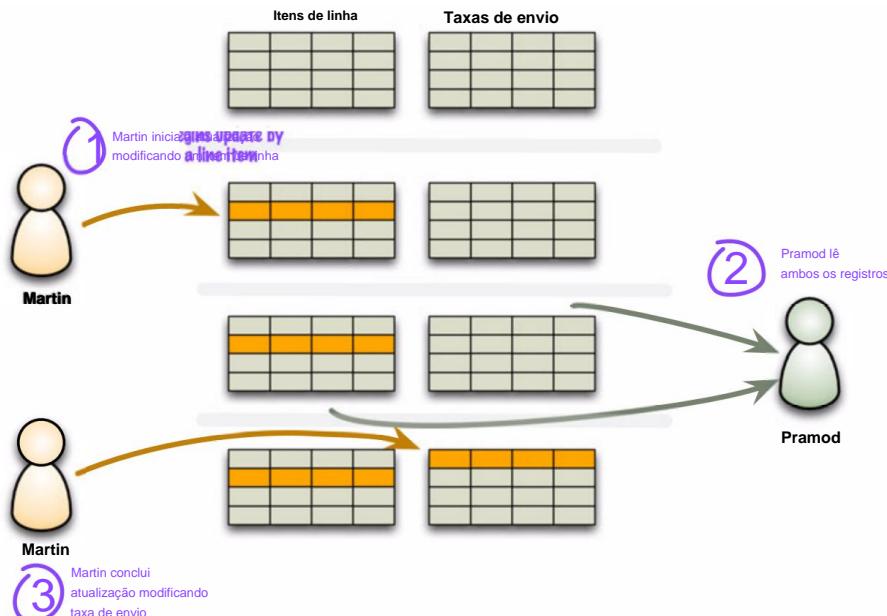


Figura 5.1 Um conflito de leitura e escrita na consistência lógica

Nós nos referimos a esse tipo de consistência como **consistência lógica**: garantindo que diferentes itens de dados façam sentido juntos. Para evitar um conflito de leitura-escrita logicamente inconsistente, bancos de dados relacionais dão suporte à noção de transações. Desde que Martin envolva suas duas escritas em uma transação, o sistema garante que Pramod lerá ambos os itens de dados antes da atualização ou ambos depois da atualização.

Uma alegação comum que ouvimos é que bancos de dados NoSQL não suportam transações e, portanto, não podem ser consistentes. Tal alegação é, em grande parte, errada porque ignora muitos detalhes importantes. Nossa primeira esclarecimento é que qualquer declaração sobre a falta de transações geralmente se aplica apenas a alguns bancos de dados NoSQL, em particular os orientados a agregados. Em contraste, bancos de dados de grafos tendem a suportar transações ACID da mesma forma que bancos de dados relacionais.

Em segundo lugar, bancos de dados orientados a agregados suportam atualizações atômicas, mas apenas dentro de um único agregado. Isso significa que você terá consistência lógica dentro de um agregado, mas não entre agregados. Então, no exemplo, você poderia evitar essa inconsistência se o pedido, a taxa de entrega e os itens de linha forem todos parte de um único agregado de pedido.

Claro que nem todos os dados podem ser colocados no mesmo agregado, então qualquer atualização que afete vários agregados deixa em aberto um momento em que os clientes poderiam executar uma leitura inconsistente. O período de tempo em que uma inconsistência está presente é chamado de **janela de inconsistência**. Um sistema NoSQL pode ter uma janela de inconsistência bem curta: como um ponto de dados, a documentação da Amazon diz que a janela de inconsistência para seu serviço SimpleDB é geralmente menor que um segundo.

Este exemplo de uma leitura logicamente inconsistente é o exemplo clássico que você verá em qualquer livro que toque programação de banco de dados. Uma vez que você introduz a replicação, no entanto, você obtém um tipo totalmente novo de inconsistência. Vamos imaginar que há um último quarto de hotel para um evento desejável. O sistema de reserva de hotel roda em muitos nós. Martin e Cindy são um casal considerando este quarto, mas eles estão discutindo isso no telefone porque Martin está em Londres e Cindy está em Boston.

Enquanto isso, Pramod, que está em Mumbai, vai e reserva o último quarto. Isso atualiza a disponibilidade do quarto replicado, mas a atualização chega a Boston mais rápido do que a Londres. Quando Martin e Cindy abrem seus navegadores para ver se o quarto está disponível, Cindy o vê reservado e Martin o vê livre. Esta é outra leitura inconsistente — mas é uma violação de uma forma diferente de consistência que chamamos de **consistência de replicação**: garantir que o mesmo item de dados tenha o mesmo valor quando lido de réplicas diferentes (veja a Figura 5.2).

Eventualmente, é claro, as atualizações serão propagadas completamente, e Martin verá que a sala está totalmente reservada. Portanto, essa situação é geralmente chamada de **eventualmente consistente**, o que significa que a qualquer momento os nós podem ter inconsistências de replicação, mas, se não houver mais atualizações, eventualmente todos os nós serão atualizados para o mesmo valor. Dados desatualizados são geralmente chamados de **obsoletos**, o que nos lembra que um cache é outra forma de replicação — essencialmente seguindo o modelo de distribuição mestre-escravo.

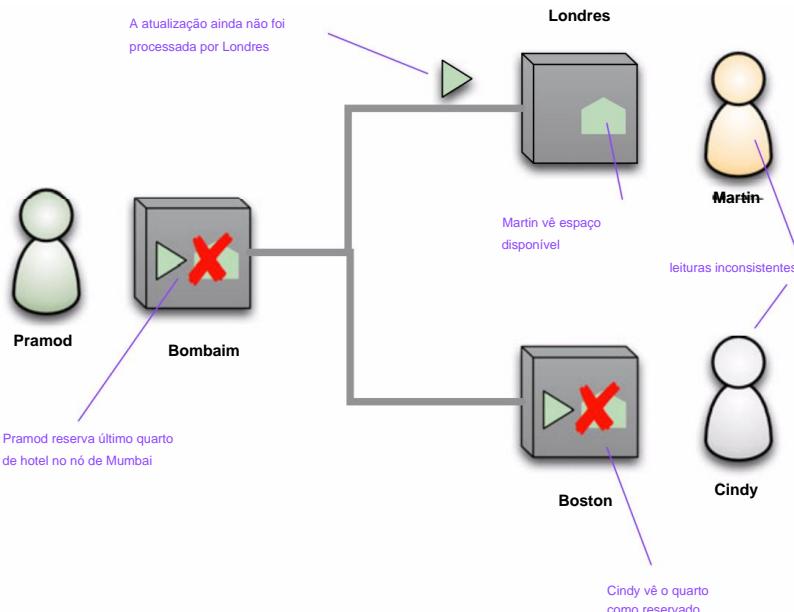


Figura 5.2 Um exemplo de inconsistência de replicação

Embora a consistência da replicação seja independente da consistência lógica, a replicação pode exacerbar uma inconsistência lógica ao aumentar sua janela de inconsistência. Duas atualizações diferentes no mestre podem ser executadas em rápida sucessão, deixando uma janela de inconsistência de milissegundos. Mas atrasos na rede podem significar que a mesma janela de inconsistência dura muito mais em um escravo.

Garantias de consistência não são algo global para um aplicativo. Normalmente, você pode especificar o nível de consistência que deseja com solicitações individuais.

Isso permite que você use consistência fraca na maioria das vezes, quando isso não é um problema, mas solicite consistência forte quando isso é um problema.

A presença de uma janela de inconsistência significa que pessoas diferentes verão coisas diferentes ao mesmo tempo. Se Martin e Cindy estiverem olhando salas durante uma chamada transatlântica, isso pode causar confusão. É mais comum que os usuários ajam de forma independente, e então isso não é um problema. Mas janelas de inconsistência podem ser particularmente problemáticas quando você tem inconsistências consigo mesmo. Considere o exemplo de postar comentários em uma entrada de blog. Poucas pessoas vão se preocupar com janelas de inconsistência de até mesmo alguns minutos enquanto as pessoas estão digitando seus pensamentos mais recentes. Frequentemente, os sistemas lidam com a carga de tais sites executando em um cluster e balanceando a carga de solicitações de entrada para diferentes nós.

Aí está um perigo: você pode postar uma mensagem usando um nó e, em seguida, atualizar seu navegador, mas a atualização vai para um nó diferente que ainda não recebeu sua postagem — e parece que sua postagem foi perdida.

Em situações como essa, você pode tolerar janelas de inconsistência razoavelmente longas, mas precisa de **consistência de leitura-suas-escritas**, o que significa que, depois de fazer uma atualização, você tem a garantia de continuar vendo essa atualização. Uma maneira de obter isso em um sistema eventualmente consistente é fornecer **consistência de sessão**: dentro da sessão de um usuário, há consistência de leitura-suas-escritas. Isso significa que o usuário pode perder essa consistência caso sua sessão termine por algum motivo ou caso o usuário acesse o mesmo sistema simultaneamente de computadores diferentes, mas esses casos são relativamente raros.

Existem algumas técnicas para fornecer consistência de sessão. Uma maneira comum, e geralmente a mais fácil, é ter uma **sessão fixa**: uma sessão que está vinculada a um nó (isso também é chamado de **afinidade de sessão**). Uma sessão fixa permite que você garanta que, enquanto você mantiver a consistência de leitura-suas-escritas em um nó, você a obterá para sessões também. A desvantagem é que as sessões fixas reduzem a capacidade do平衡ador de carga de fazer seu trabalho.

Outra abordagem para consistência de sessão é usar carimbos de versão ("Carimbos de versão", p. 61) e garantir que cada interação com o armazenamento de dados inclua o carimbo de versão mais recente visto por uma sessão. O nó do servidor deve então garantir que ele tenha as atualizações que incluem esse carimbo de versão antes de responder a uma solicitação.

Manter a consistência da sessão com sessões persistentes e replicação mestre-escravo pode ser estranho se você quiser ler dos escravos para melhorar o desempenho de leitura, mas ainda precisar gravar no mestre. Uma maneira de lidar com isso é enviar gravações para o escravo, que então assume a responsabilidade de encaminhá-las para o mestre, mantendo a consistência da sessão para seu cliente. Outra abordagem é alternar a sessão para o mestre temporariamente ao fazer uma gravação, apenas o tempo suficiente para que as leituras sejam feitas do mestre até que os escravos alcancem a atualização.

Estamos falando sobre consistência de replicação no contexto de um armazenamento de dados, mas também é um fator importante no design geral do aplicativo. Mesmo um sistema de banco de dados simples terá muitas ocasiões em que os dados são apresentados a um usuário, o usuário cogita e, em seguida, atualiza esses dados. Geralmente é uma má ideia manter uma transação aberta durante a interação do usuário porque há um perigo real de conflitos quando o usuário tenta fazer sua atualização, o que leva a abordagens como bloqueios offline [Fowler PoEAA].

5.3 Consistência Relaxante

Consistência é uma coisa boa — mas, infelizmente, às vezes temos que sacrificá-la. É sempre possível projetar um sistema para evitar inconsistências, mas muitas vezes é impossível fazer isso sem fazer sacrifícios insuportáveis em outras características do sistema. Como resultado, muitas vezes temos que trocar consistência por outra coisa.

Enquanto alguns arquitectos veem isto como um desastre, nós vemos isto como parte do inevitável

tradeoffs envolvidos no design do sistema. Além disso, diferentes domínios têm diferentes tolerâncias para inconsistência, e precisamos levar essa tolerância em consideração ao tomarmos nossas decisões.

Negociar a consistência é um conceito familiar mesmo em sistemas de banco de dados relacionais de servidor único. Aqui, nossa principal ferramenta para impor consistência é a transação, e as transações podem fornecer fortes garantias de consistência. No entanto, os sistemas de transação geralmente vêm com a capacidade de relaxar os níveis de isolamento, permitindo que as consultas leiam dados que ainda não foram confirmados, e na prática vemos a maioria dos aplicativos relaxar a consistência do nível de isolamento mais alto (serializado) para obter desempenho eficaz. Mais comumente vemos pessoas usando o nível de transação de leitura confirmada, que elimina alguns conflitos de leitura e gravação, mas permite outros.

Muitos sistemas renunciam a transações completamente porque o impacto das transações no desempenho é muito alto. Vimos isso de algumas maneiras diferentes. Em pequena escala, vimos a popularidade do MySQL durante os dias em que ele não suportava transações. Muitos sites gostaram da alta velocidade do MySQL e estavam preparados para viver sem transações. Na outra ponta da escala, alguns sites muito grandes, como o eBay [Pritchett], tiveram que renunciar a transações para ter um desempenho aceitável — isso é particularmente verdadeiro quando você precisa introduzir sharding. Mesmo sem essas restrições, muitos construtores de aplicativos precisam interagir com sistemas remotos que não podem ser incluídos adequadamente dentro de um limite de transação, portanto, atualizar fora das transações é uma ocorrência bastante comum para aplicativos corporativos.

5.3.1 O Teorema CAP

No mundo NoSQL é comum se referir ao teorema CAP como a razão pela qual você pode precisar relaxar a consistência. Ele foi originalmente proposto por Eric Brewer em 2000 [Brewer] e recebeu uma prova formal de Seth Gilbert e Nancy Lynch [Lynch e Gilbert] alguns anos depois. (Você também pode ouvir isso sendo chamado de Conjectura de Brewer.)

A declaração básica do teorema CAP é que, dadas as três propriedades de Consistência, Disponibilidade e Tolerância de partição, você só pode obter duas. Obviamente, isso depende muito de como você define essas três propriedades, e opiniões divergentes levaram a vários debates sobre quais são as consequências reais do teorema CAP.

Consistência é basicamente como a definimos até agora. **Disponibilidade** tem um significado particular no contexto do CAP — significa que se você pode falar com um nó no cluster, ele pode ler e gravar dados. Isso é sutilmente diferente do significado usual, que exploraremos mais tarde. **Tolerância de partição** significa que o cluster pode sobreviver a quebras de comunicação no cluster que separam o cluster em várias partições incapazes de se comunicar entre si (situação conhecida como **split brain**, veja Figura 5.3).

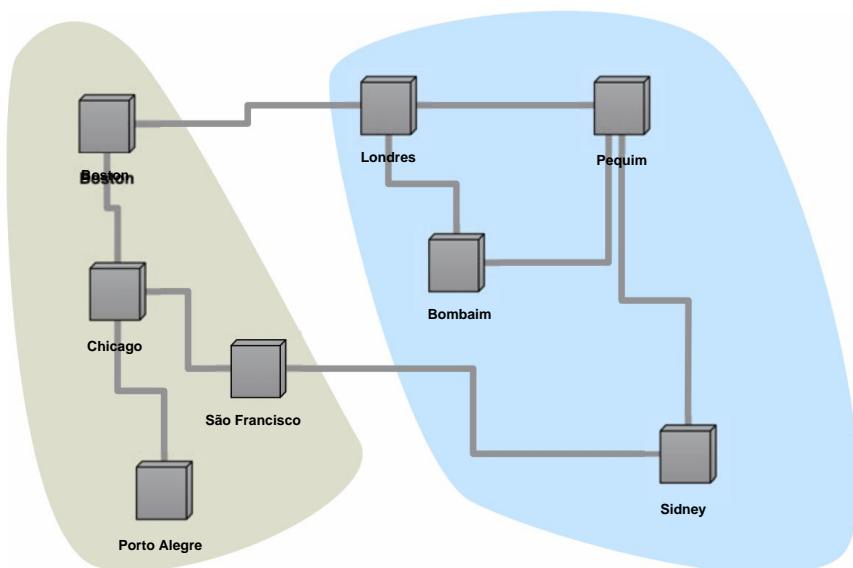


Figura 5.3 Com duas interrupções nas linhas de comunicação, a rede se divide em dois grupos.

Um sistema de servidor único é o exemplo óbvio de um sistema CA — um sistema que tem Consistência e Disponibilidade, mas não Tolerância de Partição. Uma única máquina não pode particionar, então ela não precisa se preocupar com tolerância de partição. Há apenas um nó — então se ele estiver ativo, ele está disponível. Estar ativo e manter a consistência é razoável. Este é o mundo em que a maioria dos sistemas de banco de dados relacionais vive.

É teoricamente possível ter um cluster CA. No entanto, isso significaria que se uma partição ocorresse no cluster, todos os nós no cluster ficariam inativos, de modo que nenhum cliente poderia falar com um nó. Pela definição usual de "disponível", isso significaria uma falta de disponibilidade, mas é aqui que o uso especial de "disponibilidade" do CAP fica confuso. O CAP define "disponibilidade" como "cada solicitação recebida por um nó não falho no sistema deve resultar em uma resposta".

[Lynch e Gilbert]. Então, um nó com falha e sem resposta não infere uma falta de disponibilidade de CAP.

Isso implica que você pode construir um cluster CA, mas você tem que garantir que ele só particione raramente e completamente. Isso pode ser feito, pelo menos dentro de um data center, mas geralmente é proibitivamente caro. Lembre-se de que, para derrubar todos os nós em um cluster em uma partição, você também tem que detectar a partição em tempo hábil — o que por si só não é pouca coisa.

Então, os clusters precisam ser tolerantes a partições de rede. E aqui está o ponto real do teorema CAP. Embora o teorema CAP seja frequentemente declarado como "você só pode obter dois de três", na prática o que ele está dizendo é que em um sistema que pode sofrer partições, como os sistemas distribuídos, você tem que negociar a consistência.

versus disponibilidade. Esta não é uma decisão binária; frequentemente, você pode negociar um pouco de consistência para obter alguma disponibilidade. O sistema resultante não seria nem perfeitamente consistente nem perfeitamente disponível — mas teria uma combinação que é razoável para suas necessidades particulares.

Um exemplo deve ajudar a ilustrar isso. Martin e Pramod estão ambos tentando reservar o último quarto de hotel em um sistema que usa distribuição peer-to-peer com dois nós (Londres para Martin e Mumbai para Pramod). Se quisermos garantir consistência, então quando Martin tenta reservar seu quarto no nó Londres, esse nó deve se comunicar com o nó Mumbai antes de confirmar a reserva.

Essencialmente, ambos os nós devem concordar com a serialização de suas solicitações. Isso nos dá consistência — mas se o link de rede quebrar, nenhum sistema pode reservar nenhum quarto de hotel, sacrificando a disponibilidade.

Uma maneira de melhorar a disponibilidade é designar um nó como o mestre para um hotel específico e garantir que todas as reservas sejam processadas por esse mestre. Se esse mestre for Mumbai, Mumbai ainda poderá processar reservas de hotel para esse hotel e Pramod obterá o último quarto. Se usarmos replicação mestre-escravo, os usuários de Londres poderão ver as informações inconsistentes do quarto, mas não poderão fazer uma reserva e, portanto, causar uma inconsistência de atualização. No entanto, os usuários esperam que isso possa acontecer nessa situação — então, novamente, o compromisso funciona para esse caso de uso específico.

Isso melhora a situação, mas ainda não podemos reservar um quarto no nó de Londres para o hotel cujo master está em Mumbai se a conexão cair. Na terminologia CAP, isso é uma falha de disponibilidade, pois Martin pode falar com o nó de Londres, mas o nó de Londres não pode atualizar os dados. Para obter mais disponibilidade, podemos permitir que ambos os sistemas continuem aceitando reservas de hotel, mesmo se o link de rede quebrar. O perigo aqui é que Martin e Pramod reservem o último quarto de hotel. No entanto, dependendo de como esse hotel opera, isso pode ser bom. Muitas vezes, as empresas de viagens toleram uma certa quantidade de overbooking para lidar com não comparecimentos. Por outro lado, alguns hotéis sempre mantêm alguns quartos livres, mesmo quando estão totalmente reservados, para poder trocar um hóspede de um quarto com problemas ou para acomodar uma reserva tardia de alto status.

Alguns podem até cancelar a reserva com um pedido de desculpas assim que detectarem o conflito, argumentando que o custo disso é menor do que o custo de perder reservas devido a falhas de rede.

O exemplo clássico de permitir gravações inconsistentes é o carrinho de compras, conforme discutido no Dynamo [Dynamo da Amazon]. Nesse caso, você sempre tem permissão para gravar no seu carrinho de compras, mesmo que falhas de rede signifiquem que você acabe com vários carrinhos de compras. O processo de checkout pode mesclar os dois carrinhos de compras colocando a união dos itens dos carrinhos em um único carrinho e retornando-o. Quase sempre essa é a resposta correta — mas se não for, o usuário tem a oportunidade de olhar o carrinho antes de concluir o pedido.

A lição aqui é que, embora a maioria dos desenvolvedores de software trate a consistência de atualização como The Way Things Must Be, há casos em que você pode lidar graciosamente com respostas inconsistentes a solicitações. Essas situações estão intimamente ligadas à

domínio e exigem conhecimento de domínio para saber como resolver. Portanto, você geralmente não pode procurar resolvê-los puramente dentro da equipe de desenvolvimento — você tem que falar com especialistas em domínio. Se você puder encontrar uma maneira de lidar com atualizações inconsistentes, isso lhe dará mais opções para aumentar a disponibilidade e o desempenho. Para um carrinho de compras, significa que os compradores sempre podem comprar, e fazer isso rapidamente. E como americanos patriotas, sabemos o quanto vital é apoiar o Nosso Destino de Varejo.

Uma lógica similar se aplica à consistência de leitura. Se você estiver negociando instrumentos financeiros em uma bolsa computadorizada, você pode não ser capaz de tolerar quaisquer dados que não estejam atualizados. No entanto, se você estiver postando um item de notícias em um site de mídia, você pode ser capaz de tolerar páginas antigas por minutos.

Nesses casos, você precisa saber o quanto tolerante você é com leituras obsoletas e qual pode ser a duração da janela de inconsistência — geralmente em termos de comprimento médio, pior caso e alguma medida da distribuição para os comprimentos. Itens de dados diferentes podem ter tolerâncias diferentes para obsolescência e, portanto, podem precisar de configurações diferentes na sua configuração de replicação.

Os defensores do NoSQL costumam dizer que, em vez de seguir as propriedades ACID de transações relacionais, os sistemas NoSQL seguem as propriedades BASE (Basically Available, Soft state, Eventual consistent) [Brewer]. Embora sintamos que devemos mencionar a sigla BASE aqui, não achamos que seja muito útil. A sigla é ainda mais artificial do que ACID, e nem “basically available” nem “soft state” foram bem definidos. Também devemos enfatizar que, quando Brewer introduziu a noção de BASE, ele viu a troca entre ACID e BASE como um espectro, não uma escolha binária.

Incluímos esta discussão do teorema CAP porque ele é frequentemente usado (e abusado) ao falar sobre as compensações envolvendo consistência em bancos de dados distribuídos. No entanto, geralmente é melhor pensar não sobre a compensação entre consistência e disponibilidade, mas sim entre consistência e *latência*. Podemos resumir grande parte da discussão sobre consistência na distribuição dizendo que podemos melhorar a consistência ao envolver mais nós na interação, mas cada nó que adicionamos aumenta o tempo de resposta dessa interação. Podemos então pensar na disponibilidade como o limite de latência que estamos preparados para tolerar; uma vez que a latência fica muito alta, desistimos e tratamos os dados como indisponíveis — o que se encaixa perfeitamente em sua definição no contexto do CAP.

5.4 Durabilidade Relaxante

Até agora falamos sobre consistência, que é a maior parte do que as pessoas querem dizer quando falam sobre as propriedades ACID de transações de banco de dados. A chave para Consistência é serializar solicitações formando unidades de trabalho Atômicas e Isoladas. Mas a maioria das pessoas zombaria de relaxar a durabilidade — afinal, qual é o sentido de um armazenamento de dados se ele pode perder atualizações?

Acontece que há casos em que você pode querer trocar alguma durabilidade por um desempenho maior. Se um banco de dados pode ser executado principalmente na memória, aplicar atualizações à sua representação na memória e liberar periodicamente as alterações no disco, então ele pode ser capaz de fornecer uma responsividade substancialmente maior às solicitações. O custo é que, se o servidor travar, todas as atualizações desde a última liberação serão perdidas.

Um exemplo de onde essa troca pode valer a pena é armazenar o estado da sessão do usuário. Um grande site pode ter muitos usuários e manter informações temporárias sobre o que cada usuário está fazendo em algum tipo de estado de sessão. Há muita atividade nesse estado, criando muita demanda, o que afeta a capacidade de resposta do site.

O ponto vital é que perder os dados da sessão não é uma tragédia tão grande — isso criará algum incômodo, mas talvez menos do que um site mais lento causaria. Isso o torna um bom candidato para gravações não duráveis. Muitas vezes, você pode especificar as necessidades de durabilidade em uma base chamada por chamada, para que atualizações mais importantes possam forçar uma descarga no disco.

Outro exemplo de durabilidade relaxante é capturar dados telemétricos de dispositivos físicos. Pode ser que você prefira capturar dados em uma taxa mais rápida, ao custo de perder as últimas atualizações caso o servidor caia.

Outra classe de tradeoffs de durabilidade surge com dados replicados. Uma falha de **durabilidade de replicação** ocorre quando um nó processa uma atualização, mas falha antes que essa atualização seja replicada para os outros nós. Um caso simples disso pode acontecer se você tiver um modelo de distribuição mestre-escravo, onde os escravos nomeiam um novo mestre automaticamente, caso o mestre existente falhe. Se um mestre falhar, todas as gravações não passadas para as réplicas serão efetivamente perdidas. Se o mestre voltar a ficar online, essas atualizações entrarão em conflito com as atualizações que ocorreram desde então.

Pensamos nisso como um problema de durabilidade porque você acha que sua atualização foi bem-sucedida desde que o mestre a reconheceu, mas uma falha do nó mestre fez com que ela fosse perdida.

Se você estiver suficientemente confiante em colocar o mestre de volta online rapidamente, esta é uma razão para não fazer failover automático para um escravo. Caso contrário, você pode melhorar a durabilidade da replicação garantindo que o mestre espere algumas réplicas para reconhecer a atualização antes que o mestre a reconheça para o cliente. Obviamente, no entanto, isso tornará as atualizações mais lentas e tornará o cluster indisponível se os escravos falharem — então, novamente, temos uma troca, dependendo de quão vital é a durabilidade. Assim como com a durabilidade básica, é útil para chamadas individuais indicarem qual nível de durabilidade elas precisam.

5.5 Quóruns

Quando você está negociando consistência ou durabilidade, não é uma proposta de tudo ou nada. Quanto mais nós envolve em uma solicitação, maior é a chance de evitar uma inconsistência. Isso naturalmente leva à pergunta: Quantos nós precisam estar envolvidos para obter uma consistência forte?

Imagine alguns dados replicados em três nós. Você não precisa que todos os nós reconheçam uma gravação para garantir uma consistência forte; tudo o que você precisa é de dois deles — uma maioria. Se você tiver gravações conflitantes, apenas uma pode obter a maioria. Isso é chamado de **quorum de gravação** e expresso em uma desigualdade um pouco pretensiosa de $W > N/2$, significando que o número de nós que participam da gravação (W) deve ser maior que a metade do número de nós envolvidos na replicação (N). O número de réplicas é frequentemente chamado de **fator de replicação**.

Similmente ao quorum de gravação, há a noção de quorum de leitura: quantos nós você precisa contatar para ter certeza de que tem a alteração mais atualizada. O quorum de leitura é um pouco mais complicado porque depende de quantos nós precisam confirmar uma gravação.

Vamos considerar um fator de replicação de 3. Se todas as gravações precisarem de dois nós para confirmar ($W = 2$), então precisamos contatar pelo menos dois nós para ter certeza de que obteremos os dados mais recentes. Se, no entanto, as gravações forem confirmadas apenas por um único nó ($W = 1$), precisamos falar com todos os três nós para ter certeza de que temos as últimas atualizações. Neste caso, como não temos um quorum de gravação, podemos ter um conflito de atualização, mas ao contatar leitores suficientes, podemos ter certeza de detectá-lo. Assim, podemos obter leituras fortemente consistentes, mesmo se não tivermos uma consistência forte em nossas gravações.

Essa relação entre o número de nós que você precisa contatar para uma leitura (R), aqueles que confirmam uma gravação (W) e o fator de replicação (N) pode ser capturada em uma desigualdade: você pode ter uma leitura fortemente consistente se $R + W > N$.

Essas desigualdades são escritas com um modelo de distribuição ponto a ponto em mente. Se você tem uma distribuição mestre-escravo, você só precisa escrever no mestre para evitar conflitos de gravação-gravação, e similarmente só precisa ler do mestre para evitar conflitos de leitura-gravação. Com essa notação, é comum confundir o número de nós no cluster com o fator de replicação, mas eles geralmente são diferentes. Eu posso ter 100 nós no meu cluster, mas tenho apenas um fator de replicação de 3, com a maior parte da distribuição ocorrendo devido ao sharding.

De fato, a maioria das autoridades sugere que um fator de replicação de 3 é suficiente para ter boa resiliência. Isso permite que um único nó falhe enquanto ainda mantém quora para leituras e gravações. Se você tiver rebalanceamento automático, não demorará muito para o cluster criar uma terceira réplica, então as chances de perder uma segunda réplica antes que uma substituição apareça são pequenas.

O número de nós que participam de uma operação pode variar de acordo com a operação. Ao escrever, podemos exigir quorum para alguns tipos de atualizações, mas não para outros, dependendo de quanto valorizamos a consistência e a disponibilidade. Da mesma forma, uma leitura que precisa de velocidade, mas pode tolerar obsolescência, deve contatar menos nós.

Muitas vezes você pode precisar levar ambos em consideração. Se você precisa de leituras rápidas e fortemente consistentes, você pode exigir que as gravações sejam reconhecidas por todos os nós, permitindo assim que as leituras contatem apenas um ($N = 3$, $W = 3$, $R = 1$). Isso significaria que suas gravações são lentas, já que elas têm que contatar todos os três nós, e você

não ser capaz de tolerar perder um nó. Mas em algumas circunstâncias essa pode ser a troca a ser feita.

O ponto de tudo isso é que você tem uma gama de opções para trabalhar e pode escolher qual combinação de problemas e vantagens preferir. Alguns escritores sobre NoSQL falam sobre uma troca simples entre consistência e disponibilidade; esperamos que agora você perceba que é mais flexível — e mais complicado — do que isso.

5.6 Leituras Adicionais

Há todo tipo de posts de blog e artigos interessantes na Internet sobre consistência em sistemas distribuídos, mas a fonte mais útil para nós foi [Tanenbaum e Van Steen]. Ele faz um excelente trabalho de organizar muitos dos fundamentos dos sistemas distribuídos e é o melhor lugar para ir se você quiser se aprofundar mais do que fizemos neste capítulo.

Quando estávamos terminando este livro, a *IEEE Computer* publicou uma edição especial [*IEEE Computer* de fevereiro de 2012] sobre a crescente influência do teorema CAP, que é uma fonte útil de esclarecimento adicional sobre este tópico.

5.7 Pontos-chave

- Conflitos de gravação-gravação ocorrem quando dois clientes tentam gravar os mesmos dados ao mesmo tempo. Conflitos de leitura-gravação ocorrem quando um cliente lê dados inconsistentes no meio da gravação de outro cliente.
- Abordagens pessimistas bloqueiam registros de dados para evitar conflitos. Otimistas abordagens detectam conflitos e os corrigem.
- Os sistemas distribuídos veem conflitos de leitura-escrita devido a alguns nós terem recebido atualizações enquanto outros nós não. Consistência eventual significa que em algum ponto o sistema se tornará consistente uma vez que todas as escritas tenham se propagado para todos os nós.
- Os clientes geralmente querem consistência de leitura-suas-escritas, o que significa que um cliente pode escrever e então ler imediatamente o novo valor. Isso pode ser difícil se a leitura e a escrita acontecerem em nós diferentes.
- Para obter uma boa consistência, você precisa envolver muitos nós em operações de dados, mas isso aumenta a latência. Então, muitas vezes, você tem que negociar consistência versus latência.

- O teorema CAP afirma que, se você obtiver uma partição de rede, terá que equilibrar a disponibilidade dos dados com a consistência.
- A durabilidade também pode ser compensada pela latência, principalmente se você quiser para sobreviver a falhas com dados replicados.
- Você não precisa entrar em contato com todos os replicantes para preservar uma consistência forte com a replicação; você só precisa de um quórum grande o suficiente.

P2P

Capítulo 6

Selos de versão

Usado no
caso
Otimistic

Muitos críticos dos bancos de dados NoSQL se concentram na falta de suporte para transações. Transações são uma ferramenta útil que ajuda os programadores a dar suporte à consistência. Uma razão pela qual muitos proponentes do NoSQL se preocupam menos com a falta de transações é que os bancos de dados NoSQL orientados a agregados dão suporte a atualizações atômicas dentro de um agregado — e os agregados são projetados para que seus dados formem uma unidade natural de atualização. Dito isso, é verdade que as necessidades transacionais são algo a ser levado em conta quando você decide qual banco de dados usar.

Como parte disso, é importante lembrar que as transações têm limitações. Mesmo dentro de um sistema transacional, ainda temos que lidar com atualizações que exigem intervenção humana e geralmente não podem ser executadas dentro de transações porque envolveriam manter uma transação aberta por muito tempo. Podemos lidar com isso usando carimbos **de versão** — que acabam sendo úteis em outras situações também, particularmente à medida que nos afastamos do modelo de distribuição de servidor único.

6.1 Transações comerciais e de sistema

A necessidade de oferecer suporte à consistência de atualizações sem transações é, na verdade, um recurso comum dos sistemas, mesmo quando eles são construídos sobre bancos de dados transacionais. Quando os usuários pensam em transações, eles geralmente se referem a **transações comerciais**. Uma transação comercial pode ser algo como navegar em um catálogo de produtos, escolher uma garrafa de Talisker a um bom preço, preencher informações de cartão de crédito e confirmar o pedido. No entanto, tudo isso geralmente não ocorrerá dentro da **transação do sistema** fornecida pelo banco de dados porque isso significaria bloquear os elementos do banco de dados enquanto o usuário está tentando encontrar seu cartão de crédito e é chamado para almoçar por seus colegas.

Normalmente, os aplicativos só iniciam uma transação do sistema no final da interação com o usuário, de modo que os bloqueios são mantidos apenas por um curto período de tempo. O problema, no entanto, é que cálculos e decisões podem ter sido feitos com base em dados que foram alterados. A lista de preços pode ter atualizado o preço do Talisker, ou alguém pode ter atualizado o endereço do cliente, alterando as taxas de envio.

As técnicas amplas para lidar com isso são a simultaneidade offline [Fowler PoEAA], útil também em situações NoSQL. Uma abordagem particularmente útil é o Optimistic Offline Lock [Fowler PoEAA], uma forma de atualização condicional em que uma operação de cliente relê qualquer informação da qual a transação comercial depende e verifica se ela não mudou desde que foi originalmente lida e exibida ao usuário. Uma boa maneira de fazer isso é garantir que os registros no banco de dados contenham alguma forma de **carimbo de versão**: um campo que muda toda vez que os dados subjacentes no registro mudam. Ao ler os dados, você mantém uma nota do carimbo de versão, para que, ao gravar dados, você possa verificar se a versão mudou.

Você pode ter se deparado com essa técnica com atualização de recursos com HTTP [HTTP]. Uma maneira de fazer isso é usar etags. Sempre que você obtém um recurso, o servidor responde com uma etag no cabeçalho. Essa etag é uma string opaca que indica a versão do recurso. Se você então atualizar esse recurso, poderá usar uma atualização condicional fornecendo a etag que você obteve do seu último GET.

Se o recurso tiver sido alterado no servidor, as etags não corresponderão e o servidor recusará a atualização, retornando uma resposta 412 (Falha na pré-condição).

etrag
412

Alguns bancos de dados fornecem um mecanismo semelhante de atualização condicional que permite que você garanta que as atualizações não serão baseadas em dados obsoletos. Você pode fazer essa verificação sozinho, embora tenha que garantir que nenhuma outra thread possa ser executada no recurso entre sua leitura e sua atualização. (Às vezes, isso é chamado de **operação compare-and-set (CAS)**, cujo nome vem das operações CAS feitas em processadores. A diferença é que um CAS do processador compara um valor antes de defini-lo, enquanto uma atualização condicional do banco de dados compara um carimbo de versão do valor.)

Existem várias maneiras de construir seus carimbos de versão. Você pode usar um contador, sempre incrementando-o quando atualizar o recurso. Os contadores são úteis, pois facilitam saber se uma versão é mais recente do que outra.

Por outro lado, eles exigem que o servidor gere o valor do contador e também precisam de um único mestre para garantir que os contadores não sejam duplicados.

Outra abordagem é criar um GUID, um grande número aleatório que é garantido ser único. Eles usam alguma combinação de datas, informações de hardware e quaisquer outras fontes de aleatoriedade que eles possam captar. O bom dos GUIDs é que eles podem ser gerados por qualquer um e você nunca obterá uma duplicata; uma desvantagem é que eles são grandes e não podem ser comparados diretamente para atualidade.

Uma terceira abordagem é fazer um hash do conteúdo do recurso. Com um tamanho de chave de hash grande o suficiente, um hash de conteúdo pode ser globalmente único como um GUID e também pode ser gerado por qualquer pessoa; a vantagem é que eles são determinísticos — qualquer

6.2 Carimbos de versão em vários nós

node gerará o mesmo hash de conteúdo para os mesmos dados de recurso. No entanto, como GUIDs, eles não podem ser comparados diretamente quanto à atualidade e podem ser longos.

Uma quarta abordagem é usar o timestamp da última atualização. Como contadores, eles são razoavelmente curtos e podem ser comparados diretamente para atualidade, mas têm a vantagem de não precisar de um único mestre. Várias máquinas podem gerar timestamps — mas para funcionar corretamente, seus relógios precisam ser mantidos em sincronia. Um nó com um relógio ruim pode causar todos os tipos de corrupções de dados. Também há o perigo de que, se o timestamp for muito granular, você possa obter duplicatas — não é bom usar timestamps com precisão de milissegundos se você obtém muitas atualizações por milissegundo.

Você pode misturar as vantagens desses diferentes esquemas de carimbo de versão usando mais de um deles para criar um carimbo composto. Por exemplo, o CouchDB usa uma combinação de contador e hash de conteúdo. Na maioria das vezes, isso permite que carimbos de versão sejam comparados para atualidade, mesmo quando você usa replicação ponto a ponto. Se dois pares atualizarem ao mesmo tempo, a combinação da mesma contagem e diferentes hashes de conteúdo facilita a detecção do conflito.

Além de ajudar a evitar conflitos de atualização, os carimbos de versão também são úteis para proporcionando consistência de sessão (p. 52).

6.2 Carimbos de versão em vários nós

O carimbo de versão básico funciona bem quando você tem uma única fonte autoritativa para dados, como um único servidor ou replicação mestre-escravo. Nesse caso, o carimbo de versão é controlado pelo mestre. Todos os escravos seguem os carimbos do mestre. Mas esse sistema precisa ser aprimorado em um modelo de distribuição ponto a ponto porque não há mais um único lugar para definir os carimbos de versão.

Se você estiver pedindo dados a dois nós, você corre o risco de que eles possam lhe dar respostas diferentes. Se isso acontecer, sua reação pode variar dependendo da causa dessa diferença. Pode ser que uma atualização tenha alcançado apenas um nó, mas não o outro, nesse caso você pode aceitar a mais recente (assumindo que você pode dizer qual é). Alternativamente, você pode ter encontrado uma atualização inconsistente, nesse caso você precisa decidir como lidar com isso. Nessa situação, um simples GUID ou etag não será suficiente, já que eles não lhe dizem o suficiente sobre os relacionamentos.

A forma mais simples de carimbo de versão é um contador. Cada vez que um nó atualiza os dados, ele incrementa o contador e coloca o valor do contador no carimbo de versão. Se você tem réplicas escravas azuis e verdes de um único mestre, e o nó azul responde com um carimbo de versão de 4 e o nó verde com 6, você sabe que a resposta do verde é mais recente.

Em casos de múltiplos mestres, precisamos de algo mais sofisticado. Uma abordagem, usada por sistemas de controle de versão distribuídos, é garantir que todos os nós contenham um histórico de carimbos de versão. Dessa forma, você pode ver se a resposta do nó azul é um ancestral da resposta do verde. Isso exigiria que os clientes mantivessem os históricos de carimbos de versão ou que os nós do servidor mantivessem os históricos de carimbos de versão e os incluíssem quando solicitados por dados. Isso também detecta uma inconsistência, que veríamos se obtivermos dois carimbos de versão e nenhum deles tiver o outro em seus históricos. Embora os sistemas de controle de versão mantenham esses tipos de históricos, eles não são encontrados em bancos de dados NoSQL.

Uma abordagem simples, mas problemática, é usar timestamps. O principal problema aqui é que geralmente é difícil garantir que todos os nós tenham uma noção consistente de tempo, principalmente se as atualizações podem acontecer rapidamente. Se o relógio de um nó ficar fora de sincronia, isso pode causar todos os tipos de problemas. Além disso, você não pode detectar conflitos de gravação-gravação com timestamps, então isso só funcionaria bem para o caso de mestre único — e então um contador geralmente é melhor.

A abordagem mais comum usada por sistemas NoSQL peer-to-peer é uma forma especial de carimbo de versão que chamamos de carimbo vetorial. Em essência, um **carimbo vetorial** é um conjunto de contadores, um para cada nó. Um carimbo vetorial para três nós (azul, verde, preto) seria algo como [azul: 43, verde: 54, preto: 12].

Cada vez que um nó tem uma atualização interna, ele atualiza seu próprio contador, então uma atualização no nó verde mudaria o vetor para [azul: 43, verde: 55, preto: 12]. Sempre que dois nós se comunicam, eles sincronizam seus carimbos de vetor.

Há várias variações de como exatamente essa sincronização é feita. Estamos cunhando o termo “selo vetorial” como um termo geral neste livro; você também encontrará **relógios vetoriais** e **vetores de versão** — essas são formas específicas de selos vetoriais que diferem em como sincronizam.

Usando esse esquema, você pode dizer se um carimbo de versão é mais novo que outro porque o carimbo mais novo terá todos os seus contadores maiores ou iguais aos do carimbo mais antigo. Então [azul: 1, verde: 2, preto: 5] é mais novo que [azul: 1, verde: 1, preto: 5] já que um dos seus contadores é maior. Se ambos os carimbos tiverem um contador maior que o outro, por exemplo, [azul: 1, verde: 2, preto: 5] e [azul: 2, verde: 1, preto: 5], então você tem um conflito de escrita-escrita.

Pode haver valores ausentes no vetor, em cujo caso tratamos o valor ausente como 0. Então [azul: 6, preto: 2] seria tratado como [azul: 6, verde: 0, preto: 2]. Isso permite que você adicione facilmente novos nós sem invalidar os carimbos de vetor existentes.

Os carimbos vetoriais são uma ferramenta valiosa que identifica inconsistências, mas não as resolve. Qualquer resolução de conflito dependerá do domínio em que você está trabalhando. Isso faz parte do tradeoff consistência-latência. Ou você tem que conviver com o fato de que partições de rede podem tornar seu sistema indisponível, ou você tem que detectar e lidar com inconsistências.

6.3 Pontos-chave

- Os carimbos de versão ajudam a detectar conflictos de simultaneidade. Quando você lê dados e os atualiza, você pode verificar o carimbo de versão para garantir que ninguém atualizou os dados entre sua leitura e gravação.
- Os carimbos de versão podem ser implementados usando contadores, GUIDs, hashes de conteúdo, carimbos de data/hora ou uma combinação destes.
- Com sistemas distribuídos, um vetor de carimbos de versão permite detectar quando diferentes nós têm atualizações conflitantes.

Esta página foi deixada em branco intencionalmente

Capítulo 7

Mapa-Reduzir

O aumento de bancos de dados orientados a agregados se deve em grande parte ao crescimento de clusters. Executar em um cluster significa que você tem que fazer suas trocas no armazenamento de dados de forma diferente do que quando executado em uma única máquina. Os clusters não mudam apenas as regras para armazenamento de dados — eles também mudam as regras para computação. Se você armazena muitos dados em um cluster, processar esses dados de forma eficiente significa que você precisa pensar de forma diferente sobre como organizar seu processamento.

Com um banco de dados centralizado, geralmente há duas maneiras de executar a lógica de processamento nele: no próprio servidor de banco de dados ou em uma máquina cliente. Executá-lo em uma máquina cliente oferece mais flexibilidade na escolha de um ambiente de programação, o que geralmente torna os programas mais fáceis de criar ou estender. Isso tem o custo de ter que transportar muitos dados do servidor de banco de dados. Se você precisa atingir muitos dados, então faz sentido fazer o processamento no servidor, pagando o preço em conveniência de programação e aumentando a carga no servidor de banco de dados.

Quando você tem um cluster, há boas notícias imediatamente — você tem muitas máquinas para espalhar a computação. No entanto, você também ainda precisa tentar reduzir a quantidade de dados que precisa ser transferida pela rede, fazendo o máximo de processamento possível no mesmo nó que os dados de que ele precisa.

O padrão map-reduce (uma forma de Scatter-Gather [Hohpe e Woolf]) é uma maneira de organizar o processamento de forma a aproveitar várias máquinas em um cluster, mantendo o máximo de processamento e os dados necessários juntos na mesma máquina. Ele ganhou destaque pela primeira vez com a estrutura MapReduce do Google [Dean e Ghemawat]. Uma implementação de código aberto amplamente usada faz parte do projeto Hadoop, embora vários bancos de dados incluam suas próprias implementações. Como na maioria dos padrões, há diferenças em detalhes entre essas implementações, então nos concentraremos no conceito geral. O nome “map-reduce” revela sua inspiração nas operações de map e reduce em coleções em linguagens de programação funcional.

7.1 Map-Reduce básico

Para explicar a ideia básica, começaremos com um exemplo que já repetimos à exaustão — o de clientes e pedidos. Vamos supor que escolhemos pedidos como nosso agregado, com cada pedido tendo itens de linha. Cada item de linha tem um ID de produto, quantidade e o preço cobrado. Esse agregado faz muito sentido, pois geralmente as pessoas querem ver o pedido inteiro em um acesso. Temos muitos pedidos, então fragmentamos o conjunto de dados em muitas máquinas.

No entanto, as pessoas de análise de vendas querem ver um produto e sua receita total dos últimos sete dias. Este relatório não se encaixa na estrutura agregada que temos — o que é a desvantagem de usar agregados. Para obter o relatório de receita do produto, você terá que visitar cada máquina no cluster e examinar muitos registros em cada máquina.

Este é exatamente o tipo de situação que exige map-reduce. O primeiro estágio em um trabalho de map-reduce é o mapa. Um mapa é uma função cuja entrada é um único agregado e cuja saída é um monte de pares de chave-valor. Neste caso, a entrada seria um pedido. A saída seria pares de chave-valor correspondentes aos itens de linha. Cada um teria o ID do produto como a chave e um mapa incorporado com a quantidade e o preço como os valores (veja a Figura 7.1).

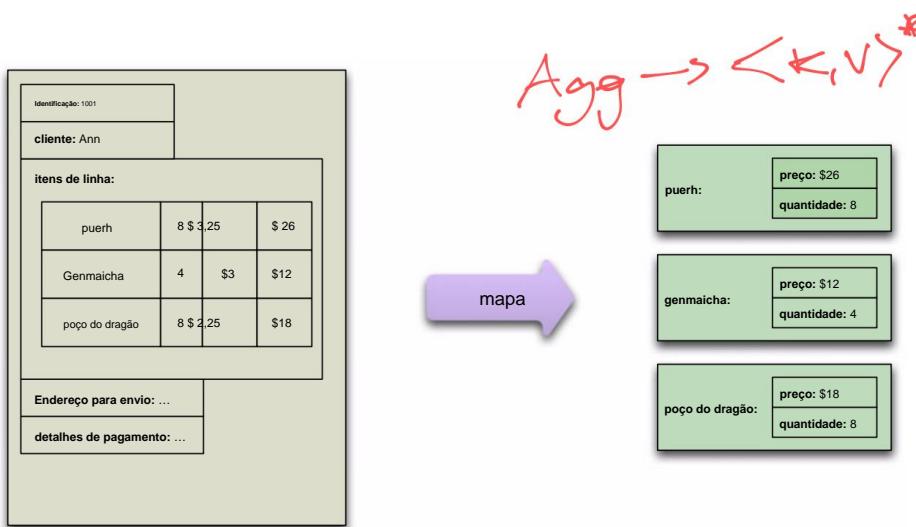


Figura 7.1 Uma função de mapa lê registros do banco de dados e emite pares chave-valor.

Cada aplicação da função map é independente de todas as outras. Isso permite que elas sejam seguramente paralelizáveis, de modo que uma estrutura map-reduce possa criar tarefas map eficientes em cada nó e alojar livremente cada ordem para uma tarefa map.

Isso produz uma grande quantidade de paralelismo e localidade de acesso a dados. Para este exemplo,

7.2 Particionamento e combinação

estamos apenas selecionando um valor do registro, mas não há razão para que não possamos executar alguma função arbitrariamente complexa como parte do mapa, desde que isso dependa apenas do valor de um agregado de dados.

Uma operação de mapa opera somente em um único registro; a função reduce pega múltiplas saídas de mapa com a mesma chave e combina seus valores. Então, uma função map pode render 1000 itens de linha de pedidos para “Refatoração de Banco de Dados”; a função reduce reduziria para um, com os totais para quantidade e receita. Enquanto a função map é limitada a trabalhar somente em dados de um único agregado, a função reduce pode usar todos os valores emitidos para uma única chave (veja Figura 7.2).

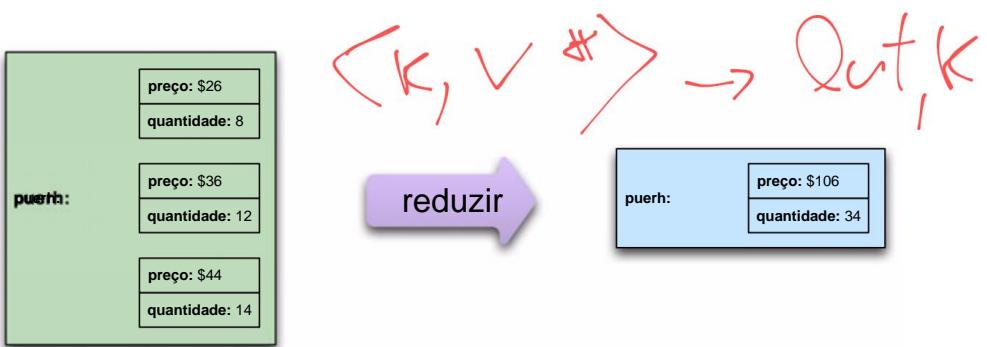


Figura 7.2 Uma função de redução pega vários pares de chave-valor com a mesma chave e os agrupa em um.

O framework map-reduce organiza as tarefas de mapeamento para serem executadas nos nós corretos para processar todos os documentos e para que os dados sejam movidos para a função reduce. Para facilitar a escrita da função reduce, o framework coleta todos os valores para um único par e chama a função reduce uma vez com a chave e a coleção de todos os valores para essa chave. Então, para executar um trabalho map-reduce, você só precisa escrever essas duas funções.

7.2 Particionamento e combinação

Na forma mais simples, pensamos em um trabalho map-reduce como tendo uma única função reduce. As saídas de todas as tarefas map em execução nos vários nós são concatenadas e enviadas para o reduce. Embora isso funcione, há coisas que podemos fazer para aumentar o paralelismo e reduzir a transferência de dados (veja a Figura 7.3).

A primeira coisa que podemos fazer é aumentar o paralelismo partionando a saída dos mapeadores. Cada função reduce opera nos resultados de uma única chave. Isso

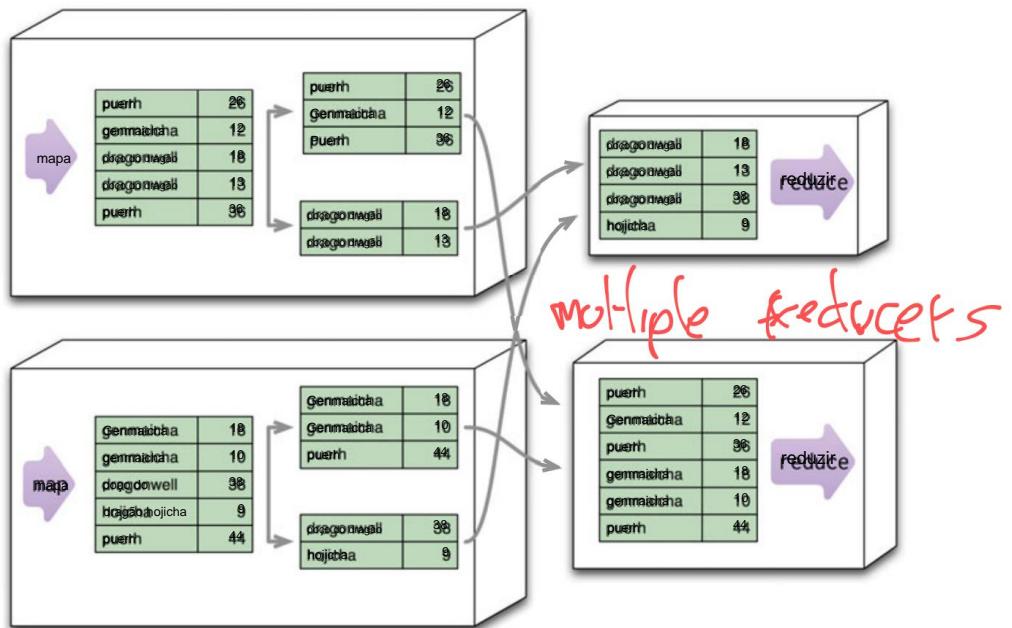


Figura 7.3 O particionamento permite que funções de redução sejam executadas em paralelo em teclas diferentes.

é uma limitação — significa que você não pode fazer nada na redução que opere entre chaves — mas também é um benefício, pois permite que você execute vários redutores em paralelo. Para aproveitar isso, os resultados do mapeador são divididos com base na chave de cada nó de processamento. Normalmente, várias chaves são agrupadas em partições. A estrutura então pega os dados de todos os nós para uma partição, combina-os em um único grupo para essa partição e os envia para um redutor. Vários redutores podem então operar nas partições em paralelo, com os resultados finais mesclados. (Esta etapa também é chamada de “embaralhamento” e as partições são às vezes chamadas de “buckets” ou “regiões”.)

O próximo problema com o qual podemos lidar é a quantidade de dados sendo movidos de nó para nó entre os estágios de mapa e redução. Muitos desses dados são repetitivos, consistindo em vários pares de chave-valor para a mesma chave. Uma função combinadora corta esses dados combinando todos os dados para a mesma chave em um único valor (veja a Figura 7.4). Uma função combinadora é, em essência, uma função redutora — de fato, em muitos casos, a mesma função pode ser usada para combinar como a redução final. A função reduce precisa de uma forma especial para que isso funcione: sua saída deve corresponder à sua entrada. Chamamos essa função de **combinable reducer**.

Nem todas as funções reduce são combináveis. Considere uma função que conta o número de clientes únicos para um produto específico. A função map para tal operação precisaria emitir o produto e o cliente. O reducer pode então combiná-los e contar quantas vezes cada cliente aparece para um

$\langle K, V \rangle \rightarrow \text{Combiner} \rightarrow \langle K, V \rangle$

7.2 Particionamento e combinação

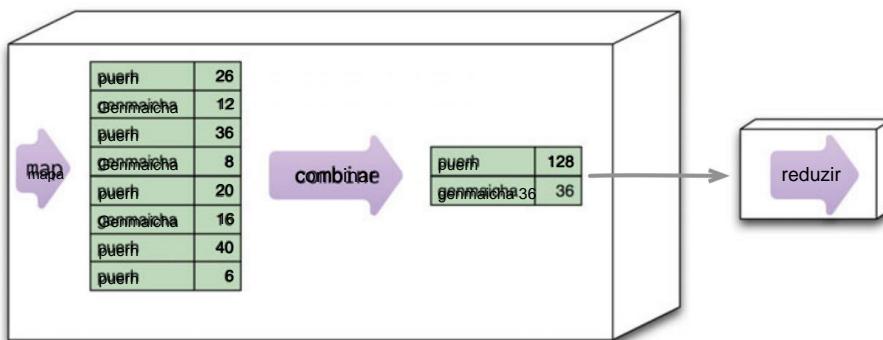


Figura 7.4 A combinação reduz os dados antes de enviá-los pela rede.

produto específico, emitindo o produto e a contagem (ver Figura 7.5). Mas isso a saída do redutor é diferente de sua entrada, portanto ele não pode ser usado como um combinador. Você ainda pode executar uma função de combinação aqui: uma que apenas elimina duplicatas pares produto-cliente, mas será diferente do redutor final.

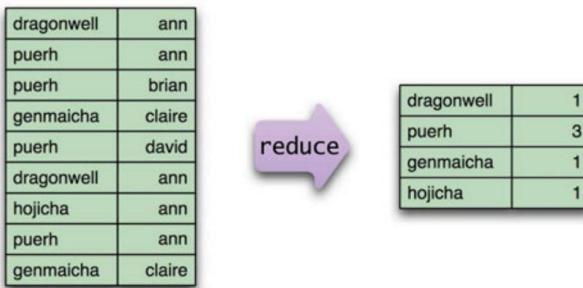


Figura 7.5 Esta função de redução, que conta quantos clientes únicos encomendam um chá em particular, não é combinável.

Combine: parallel + series

Quando você combina redutores, a estrutura map-reduce pode ser usada com segurança não só são executados em paralelo (para reduzir diferentes partições), mas também em série para reduzir a mesma partição em diferentes momentos e lugares. Além de permitir a combinação para ocorrer em um nó antes da transmissão de dados, você também pode começar a combinar antes os mappers terminaram. Isso fornece um pouco mais de flexibilidade extra ao processamento de redução de mapa. Algumas estruturas de redução de mapa exigem que todos os redutores sejam combinando redutores, o que maximiza essa flexibilidade. Se você precisar fazer um redutor não combinado com uma dessas estruturas, você precisará separar o processamento em etapas de mapeamento e redução em pipeline.

7.3 Compondo cálculos de Map-Reduce

A abordagem map-reduce é uma maneira de pensar sobre processamento concorrente que troca flexibilidade em como você estrutura sua computação por um modelo relativamente direto para paralelizar a computação em um cluster. Como é uma troca, há restrições sobre o que você pode fazer em seus cálculos.

Dentro de uma tarefa de mapa, você só pode operar em um único agregado. Dentro de uma tarefa de redução, você só pode operar em uma única chave. Isso significa que você tem que pensar diferente sobre estruturar seus programas para que eles funcionem bem dentro dessas restrições.

Uma limitação simples é que você tem que estruturar seus cálculos em torno de operações que se encaixem bem com a noção de uma operação de redução. Um bom exemplo disso é calcular médias. Vamos considerar o tipo de pedidos que temos observado até agora; suponha que queremos saber a quantidade média pedida de cada produto. Uma propriedade importante das médias é que elas não são componíveis — ou seja, se eu pegar dois grupos de pedidos, não posso combinar suas médias sozinhas. Em vez disso, preciso pegar o valor total e a contagem de pedidos de cada grupo, combiná-los e, então, calcular a média da soma e contagem combinadas (veja a Figura 7.6).

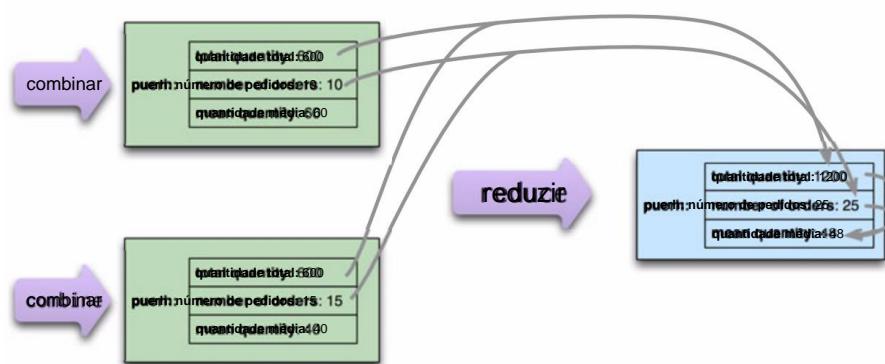


Figura 7.6 Ao calcular médias, a soma e a contagem podem ser combinadas no cálculo de redução, mas a média deve ser calculada a partir da soma e da contagem combinadas.

Essa noção de procurar cálculos que reduzem nitidamente também afeta como fazemos contagens. Para fazer uma contagem, a função de mapeamento emitirá campos de contagem com um valor de 1, que podem ser somados para obter uma contagem total (veja Figura 7.7).

7.3 Compondo cálculos de Map-Reduce

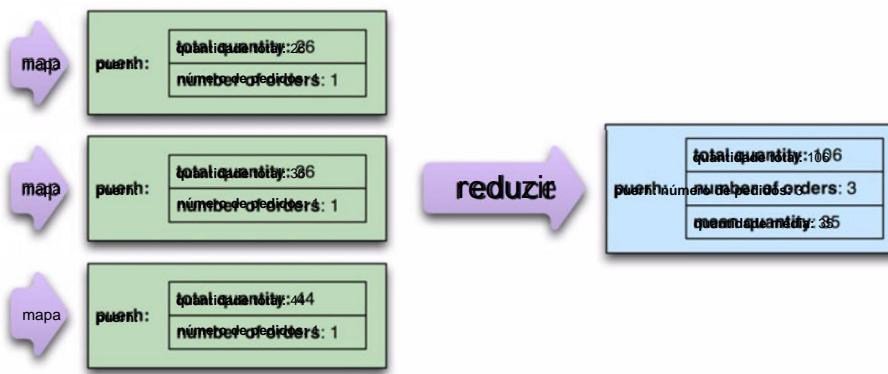


Figura 7.7 Ao fazer uma contagem, cada mapa emite 1, que pode ser somado para obter um total.

7.3.1 Um exemplo de Map-Reduce em dois estágios

À medida que os cálculos de redução de mapa se tornam mais complexos, é útil dividi-los em estágios usando uma abordagem de pipes e filtros, com a saída de um estágio servindo como entrada para o próximo, semelhante aos pipelines no UNIX.

Considere um exemplo em que queremos comparar as vendas de produtos para cada mês de 2011 com o ano anterior. Para fazer isso, dividiremos os cálculos em dois estágios. O primeiro estágio produzirá registros mostrando os números agregados para um único produto em um único mês do ano. O segundo estágio então usa esses como entradas e produz o resultado para um único produto comparando os resultados de um mês com o mesmo mês no ano anterior (veja a Figura 7.8).

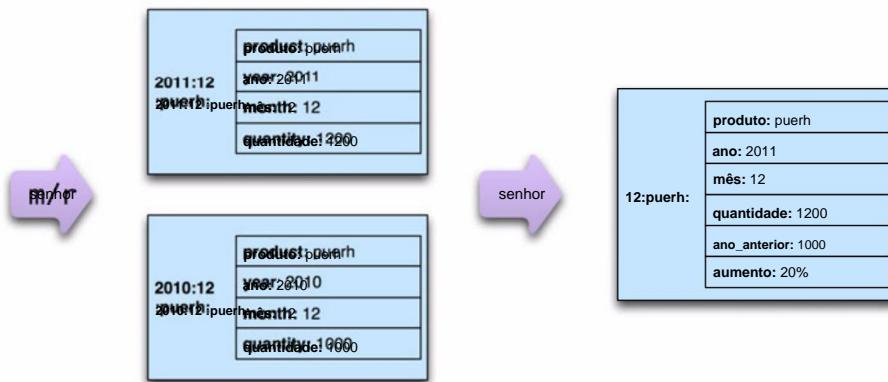


Figura 7.8 Um cálculo dividido em duas etapas de redução de mapa, que serão expandidas nas próximas três figuras

Um primeiro estágio (Figura 7.9) leia os registros de pedidos originais e produziria um série de pares de chave-valor para as vendas de cada produto por mês.

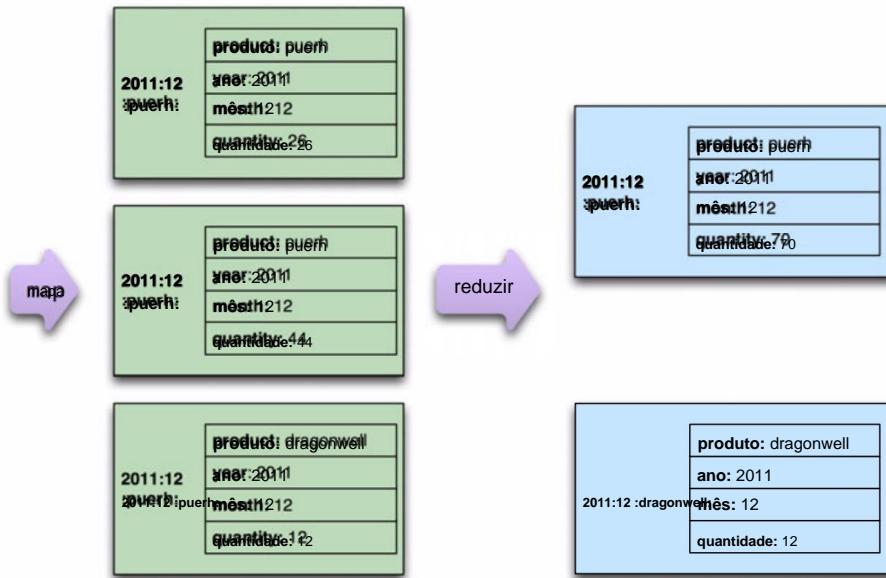


Figura 7.9 Criando registros para vendas mensais de um produto

Este estágio é similar aos exemplos de map-reduce que vimos até agora. O único recurso novo é usar uma chave composta para que possamos reduzir registros com base nos valores de vários campos.

Os mapeadores de segundo estágio (Figura 7.10) processam essa saída dependendo do ano. Um registro de 2011 preenche a quantidade do ano atual, enquanto um registro de 2010 preenche uma quantidade do ano anterior. Registros de anos anteriores (como 2009) não resultam em nenhuma saída de mapeamento sendo emitida.

A redução neste caso (Figura 7.11) é uma fusão de registros, onde a combinação dos valores por soma permite que duas saídas de anos diferentes sejam reduzidas a um único valor (com um cálculo baseado nos valores reduzidos incluídos para garantir).

Decompor este relatório em várias etapas de map-reduce facilita a escrita. Como muitos exemplos de transformação, uma vez que você tenha encontrado uma estrutura de transformação que facilite a composição de etapas, geralmente é mais fácil compor muitas etapas pequenas juntas do que tentar amontoar montes de lógica em uma única etapa.

7.3 Compondo cálculos de Map-Reduce

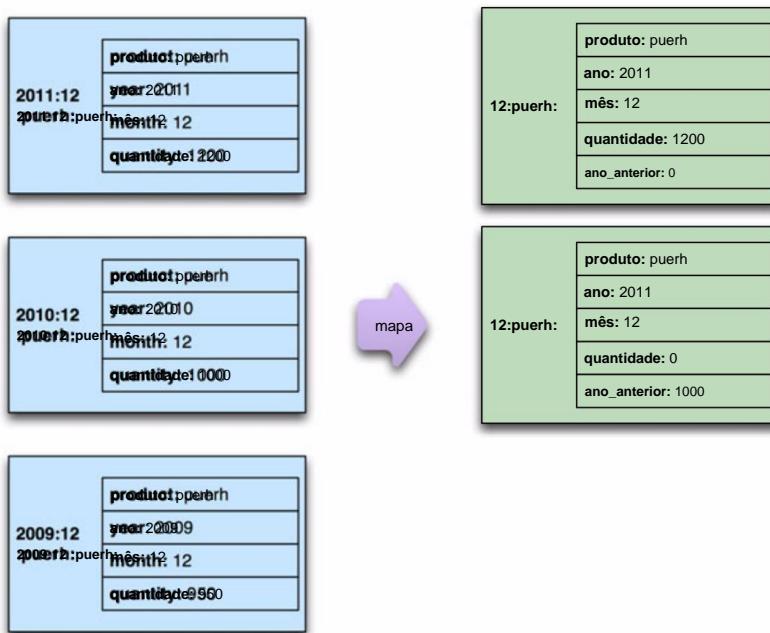


Figura 7.10 O mapeador de segundo estágio cria registros base para comparações ano a ano.

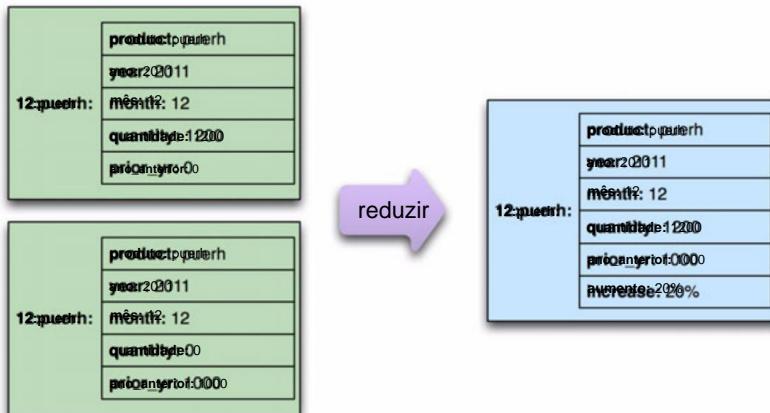


Figura 7.11 A etapa de redução é uma mesclagem de registros incompletos.

Outra vantagem é que a saída intermediária pode ser útil para saídas diferentes também, então você pode obter alguma reutilização. Essa reutilização é importante, pois economiza tempo tanto na programação quanto na execução. Os registros intermediários podem ser salvos no armazenamento de dados, formando uma visualização materializada (“Visualizações Materializadas”, p. 30). Os estágios iniciais das operações de redução de mapa são particularmente valiosos para salvar, pois geralmente representam a maior quantidade de acesso a dados, então construí-los uma vez como base para muitos usos posteriores economiza muito trabalho. Como em qualquer atividade de reutilização, no entanto, é importante construí-los a partir da experiência com consultas reais, pois a reutilização especulativa raramente cumpre sua promessa. Portanto, é importante observar as formas de várias consultas conforme elas são construídas e fatorar as partes comuns dos cálculos em visualizações materializadas.

Map-reduce é um padrão que pode ser implementado em qualquer linguagem de programação. No entanto, as restrições do estilo o tornam um bom ajuste para linguagens projetadas especificamente para computações de map-reduce. Apache Pig [Pig], um desdobramento do projeto Hadoop [Hadoop], é uma linguagem construída especificamente para facilitar a escrita de programas map-reduce. Certamente torna muito mais fácil trabalhar com Hadoop do que com as bibliotecas Java subjacentes. Em uma linha semelhante, se você quiser especificar programas map-reduce usando uma sintaxe semelhante a SQL, há hive [Hive], outro desdobramento do Hadoop.

O padrão map-reduce é importante de se conhecer, mesmo fora do contexto de bancos de dados NoSQL. O sistema map-reduce original do Google operava em arquivos armazenados em um sistema de arquivos distribuído — uma abordagem usada pelo projeto Hadoop de código aberto. Embora seja preciso pensar um pouco para se acostumar com as restrições de estruturar cálculos em etapas map-reduce, o resultado é um cálculo inherentemente adequado para execução em um cluster. Ao lidar com altos volumes de dados, você precisa adotar uma abordagem orientada a cluster. Bancos de dados orientados a agregados se encaixam bem com esse estilo de cálculo. Acreditamos que, nos próximos anos, muito mais organizações processarão os volumes de dados que exigem uma solução orientada a cluster — e o padrão map-reduce verá mais e

mais uso.

7.3.2 Mapa-Redução Incremental

Os exemplos que discutimos até agora são computações completas de map-reduce, onde começamos com entradas brutas e criamos uma saída final. Muitas computações de map-reduce demoram um pouco para serem executadas, mesmo com hardware em cluster, e novos dados continuam chegando, o que significa que precisamos executar novamente a computação para manter a saída atualizada. Começar do zero a cada vez pode levar muito tempo, então geralmente é útil estruturar uma computação de map-reduce para permitir atualizações incrementais, de modo que apenas a computação mínima precise ser feita.

Os estágios de mapa de um map-reduce são fáceis de manipular incrementalmente — somente se os dados de entrada mudarem é que o mapper precisará ser executado novamente. Como os mapas são isolados uns dos outros, as atualizações incrementais são diretas.

O caso mais complexo é a etapa de redução, pois ela reúne as saídas de muitos mapas e qualquer alteração nas saídas do mapa pode disparar uma nova redução. Essa recomputação pode ser diminuída dependendo de quão paralela é a etapa de redução. Se estivermos particionando os dados para redução, qualquer partição que não for alterada não precisa ser reduzida novamente. Da mesma forma, se houver uma etapa combinadora, ela não precisa ser executada novamente se seus dados de origem não tiverem sido alterados.

Se nosso redutor for combinável, há mais algumas oportunidades para evitar a computação. Se as alterações forem aditivas — isto é, se estivermos apenas adicionando novos registros, mas não alterando ou excluindo nenhum registro antigo — então podemos simplesmente executar a redução com o resultado existente e as novas adições. Se houver alterações destrutivas, ou seja, atualizações e exclusões, então podemos evitar alguma recomputação dividindo a operação de redução em etapas e recalculando apenas as etapas cujas entradas foram alteradas — essencialmente, usando uma *Rede de Dependência* [Fowler DSL] para organizar a computação.

A estrutura de mapeamento e redução controla muito disso, então você precisa entender como uma estrutura específica oferece suporte à operação incremental.

7.4 Leituras Adicionais

Se você for usar cálculos de map-reduce, seu primeiro porto de escala será a documentação do banco de dados específico que você está usando. Cada banco de dados tem sua própria abordagem, vocabulário e peculiaridades, e é com isso que você precisa estar familiarizado. Além disso, há uma necessidade de capturar informações mais gerais sobre como estruturar trabalhos de map-reduce para maximizar a manutenibilidade e o desempenho. Ainda não temos nenhum livro específico para indicar, mas suspeitamos que uma boa fonte, embora facilmente esquecida, são os livros sobre Hadoop. Embora o Hadoop não seja um banco de dados, é uma ferramenta que usa muito o map-reduce, então escrever uma tarefa de map-reduce eficaz com o Hadoop provavelmente será útil em outros contextos (sujeito às mudanças em detalhes entre o Hadoop e quaisquer sistemas que você esteja usando).

7.5 Pontos-chave

- Map-reduce é um padrão que permite que os cálculos sejam paralelizados em um conjunto.
- A tarefa de mapa lê dados de um agregado e os reduz a pares de chave-valor relevantes. Os mapas leem apenas um único registro por vez e, portanto, podem ser paralelizados e executados no nó que armazena o registro.

Map Agg $\rightarrow \langle k, v \rangle$

Reduce [$\langle k, v \rangle^*$] \rightarrow Output

- Tarefas Reduce pegam muitos valores para uma única saída de chave de tarefas map e os resumem em uma única saída. Cada reducer opera no resultado de uma única chave, então ele pode ser paralelizado por chave.
- Reducers que têm o mesmo formato para entrada e saída podem ser combinados em pipelines. Isso melhora o paralelismo e reduz a quantidade de dados a serem transferidos.
- As operações de redução de mapa podem ser compostas em pipelines onde a saída de uma redução é a entrada para o mapa de outra operação.
- Se o resultado de um cálculo de redução de mapa for amplamente utilizado, ele poderá ser armazenado como uma visualização materializada.
- As visualizações materializadas podem ser atualizadas por meio de operações incrementais de redução de mapa que apenas calculam as alterações na visualização em vez de recalcular tudo do zero.

Parte II

Implement

Esta página foi deixada em branco intencionalmente

Capítulo 8

Bancos de dados de chave-valor

Um armazenamento de chave-valor é uma tabela hash simples, usada principalmente quando todo o acesso ao banco de dados é via chave primária. Pense em uma tabela em um RDBMS tradicional com duas colunas, como ID e NAME, a coluna ID sendo a chave e a coluna NAME armazenando o valor. Em um RDBMS, a coluna NAME é restrita a armazenar dados do tipo String. O aplicativo pode fornecer um ID e VALUE e persistir o par; se o ID já existir, o valor atual será substituído, caso contrário, uma nova entrada será criada. Vamos ver como a terminologia se compara no Oracle e no Riak.

Oráculo	Riak
instância de banco de dados	Aglomerado de Riak
mesa	balde
linha	valor-chave
barulhento	chave

8.1 O que é um armazenamento de chave-valor

Os armazenamentos de chave-valor são os armazenamentos de dados NoSQL mais simples de usar de uma perspectiva de API. O cliente pode obter o valor para a chave, colocar um valor para uma chave ou excluir uma chave do armazenamento de dados. O valor é um blob que o armazenamento de dados apenas armazena, sem se importar ou saber o que está dentro; é responsabilidade do aplicativo entender o que foi armazenado. Como os armazenamentos de chave-valor sempre usam acesso de chave primária, eles geralmente têm ótimo desempenho e podem ser facilmente dimensionados.

Alguns dos bancos de dados de chave-valor populares são Riak [Riak], Redis (frequentemente chamado de servidor de estrutura de dados) [Redis], Memcached DB e seus sabores [Memcached], Berkeley DB [Berkeley DB], HamsterDB (especialmente adequado para uso incorporado)

[HamsterDB], Amazon DynamoDB [Dynamo da Amazon] (não de código aberto) e Project Voldemort [Project Voldemort] (uma implementação de código aberto do Amazon DynamoDB).

Em alguns armazenamentos de chave-valor, como o Redis, o agregado que está sendo armazenado não precisa ser um objeto de domínio — pode ser qualquer estrutura de dados. O Redis suporta o armazenamento de listas, conjuntos, hashes e pode fazer operações de intervalo, diff, união e interseção. Esses recursos permitem que o Redis seja usado de mais maneiras diferentes do que um armazenamento de chave-valor padrão.

Há muito mais bancos de dados de chave-valor e muitos novos estão sendo trabalhados neste momento. Para manter as discussões neste livro mais fáceis, vamos nos concentrar principalmente no Riak. O Riak nos permite armazenar chaves em buckets, que são apenas uma maneira de segmentar as chaves — pense nos buckets como namespaces simples para as chaves.

Se quiséssemos armazenar dados de sessão do usuário, informações do carrinho de compras e preferências do usuário no Riak, poderíamos simplesmente armazenar todos eles no mesmo bucket com uma única chave e um único valor para todos esses objetos. Nesse cenário, teríamos um único objeto que armazena todos os dados e é colocado em um único bucket (Figura 8.1).

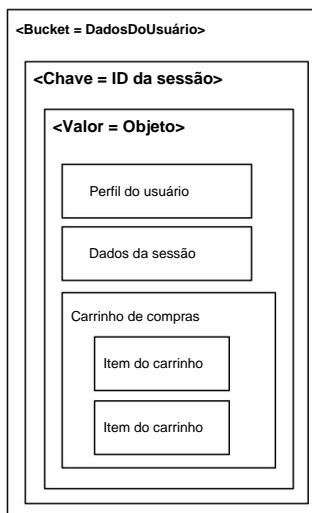


Figura 8.1 Armazenando todos os dados em um único bucket

A desvantagem de armazenar todos os diferentes objetos (agregados) no único bucket seria que um bucket armazenaria diferentes tipos de agregados, aumentando a chance de conflitos de chaves. Uma abordagem alternativa seria anexar o nome do objeto à chave, como 288790b8a421_userProfile, para que possamos obter objetos individuais conforme necessário (Figura 8.2).

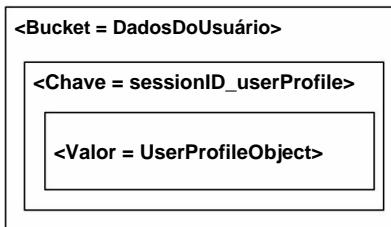


Figura 8.2 Altere o design da chave para segmentar os dados em um único bucket.

Também poderíamos criar buckets que armazenam dados específicos. No Riak, eles são conhecidos como **buckets de domínio**, permitindo que a serialização e a desserialização sejam manipuladas pelo driver cliente.

```

Bucket bucket = client.fetchBucket(bucketName).execute();
DomainBucket<UserProfile> profileBucket =
DomainBucket.builder(bucket, UserProfile.class).build();
  
```

Usar buckets de domínio ou buckets diferentes para objetos diferentes (como UserProfile e ShoppingCart) segmenta os dados em buckets diferentes, permitindo que você leia somente o objeto necessário sem precisar alterar o design da chave.

Os armazenamentos de chave-valor, como o Redis, também oferecem suporte ao armazenamento de estruturas de dados aleatórias, que podem ser conjuntos, hashes, strings e assim por diante. Esse recurso pode ser usado para armazenar listas de coisas, como estados ou addressTypes, ou uma matriz de visitas do usuário.

8.2 Recursos do Key-Value Store

Ao usar qualquer armazenamento de dados NoSQL, há uma necessidade inevitável de entender como os recursos se comparam aos armazenamentos de dados RDBMS padrão aos quais estamos tão acostumados. O principal motivo é entender quais recursos estão faltando e como a arquitetura do aplicativo precisa mudar para usar melhor os recursos de um armazenamento de dados de chave-valor. Alguns dos recursos que discutiremos para todos os armazenamentos de dados NoSQL são consistência, transações, recursos de consulta, estrutura dos dados e dimensionamento.

8.2.1 Consistência

A consistência é aplicável somente para operações em uma única chave, já que essas operações são get, put ou delete em uma única chave. Escritas otimistas podem ser executadas, mas são muito caras de implementar, porque uma mudança no valor não pode ser determinada pelo armazenamento de dados.

Em implementações de armazenamento de chave-valor distribuídas como Riak, o modelo de consistência *eventualmente consistente* (p. 50) é implementado. Como o valor pode já ter

replicado para outros nós, o Riak tem duas maneiras de resolver conflitos de atualização: ou a gravação mais recente vence e as gravações mais antigas perdem, ou ambos (todos) os valores são retornados, permitindo que o cliente resolva o conflito.

No Riak, essas opções podem ser configuradas durante a criação do bucket. Os buckets são apenas uma maneira de criar namespaces para que as colisões de chaves possam ser reduzidas — por exemplo, todas as chaves do cliente podem residir no bucket do cliente. Ao criar um bucket, valores padrão para consistência podem ser fornecidos, por exemplo, que uma gravação é considerada boa somente quando os dados são consistentes em todos os nós onde os dados são armazenados.

Balde balde = conexão

```
.createBucket(bucketName) .withRetrier(tentativas(3)) .allowSiblings(irmãosPermitidos) .nVal(númeroDeRéplicas)
```

Se precisarmos que os dados em cada nó sejam consistentes, podemos aumentar o `numberOfNodesToRespondToWrite` definido por `w` para ser o mesmo que `nVal`. Claro que fazer isso diminuirá o desempenho de gravação do cluster. Para melhorar os conflitos de gravação ou leitura, podemos alterar o sinalizador `allowSiblings` durante a criação do bucket: Se estiver definido como `false`, deixamos a última gravação vencer e não criamos irmãos.

8.2.2 Transações

Diferentes produtos do tipo de armazenamento de valor-chave têm diferentes especificações de transações. Em geral, não há garantias sobre as gravações. Muitos armazenamentos de dados implementam transações de maneiras diferentes. Riak usa o conceito de quorum (“Quorums,” p. 57) implementado usando o valor `W` — quorum de gravação — durante a chamada da API de gravação.

Suponha que temos um cluster Riak com um fator de replicação de 5 e fornecemos o valor `W` de 3. Ao escrever, a escrita é relatada como bem-sucedida somente quando é escrita e relatada como um sucesso em pelo menos três dos nós. Isso permite que o Riak tenha tolerância à escrita; em nosso exemplo, com `N` igual a 5 e com um valor `W` de 3, o cluster pode tolerar $N - W = 2$ nós inativos para operações de escrita, embora ainda tenhamos perdido alguns dados nesses nós para leitura.

8.2.3 Recursos de consulta

Todos os armazenamentos de chave-valor podem consultar pela chave — e é basicamente isso. Se você tiver requisitos para consultar usando algum atributo da coluna de valor, não será possível usar o banco de dados: seu aplicativo precisa ler o valor para descobrir se o atributo atende às condições.

8.2 Recursos do Key-Value Store

Consultar por chave também tem um efeito colateral interessante. E se não soubermos a chave, especialmente durante consultas ad-hoc durante a depuração? A maioria dos armazenamentos de dados não fornecerá uma lista de todas as chaves primárias; mesmo se fornecessem, recuperar listas de chaves e então consultar o valor seria muito trabalhoso. Alguns bancos de dados de chave-valor contornam isso fornecendo a capacidade de pesquisar dentro do valor, como [o Riak Search](#), que permite consultar os dados da mesma forma que você os consultaria usando índices Lucene.

Ao usar armazenamentos de chave-valor, muito pensamento deve ser dado ao design da chave. A chave pode ser gerada usando algum algoritmo? A chave pode ser fornecida pelo usuário (ID do usuário, e-mail, etc.)? Ou derivada de timestamps ou outros dados que podem ser derivados fora do banco de dados?

Essas características de consulta tornam os armazenamentos de chave-valor prováveis candidatos para armazenar dados de sessão (com o ID da sessão como chave), dados de carrinho de compras, perfis de usuário e assim por diante. A propriedade expiry_secs pode ser usada para expirar chaves após um certo intervalo de tempo, especialmente para objetos de sessão/carrinho de compras.

```
Balde balde = getBucket(bucketName); IRiakObject  
riakObject = bucket.store(chave, valor).execute();
```

Ao escrever no bucket Riak usando a API store , o objeto é armazenado para a chave fornecida. Da mesma forma, podemos obter o valor armazenado para a chave usando a API fetch .

```
Bucket bucket = getBucket(bucketName); IRiakObject  
riakObject = bucket.fetch(key).execute(); byte[] bytes = riakObject.getValue(); String  
value = new String(bytes);
```

O Riak fornece uma interface baseada em HTTP, para que todas as operações possam ser realizadas a partir do navegador da web ou na linha de comando usando curl. Vamos salvar esses dados no Riak:

```
{ "lastVisit":1324669989288, "user":
```

```
 { "customerId":"91cfdf5bcb7c", "name":"buyer",  
  "countryCode":"US",  
  "tzOffset":0 }
```

```
}
```

Use o comando curl para POSTAR os dados, armazenando-os na sessão bucket com a chave a7e618d9db25 (temos que fornecer esta chave):

```
curl -v -X POST -d '
{ "lastVisit":1324669989288, "user":
  {"customerId":"91cfdf5bcb7c", "name":"buyer",
   "countryCode":"US",
   "tzOffset":0} }'
```

```
-H "Tipo de conteúdo: application/json" http://
localhost:8098/buckets/session/keys/a7e618d9db25
```

Os dados para a chave a7e618d9db25 podem ser obtidos usando o comando curl :

```
curl -i http://localhost:8098/buckets/session/keys/a7e618d9db25
```

8.2.4 Estrutura de Dados

Bancos de dados de chave-valor não se importam com o que é armazenado na parte de valor do par chave-valor. O valor pode ser um blob, texto, JSON, XML e assim por diante. No Riak, podemos usar o Content-Type na solicitação POST para especificar o tipo de dados.

8.2.5 Escala

Muitos armazenamentos de chave-valor escalam usando sharding ("Sharding," p. 38). Com o sharding, o valor da chave determina em qual nó a chave é armazenada. Vamos supor que estamos sharding pelo primeiro caractere da chave; se a chave for f4b19d79587d, que começa com um f, ela será enviada para um nó diferente da chave ad9c7a396542.

Esse tipo de configuração de fragmentação pode aumentar o desempenho à medida que mais nós são adicionados ao cluster.

O sharding também introduz alguns problemas. Se o nó usado para armazenar f cair, os dados armazenados naquele nó se tornam indisponíveis, e novos dados não podem ser gravados com chaves que começam com f.

Armazenamentos de dados como o Riak permitem que você controle os aspectos do Teorema CAP ("O Teorema CAP", p. 53): N (número de nós para armazenar as réplicas de chave-valor), R (número de nós que precisam ter os dados sendo buscados antes que a leitura seja considerada bem-sucedida) e W (o número de nós nos quais a gravação precisa ser feita antes que seja considerada bem-sucedida).

Vamos supor que temos um cluster Riak de 5 nós. Definir N como 3 significa que todos os dados são replicados para pelo menos três nós, definir R como 2 significa que quaisquer dois nós devem responder a uma solicitação GET para que ela seja considerada bem-sucedida, e definir W como 2 garante que a solicitação PUT seja gravada em dois nós antes que a gravação seja considerada bem-sucedida.

Essas configurações nos permitem ajustar falhas de nós para operações de leitura ou gravação. Com base em nossa necessidade, podemos alterar esses valores para melhor disponibilidade de leitura ou disponibilidade de gravação. Em geral, escolha um valor W para corresponder às suas necessidades de consistência; esses valores podem ser definidos como padrões durante a criação do bucket.

8.3 Casos de uso adequados

Vamos discutir alguns dos problemas em que os armazenamentos de chave-valor são uma boa opção.

8.3.1 Armazenando informações da sessão

Geralmente, cada sessão web é única e recebe um valor de sessionid exclusivo .

Os aplicativos que armazenam o sessionid no disco ou em um RDBMS se beneficiarão muito da mudança para um armazenamento de chave-valor, já que tudo sobre a sessão pode ser armazenado por uma única solicitação PUT ou recuperado usando GET. Essa operação de solicitação única a torna muito rápida, já que tudo sobre a sessão é armazenado em um único objeto. Soluções como Memcached são usadas por muitos aplicativos da web, e Riak pode ser usado quando a disponibilidade é importante.

8.3.2 Perfis de usuário, preferências

Quase todo usuário tem um userid, nome de usuário ou algum outro atributo exclusivo, bem como preferências como idioma, cor, fuso horário, a quais produtos o usuário tem acesso e assim por diante. Tudo isso pode ser colocado em um objeto, então obter as preferências de um usuário leva uma única operação GET . Da mesma forma, perfis de produtos podem ser armazenados.

8.3.3 Dados do carrinho de compras

Sites de e-commerce têm carrinhos de compras vinculados ao usuário. Como queremos que os carrinhos de compras estejam disponíveis o tempo todo, em todos os navegadores, máquinas e sessões, todas as informações de compras podem ser colocadas no valor onde a chave é o userid. Um cluster Riak seria mais adequado para esses tipos de aplicativos.

8.4 Quando não usar

Existem espaços problemáticos onde os armazenamentos de chave-valor não são a melhor solução.

8.4.1 Relacionamentos entre Dados

Se você precisa ter relacionamentos entre diferentes conjuntos de dados ou correlacionar os dados entre diferentes conjuntos de chaves, os armazenamentos de chave-valor não são a melhor solução a ser usada, embora alguns armazenamentos de chave-valor fornecam recursos de link-walking.

8.4.2 Transações multioperacionais

Se você estiver salvando várias chaves e houver uma falha ao salvar qualquer uma delas, e você quiser reverter ou reverter o restante das operações, os armazenamentos de chave-valor não são a melhor solução a ser usada.

8.4.3 Consulta por Dados

Se você precisar pesquisar as chaves com base em algo encontrado na parte de valor dos pares chave-valor, os armazenamentos chave-valor não funcionarão bem para você.

Não há como inspecionar o valor no lado do banco de dados, com exceção de alguns produtos como o Riak Search ou mecanismos de indexação como o Lucene [Lucene] ou o Solr [Solr].

8.4.4 Operações por Conjuntos

Como as operações são limitadas a uma chave por vez, não há como operar em várias chaves ao mesmo tempo. Se você precisa operar em várias chaves, você tem que lidar com isso do lado do cliente.

Capítulo 9

Bancos de dados de documentos

Documentos são o conceito principal em bancos de dados de documentos. O banco de dados armazena e recupera documentos, que podem ser XML, JSON, BSON e assim por diante. Estes documentos são estruturas de dados em árvore hierárquicas e autodescritivas que podem consistir de mapas, coleções e valores escalares. Os documentos armazenados são semelhantes a uns aos outros, mas não precisam ser exatamente os mesmos. Bancos de dados de documentos armazene documentos na parte de valor do armazenamento de chave-valor; pense em documentos bancos de dados como armazenamentos de chave-valor onde o valor é examinável. Vamos ver como a terminologia é comparada no Oracle e no MongoDB.

Oráculo	MongoDB
instância de banco de dados	Instância do MongoDB
esquema	banco de dados
mesa	coleção
linha	documento
barulhento	_eu ia
juntar	Referência DB

O `_id` é um campo especial que é encontrado em todos os documentos no Mongo, assim como ROWID no Oracle. No MongoDB, `_id` pode ser atribuído pelo usuário, desde que seja exclusivo.

9.1 O que é um banco de dados de documentos?

```
{
  "firstname": "Martin", "curtidas":
    [ "Ciclismo",
      "Fotografia" ], "lastcity":
    "Boston", "lastVisited":
}

}
```

O documento acima pode ser considerado uma linha em um RDBMS tradicional. Vamos veja outro documento:

```
{
  "nome": "Pramod", "cidades
visitadas": [ "Chicago", "Londres", "Pune", "Bangalore" ], "endereços": [ { "estado": "AK", "cidade":
"DILLINGHAM", "tipo":
"R" }, { "estado": "MH",
"cidade": "PUNE", "tipo":
"R" }
],
"última cidade": "Chicago"
}
```

Olhando para os documentos, podemos ver que eles são semelhantes, mas têm diferenças em nomes de atributos. Isso é permitido em bancos de dados de documentos. O esquema dos dados pode diferir entre os documentos, mas esses documentos ainda podem pertencer à mesma coleção — diferentemente de um RDBMS, onde cada linha em uma tabela tem que seguir o mesmo esquema. Representamos uma lista de cidades visitadas como uma matriz, ou uma lista de endereços como uma lista de documentos incorporados dentro do documento principal.

Incorporar documentos filho como subobjetos dentro de documentos proporciona fácil acesso e melhor desempenho.

Se você olhar os documentos, verá que alguns dos atributos são semelhantes, como `firstname` ou `city`. Ao mesmo tempo, há atributos no segundo documento que não existem no primeiro documento, como `endereços`, enquanto `likes` está no primeiro documento, mas não no segundo.

Essa representação diferente de dados não é a mesma que no RDBMS, onde cada coluna tem que ser definida e, se não tiver dados, é marcada como vazia ou definida como nula. Em documentos, não há atributos vazios; se um determinado atributo não for encontrado, presumimos que ele não foi definido ou não é relevante para o documento. Os documentos permitem que novos atributos sejam criados sem a necessidade de defini-los ou alterar os documentos existentes.

Alguns dos bancos de dados de documentos populares que vimos são MongoDB [MongoDB], CouchDB [CouchDB], Terrastore [Terrastore], OrientDB [OrientDB], RavenDB [RavenDB] e, claro, o conhecido e frequentemente criticado Lotus Notes [Notes Storage Facility] que usa armazenamento de documentos.

9.2 Características

Embora existam muitos bancos de dados de documentos especializados, usaremos o MongoDB como um representante do conjunto de recursos. Tenha em mente que cada produto tem alguns recursos que podem não ser encontrados em outros bancos de dados de documentos.

Vamos dedicar algum tempo para entender como o MongoDB funciona. Cada instância do MongoDB tem vários *bancos de dados*, e cada banco de dados pode ter várias *coleções*.

Quando comparamos isso com RDBMS, uma instância RDBMS é a mesma que uma instância MongoDB, os esquemas em RDBMS são semelhantes aos bancos de dados MongoDB, e as tabelas RDBMS são coleções em MongoDB. Quando armazenamos um documento, temos que escolher a qual banco de dados e coleção esse documento pertence — por exemplo, `database.collection.insert(document)`, que geralmente é representado como `db.coll.insert(document)`.

9.2.1 Consistência

A consistência no banco de dados MongoDB é configurada usando os **conjuntos de réplicas** e escolhendo esperar que as gravações sejam replicadas para todos os escravos ou um número determinado de escravos. Cada gravação pode especificar o número de servidores para os quais a gravação deve ser propagada antes de retornar como bem-sucedida.

Um comando como `db.runCommand({ getlasterror : 1 , w : "majority" })` informa ao banco de dados quão forte é a consistência que você deseja. Por exemplo, se você tiver um servidor e especificar `w` como `mostority`, a gravação retornará imediatamente, pois há apenas um nó. Se você tiver três nós no conjunto de réplicas e especificar `w` como `mostority`, a gravação terá que ser concluída em no mínimo dois nós antes de ser relatada como um sucesso. Você pode aumentar o valor `w` para uma consistência mais forte, mas sofrerá no desempenho da gravação, pois agora as gravações precisam ser concluídas em mais nós. Os conjuntos de réplicas também permitem que você aumente o desempenho da leitura, permitindo a leitura de escravos ao definir `slaveOk`; esse parâmetro pode ser definido na conexão, no banco de dados, na coleção ou individualmente para cada operação.

```
Mongo mongo = novo Mongo("localhost:27017");
mongo.slaveOk();
```

Aqui estamos definindo `slaveOk` por operação, para que possamos decidir qual as operações podem trabalhar com dados do nó escravo.

```
Coleção DBCollection = getOrderCollection(); consulta
BasicDBObject = new BasicDBObject();
consulta.put("nome", "Martin"); cursor
DBCursor = coleção.find(consulta).slaveOk();
```

Semelhante a várias opções disponíveis para leitura, você pode alterar as configurações para obter uma consistência de gravação forte, se desejar. Por padrão, uma gravação é relatada como bem-sucedida quando o banco de dados a recebe; você pode alterar isso para esperar que as gravações sejam sincronizadas com o disco ou propagadas para dois ou mais escravos. Isso é conhecido como WriteConcern: você garante que certas gravações sejam gravadas no mestre e em alguns escravos definindo WriteConcern como REPLICAS_SAFE. Abaixo, é mostrado o código em que estamos definindo o WriteConcern para todas as gravações em uma coleção:

```
DBCollection compras = database.getCollection("compras");
compras.setWriteConcern(REPLICAS_SAFE);
```

WriteConcern também pode ser definido por operação, especificando-o no comando save:

```
WriteResult resultado = shopping.insert(pedido, REPLICAS_SAFE);
```

Há uma compensação que você precisa pensar cuidadosamente, com base nas necessidades do seu aplicativo e nos requisitos do seu negócio, para decidir quais configurações fazem sentido para o slaveOk durante a leitura ou qual nível de segurança você deseja durante a gravação com o WriteConcern.

9.2.2 Transações

Transações, no sentido tradicional de RDBMS, significam que você pode começar a modificar o banco de dados com comandos insert, update ou delete em diferentes tabelas e então decidir se quer manter as alterações ou não usando commit ou rollback. Essas construções geralmente não estão disponíveis em soluções NoSQL — uma gravação é bem-sucedida ou falha. Transações no nível de documento único são conhecidas como **transações atômicas**. Transações envolvendo mais de uma operação não são possíveis, embora existam produtos como o RavenDB que suportam transações em várias operações.

Por padrão, todas as gravações são relatadas como bem-sucedidas. Um controle mais fino sobre a gravação pode ser obtido usando o parâmetro WriteConcern . Garantimos que a ordem seja gravada em mais de um nó antes de ser relatada como bem-sucedida usando WriteConcern.REPLICAS_SAFE. Diferentes níveis de WriteConcern permitem que você escolha o nível de segurança durante as gravações; por exemplo, ao gravar entradas de log, você pode usar o nível mais baixo de segurança, WriteConcern.NONE.

```
final Mongo mongo = novo Mongo(mongoURI);
mongo.setWriteConcern(REPLICAS_SAFE);
DBCollection compras =
    mongo.getDB(orderDatabase) .getCollection(shoppingCollection);
```

```
tente
    { WriteResult resultado = shopping.insert(pedido, REPLICAS_SAFE);
//As gravações foram feitas no primário e em pelo menos um secundário }
catch (MongoException writeException) {
//As gravações não foram feitas em pelo menos dois nós, incluindo o primário
    dealWithWriteFailure(ordem, writeException);
}
}
```

9.2.3 Disponibilidade

O teorema CAP (“The CAP Theorem,” p. 53) determina que podemos ter apenas dois: Consistência, Disponibilidade e Tolerância de Partição. Os bancos de dados de documentos tentam melhorar a disponibilidade replicando dados usando a configuração mestre-escravo. Os mesmos dados estão disponíveis em vários nós e os clientes podem obter os dados mesmo quando o nó primário está inativo. Normalmente, o código do aplicativo não precisa determinar se o nó primário está disponível ou não. O MongoDB implementa a replicação, fornecendo alta disponibilidade usando **conjuntos de réplicas**.

Em um conjunto de réplicas, há dois ou mais nós participando de uma replicação mestre-escravo assíncrona. Os nós do conjunto de réplicas elegem o mestre, ou primário, entre si. Supondo que todos os nós tenham direitos de voto iguais, alguns nós podem ser favorecidos por estarem mais próximos dos outros servidores, por terem mais RAM, e assim por diante; os usuários podem afetar isso atribuindo uma prioridade — um número entre 0 e 1000 — a um nó.

Todas as solicitações vão para o nó mestre e os dados são replicados para os nós escravos. Se o nó mestre cair, os nós restantes no conjunto de réplicas votam entre si para eleger um novo mestre; todas as solicitações futuras são roteadas para o novo mestre, e os nós escravos começam a obter dados do novo mestre. Quando o nó que falhou volta a ficar online, ele se junta como um escravo e alcança o resto dos nós, puxando todos os dados de que precisa para ficar atualizado.

A Figura 9.1 é um exemplo de configuração de conjuntos de réplicas. Temos dois nós, **mongo A** e **mongo B**, executando o banco de dados MongoDB no data center primário, e **mongo C** no data center secundário. Se quisermos que os nós no data center primário sejam eleitos como nós primários, podemos atribuir a eles uma prioridade maior do que os outros nós. Mais nós podem ser adicionados aos conjuntos de réplicas sem precisar colocá-los offline.

O aplicativo grava ou lê do nó primário (mestre). Quando a conexão é estabelecida, o aplicativo precisa se conectar somente a um nó (primário ou não, não importa) no conjunto de réplicas, e o restante dos nós são descobertos automaticamente. Quando o nó primário cai, o driver fala com o novo primário eleito pelo conjunto de réplicas. O aplicativo não precisa gerenciar nenhuma das falhas de comunicação ou critérios de seleção de nós. Usar conjuntos de réplicas dá a você a capacidade de ter um armazenamento de dados de documentos altamente disponível.

Os conjuntos de réplicas são geralmente usados para redundância de dados, failover automatizado, dimensionamento de leitura, manutenção de servidor sem tempo de inatividade e recuperação de desastres. Semelhante

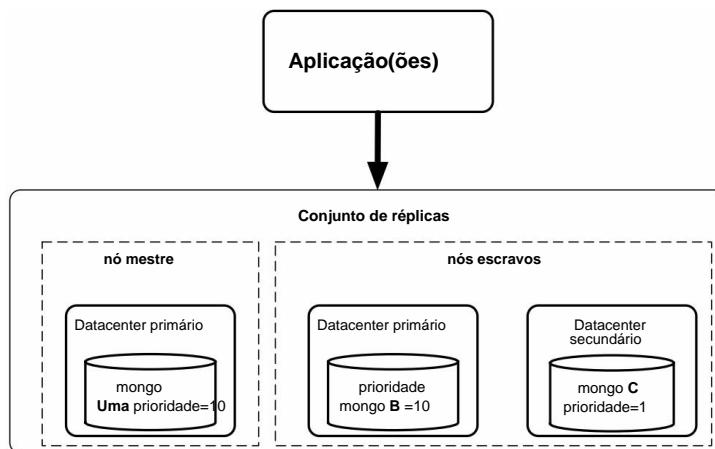


Figura 9.1 Configuração do conjunto de réplicas com maior prioridade atribuída aos nós no mesmo datacenter

configurações de disponibilidade podem ser obtidas com CouchDB, RavenDB, Terrastore e outros produtos.

9.2.4 Recursos de consulta

Os bancos de dados de documentos fornecem diferentes recursos de consulta. O CouchDB permite que você faça consultas por meio de visualizações — consultas complexas em documentos que podem ser materializadas (“Visualizações Materializadas”, p. 30) ou dinâmicas (pense nelas como visualizações RDBMS que são materializadas ou não). Com o CouchDB, se você precisar agregar o número de avaliações de um produto, bem como a classificação média, você pode adicionar uma visualização implementada por meio do map-reduce (“Map-Reduce Básico”, p. 68) para retornar a contagem de avaliações e a média de suas classificações.

Quando há muitas solicitações, você não quer calcular a contagem e a média para cada solicitação; em vez disso, você pode adicionar uma visualização materializada que pré-calcula os valores e armazena os resultados no banco de dados. Essas visualizações materializadas são atualizadas quando consultadas, se algum dado foi alterado desde a última atualização.

Um dos bons recursos dos bancos de dados de documentos, em comparação com os armazenamentos de chave-valor, é que podemos consultar os dados dentro do documento sem ter que recuperar o documento inteiro por sua chave e então introspectar o documento. Esse recurso aproxima esses bancos de dados do modelo de consulta RDBMS.

O MongoDB tem uma linguagem de consulta que é expressa via JSON e tem construções como \$query para a cláusula where , \$orderby para classificar os dados ou \$explain para mostrar o plano de execução da consulta. Há muitas outras construções como essas que podem ser combinadas para criar uma consulta MongoDB.

Vamos dar uma olhada em certas consultas que podemos fazer no MongoDB. Suponha que queremos retornar todos os documentos em uma coleção de pedidos (todas as linhas na tabela de pedidos). O SQL para isso seria:

```
SELEÇÃO * DO pedido
```

A consulta equivalente no shell Mongo seria:

```
db.pedido.encontrar()
```

Selecionar os pedidos para um único customerId de 883c2c5b4e5b seria:

```
SELEÇÃO * DO pedido ONDE customerId = "883c2c5b4e5b"
```

A consulta equivalente no Mongo para obter todos os pedidos para um único customerId de 883c2c5b4e5b:

```
db.order.find({"IDcliente": "883c2c5b4e5b"})
```

Da mesma forma, selecionar orderId e orderDate para um cliente no SQL seria:

```
SELEÇÃO orderId,orderDate DE order ONDE customerId = "883c2c5b4e5b"
```

e o equivalente em Mongo seria:

```
db.order.find({customerId: "883c2c5b4e5b"},{orderId:1,orderDate:1})
```

Da mesma forma, consultas para contar, somar e assim por diante estão todas disponíveis. Como os documentos são objetos agregados, é realmente fácil consultar documentos que precisam ser correspondidos usando os campos com objetos filhos. Digamos que queremos consultar todos os pedidos em que um dos itens pedidos tem um nome como Refactoring. O SQL para esse requisito seria:

```
SELEÇÃO * DE customerOrder, orderItem, produto ONDE customerOrder.orderId  
= orderItem.customerOrderId E orderItem.productId = product.productId E product.name  
COMO '%Refactoring%'
```

e a consulta Mongo equivalente seria:

```
db.orders.find({"items.product.name": /Refatoração/})
```

A consulta para MongoDB é mais simples porque os objetos são incorporados dentro de um único documento e você pode consultar com base nos documentos filhos incorporados.

9.2.5 Escala

A ideia de escalar é adicionar nós ou alterar o armazenamento de dados sem simplesmente migrar o banco de dados para uma caixa maior. Não estamos falando sobre tornar o aplicativo

alterações para lidar com mais carga; em vez disso, estamos interessados em quais recursos estão no banco de dados para que ele possa lidar com mais carga.

O dimensionamento para cargas de leitura pesada pode ser alcançado adicionando mais escravos de leitura, para que todas as leituras possam ser direcionadas aos escravos. Dado um aplicativo de leitura pesada, com nosso cluster de conjunto de réplicas de 3 nós, podemos adicionar mais capacidade de leitura ao cluster conforme a carga de leitura aumenta apenas adicionando mais nós escravos ao conjunto de réplicas para executar leituras com o sinalizador slaveOk (Figura 9.2). Este é o dimensionamento horizontal para leituras.

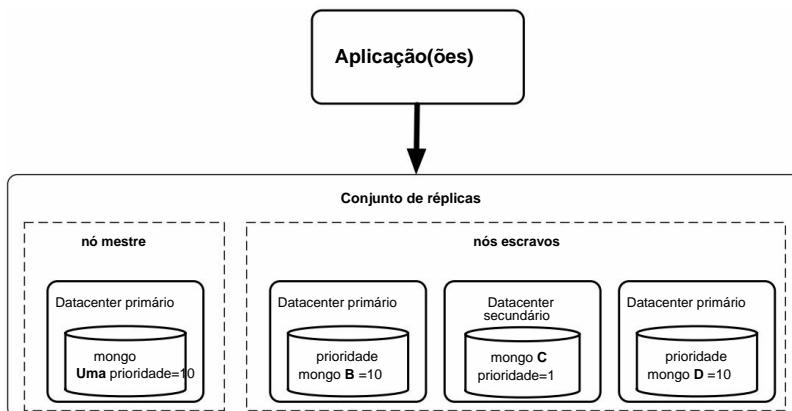


Figura 9.2 Adicionando um novo nó, mongo D, a um cluster de conjunto de réplicas existente

Depois que o novo nó, mongo D, for iniciado, ele precisará ser adicionado ao conjunto de réplicas.

```
rs.add("mongod:27017");
```

Quando um novo nó é adicionado, ele será sincronizado com os nós existentes, se juntará ao conjunto de réplicas como nó secundário e começará a atender solicitações de leitura. Uma vantagem dessa configuração é que não precisamos reiniciar nenhum outro nó e também não há tempo de inatividade para o aplicativo.

Quando queremos escalar para gravação, podemos começar a fragmentar ("Sharding," p. 38) os dados. A fragmentação é semelhante às partições em RDBMS, onde dividimos os dados por valor em uma determinada coluna, como estado ou ano. Com RDBMS, as partições geralmente estão no mesmo nó, então o aplicativo cliente não precisa consultar uma partição específica, mas pode continuar consultando a tabela base; o RDBMS cuida de encontrar a partição certa para a consulta e retorna os dados.

No sharding, os dados também são divididos por determinado campo, mas depois movidos para diferentes nós do Mongo. Os dados são movidos dinamicamente entre nós para garantir que os shards estejam sempre平衡ados. Podemos adicionar mais nós ao cluster e aumentar o número de nós graváveis, permitindo o dimensionamento horizontal para gravações.

```
db.runCommand( { shardcollection : "ecommerce.customer", chave : {nome : 1} } )
```

Dividir os dados no primeiro nome do cliente garante que os dados sejam balanceados entre os shards para desempenho de gravação ideal; além disso, cada shard pode ser um conjunto de réplicas, garantindo melhor desempenho de leitura dentro do shard (Figura 9.3). Quando adicionamos um novo shard a esse cluster shard existente, os dados agora serão平衡ados entre quatro shards em vez de três. Como toda essa movimentação de dados e refatoração de infraestrutura está acontecendo, o aplicativo não experimentará nenhum tempo de inatividade, embora o cluster possa não ter desempenho ideal quando grandes quantidades de dados estão sendo movidas para rebalancear os shards.

A chave de shard desempenha um papel importante. Você pode querer colocar seus shards de banco de dados MongoDB mais próximos de seus usuários, então o sharding baseado na localização do usuário pode ser uma boa ideia. Ao sharding por localização do cliente, todos os dados do usuário para a Costa Leste dos EUA estão nos shards que são servidos da Costa Leste, e todos os dados do usuário para a Costa Oeste estão nos shards que estão na Costa Oeste.

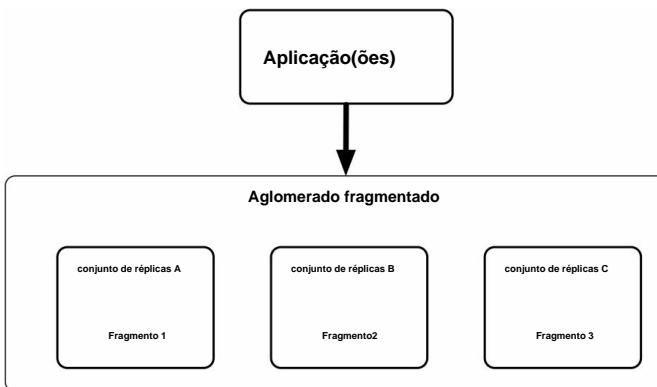


Figura 9.3 Configuração fragmentada do MongoDB onde cada fragmento é um conjunto de réplicas

9.3 Casos de uso adequados

9.3.1 Registro de eventos

Os aplicativos têm diferentes necessidades de registro de eventos; dentro da empresa, há muitos aplicativos diferentes que desejam registrar eventos. Os bancos de dados de documentos podem armazenar todos esses diferentes tipos de eventos e podem atuar como um armazenamento de dados central para armazenamento de eventos. Isso é especialmente verdadeiro quando o tipo de dados capturados pelos eventos continua mudando. Os eventos podem ser fragmentados pelo nome do aplicativo onde o evento se originou ou pelo tipo de evento, como order_processed ou customer_logged.

9.3.2 Sistemas de gerenciamento de conteúdo, plataformas de blogs

Como os bancos de dados de documentos não têm esquemas predefinidos e geralmente entendem documentos JSON, eles funcionam bem em sistemas de gerenciamento de conteúdo ou aplicativos para publicação de sites, gerenciamento de comentários de usuários, registros de usuários, perfis e documentos voltados para a web.

9.3.3 Web Analytics ou Análise em Tempo Real

Os bancos de dados de documentos podem armazenar dados para análises em tempo real; como partes do documento podem ser atualizadas, é muito fácil armazenar visualizações de página ou visitantes únicos, e novas métricas podem ser facilmente adicionadas sem alterações de esquema.

9.3.4 Aplicações de comércio eletrônico

Os aplicativos de comércio eletrônico geralmente precisam ter esquemas flexíveis para produtos e pedidos, bem como a capacidade de desenvolver seus modelos de dados sem refatoração de banco de dados dispendiosa ou migração de dados (“Alterações de esquema em um armazenamento de dados NoSQL”, p. 128).

9.4 Quando não usar

Existem áreas problemáticas em que os bancos de dados de documentos não são a melhor solução.

9.4.1 Transações complexas abrangendo diferentes operações

Se você precisa ter operações atômicas entre documentos, então bancos de dados de documentos podem não ser para você. No entanto, há alguns bancos de dados de documentos que suportam esses tipos de operações, como o RavenDB.

9.4.2 Consultas em Estrutura Agregada Variável

Esquema flexível significa que o banco de dados não impõe nenhuma restrição ao esquema. Os dados são salvos na forma de entidades de aplicativo. Se você precisar consultar essas entidades ad hoc, suas consultas mudarão (em termos de RDBMS, isso significaria que, à medida que você une critérios entre tabelas, as tabelas a serem unidas continuam mudando). Como os dados são salvos como um agregado, se o design do agregado estiver mudando constantemente, você precisa salvar os agregados no nível mais baixo de granularidade — basicamente, você precisa normalizar os dados. Nesse cenário, os bancos de dados de documentos podem não funcionar.

Capítulo 10

Lojas da família Column

Armazenamentos de família de colunas, como Cassandra [Cassandra], HBase [Hbase], Hypertable [Hypertable] e Amazon SimpleDB [Amazon SimpleDB], permitem que você armazene dados com chaves mapeadas para valores e os valores agrupados em várias colunas famílias, sendo cada família de colunas um mapa de dados.

RDBMS	Cassandra
instância de banco de dados	conjunto
banco de dados	espaço de chaves
mesa	família de colunas
linha	linha
coluna (o mesmo para todas as linhas)	coluna (pode ser diferente por linha)

10.1 O que é um armazenamento de dados de família de colunas?

Existem muitos bancos de dados de famílias de colunas. Neste capítulo, falaremos sobre Cassandra, mas também faz referência a outros bancos de dados de famílias de colunas para discutir recursos que podem ser de interesse em cenários específicos.

Os bancos de dados de famílias de colunas armazenam dados em famílias de colunas como linhas que têm muitas colunas associadas a uma chave de linha (Figura 10.1). Famílias de colunas são grupos de dados relacionados que são frequentemente acessados juntos. Para um Cliente, frequentemente acessaríamos suas informações de Perfil ao mesmo tempo, mas não seus Pedidos.

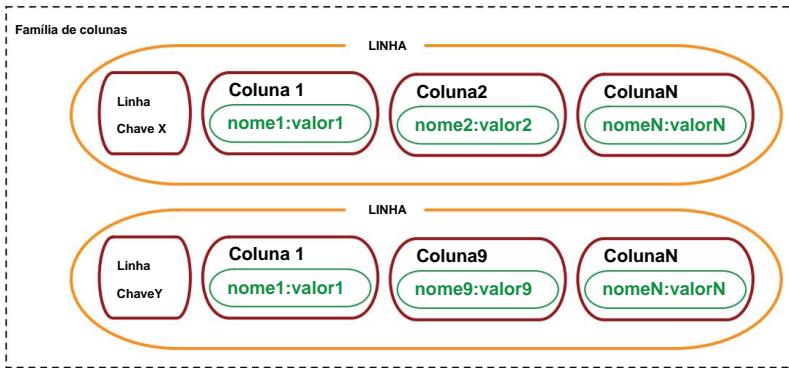


Figura 10.1 Modelo de dados de Cassandra com famílias de colunas

Cassandra é um dos bancos de dados de família de colunas mais populares; existem outros, como HBase, Hypertable e Amazon DynamoDB [Amazon DynamoDB].

Cassandra pode ser descrito como rápido e facilmente escalável com operações de gravação espalhadas pelo cluster. O cluster não tem um nó mestre, então qualquer leitura e gravação podem ser manipuladas por qualquer nó no cluster.

10.2 Características

Vamos começar observando como os dados são estruturados no Cassandra. A unidade básica de armazenamento no Cassandra é uma coluna. Uma coluna do Cassandra consiste em um par nome-valor em que o nome também se comporta como a chave. Cada um desses pares chave-valor é uma única coluna e é sempre armazenado com um valor de registro de data e hora. O registro de data e hora é usado para expirar dados, resolver conflitos de gravação, lidar com dados obsoletos e fazer outras coisas. Quando os dados da coluna não são mais usados, o espaço pode ser recuperado posteriormente durante uma fase de compactação.

```
{
  nome: "firstName",
  valor: "Martin",
  carimbo de data/hora: 12345667890
}
```

A coluna tem uma chave de firstName e o valor de Martin e tem um timestamp anexado a ela. Uma linha é uma coleção de colunas anexadas ou vinculadas a uma chave; uma coleção de linhas semelhantes forma uma família de colunas. Quando as colunas em uma família de colunas são colunas simples, a família de colunas é conhecida como **família de colunas padrão**.

```
//família de colunas { //  
  
linha  
  "pramod-sadalage" : {  
    primeiroNome: "Pramod",  
    últimoNome: "Sadalage",  
    últimaVisita: "2012/12/12"  
  
  } //linha  
  "martin-fowler" : { primeiroNome:  
    "Martin", últimoNome: "Fowler",  
    localização: "Boston"  
  
} }
```

Cada família de colunas pode ser comparada a um contêiner de linhas em uma tabela RDBMS onde a chave identifica a linha e a linha consiste em várias colunas.

A diferença é que várias linhas não precisam ter as mesmas colunas, e colunas podem ser adicionadas a qualquer linha a qualquer momento sem precisar adicioná-las a outras linhas. Temos a linha pramod-sadalage e a linha martin-fowler com colunas diferentes; ambas as linhas são parte da família de colunas.

Quando uma coluna consiste em um mapa de colunas, então temos uma **supercoluna**. Uma super coluna consiste em um nome e um valor que é um mapa de colunas. Pense em uma super coluna como um contêiner de colunas.

```
{ nome: "livro:978-0767905923", valor: { autor:  
  "Mitch  
  Albon", título: "Terças-feiras com  
  Morrie", isbn: "978-0767905923"  
  
} }
```

Quando usamos supercolunas para criar uma família de colunas, obtemos uma **superfamília de colunas**.

```
//família de supercolunas { //

    nome
    da linha: "faturamento:martin-fowler",
    valor:
        { endereço:
            { nome: "endereço:padrão", valor:

                { nomecompleto: "Martin Fowler",
                    rua:"100 N. Main Street", CEP:
                    "20145" } },

        faturamento:
            { nome: "faturamento:padrão",
                valor:
                    { cartãodecrédito: "8888-8888-8888-8888",
                        datadeexp: "12/2016" } }

    } //

    nome da linha: "faturamento:pramod-
    sadalage",
    valor:
        { endereço: { nome:

            "endereço:padrão", valor: { nomecompleto:
                "Pramod Sadalage", rua:"100 E. State
                Parkway", CEP:

                "54130" } },
            faturamento: { nome:

                "faturamento:padrão", valor: { cartãodecrédito:
                    "9999-8888-7777-4444",

                    datadevalidade: "01/2016" } }

        }
    }
```

Famílias de supercolunas são boas para manter dados relacionados juntos, mas quando algumas das colunas não são necessárias na maior parte do tempo, elas ainda são buscadas e desserializadas pelo Cassandra, o que pode não ser o ideal.

Cassandra coloca as famílias de colunas padrão e super em **keyspaces**. Um keyspace é semelhante a um banco de dados em RDBMS onde todas as famílias de colunas relacionadas ao aplicativo são armazenadas. Keyspaces precisam ser criados para que as famílias de colunas possam ser atribuídas a eles:

```
criar keyspace ecommerce
```

10.2.1 Consistência

Quando uma gravação é recebida pelo Cassandra, os dados são primeiro registrados em um log de confirmação, depois gravados em uma estrutura na memória conhecida como **memtable**. Uma operação de gravação é considerada bem-sucedida quando é gravada no log de confirmação e na memtable. As gravações são agrupadas na memória e periodicamente gravadas em estruturas conhecidas como **SSTable**. As SSTables não são gravadas novamente após serem liberadas; se houver alterações nos dados, uma nova SSTable é gravada. As SSTables não utilizadas são recuperadas por **compactação**.

Vamos dar uma olhada na operação de leitura para ver como as configurações de consistência afetam. Se tivermos uma configuração de consistência de UM como padrão para todas as operações de leitura, então quando uma solicitação de leitura for feita, o Cassandra retornará os dados da primeira réplica, mesmo se os dados estiverem obsoletos. Se os dados estiverem obsoletos, as leituras subsequentes obterão os dados mais recentes (mais novos); esse processo é conhecido como **reparo de leitura**. O nível de consistência baixo é bom para usar quando você não se importa se obtém dados obsoletos e/ou se tem altos requisitos de desempenho de leitura.

Similarmente, se você estiver fazendo gravações, o Cassandra gravaria no log de commit de um nó e retornaria uma resposta ao cliente. A consistência do ONE é boa se você tem requisitos de desempenho de gravação muito altos e também não se importa se algumas gravações forem perdidas, o que pode acontecer se o nó cair antes que a gravação seja replicada para outros nós.

```
quorum = novo ConfigurableConsistencyLevel();
quorum.setDefaultReadConsistencyLevel(HConsistencyLevel.QUORUM);
quorum.setDefaultWriteConsistencyLevel(HConsistencyLevel.QUORUM);
```

Usar a configuração de consistência QUORUM para operações de leitura e gravação garante que a maioria dos nós responda à leitura e a coluna com o carimbo de data/hora mais recente seja retornada ao cliente, enquanto as réplicas que não têm os dados mais recentes são reparadas por meio das operações de reparo de leitura. Durante as operações de gravação, a configuração de consistência QUORUM significa que a gravação tem que se propagar para a maioria dos nós antes de ser considerada bem-sucedida e o cliente ser notificado.

Usar ALL como nível de consistência significa que todos os nós terão que responder a leituras ou gravações, o que tornará o cluster não tolerante a falhas — mesmo quando um nó está inativo, a gravação ou leitura é bloqueada e relatada como uma falha. Portanto, cabe aos projetistas do sistema ajustar os níveis de consistência conforme os requisitos do aplicativo mudam. Dentro do mesmo aplicativo, pode haver diferentes requisitos de consistência; eles também podem mudar com base em cada operação, por exemplo, mostrar comentários de revisão para um produto tem diferentes requisitos de consistência em comparação à leitura do status do último pedido feito pelo cliente.

Durante a criação do **keyspace**, podemos configurar quantas réplicas dos dados precisamos armazenar. Esse número determina o fator de replicação dos dados. Se você tiver um fator de replicação de 3, os dados serão copiados para três nós. Ao gravar e ler dados com o Cassandra, se você especificar os valores de consistência de 2, você

entenda que $R + W$ é maior que o fator de replicação ($2 + 2 > 3$), o que proporciona melhor consistência durante gravações e leituras.

Podemos executar o comando node repair para o keyspace e forçar o Cassandra a comparar cada chave pela qual ele é responsável com o restante das réplicas. Como essa operação é cara, também podemos apenas reparar uma família de colunas específica ou uma lista de famílias de colunas:

```
reparar comércio eletrônico
```

```
reparar ecommerce customerInfo
```

Enquanto um nó está inativo, os dados que deveriam ser armazenados por esse nó são passados para outros nós. Conforme o nó volta a ficar online, as alterações feitas nos dados são passadas de volta para o nó. Essa técnica é conhecida como **hinted handoff**. O hinted handoff permite uma restauração mais rápida de nós com falha.

10.2.2 Transações

O Cassandra não tem transações no sentido tradicional, onde poderíamos iniciar várias gravações e então decidir se queremos confirmar as alterações ou não.

No Cassandra, uma gravação é atômica no nível da linha, o que significa que inserir ou atualizar colunas para uma determinada chave de linha será tratado como uma única gravação e terá sucesso ou falhará. As gravações são primeiros gravadas em logs de confirmação e memtables, e são consideradas boas somente quando a gravação em log de confirmação e memtable foi bem-sucedida. Se um nó ficar inativo, o log de confirmação será usado para aplicar alterações ao nó, assim como o **log de refazer** no Oracle.

Você pode usar bibliotecas de transações externas, como ZooKeeper [ZooKeeper], para sincronizar suas gravações e leituras. Há também bibliotecas como Cages [Cages] que permitem que você envolva suas transações no ZooKeeper.

10.2.3 Disponibilidade

O Cassandra é altamente disponível por design, já que não há mestre no cluster e cada nó é um par no cluster. A disponibilidade de um cluster pode ser aumentada reduzindo o nível de consistência das solicitações. A disponibilidade é governada pela fórmula $(R + W) > N$ ("Quorums," p. 57) onde W é o número mínimo de nós onde a gravação deve ser gravada com sucesso, R é o número mínimo de nós que devem responder com sucesso a uma leitura e N é o número de nós que participam da replicação de dados. Você pode ajustar a disponibilidade alterando os valores de R e W para um valor fixo de N .

Em um cluster Cassandra de 10 nós com um fator de replicação para o keyspace definido como 3 ($N = 3$), se definirmos $R = 2$ e $W = 2$, então teremos $(2 + 2) > 3$. Neste cenário, quando um nó cai, a disponibilidade não é muito afetada, pois os dados podem ser recuperados dos outros dois nós. Se $W = 2$ e $R = 1$, quando dois nós estão inativos, o cluster não está disponível para gravação, mas ainda podemos ler. Da mesma forma, se

$R = 2$ e $W = 1$, podemos escrever, mas o cluster não está disponível para leitura. Com a equação $R + W > N$, você está tomando decisões conscientes sobre compensações de consistência.

Você deve configurar seus espaços de chaves e operações de leitura/gravação com base em suas necessidades — maior disponibilidade para gravação ou maior disponibilidade para leitura.

10.2.4 Recursos de consulta

Ao projetar o modelo de dados no Cassandra, é aconselhável tornar as colunas e famílias de colunas otimizadas para leitura dos dados, pois ele não tem uma linguagem de consulta rica; como os dados são inseridos nas famílias de colunas, os dados em cada linha são classificados por nomes de coluna. Se tivermos uma coluna que é recuperada com muito mais frequência do que outras colunas, é melhor em termos de desempenho usar esse valor para a chave de linha.

10.2.4.1 Consultas básicas

Consultas básicas que podem ser executadas usando um cliente Cassandra incluem GET, SET e DEL. Antes de começar a consultar dados, temos que emitir o comando `keyspace use ecommerce;`. Isso garante que todas as nossas consultas sejam executadas no keyspace em que colocamos nossos dados. Antes de começar a usar a família de colunas no keyspace, temos que definir a família de colunas.

CRIAR FAMÍLIA DE COLUNAS Cliente

```
COM comparador = UTF8Type
E classe_de_validação_de_chave=UTF8Type
E metadados_da_coluna =
[ {nome_da_coluna: cidade, classe_de_validação: Tipo_UTF8} {nome_da_coluna:
nome, classe_de_validação: Tipo_UTF8} {nome_da_coluna: web,
classe_de_validação: Tipo_UTF8} ];
```

Temos uma família de colunas chamada Cliente com colunas de nome, cidade e web, e estamos inserindo dados na família de colunas com um cliente Cassandra.

```
SET Cliente['mfowler']['cidade']='Boston'; SET Cliente['mfowler']
['nome']='Martin Fowler'; SET Cliente['mfowler']['web']='www.martinfowler.com';
```

Usando o cliente Java Hector [Hector], podemos inserir os mesmos dados na família de colunas.

```
ColumnFamilyTemplate<String, String> template =
    cassandra.getColumnFamilyTemplate();
ColumnFamilyUpdater<String, String> atualizador =
    template.createUpdater(chave); para (String
nome : valores.keySet()) {
    atualizador.setString(nome, valores.get(nome));
}
```

```
tente
    { template.update(updater); }
pegue (HectorException e)
    { handleException(e);
}
```

Podemos ler os dados de volta usando o comando GET . Há várias maneiras de obter os dados; podemos obter toda a família de colunas.

OBTER Cliente['mfowler'];

Podemos até obter apenas a coluna em que estamos interessados da família de colunas.

OBTER Cliente['mfowler']['web'];

Obter a coluna específica que precisamos é mais eficiente, pois apenas os dados com os quais nos importamos são retornados — o que economiza muita movimentação de dados, especialmente quando a família de colunas tem um grande número de colunas. Atualizar os dados é o mesmo que usar o comando SET para a coluna que precisa ser definida para o novo valor.

Usando o comando DEL , podemos excluir uma coluna ou toda a família de colunas.

Cliente DEL['mfowler']['cidade'];

Cliente DEL['mfowler'];

10.2.4.2 Consultas e Indexação Avançadas O Cassandra

permite que você indexe colunas que não sejam as chaves para a família de colunas. Podemos definir um índice na coluna city .

```
ATUALIZAR FAMÍLIA DE COLUNAS Cliente
COM comparador = UTF8Type
E metadados_da_coluna = [{nome_da_coluna: cidade,
    classe_de_validação: Tipo_UTF8,
    tipo_de Índice: CHAVES}];
```

Agora podemos consultar diretamente a coluna indexada.

OBTER cliente ONDE cidade = 'Boston';

Esses índices são implementados como índices *mapeados em bits* e apresentam bom desempenho para valores de coluna de baixa cardinalidade.

10.2.4.3 Linguagem de consulta Cassandra (CQL)

Cassandra tem uma linguagem de consulta que suporta comandos do tipo SQL, conhecida como Cassandra Query Language (CQL). Podemos usar os comandos CQL para criar uma família de colunas.

```
CRIAR COLUMNFAMILY Cliente (
    CHAVE varchar CHAVE PRIMÁRIA,
    nome varchar,
    cidade varchar, web
    varchar);
```

Inserimos os mesmos dados usando CQL.

```
INSERIR EM Cliente (CHAVE,nome,cidade,web)
    VALORES ('mfowler', 'Martin
              Fowler', 'Boston',
              'www.martinfowler.com');
```

Podemos ler dados usando o comando SELECT . Aqui lemos todas as colunas:

```
SELEÇÃO * DO Cliente
```

Ou podemos simplesmente SELECIONAR as colunas que precisamos.

```
SELEÇÃO nome, web DE Cliente
```

As colunas de indexação são criadas usando o comando CREATE INDEX e podem ser usadas para consultar os dados.

```
SELEÇÃO nome, web DE Cliente ONDE cidade='Boston'
```

CQL tem muito mais recursos para consultar dados, mas não tem todos os recursos que SQL tem. CQL não permite junções ou subconsultas, e suas cláusulas where são tipicamente simples.

10.2.5 Escala

Escalar um cluster Cassandra existente é uma questão de adicionar mais nós. Como nenhum nó é um mestre, quando adicionamos nós ao cluster, estamos melhorando a capacidade do cluster de suportar mais gravações e leituras. Esse tipo de escala horizontal permite que você tenha o máximo de tempo de atividade, pois o cluster continua atendendo às solicitações dos clientes enquanto novos nós são adicionados ao cluster.

10.3 Casos de uso adequados

Vamos discutir alguns dos problemas em que bancos de dados de famílias de colunas são uma boa opção.

10.3.1 Registro de eventos

Os bancos de dados de família de colunas com sua capacidade de armazenar qualquer estrutura de dados são uma ótima opção para armazenar informações de eventos, como estado do aplicativo ou erros

encontrados pelo aplicativo. Dentro da empresa, todos os aplicativos podem gravar seus eventos no Cassandra com suas próprias colunas e a rowkey do formulário appname:timestamp. Como podemos escalar gravações, o Cassandra funcionaria idealmente para um sistema de registro de eventos (Figura 10.2).



Figura 10.2 Registro de eventos com Cassandra

10.3.2 Sistemas de gerenciamento de conteúdo, plataformas de blogs

Usando famílias de colunas, você pode armazenar entradas de blog com tags, categorias, links e trackbacks em colunas diferentes. Comentários podem ser armazenados na mesma linha ou movidos para um keyspace diferente; similarmente, usuários de blog e os blogs reais podem ser colocados em famílias de colunas diferentes.

10.3.3 Contadores

Frequentemente, em aplicativos da web, você precisa contar e categorizar visitantes de uma página para calcular análises. Você pode usar o CounterColumnType durante a criação de uma família de colunas.

```
CRIAR FAMÍLIA DE COLUNA visit_counter
COM default_validation_class=CounterColumnType
E key_validation_class=UTF8Type E comparador=UTF8Type;
```

Depois que uma família de colunas é criada, você pode ter colunas arbitrárias para cada página visitados dentro do aplicativo web para cada usuário.

```
INCR visit_counter['mfowler'][home] POR 1;
INCR visit_counter['mfowler'][produtos] POR 1;
INCR visit_counter['mfowler'][contactus] POR 1;
```

Incrementando contadores usando CQL:

```
ATUALIZAR visit_counter DEFINIR home = home + 1 ONDE KEY='mfowler'
```

10.3.4 Uso expirado

Você pode fornecer acesso de demonstração aos usuários ou pode querer mostrar banners de anúncios em um site por um tempo específico. Você pode fazer isso usando **colunas que expiram**: o Cassandra permite que você tenha colunas que, após um tempo determinado, são excluídas automaticamente. Este tempo é conhecido como TTL (Time To Live) e é definido em segundos. A coluna

é excluído após o TTL ter decorrido; quando a coluna não existe, o acesso pode ser revogado ou o banner pode ser removido.

```
DEFINIR Cliente['mfowler']['demo_access'] = 'permitido' COM ttl=2592000;
```

10.4 Quando não usar

Há problemas para os quais bancos de dados de família de colunas não são as melhores soluções, como sistemas que exigem transações ACID para gravações e leituras. Se você precisa que o banco de dados agregue os dados usando consultas (como SUM ou AVG), você tem que fazer isso no lado do cliente usando dados recuperados pelo cliente de todas as linhas.

Cassandra não é ótimo para protótipos iniciais ou picos tecnológicos iniciais: durante os estágios iniciais, não temos certeza de como os padrões de consulta podem mudar e, conforme os padrões de consulta mudam, temos que mudar o design da família de colunas. Isso causa atrito para a equipe de inovação de produtos e desacelera a produtividade do desenvolvedor.

Os RDBMS impõem um alto custo na alteração de esquema, que é compensado por um baixo custo de alteração de consulta; no Cassandra, o custo pode ser maior para alteração de consulta em comparação à alteração de esquema.

Esta página foi deixada em branco intencionalmente

Capítulo 11

Bancos de dados de gráficos

Bancos de dados de grafos permitem que você armazene entidades e relacionamentos entre essas entidades. Entidades também são conhecidas como nós, que têm propriedades. Pense em um nó como uma instância de um objeto no aplicativo. Relações são conhecidas como arestas que podem ter propriedades. Arestas têm significância direcional; nós são organizados por relacionamentos que permitem que você encontre padrões interessantes entre os nós.

A organização do gráfico permite que os dados sejam armazenados uma vez e depois interpretados de diferentes maneiras com base em relacionamentos.

11.1 O que é um banco de dados de gráficos?

No gráfico de exemplo na Figura 11.1, vemos um monte de nós relacionados entre si. Nós são entidades que têm propriedades, como nome. O nó de Martin é, na verdade, um **nó** que tem a **propriedade** de nome definida como Martin.

Também vemos que as arestas têm tipos, como likes, author e assim por diante. Essas propriedades nos permitem organizar os nós; por exemplo, os nós Martin e Pramod têm uma **aresta** conectando-os com um tipo de relacionamento de friend. As arestas podem ter várias propriedades. Podemos atribuir uma propriedade since no tipo de relacionamento friend entre Martin e Pramod. Os tipos de relacionamento têm significância direcional; o tipo de relacionamento friend é bidirecional, mas likes não é. Quando Dawn gosta de NoSQL Distilled, isso não significa automaticamente que NoSQL Distilled gosta de Dawn.

Uma vez que tenhamos um gráfico desses nós e arestas criado, podemos consultar o gráfico de muitas maneiras, como "obter todos os nós empregados pela Big Co que gostam de NoSQL Distilled". Uma consulta no gráfico também é conhecida como **percorrer** o gráfico. Uma vantagem dos bancos de dados de gráficos é que podemos alterar os requisitos de travessia sem ter que alterar os nós ou arestas. Se quisermos "obter todos os nós que gostam de NoSQL Distilled", podemos fazer isso sem ter que alterar os dados existentes ou o modelo do banco de dados, porque podemos percorrer o gráfico da maneira que quisermos.

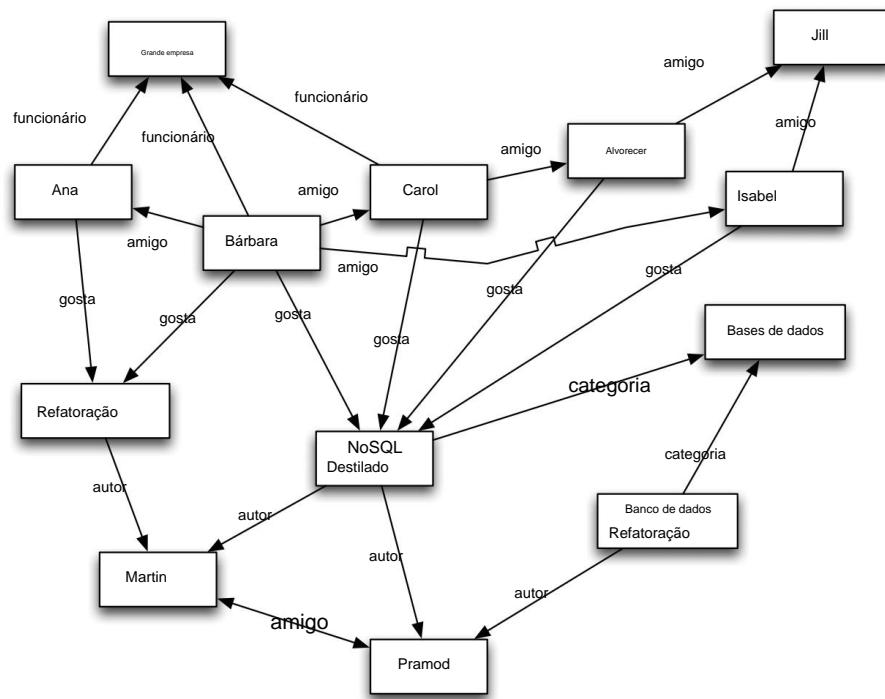


Figura 11.1 Um exemplo de estrutura de gráfico

Geralmente, quando armazenamos uma estrutura semelhante a um gráfico em RDBMS, é para um único tipo de relacionamento ("quem é meu gerente" é um exemplo comum). Adicionar outro relacionamento à mistura geralmente significa muitas mudanças de esquema e movimentação de dados, o que não é o caso quando usamos bancos de dados de gráficos. Da mesma forma, em bancos de dados relacionais, modelamos o gráfico de antemão com base no Traversal que queremos; se o Traversal mudar, os dados terão que mudar.

Em bancos de dados de grafos, percorrer as junções ou relacionamentos é muito rápido. O relacionamento entre nós não é calculado no momento da consulta, mas é realmente persistido como um relacionamento. Percorrer relacionamentos persistentes é mais rápido do que calculá-los para cada consulta.

Os nós podem ter diferentes tipos de relacionamentos entre eles, permitindo que você represente relacionamentos entre as entidades de domínio e tenha relacionamentos secundários para coisas como categoria, caminho, árvores de tempo, quad-trees para indexação espacial ou listas vinculadas para acesso classificado. Como não há limite para o número e o tipo de relacionamentos que um nó pode ter, todos eles podem ser representados no mesmo banco de dados de gráfico.

11.2 Características

Há muitos bancos de dados de grafos disponíveis, como Neo4J [Neo4J], Infinite Graph [Infinite Graph], OrientDB [OrientDB] ou FlockDB [FlockDB] (que é um caso especial: um banco de dados de grafos que suporta apenas relacionamentos de profundidade única ou listas de adjacência, onde você não pode atravessar mais de um nível de profundidade para relacionamentos). Tomaremos o Neo4J como um representante das soluções de banco de dados de grafos para discutir como elas funcionam e como podem ser usadas para resolver problemas de aplicação.

No Neo4J, criar um gráfico é tão simples quanto criar dois nós e então criar um relacionamento. Vamos criar dois nós, Martin e Pramod:

```
Nó martin = graphDb.createNode();
martin.setProperty("nome", "Martin");
```

```
Nó pramod = graphDb.createNode();
pramod.setProperty("nome", "Pramod");
```

Atribuímos à propriedade name dos dois nós os valores de Martin e Pramod. Uma vez que temos mais de um nó, podemos criar um relacionamento:

```
martin.createRelationshipTo(pramod, AMIGO);
pramod.createRelationshipTo(martin, AMIGO);
```

Temos que criar relacionamento entre os nós em ambas as direções, pois a direção do relacionamento importa: por exemplo, um nó de produto pode ser apreciado pelo usuário, mas o produto não pode ser apreciado pelo usuário. Essa direcionalidade ajuda a projetar um modelo de domínio rico (Figura 11.2). Os nós sabem sobre relacionamentos de ENTRADA e SAÍDA que são percorríveis em ambas as direções.

Relacionamentos são cidadãos de primeira classe em bancos de dados de grafos; a maior parte do valor de bancos de dados de grafos é derivada dos relacionamentos. Relacionamentos não têm apenas um tipo, um nó inicial e um nó final, mas podem ter propriedades próprias.

Usando essas propriedades nos relacionamentos, podemos adicionar inteligência ao relacionamento — por exemplo, desde quando eles se tornaram amigos, qual é a distância entre os nós ou quais aspectos são compartilhados entre os nós. Essas propriedades nos relacionamentos podem ser usadas para consultar o gráfico.

Como a maior parte do poder dos bancos de dados de grafos vem dos relacionamentos e suas propriedades, muito pensamento e trabalho de design são necessários para modelar os relacionamentos no domínio com o qual estamos tentando trabalhar. Adicionar novos tipos de relacionamento é fácil; alterar nós existentes e seus relacionamentos é semelhante à migração de dados ("Migrations in Graph Databases," p. 131), porque essas alterações terão que ser feitas em cada nó e cada relacionamento nos dados existentes.

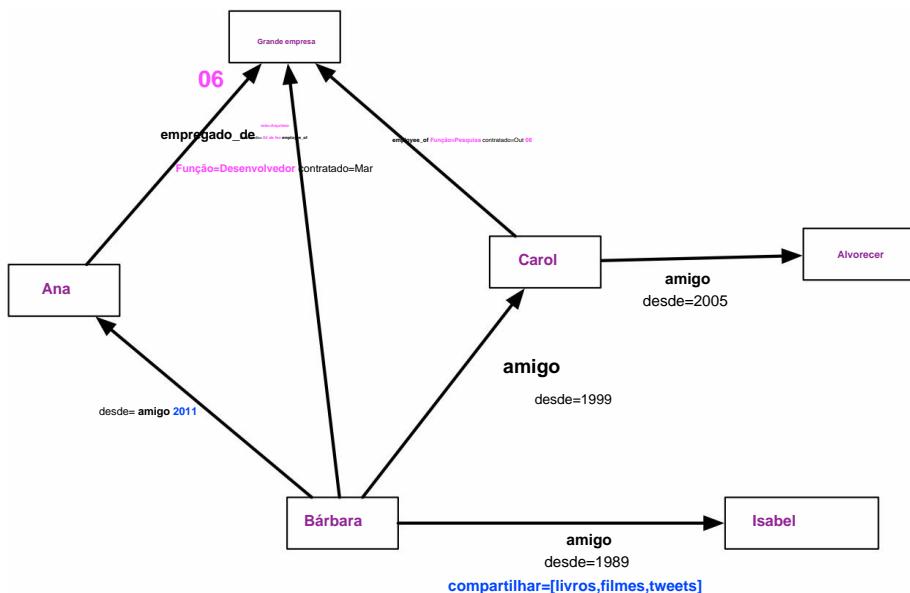


Figura 11.2 Relacionamentos com propriedades

11.2.1 Consistência

Como os bancos de dados de gráficos estão operando em nós conectados, a maioria das soluções de bancos de dados de gráficos geralmente não suportam a distribuição dos nós em servidores diferentes. Existem algumas soluções, no entanto, que suportam a distribuição de nós em um cluster de servidores, como o Infinite Graph. Dentro de um único servidor, os dados são sempre consistentes, especialmente no Neo4J, que é totalmente compatível com ACID. Ao executar o Neo4J em um cluster, uma gravação no mestre é eventualmente sincronizada com os escravos, enquanto os escravos estão sempre disponíveis para leitura. As gravações nos escravos são permitidas e são imediatamente sincronizadas com o mestre; outros escravos não serão sincronizados imediatamente, no entanto — eles terão que esperar que os dados se propaguem do mestre.

Bancos de dados de grafos garantem consistência por meio de transações. Eles não permitem relacionamentos pendentes: o nó inicial e o nó final sempre precisam existir, e os nós só podem ser excluídos se não tiverem nenhum relacionamento anexado a eles.

11.2.2 Transações

O Neo4J é compatível com ACID. Antes de alterar qualquer nó ou adicionar qualquer relacionamento a nós existentes, temos que iniciar uma transação. Sem envolver operações em transações, obteremos uma `NotInTransactionException`. Operações de leitura podem ser feitas sem iniciar uma transação.

```
Transação transação = database.beginTx(); tente { Nó nó =
database.createNode(); nó.setProperty("nome",
"NoSQL Destilled"); nó.setProperty("publicado", "2012");
transação.sucesso(); } finalmente { transação.finish();
}

}
```

No código acima, iniciamos uma transação no banco de dados, criamos um nó e definimos propriedades nele. Marcamos a transação como sucesso e finalmente a concluímos por finish. Uma transação tem que ser marcada como sucesso, caso contrário, o Neo4J assume que foi uma falha e a reverte quando finish é emitido. Definir sucesso sem emitir finish também não confirma os dados no banco de dados.

Essa maneira de gerenciar transações deve ser lembrada durante o desenvolvimento, pois difere da maneira padrão de fazer transações em um RDBMS.

11.2.3 Disponibilidade

O Neo4J, a partir da versão 1.8, atinge alta disponibilidade ao fornecer escravos replicados. Esses escravos também podem manipular gravações: quando são gravados, eles sincronizam a gravação com o mestre atual, e a gravação é confirmada primeiro no mestre e depois no escravo. Outros escravos eventualmente receberão a atualização. Outros bancos de dados de gráficos, como Infinite Graph e FlockDB, fornecem armazenamento distribuído dos nós.

O Neo4J usa o Apache ZooKeeper [ZooKeeper] para manter o controle dos últimos IDs de transação persistidos em cada nó escravo e no nó mestre atual. Uma vez que um servidor é inicializado, ele se comunica com o ZooKeeper e descobre qual servidor é o mestre. Se o servidor for o primeiro a se juntar ao cluster, ele se torna o mestre; quando um mestre cai, o cluster elege um mestre dos nós disponíveis, fornecendo assim alta disponibilidade.

11.2.4 Recursos de consulta

Bancos de dados de gráficos são suportados por linguagens de consulta como Gremlin [Gremlin]. Gremlin é uma linguagem específica de domínio para percorrer gráficos; ela pode percorrer todos os bancos de dados de gráficos que implementam o gráfico de propriedade Blueprints [Blueprints]. O Neo4J também tem a linguagem de consulta Cypher [Cypher] para consultar o gráfico. Fora dessas linguagens de consulta, o Neo4J permite que você consulte o gráfico para propriedades dos nós, percorra o gráfico ou navegue pelos relacionamentos dos nós usando vinculações de linguagem.

Propriedades de um nó podem ser indexadas usando o serviço de indexação. Similarmente, propriedades de relacionamentos ou arestas podem ser indexadas, então um nó ou aresta pode ser encontrado pelo valor. Índices devem ser consultados para encontrar o nó inicial para começar uma travessia. Vamos dar uma olhada na busca pelo nó usando indexação de nó.

Se tivermos o gráfico mostrado na Figura 11.1, podemos indexar os nós à medida que eles são adicionados ao banco de dados, ou podemos indexar todos os nós posteriormente, iterando sobre eles. Primeiro precisamos criar um índice para os nós usando o **IndexManager**.

```
Índice<Nó> nodeIndex = graphDb.index().forNodes("nós");
```

Estamos indexando os nós para a propriedade name . O Neo4J usa o Lucene [Lucene] como seu serviço de indexação. Veremos mais tarde que também podemos usar a capacidade de pesquisa de texto completo do Lucene. Quando novos nós são criados, eles podem ser adicionados ao índice.

```
Transação transação = graphDb.beginTx(); tente { Índice<Nó> nodeIndex
=
graphDb.index().forNodes("nós"); nodeIndex.add(martin, "nome", martin.getProperty("nome"));
nodeIndex.add(pramod, "nome", pramod.getProperty("nome")); transação.sucesso(); } finalmente
{ transação.finish();
```

```
}
```

Adicionar nós ao índice é feito dentro do contexto de uma transação. Uma vez que os nós são indexados, podemos pesquisá-los usando a propriedade indexed. Se procurarmos o nó com o nome de Barbara, consultaríamos o índice para a propriedade de name ter um valor de Barbara.

```
Nó nó = nodeIndex.get("nome", "Barbara").getSingle();
```

Obtemos o nó cujo nome é Martin; dado o nó, podemos obter todos os seus relacionamentos.

```
Nó martin = nodeIndex.get("nome", "Martin").getSingle(); todosRelacionamentos =
martin.getRelacionamentos();
```

Podemos ter relacionamentos de ENTRADA ou SAÍDA .

```
incomingRelations = martin.getRelationships(Direção.INCOMING);
```

Também podemos aplicar filtros direcionais nas consultas ao consultar um relacionamento. Com o gráfico na Figura 11.1, se quisermos encontrar todas as pessoas que gostam de NoSQL Distilled, podemos encontrar o nó NoSQL Distilled e então obter seus relacionamentos com Direction.INCOMING . Neste ponto, também podemos adicionar o tipo de relacionamento ao filtro de consulta, já que estamos procurando apenas por nós que GOSTAM de NoSQL Distilled.

```
Nó nosqlDistilled = nodeIndex.get("nome", "NoSQL Destilled").getSingle();
```

```
relacionamentos = nosqlDistilled.getRelationships(INCOMING, LIKES); para (Relacionamento relacionamento :
relacionamentos) { likesNoSQLDistilled.add(relationship.getStartNode()); }
```

Encontrar nós e suas relações imediatas é fácil, mas isso também pode ser alcançado em bancos de dados RDBMS. Bancos de dados de grafos são realmente poderosos quando você quer percorrer os grafos em qualquer profundidade e especificar um nó inicial para a travessia. Isso é especialmente útil quando você está tentando encontrar nós que são relacionados ao nó inicial em mais de um nível abaixo. À medida que a profundidade do grafo aumenta, faz mais sentido percorrer os relacionamentos usando um Traverser onde você pode especificar que está procurando por tipos de relacionamentos INCOMING, OUTGOING ou BOTH . Você também pode fazer o traverser ir de cima para baixo ou lateralmente no grafo usando valores de Ordem de BREADTH_FIRST ou DEPTH_FIRST . A travessia tem que começar em algum nó — neste exemplo, tentamos encontrar todos os nós em qualquer profundidade que são relacionados como um FRIEND com Barbara:

```
Nó barbara = nodeIndex.get("nome", "Barbara").getSingle();
```

```
Amigos do TraverserTraverser = barbara.traverse(Ordem.BREADTH_FIRST, StopEvaluator.END_OF_GRAPH,
    ReturnableEvaluator.ALL_BUT_START_NODE,
    EdgeType.FRIEND, Direção.OUTGOING);
```

O friendsTraverser nos fornece uma maneira de encontrar todos os nós que estão relacionados a Barbara onde o tipo de relacionamento é FRIEND. Os nós podem estar em qualquer profundidade — amigo de um amigo em qualquer nível — permitindo que você explore estruturas de árvore.

Um dos bons recursos dos bancos de dados de grafos é encontrar caminhos entre dois nós — determinar se há vários caminhos, encontrar todos os caminhos ou o caminho mais curto. No grafo da Figura 11.1, sabemos que Barbara está conectada a Jill por dois caminhos distintos; para encontrar todos esses caminhos e a distância entre Barbara e Jill ao longo desses diferentes caminhos, podemos usar

```
Nó barbara = nodeIndex.get("nome", "Barbara").getSingle(); Nó jill = nodeIndex.get("nome",
    "Jill").getSingle(); PathFinder<Caminho> finder = GraphAlgoFactory.allPaths(
    Traversal.expanderForTypes(AMIGO,Direção.SAÍDA)
    ,MAX_DEPTH);
Iterável<Caminho> caminhos = finder.findAllPaths(barbara, jill);
```

Este recurso é usado em redes sociais para mostrar relacionamentos entre quaisquer dois nós. Para encontrar todos os caminhos e a distância entre os nós para cada caminho, primeiro obtemos uma lista de caminhos distintos entre os dois nós. O comprimento de cada caminho é o **número de saltos** no gráfico necessários para atingir o nó de destino a partir do nó inicial. Frequentemente, você precisa obter o caminho mais curto entre dois nós; dos dois caminhos de Barbara para Jill, o caminho mais curto pode ser encontrado usando

```
PathFinder<Caminho> finder = GraphAlgoFactory.shortestPath(
    Traversal.expanderForTypes(AMIGO, Direção.SAÍDA)
    , PROFUNDIDADE_MÁXIMA);
Iterável<Caminho> caminhos = finder.findAllPaths(barbara, jill);
```

Muitos outros algoritmos de gráfico podem ser aplicados ao gráfico em questão, como o algoritmo de Dijkstra [Dijkstra's] para encontrar o caminho mais curto ou mais barato entre os nós.

```
START beginningNode = (especificação do nó inicial)
MATCH (relacionamento, correspondência de padrões)
ONDE (condição de filtragem: em dados em nós e relacionamentos)
RETORNAR (O que retornar: nós, relacionamentos, propriedades)
ORDENAR POR (propriedades para ordenar por)
SKIP (nós para pular do topo)
LIMIT (resultados limite)
```

O Neo4J também fornece a linguagem de consulta **Cypher** para consultar o gráfico. O Cypher precisa de um nó para INICIAR a consulta. O nó inicial pode ser identificado por seu ID de nó, uma lista de IDs de nó ou pesquisas de índice. O Cypher usa a palavra-chave MATCH para padrões de correspondência em relacionamentos; a palavra-chave WHERE filtra as propriedades em um nó ou relacionamento. A palavra-chave RETURN especifica o que é retornado pela consulta — nós, relacionamentos ou campos nos nós ou relacionamentos.

O Cypher também fornece métodos para ORDENAR, AGREGAR, PULAR e LIMITAR os dados. Na Figura 11.2, encontramos todos os nós conectados à Barbara, tanto de entrada quanto de saída, usando 0 --.

```
INICIAR barbara = nó:nodelIndex(nome = "Barbara")
CORRESPONDÊNCIA (barbara)--(connected_node)
RETORNAR nó_conectado
```

Quando estivermos interessados em significância direcional, podemos usar

```
CORRESPONDÊNCIA (barbara)<--(connected_node)
```

para relacionamentos de entrada ou

```
CORRESPONDÊNCIA (barbara)-->(nó_conectado)
```

para relacionamentos de saída. A correspondência também pode ser feita em relacionamentos específicos usando a convenção :RELATIONSHIP_TYPE e retornando os campos ou nós necessários.

```
INICIAR barbara = nó:nodelIndex(nome = "Barbara")
CORRESPONDÊNCIA (barbara)-[:AMIGO]->(friend_node)
RETORNAR friend_node.name,friend_node.location
```

Começamos com Barbara, encontramos todos os relacionamentos de saída com o tipo FRIEND e retornamos os nomes dos amigos. A consulta do tipo de relacionamento funciona apenas para a profundidade de um nível; podemos fazê-la funcionar para profundidades maiores e descobrir a profundidade de cada um dos nós de resultado.

```
INICIAR barbara=node:index(nome = "Barbara")
CORRESPONDÊNCIA caminho = barbara-[:AMIGO*1..3]->end_node
RETORNAR barbara.nome,end_node.nome, comprimento(caminho)
```

Similarmente, podemos consultar relacionamentos onde uma propriedade de relacionamento específica existe. Também podemos filtrar nas propriedades de relacionamentos e consultar se uma propriedade existe ou não.

```
INICIAR barbara = nó:index(nome = "Barbara")
CORRESPONDÊNCIA (barbara)-[relação]->(nó_relacionado)
ONDE tipo(relação) = 'AMIGO' E relação.compartilhar RETORNA nó_relacionado.nome,
relação.desde
```

Existem muitos outros recursos de consulta na linguagem Cypher que podem ser usados para consultar gráficos de banco de dados.

11.2.5 Escala

Em bancos de dados NoSQL, uma das técnicas de dimensionamento comumente usadas é o sharding, onde os dados são divididos e distribuídos em diferentes servidores. Com bancos de dados de grafos, o sharding é difícil, pois os bancos de dados de grafos não são orientados a agregados, mas sim a relacionamentos. Como qualquer nó pode ser relacionado a qualquer outro nó, armazenar nós relacionados no mesmo servidor é melhor para a travessia de grafos. Atravessar um grafo quando os nós estão em máquinas diferentes não é bom para o desempenho.

Sabendo dessa limitação dos bancos de dados de gráficos, ainda podemos escalá-los usando algumas técnicas comuns descritas por Jim Webber [Webber Neo4J Scaling].

Em termos gerais, há três maneiras de escalar bancos de dados de grafos. Como as máquinas agora podem vir com muita RAM, podemos adicionar RAM suficiente ao servidor para que o conjunto de trabalho de nós e relacionamentos seja mantido inteiramente na memória.

Essa técnica só é útil se o conjunto de dados com o qual estamos trabalhando couber em uma quantidade realista de RAM.

Podemos melhorar o dimensionamento de leitura do banco de dados adicionando mais escravos com acesso somente leitura aos dados, com todas as gravações indo para o mestre. Esse padrão de escrever uma vez e ler de muitos servidores é uma técnica comprovada em clusters MySQL e é realmente útil quando o conjunto de dados é grande o suficiente para não caber na RAM de uma única máquina, mas pequeno o suficiente para ser replicado em várias máquinas.

Os escravos também podem contribuir para a disponibilidade e o dimensionamento de leitura, pois podem ser configurados para nunca se tornarem mestres, permanecendo sempre somente leitura.

Quando o tamanho do conjunto de dados torna a replicação impraticável, podemos fragmentar (veja a seção "Fragmentação" na p. 38) os dados do lado do aplicativo usando conhecimento específico do domínio. Por exemplo, nós relacionados à América do Norte podem ser criados em um servidor, enquanto os nós relacionados à Ásia em outro. Essa fragmentação no nível do aplicativo precisa entender que os nós são armazenados em bancos de dados fisicamente diferentes (Figura 11.3).

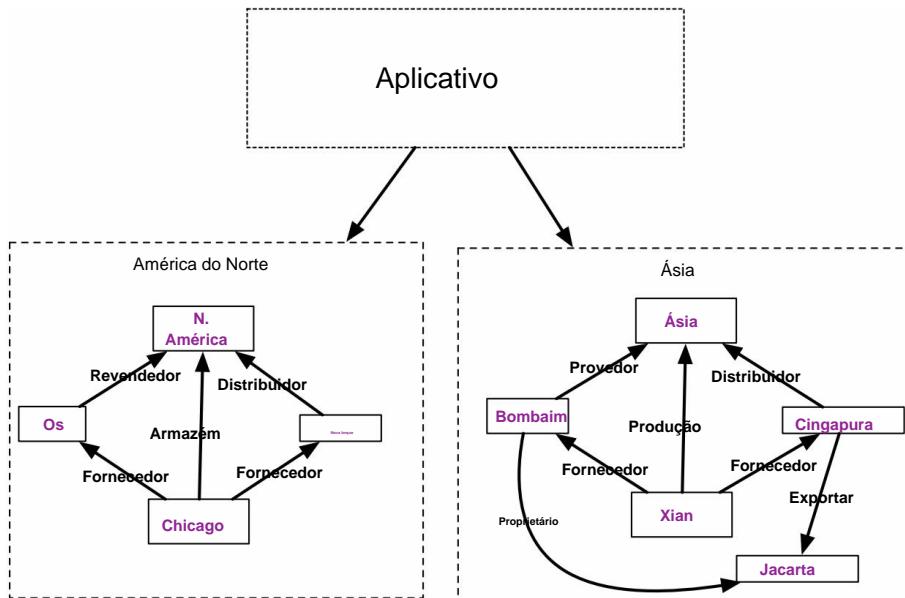


Figura 11.3 Fragmentação de nós em nível de aplicativo

11.3 Casos de uso adequados

Vejamos alguns casos de uso adequados para bancos de dados gráficos.

11.3.1 Dados conectados

Redes sociais são onde bancos de dados de gráficos podem ser implantados e usados de forma muito eficaz. Esses gráficos sociais não precisam ser apenas do tipo amigo; por exemplo, eles podem representar funcionários, seus conhecimentos e onde eles trabalharam com outros funcionários em diferentes projetos. Qualquer domínio rico em links é bem adequado para bancos de dados de gráficos.

Se você tiver relacionamentos entre entidades de domínios diferentes (como social, espacial, comercial) em um único banco de dados, poderá tornar esses relacionamentos mais valiosos fornecendo a capacidade de atravessar domínios.

11.3.2 Serviços de roteamento, despacho e baseados em localização

Cada local ou endereço que tem uma entrega é um nó, e todos os nós onde a entrega tem que ser feita pelo entregador podem ser modelados como um gráfico de nós. Os relacionamentos entre nós podem ter a propriedade de distância, portanto

permitindo que você entregue as mercadorias de forma eficiente. Propriedades de distância e localização também podem ser usadas em gráficos de locais de interesse, para que seu aplicativo possa fornecer recomendações de bons restaurantes ou opções de entretenimento nas proximidades. Você também pode criar nós para seus pontos de venda, como livrarias ou restaurantes, e notificar os usuários quando eles estiverem próximos de qualquer um dos nós para fornecer serviços baseados em localização.

11.3.3 Mecanismos de Recomendação

À medida que nós e relacionamentos são criados no sistema, eles podem ser usados para fazer recomendações como "seus amigos também compraram este produto" ou "ao faturar este item, esses outros itens geralmente são faturados". Ou pode ser usado para fazer recomendações a viajantes mencionando que quando outros visitantes vêm a Barcelona, eles geralmente visitam as criações de Antonio Gaudi.

Um efeito colateral interessante do uso de bancos de dados de gráficos para recomendações é que, à medida que o tamanho dos dados cresce, o número de nós e relacionamentos disponíveis para fazer as recomendações aumenta rapidamente. Os mesmos dados também podem ser usados para minerar informações — por exemplo, quais produtos são sempre comprados juntos ou quais itens são sempre faturados juntos; alertas podem ser gerados quando essas condições não são atendidas. Como outros mecanismos de recomendação, bancos de dados de gráficos podem ser usados para pesquisar padrões em relacionamentos para detectar fraudes em transações.

11.4 Quando não usar

Em algumas situações, bancos de dados de gráficos podem não ser apropriados. Quando você deseja atualizar todas ou um subconjunto de entidades — por exemplo, em uma solução analítica em que todas as entidades podem precisar ser atualizadas com uma propriedade alterada — bancos de dados de gráficos podem não ser ideais, pois alterar uma propriedade em todos os nós não é uma operação direta. Mesmo que o modelo de dados funcione para o domínio do problema, alguns bancos de dados podem não conseguir lidar com muitos dados, especialmente em operações de gráficos globais (aqueles que envolvem o gráfico inteiro).

Esta página foi deixada em branco intencionalmente

Capítulo 12

Migrações de esquema

12.1 Alterações de esquema

A tendência recente na discussão de bancos de dados NoSQL é destacar sua natureza *sem esquemas* — é um recurso popular que permite que os desenvolvedores se concentrem no design do domínio sem se preocupar com mudanças de esquema. Isso é especialmente verdadeiro com o surgimento de métodos ágeis [Métodos Ágeis], onde responder a requisitos em mudança é importante.

Discussões, iterações e loops de feedback envolvendo especialistas de domínio e proprietários de produtos são importantes para derivar o entendimento correto dos dados; essas discussões não devem ser prejudicadas pela complexidade do esquema de um banco de dados. Com os armazenamentos de dados NoSQL, as alterações no esquema podem ser feitas com o mínimo de atrito, melhorando a produtividade do desenvolvedor (“The Emergence of NoSQL,” p. 9). Vimos que desenvolver e manter um aplicativo no admirável mundo novo dos bancos de dados sem esquemas exige atenção especial à migração de esquemas.

12.2 Alterações de esquema no RDBMS

Ao desenvolver com tecnologias RDBMS padrão, desenvolvemos objetos, suas tabelas correspondentes e seus relacionamentos. Considere um modelo de objeto simples e um modelo de dados que tenha Customer, Order e OrderItems. O modelo ER ficaria como na Figura 12.1.

Embora esse modelo de dados suporte o modelo de objeto atual, a vida é boa. A primeira vez que há uma mudança no modelo de objeto, como a introdução de preferredShippingType no objeto Customer, temos que mudar o objeto e mudar a tabela do banco de dados, porque sem mudar a tabela o aplicativo ficará fora de sincronia com o banco de dados. Quando obtemos erros como ORA-00942: table

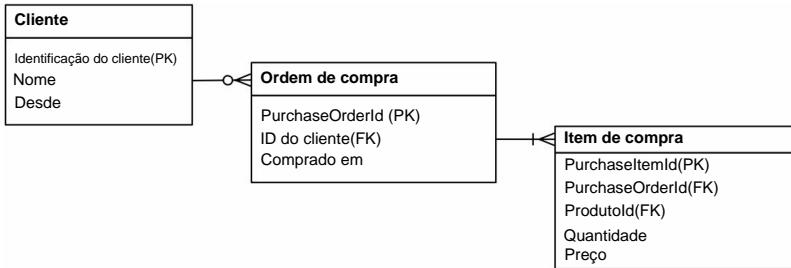


Figura 12.1 Modelo de dados de um sistema de comércio eletrônico

ou a visualização não existe ou ORA-00904: "PREFERRED_SHIPPING_TYPE": identificador inválido, sabemos que temos esse problema.

Normalmente, uma migração de esquema de banco de dados tem sido um projeto em si. Para a implementação das alterações de esquema, scripts de alteração de banco de dados são desenvolvidos, usando técnicas de diff, para todas as alterações no banco de dados de desenvolvimento. Essa abordagem de criação de scripts de migração durante o tempo de implementação/liberação é propensa a erros e não suporta métodos de desenvolvimento ágeis.

12.2.1 Migrações para Projetos Green Field

O script das alterações do esquema do banco de dados durante o desenvolvimento é melhor, pois podemos armazenar essas alterações de esquema junto com os scripts de migração de dados no mesmo arquivo de script. Esses arquivos de script devem ser nomeados com números sequenciais incrementais que refletem as versões do banco de dados; por exemplo, a primeira alteração no banco de dados pode ter um arquivo de script nomeado como 001_Description_Of_Change.sql.

Scripting changes dessa forma permite que as migrações do banco de dados sejam executadas preservando a ordem das alterações. A Figura 12.2 mostra uma pasta com todas as alterações feitas em um banco de dados até o momento.

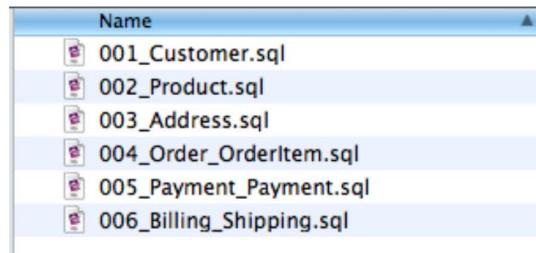


Figura 12.2 Sequência de migrações aplicadas a um banco de dados

Agora, suponha que precisamos alterar a tabela OrderItem para armazenar o DiscountedPrice e o FullPrice do item. Isso precisará de uma alteração na tabela OrderItem e será a alteração número 007 em nossa sequência de alterações, conforme mostrado na Figura 12.3.

Name
001_Customer.sql
002_Product.sql
003_Address.sql
004_Order_OrderItem.sql
005_Payment_Payment.sql
006_Billing_Shipping.sql
007_DiscountedPrice.sql

Figura 12.3 Nova alteração 007_DiscountedPrice.sql aplicada ao banco de dados

Aplicamos uma nova alteração ao banco de dados. O script dessa alteração tem o código para adicionar uma nova coluna, renomear a coluna existente e migrar os dados necessários para fazer o novo recurso funcionar. Abaixo está o script contido na alteração 007_DiscountedPrice.sql:

```
ALTER TABLE orderitem ADD discountedprice NÚMERO(18,2) NULO;
ATUALIZAR item do pedido DEFINIR preçocomdesconto = preço;
ALTER TABLE orderitem MODIFY discountedprice NÃO NULO;
ALTER TABLE orderitem RENOMEAR COLUNA price PARA fullprice;
--//@DESFAZER
ALTER TABLE orderitem RENAME preçocompleto PARA preço;
ALTER TABLE orderitem DROP COLUMN preçocomdesconto;
```

O script de alteração mostra as alterações de esquema no banco de dados, bem como as migrações de dados necessárias a serem feitas. No exemplo mostrado, estamos usando DBDeploy [DBDeploy] como a estrutura para gerenciar as alterações no banco de dados. O DBDeploy mantém uma tabela no banco de dados, chamada ChangeLog, onde todas as alterações feitas no banco de dados são armazenadas. Nessa tabela, Change_Number é o que informa a todos quais alterações foram aplicadas ao banco de dados. Esse Change_Number, que é a versão do banco de dados, é então usado para encontrar o script numerado correspondente na pasta e aplicar as alterações que ainda não foram aplicadas. Quando escrevemos um script com o número de alteração 007 e o aplicamos ao banco de dados usando o DBDeploy, o DBDeploy verificará o ChangeLog e pegará todos os scripts da pasta que ainda não foram aplicados. A Figura 12.4 é a captura de tela do DBDeploy aplicando a alteração ao banco de dados.

A melhor maneira de se integrar com o restante dos desenvolvedores é usar o repositório de controle de versão do seu projeto para armazenar todos esses scripts de alteração, para que você possa acompanhar a versão do software e do banco de dados no mesmo lugar, eliminando

```

project $>ant dbupgrade
Buildfile: /project/build.xml

init:

dbupgrade:
[dbdeploy] dbdeploy 3.0M3
[dbdeploy] Reading change scripts from directory /project/db/migrations...
[dbdeploy] Changes currently applied to database:
[dbdeploy]   1..6
[dbdeploy] Scripts available:
[dbdeploy]   1..7
[dbdeploy] To be applied:
[dbdeploy]   7
[dbdeploy] Applying #7: 007_DiscountedPrice.sql...
[dbdeploy]   -> statement 1 of 4...
[dbdeploy]   -> statement 2 of 4...
[dbdeploy]   -> statement 3 of 4...
[dbdeploy]   -> statement 4 of 4...

BUILD SUCCESSFUL
Total time: 0 seconds
project $>]

```

Figura 12.4 DBDeploy atualizando o banco de dados com número de alteração 007

possíveis incompatibilidades entre o banco de dados e o aplicativo. Existem muitas outras ferramentas para tais atualizações, incluindo Liquibase [Liquibase], MyBatis Migrator [MyBatis Migrator], DBMaintain [DBMaintain].

12.2.2 Migrações em Projetos Legados

Nem todo projeto é um campo verde. Como implementar migrações quando um projeto existente aplicativo está em produção? Descobrimos que pegar um banco de dados existente e extrair sua estrutura em scripts, junto com todo o código do banco de dados e quaisquer dados de referência, funciona como uma linha de base para o projeto. Esta linha de base não deve conter dados transacionais. Uma vez que a linha de base esteja pronta, outras alterações podem ser feitas usando a técnica de migrações descrita acima (Figura 12.5).

Um dos principais aspectos das migrações deve ser manter a compatibilidade com versões anteriores do esquema do banco de dados. Em muitas empresas, há vários aplicativos usando o banco de dados; quando mudamos o banco de dados para um aplicativo, isso a mudança não deve quebrar outros aplicativos. Podemos atingir compatibilidade com versões anteriores mantendo uma fase de transição para a mudança, conforme descrito em detalhes em *Refatoração de bancos de dados* [Ambler e Sadalage].

Durante uma **fase de transição**, o esquema antigo e o novo esquema são mantidos em paralelo e estão disponíveis para todas as aplicações que utilizam o banco de dados. Para isso, temos que introduzir código de andaime, como gatilhos, visualizações e colunas virtuais

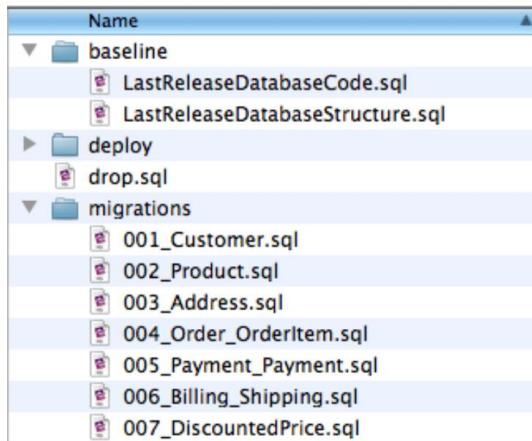


Figura 12.5 Uso de scripts de linha de base com um banco de dados legado

garantindo que outros aplicativos possam acessar o esquema do banco de dados e os dados necessários sem nenhuma alteração no código.

```
ALTER TABLE cliente ADD nome completo VARCHAR2(60);
ATUALIZAR cliente SET fullname = fname;
```

```
CRIE OU SUBSTITUA O GATILHO SyncCustomerFullName ANTES DE INSERIR OU
ATUALIZAR NO cliente REFERENCIANDO
```

ANTIGO COMO

ANTIGO NOVO COMO NOVO PARA CADA LINHA COMEÇAR

SE :NEW.fname FOR

NULO

ENTÃO

:NEW.fname := :NEW.fullname; FIM SE;

SE :NEW.fullname FOR NULO

ENTÃO :NEW.fullname := :NEW.fname

FIM SE;

FIM; /

```
--Drop Trigger e fname --quando todos os
aplicativos começam a usar customer.fullname
```

No exemplo, estamos tentando renomear a coluna customer.fname para customer.fullname , pois queremos evitar qualquer ambiguidade de fname significando fullname ou firstname. Uma renomeação direta da coluna fname e a alteração do código do aplicativo pelo qual somos responsáveis podem funcionar para o nosso aplicativo — mas não para os outros aplicativos na empresa que estão acessando o mesmo banco de dados.

Usando a técnica de fase de transição, introduzimos a nova coluna fullname, copiamos os dados para fullname, mas deixamos a coluna antiga fname por perto. Também introduzimos um gatilho BEFORE UPDATE para sincronizar dados entre as colunas antes que elas sejam confirmadas no banco de dados.

Agora, quando os aplicativos leem dados da tabela, eles lerão de fname ou de fullname , mas sempre obterão os dados corretos. Podemos remover o gatilho e a coluna fname quando todos os aplicativos tiverem passado a usar a nova coluna fullname .

É muito difícil fazer migrações de esquemas em grandes conjuntos de dados em RDBMS, especialmente se tivermos que manter o banco de dados disponível para os aplicativos, pois grandes movimentações de dados e mudanças estruturais geralmente criam bloqueios nas tabelas do banco de dados.

12.3 Alterações de esquema em um armazenamento de dados NoSQL

Um banco de dados RDBMS precisa ser alterado antes que o aplicativo seja alterado. É isso que a abordagem *sem esquema*, ou *schemaless*, tenta evitar, visando flexibilidade de alterações de esquema por entidade. Alterações frequentes no esquema são necessárias para reagir a mudanças frequentes de mercado e inovações de produtos.

Ao desenvolver com bancos de dados NoSQL, em alguns casos o esquema não precisa ser pensado de antemão. Ainda temos que projetar e pensar sobre outros aspectos, como os tipos de relacionamentos (com bancos de dados de gráficos), ou os nomes das famílias de colunas, linhas, colunas, ordem das colunas (com bancos de dados de colunas), ou como as chaves são atribuídas e qual é a estrutura dos dados dentro do objeto de valor (com armazenamentos de chave-valor). Mesmo que não tenhamos pensado sobre isso antes, ou se quisermos mudar nossas decisões, é fácil fazer isso.

A alegação de que bancos de dados NoSQL são totalmente sem esquema é enganosa; enquanto eles armazenam os dados sem considerar o esquema ao qual os dados aderem, esse esquema tem que ser definido pelo aplicativo, porque o fluxo de dados tem que ser analisado pelo aplicativo ao ler os dados do banco de dados. Além disso, o aplicativo tem que criar os dados que seriam salvos no banco de dados. Se o aplicativo não puder analisar os dados do banco de dados, temos uma incompatibilidade de esquema mesmo se, em vez do banco de dados RDBMS lançar um erro, esse erro agora for encontrado pelo aplicativo. Portanto, mesmo em bancos de dados sem esquema, o esquema dos dados tem que ser levado em consideração ao refatorar o aplicativo.

Mudanças de esquema são especialmente importantes quando há um aplicativo implantado e dados de produção existentes. Para simplificar, suponha que estamos usando um repositório de dados de documentos como o MongoDB [MongoDB] e temos o mesmo modelo de dados de antes: customer, order e orderItems.

```
{
  "_id": "4BD8AE97C47016442AF4A580",
  "customerid": 99999,
  "name": "Foo Sushi Inc", "since":
  "12/12/2012", "order":
  { "orderid":
    "4821-UXWE-122012", "orderdate": "12/12/2001", "orderItems": [{"product":
      "Biscoitos da Sorte",
      "preço": 19,99}]}
  }
}
```

Código do aplicativo para escrever esta estrutura de documento no MongoDB:

```
BasicDBObject orderItem = new BasicDBObject();
orderItem.put("produto", productName);
orderItem.put("preço", preço);
orderItems.add(orderItem);
```

Código para ler o documento de volta do banco de dados:

```
BasicDBObject item = (BasicDBObject) orderItem; String
productName = item.getString("produto"); Preço duplo =
item.getDouble("preço");
```

Alterar os objetos para adicionar preferredShippingType não requer nenhuma alteração no banco de dados, pois o banco de dados não se importa que documentos diferentes não sigam o mesmo esquema. Isso permite um desenvolvimento mais rápido e implantações fáceis. Tudo o que precisa ser implantado é o aplicativo — nenhuma alteração no lado do banco de dados é necessária. O código precisa garantir que os documentos que não têm o atributo preferredShippingType ainda possam ser analisados — e isso é tudo.

Claro que estamos simplificando a situação de mudança de esquema aqui. Vamos dar uma olhada na mudança de esquema que fizemos antes: introduzindo discountedPrice e renomeando price para fullPrice. Para fazer essa mudança, renomeamos o atributo price para fullPrice e adicionamos o atributo discountedPrice . O documento alterado é

```
{
  "_id": "5BD8AE97C47016442AF4A580",
  "customerid": 66778,
  "name": "India House", "since":
  "12/12/2012", "order":
  { "orderid":
    "4821-UXWE-222012", "orderdate":
    "12/12/2001", "orderItems":
    [{"product": "Capas para cadeiras", "fullPrice": 29,99,
      "discountedPrice": 26,99}]}
}
```

Depois que implementarmos essa mudança, novos clientes e seus pedidos poderão ser salvos e lidos novamente sem problemas, mas para pedidos existentes o preço do produto

não pode ser lido, porque agora o código está procurando por `fullPrice`, mas o documento tem apenas preço.

12.3.1 Migração Incremental

A incompatibilidade de esquemas atrai muitos novos convertidos ao mundo NoSQL. Quando o esquema é alterado no aplicativo, temos que ter certeza de converter todos os dados existentes para o novo esquema (dependendo do tamanho dos dados, esta pode ser uma operação cara). Outra opção seria ter certeza de que os dados, antes da alteração do esquema, ainda podem ser analisados pelo novo código e, quando são salvos, são salvos de volta no novo esquema. Esta técnica, conhecida como **migração incremental**, migrará dados ao longo do tempo; alguns dados podem nunca ser migrados, porque nunca foram acessados. Estamos lendo `price` e `fullPrice` do documento:

```
BasicDBObject item = (BasicDBObject) orderItem;
String productName = item.getString("produto");
Double fullPrice =
item.getDouble("preço");
if (fullPrice == null) { fullPrice =
item.getDouble("fullPrice");

}
Preçocomdescontoduplo = item.getDouble("Preçocomdesconto");
```

Ao escrever o documento de volta, o antigo atributo `price` não é salvo:

```
BasicDBObject orderItem = new BasicDBObject();
orderItem.put("produto", productName);
orderItem.put("preçocompleto", preço);
orderItem.put("preçocomdesconto", preçocomdesconto);
orderItems.add(itemopedido);
```

Ao usar migração incremental, pode haver muitas versões do objeto no lado do aplicativo que podem traduzir o esquema antigo para o novo esquema; ao salvar o objeto de volta, ele é salvo usando o novo objeto. Essa migração gradual dos dados ajuda o aplicativo a evoluir mais rápido.

A técnica de migração incremental complicará o design do objeto, especialmente porque novas mudanças estão sendo introduzidas, mas as antigas não estão sendo retiradas. Esse período entre a implantação da mudança e o último objeto no banco de dados migrando para o novo esquema é conhecido como período de transição (Figura 12.6). Mantenha-o o mais curto possível e concentre-o no escopo mínimo possível — isso ajudará você a manter seus objetos limpos.

A técnica de migração incremental também pode ser implementada com um campo `schema_version` nos dados, usado pelo aplicativo para escolher o código correto para analisar os dados nos objetos. Ao salvar, os dados são migrados para a versão mais recente e o `schema_version` é atualizado para refletir isso.

Ter uma camada de tradução adequada entre seu domínio e o banco de dados é importante para que, à medida que o esquema muda, seja possível gerenciar várias versões do

12.3 Alterações de esquema em um armazenamento de dados NoSQL

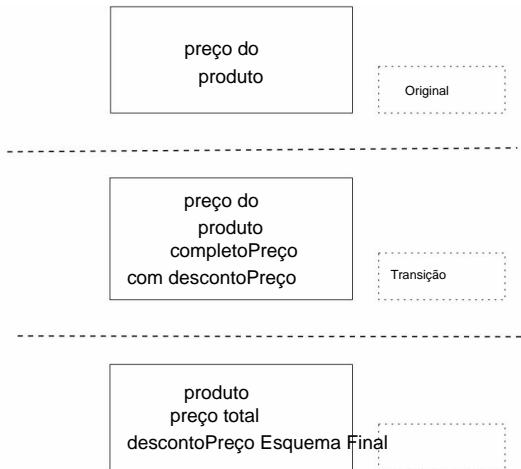


Figura 12.6 Período de transição das alterações de esquema

O esquema é restrito à camada de tradução e não vazia para todo o aplicativo.

Aplicativos móveis criam requisitos especiais. Como não podemos impor as últimas atualizações do aplicativo, o aplicativo deve ser capaz de lidar com quase todas as versões do esquema.

12.3.2 Migrações em bancos de dados de grafos

Bancos de dados de grafos têm arestas que têm tipos e propriedades. Se você alterar o tipo dessas arestas na base de código, não poderá mais atravessar o banco de dados, tornando-o inutilizável. Para contornar isso, você pode atravessar todas as arestas e alterar o tipo de cada aresta. Essa operação pode ser cara e requer que você escreva código para migrar todas as arestas no banco de dados.

Se precisarmos manter a compatibilidade com versões anteriores ou não quisermos alterar o gráfico inteiro de uma só vez, podemos simplesmente criar novas arestas entre os nós; mais tarde, quando estivermos confortáveis com a alteração, as arestas antigas podem ser descartadas. Podemos usar travessias com vários tipos de arestas para percorrer o gráfico usando os tipos de arestas novos e antigos. Essa técnica pode ajudar muito com bancos de dados grandes, especialmente se quisermos manter alta disponibilidade.

Se tivermos que alterar propriedades em todos os nós ou arestas, temos que buscar todos os nós e alterar todas as propriedades que precisam ser alteradas. Um exemplo seria adicionar NodeCreatedBy e NodeCreatedOn a todos os nós existentes para rastrear as alterações feitas em cada nó.

```
para (nó nó: database.getAllNodes())
    { nó.setProperty("NodeCreatedBy", getSystemUser());
    nó.setProperty("NodeCreatedOn", getSystemTimeStamp());
}
```

Podemos ter que alterar os dados nos nós. Novos dados podem ser derivados dos dados de nós existentes ou podem ser importados de alguma outra fonte. A migração pode ser feita buscando todos os nós usando um índice fornecido pela fonte de dados e gravando dados relevantes em cada nó.

12.3.3 Alterando a estrutura agregada

Às vezes, você precisa alterar o design do esquema, por exemplo, dividindo objetos grandes em menores que são armazenados independentemente. Suponha que você tenha um agregado de clientes que contém todos os pedidos dos clientes e você deseja separar o cliente e cada um dos seus pedidos em diferentes unidades agregadas.

Você então tem que garantir que o código pode funcionar com ambas as versões dos agregates. Se ele não encontrar os objetos antigos, ele procurará pelos novos agregados.

O código que roda em segundo plano pode ler um agregado por vez, fazer a alteração necessária e salvar os dados de volta em agregados diferentes. A vantagem de operar em um agregado por vez é que, dessa forma, você não está afetando a disponibilidade de dados para o aplicativo.

12.4 Leituras Adicionais

Para mais informações sobre migrações com bancos de dados relacionais, consulte [Ambler e Sadalage]. Embora grande parte desse conteúdo seja específico para trabalho relacional, os princípios gerais de migração também se aplicam a outros bancos de dados.

12.5 Pontos-chave

- Bancos de dados com esquemas fortes, como bancos de dados relacionais, podem ser migrados salvando cada alteração de esquema, além de sua migração de dados, em uma sequência controlada por versão.
- Bancos de dados sem esquema ainda precisam de migração cuidadosa devido ao esquema implícito em qualquer código que acessa os dados.
- Bancos de dados sem esquemas podem usar as mesmas técnicas de migração que bancos de dados com esquemas fortes.
- Bancos de dados sem esquema também podem ler dados de uma forma tolerante a alterações no esquema implícito dos dados e usar migração incremental para atualizar dados.

Capítulo 13

Persistência Poliglota

Diferentes bancos de dados são projetados para resolver diferentes problemas. Usar um único mecanismo de banco de dados para todos os requisitos geralmente leva a soluções de baixo desempenho; armazenar dados transacionais, armazenar informações de sessão em cache, percorrer gráficos de clientes e os produtos que seus amigos compraram são problemas essencialmente diferentes. Mesmo no espaço RDBMS, os requisitos de um sistema OLAP e OLTP são muito diferentes — no entanto, eles são frequentemente forçados ao mesmo esquema.

Vamos pensar em relacionamentos de dados. Soluções RDBMS são boas em impor que relacionamentos existam. Se quisermos descobrir relacionamentos, ou tivermos que encontrar dados de tabelas diferentes que pertencem ao mesmo objeto, então o uso de RDBMS começa a ser difícil.

Os mecanismos de banco de dados são projetados para executar certas operações em certas estruturas de dados e quantidades de dados muito bem, como operar em conjuntos de dados ou um armazenamento e recuperar chaves e seus valores muito rapidamente, ou armazenar documentos ricos ou gráficos complexos de informações.

13.1 Necessidades Disparas de Armazenamento de Dados

Muitas empresas tendem a usar o mesmo mecanismo de banco de dados para armazenar transações comerciais, dados de gerenciamento de sessão e para outras necessidades de armazenamento, como relatórios, BI, armazenamento de dados ou informações de registro (Figura 13.1).

Os dados da sessão, do carrinho de compras ou do pedido não precisam das mesmas propriedades de disponibilidade, consistência ou requisitos de backup. O armazenamento de gerenciamento de sessão precisa da mesma estratégia rigorosa de backup/recuperação que os dados de pedidos de e-commerce? O armazenamento de gerenciamento de sessão precisa de mais disponibilidade de uma instância do mecanismo de banco de dados para gravar/ler dados de sessão?

Em 2006, Neal Ford cunhou o termo **programação poliglota** para expressar a ideia de que os aplicativos devem ser escritos em uma mistura de linguagens para aproveitar

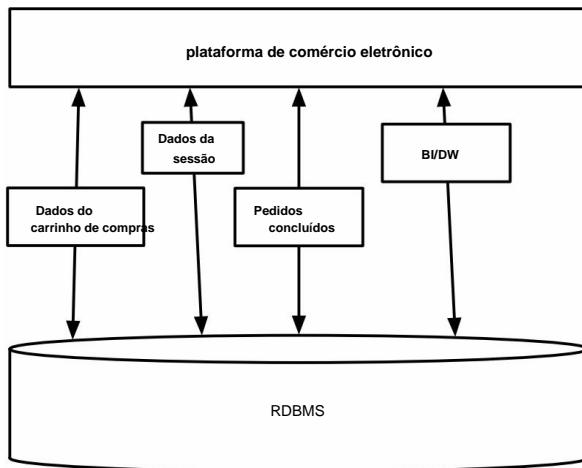


Figura 13.1 Uso de RDBMS para todos os aspectos de armazenamento do aplicativo

do fato de que diferentes línguas são adequadas para abordar diferentes problemas.

Aplicações complexas combinam diferentes tipos de problemas, portanto, escolher a linguagem certa para cada trabalho pode ser mais produtivo do que tentar encaixar todos os aspectos em uma única linguagem.

Da mesma forma, ao trabalhar em um problema de negócio de e-commerce, usar um armazenamento de dados para o carrinho de compras que seja altamente disponível e possa ser escalonado é importante, mas o mesmo armazenamento de dados não pode ajudar você a encontrar produtos comprados pelos amigos dos clientes — o que é uma questão totalmente diferente. Usamos o termo **persistência poliglota** para definir essa abordagem híbrida para persistência.

13.2 Uso do Polyglot Data Store

Vamos pegar nosso exemplo de e-commerce e usar a abordagem de persistência poliglota para ver como alguns desses armazenamentos de dados podem ser aplicados (Figura 13.2). Um armazenamento de dados de valor-chave pode ser usado para armazenar os dados do carrinho de compras antes que o pedido seja confirmado pelo cliente e também armazenar os dados da sessão para que o RDBMS não seja usado para esses dados transitórios. Os armazenamentos de valor-chave fazem sentido aqui, pois o carrinho de compras geralmente é acessado pelo ID do usuário e, uma vez confirmado e pago pelo cliente, pode ser salvo no RDBMS. Da mesma forma, os dados da sessão são codificados pelo ID da sessão.

Se precisarmos recomendar produtos aos clientes quando eles colocam produtos em seus carrinhos de compras — por exemplo, “seus amigos também compraram estes produtos”

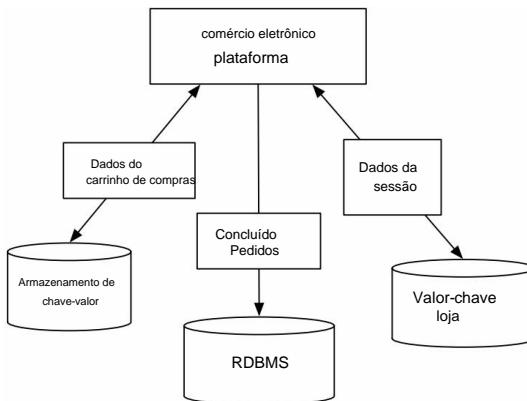


Figura 13.2 Uso de armazenamentos de chave-valor para descarregar dados de sessão e carrinho de compras

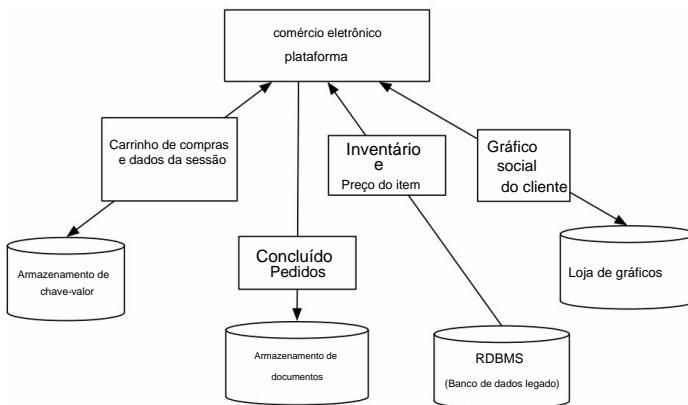


Figura 13.3 Exemplo de implementação de persistência poliglota

ou “seus amigos compraram esses acessórios para este produto” — então introduzindo um os dados do gráfico armazenados na mistura tornam-se relevantes (Figura 13.3).

Não é necessário que o aplicativo use um único armazenamento de dados para todos os seus necessidades, uma vez que diferentes bancos de dados são construídos para diferentes propósitos e nem todos problemas podem ser resolvidos elegantemente por um único banco de dados.

Mesmo utilizando bancos de dados relacionais especializados para diferentes propósitos, como coleta de dados dispositivos de armazenamento ou dispositivos analíticos dentro do mesmo aplicativo, podem ser visto como persistência poliglota.

13.3 Uso do serviço em detrimento do uso direto do armazenamento de dados

À medida que avançamos em direção a vários armazenamentos de dados no aplicativo, pode haver outros aplicativos na empresa que poderiam se beneficiar do uso de nossos armazenamentos de dados ou os dados armazenados neles. Usando nosso exemplo, o armazenamento de dados do gráfico pode servir dados para outras aplicações que precisam entender, por exemplo, quais produtos estão sendo comprados por um determinado segmento da base de clientes.

Em vez de cada aplicação falar independentemente com o banco de dados do gráfico, nós podemos encapsular o banco de dados de gráficos em um serviço para que todos os relacionamentos entre eles os nós podem ser salvos em um só lugar e consultados por todos os aplicativos (Figura 13.4). A propriedade dos dados e as APIs fornecidas pelo serviço são mais úteis do que um único aplicativo se comunicando com vários bancos de dados.

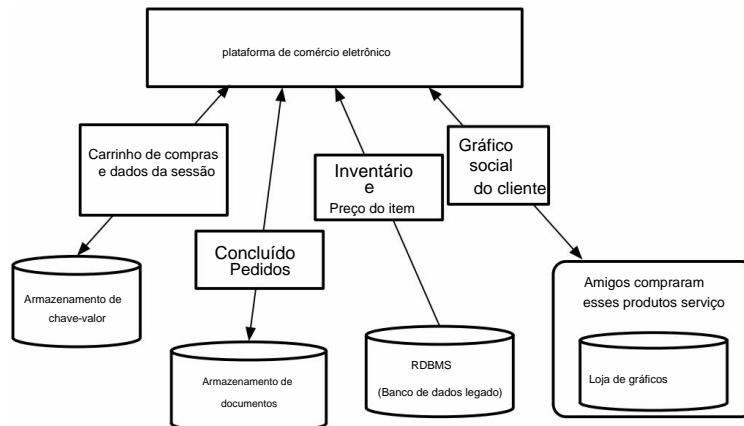


Figura 13.4 Exemplo de implementação de encapsulamento de armazenamentos de dados em serviços

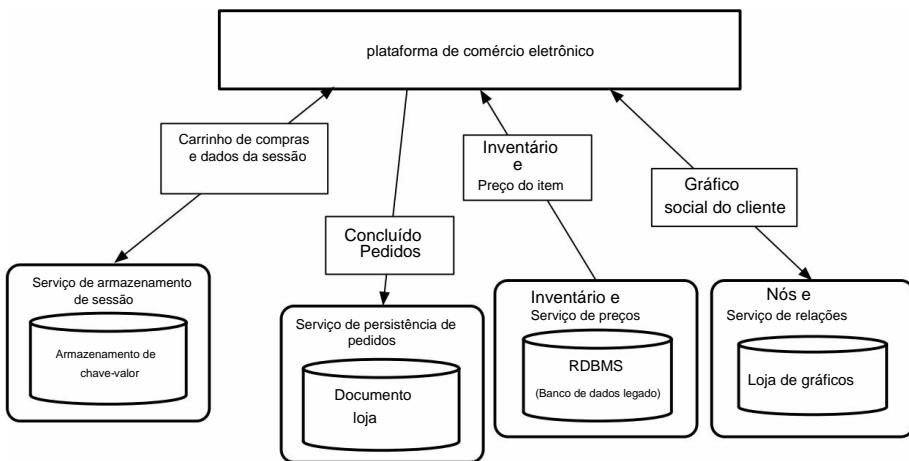
A filosofia de encapsulamento de serviços pode ser levada mais longe: você pode encapsular todos bancos de dados em serviços, permitindo que o aplicativo converse apenas com vários serviços (Figura 13.5). Isso permite que os bancos de dados dentro dos serviços evoluam sem você tendo que alterar os aplicativos dependentes.

Muitos produtos de armazenamento de dados NoSQL, como Riak [Riak] e Neo4J [Neo4J], na verdade, fornecem APIs REST prontas para uso.

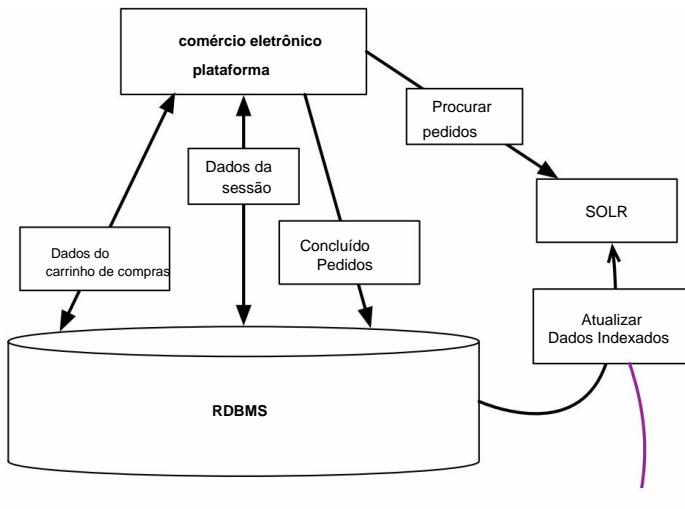
13.4 Expansão para melhor funcionalidade

Muitas vezes, não podemos realmente alterar o armazenamento de dados para um uso específico para algo diferente, devido aos aplicativos legados existentes e sua dependência de

13.4 Expansão para melhor funcionalidade

**Figura 13.5** Usando serviços em vez de falar com bancos de dados

armazenamento de dados existente. Podemos, no entanto, adicionar funcionalidades como cache para melhor desempenho, ou usar mecanismos de indexação como o Solr [Solr] para que a pesquisa pode ser mais eficiente (Figura 13.6). Quando tecnologias como esta são introduzidas, temos que garantir que os dados estejam sincronizados entre o armazenamento de dados para o aplicativo e o cache ou mecanismo de indexação.

**Figura 13.6** Usando armazenamento suplementar para aprimorar o armazenamento legado

Ao fazer isso, precisamos atualizar os dados indexados conforme os dados no banco de dados do aplicativo mudam. O processo de atualização dos dados pode ser em tempo real ou em lote, desde que garantimos que o aplicativo possa lidar com dados obsoletos no índice/mecanismo de busca. O padrão event sourcing (“Event Sourcing,” p. 142) pode ser usado para atualizar o índice.

13.5 Escolhendo a tecnologia certa

Há uma rica escolha de soluções de armazenamento de dados. Inicialmente, o pêndulo havia mudado de bancos de dados especializados para um único banco de dados RDBMS que permite que todos os tipos de modelos de dados sejam armazenados, embora com alguma abstração. A tendência agora está mudando de volta para o uso do armazenamento de dados que suporta a implementação de soluções nativamente.

Se quisermos recomendar produtos aos clientes com base no que está em seus carrinhos de compras e quais outros produtos foram comprados por clientes que compraram esses produtos, isso pode ser implementado em qualquer um dos armazenamentos de dados, persistindo os dados com os atributos corretos para responder às nossas perguntas. O truque é usar a tecnologia certa, para que, quando as perguntas mudarem, elas ainda possam ser feitas com o mesmo armazenamento de dados sem perder os dados existentes ou alterá-los para novos formatos.

Vamos voltar à nossa nova necessidade de recurso. Podemos usar RDBMS para resolver isso usando uma consulta hierárquica e modelando as tabelas adequadamente. Quando precisarmos alterar a travessia, teremos que refatorar o banco de dados, migrar os dados e começar a persistir novos dados. Em vez disso, se tivéssemos usado um armazenamento de dados que rastreia relações entre nós, poderíamos ter apenas programado as novas relações e continuar usando o mesmo armazenamento de dados com alterações mínimas.

13.6 Preocupações empresariais com a persistência poliglota

A introdução de tecnologias de armazenamento de dados NoSQL forçará os DBAs corporativos a pensar sobre como usar o novo armazenamento. A empresa está acostumada a ter ambientes RDBMS uniformes; qualquer que seja o banco de dados que uma empresa comece a usar primeiro, as chances são de que, ao longo dos anos, todos os seus aplicativos serão construídos em torno do mesmo banco de dados. Neste novo mundo de persistência poliglota, os grupos de DBA terão que se tornar mais poli-qualificados — para aprender como algumas dessas tecnologias NoSQL funcionam, como monitorar esses sistemas, fazer backup deles e retirar dados e colocá-los nesses sistemas.

Uma vez que a empresa decide usar qualquer tecnologia NoSQL, surgem questões como licenciamento, suporte, ferramentas, atualizações, drivers, auditoria e segurança. Muitos

As tecnologias NoSQL são de código aberto e têm uma comunidade ativa de apoiadores; além disso, há empresas que fornecem suporte comercial. Não há um ecossistema rico de ferramentas, mas os fornecedores de ferramentas e a comunidade de código aberto estão se atualizando, lançando ferramentas como o MongoDB Monitoring Service [Monitoring], o Datastax Ops Center [OpsCenter] ou o navegador Rekon para Riak [Rekon].

Outra área com a qual as empresas estão preocupadas é a segurança dos dados — a capacidade de criar usuários e atribuir privilégios para ver ou não os dados no nível do banco de dados. A maioria dos bancos de dados NoSQL não tem recursos de segurança muito robustos, mas isso ocorre porque eles são projetados para operar de forma diferente. No RDBMS tradicional, os dados eram fornecidos pelo banco de dados e podíamos acessá-lo usando qualquer ferramenta de consulta. Com os bancos de dados NoSQL, também há ferramentas de consulta, mas a ideia é que o aplicativo possua os dados e os forneça usando serviços. Com essa abordagem, a responsabilidade pela segurança fica com o aplicativo. Dito isso, há tecnologias NoSQL que introduzem recursos de segurança.

As empresas geralmente têm sistemas de data warehouse, BI e sistemas analíticos que podem precisar de dados de fontes de dados poliglotas. As empresas terão que garantir que as ferramentas ETL ou qualquer outro mecanismo que estejam usando para mover dados de sistemas de origem para o data warehouse possam ler dados do armazenamento de dados NoSQL.

Os fornecedores de ferramentas ETL estão lançando a capacidade de se comunicar com bancos de dados NoSQL; por exemplo, o Pentaho [Pentaho] pode se comunicar com o MongoDB e o Cassandra.

Toda empresa executa análises de algum tipo. À medida que o grande volume de dados que precisa ser capturado aumenta, as empresas estão lutando para escalar seus sistemas RDBMS para gravar todos esses dados nos bancos de dados. Um grande número de gravações e a necessidade de escalar para gravações são um ótimo caso de uso para bancos de dados NoSQL que permitem gravar grandes volumes de dados.

13.7 Complexidade de implantação

Uma vez que começamos a trilhar o caminho do uso da persistência poliglota no aplicativo, a **complexidade da implantação** precisa de consideração cuidadosa. O aplicativo agora precisa de todos os bancos de dados em produção ao mesmo tempo. Você precisará ter esses bancos de dados em seus ambientes UAT, QA e Dev. Como a maioria dos produtos NoSQL são de código aberto, há poucas ramificações de custo de licença. Eles também suportam automação de instalação e configuração. Por exemplo, para instalar um banco de dados, tudo o que precisa ser feito é baixar e descompactar o arquivo, o que pode ser automatizado usando os comandos curl e unzip . Esses produtos também têm padrões sensatos e podem ser iniciados com configuração mínima.

13.8 Pontos-chave

- A persistência poliglota consiste em usar diferentes tecnologias de armazenamento de dados para lidar com diferentes necessidades de armazenamento de dados.
- A persistência poliglota pode ser aplicada em uma empresa ou em uma única aplicativo.
- Encapsular o acesso a dados em serviços reduz o impacto das opções de armazenamento de dados em outras partes de um sistema.
- Adicionar mais tecnologias de armazenamento de dados aumenta a complexidade na programação e nas operações, portanto, as vantagens de um bom ajuste de armazenamento de dados precisam ser ponderadas em relação a essa complexidade.

Capítulo 14

Além do NoSQL

O surgimento dos bancos de dados NoSQL fez muito para sacudir e abrir o mundo dos bancos de dados, mas achamos que o tipo de banco de dados NoSQL que discutimos aqui é apenas parte do quadro da persistência poliglota. Então faz sentido gastar algum tempo discutindo soluções que não se encaixam facilmente no balde NoSQL.

14.1 Sistemas de arquivos

Bancos de dados são muito comuns, mas sistemas de arquivos são quase onipresentes. Nas últimas duas décadas, eles têm sido amplamente usados para documentos de produtividade pessoal, mas não para aplicativos empresariais. Eles não anunciam nenhuma estrutura interna, então são mais como armazenamentos de chave-valor com uma chave hierárquica. Eles também fornecem pouco controle sobre a simultaneidade além do simples bloqueio de arquivo — que por si só é semelhante à maneira como o NoSQL fornece bloqueio apenas dentro de um único agregado.

Os sistemas de arquivos têm a vantagem de serem simples e amplamente implementados. Eles lidam bem com entidades muito grandes, como vídeo e áudio. Frequentemente, bancos de dados são usados para indexar ativos de mídia armazenados em arquivos. Arquivos também funcionam muito bem para acesso sequencial, como streaming, o que pode ser útil para dados que são somente anexados.

A atenção recente aos ambientes em cluster viu um aumento de sistemas de arquivos distribuídos.

Tecnologias como o Google File System e o Hadoop [Hadoop] fornecem suporte para replicação de arquivos. Grande parte da discussão sobre map-reduce é sobre a manipulação de arquivos grandes em sistemas de cluster, com ferramentas para divisão automática de arquivos grandes em segmentos a serem processados em vários nós. De fato, um caminho de entrada comum no NoSQL é de organizações que têm usado o Hadoop.

Os sistemas de arquivos funcionam melhor para um número relativamente pequeno de arquivos grandes que podem ser processados em grandes pedaços, de preferência em um estilo de streaming. Grandes números de arquivos pequenos geralmente têm desempenho ruim — é aqui que um armazenamento de dados se torna mais eficiente. Os arquivos também não fornecem suporte para consultas sem ferramentas de indexação adicionais, como Solr [Solr].

14.2 Origem do evento

O sourcing de eventos é uma abordagem para persistência que se concentra em persistir todas as mudanças para um estado persistente, em vez de persistir o estado atual do aplicativo em si. É um padrão arquitetônico que funciona muito bem com a maioria das tecnologias de persistência, incluindo bancos de dados relacionais. Nós o mencionamos aqui porque ele também sustenta algumas das maneiras mais incomuns de pensar sobre persistência.

Considere um exemplo de um sistema que mantém um log da localização de navios (Figura 14.1). Ele tem um registro de navio simples que mantém o nome do navio e sua localização atual. Na maneira usual de pensar, quando ouvimos que o navio *King Roy* chegou em São Francisco, mudamos o valor do campo de localização do *King Roy* para São Francisco. Mais tarde, ouvimos que ele partiu, então mudamos para no mar, mudando novamente quando sabemos que ele chegou em Hong Kong.

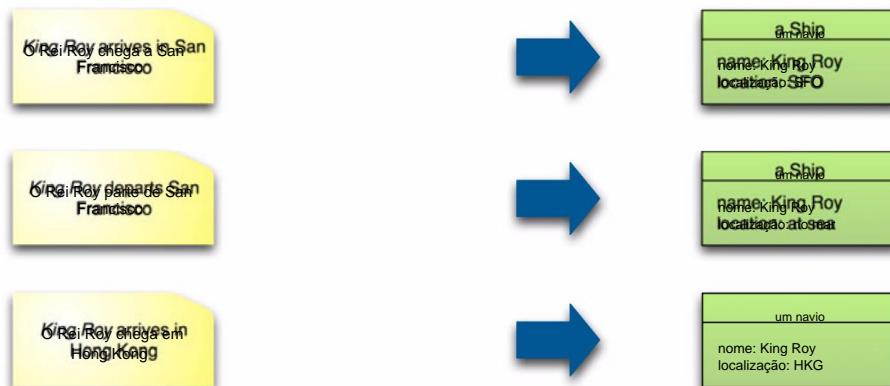


Figura 14.1 Em um sistema típico, o aviso de uma alteração causa uma atualização no estado do aplicativo.

Com um sistema de origem de eventos, o primeiro passo é construir um objeto de evento que capture as informações sobre a mudança (Figura 14.2). Esse objeto de evento é armazenado em um log de eventos durável. Finalmente, processamos o evento para atualizar o estado do aplicativo.

Como consequência, em um sistema de origem de eventos, armazenamos cada evento que causou uma mudança de estado do sistema no log de eventos, e o estado do aplicativo é inteiramente derivável desse log de eventos. A qualquer momento, podemos descartar com segurança o estado do aplicativo e reconstruí-lo a partir do log de eventos.

Em teoria, os logs de eventos são tudo o que você precisa porque você sempre pode recriar o estado do aplicativo sempre que precisar, reproduzindo o log de eventos. Na prática, isso pode ser muito lento. Como resultado, geralmente é melhor fornecer a capacidade de armazenar e recriar o estado do aplicativo em um instantâneo. Um **instantâneo** é projetado para persistir o

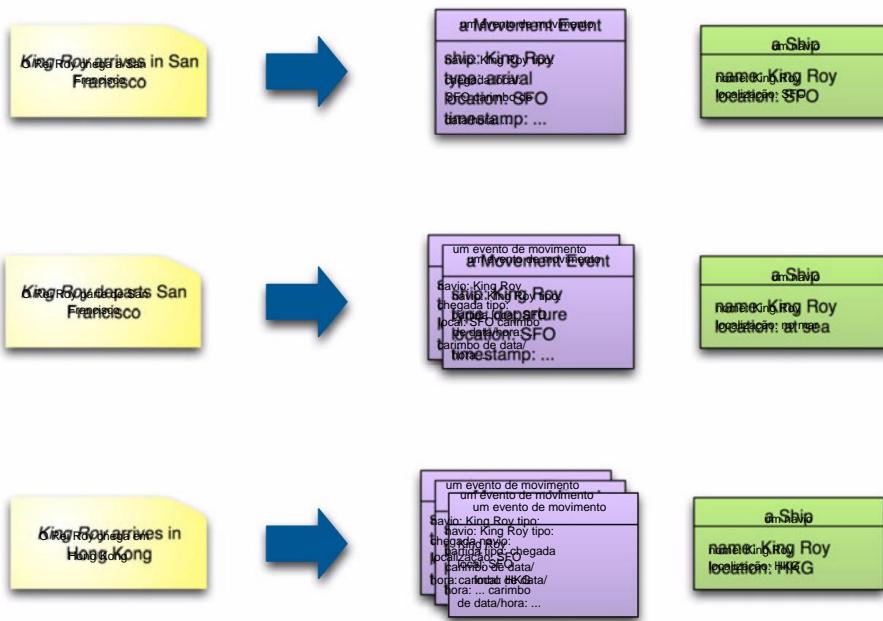


Figura 14.2 Com o fornecimento de eventos, o sistema armazena cada evento, juntamente com o estado do aplicativo derivado.

imagem de memória otimizada para recuperação rápida do estado. É um auxílio de otimização, então nunca deve ter precedência sobre o log de eventos para autoridade sobre os dados.

A frequência com que você tira um snapshot depende das suas necessidades de tempo de atividade. O snapshot não precisa estar completamente atualizado, pois você pode reconstruir a memória carregando o snapshot mais recente e, em seguida, reproduzindo todos os eventos processados desde que o snapshot foi tirado. Um exemplo de abordagem seria tirar um snapshot toda noite; se o sistema cair durante o dia, você recarregaria o snapshot da noite passada seguido pelos eventos de hoje. Se você puder fazer isso rápido o suficiente, tudo ficará bem.

Para obter um registro completo de cada alteração no estado do seu aplicativo, você precisa manter o log de eventos voltando ao início dos tempos para o seu aplicativo. Mas em muitos casos, um registro de longa duração não é necessário, pois você pode dobrar eventos mais antigos em um snapshot e usar o log de eventos somente após a data do snapshot.

Usar o event sourcing tem uma série de vantagens. Você pode transmitir eventos para vários sistemas, cada um dos quais pode construir um estado de aplicativo diferente para diferentes propósitos (Figura 14.3). Para sistemas de leitura intensiva, você pode fornecer vários nós de leitura, com esquemas potencialmente diferentes, enquanto concentra as gravações em um sistema de processamento diferente (uma abordagem amplamente conhecida como CQRS [CQRS]).

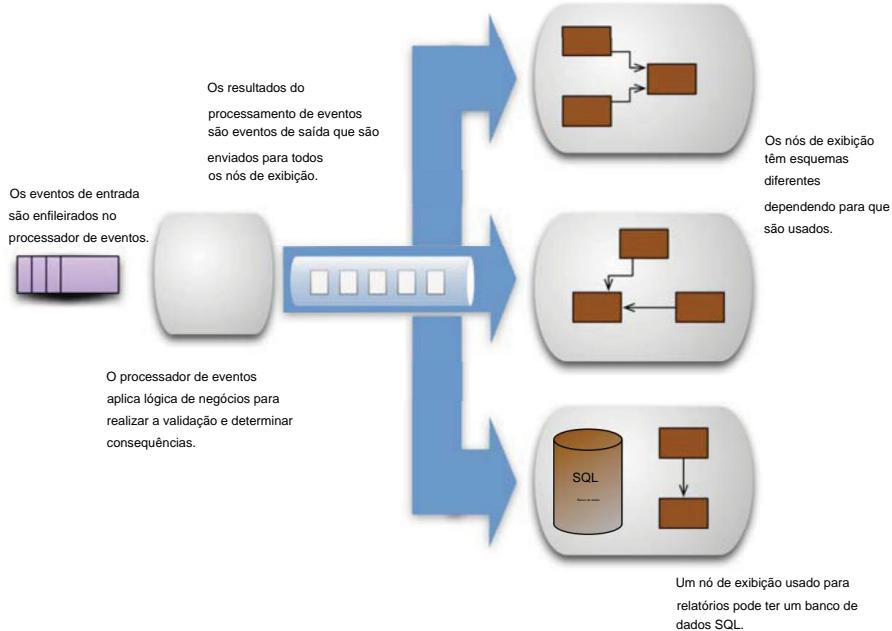


Figura 14.3 Os eventos podem ser transmitidos para vários sistemas de exibição.

O sourcing de eventos também é uma plataforma eficaz para analisar informações históricas, já que você pode replicar qualquer estado passado no log de eventos. Você também pode facilmente investigar cenários alternativos introduzindo eventos hipotéticos em uma análise processador.

O fornecimento de eventos acrescenta alguma complexidade — principalmente, você precisa garantir que todas as mudanças de estado são capturadas e armazenadas como eventos. Algumas arquiteturas e ferramentas podem tornar isso inconveniente. Qualquer colaboração com sistemas externos precisa levar em conta a origem do evento; você precisará ter cuidado com os eventos externos efeitos colaterais ao reproduzir eventos para reconstruir o estado de um aplicativo.

14.3 Imagem de Memória

Uma das consequências do fornecimento de eventos é que o log de eventos se torna o registro persistente definitivo — mas não é necessário que o estado do aplicativo seja persistente. Isso abre a opção de manter o estado do aplicativo na memória usando apenas estruturas de dados na memória. Manter todos os seus dados de trabalho em a memória oferece uma vantagem de desempenho, já que não há E/S de disco para lidar quando um evento é processado. Também simplifica a programação, pois não há necessidade para executar o mapeamento entre estruturas de dados em disco e na memória.

A limitação óbvia aqui é que você deve ser capaz de armazenar todos os dados que você precisa acessar na memória. Esta é uma opção cada vez mais viável — podemos lembrar tamanhos de disco que eram consideravelmente menores do que os tamanhos de memória atuais. Você também precisa garantir que pode se recuperar rápido o suficiente de uma falha do sistema — seja recarregando eventos do log de eventos ou executando um sistema duplicado e cortando.

Você precisará de algum mecanismo explícito para lidar com a simultaneidade. Uma rota é um sistema de memória transacional, como o que vem com a linguagem Clojure. Outra rota é fazer todo o processamento de entrada em um único thread. Projetado cuidadosamente, um processador de eventos single-threaded pode atingir uma taxa de transferência impressionante em baixa latência [Fowler Imax].

Quebrar a separação entre dados na memória e persistentes também afeta como você lida com erros. Uma abordagem comum é atualizar um modelo e reverter quaisquer alterações caso ocorra um erro. Com uma imagem de memória, você normalmente não terá um recurso de reversão automatizado; você tem que escrever o seu próprio (complicado) ou garantir que você faça uma validação completa antes de começar a aplicar quaisquer alterações.

14.4 Controle de versão

Para a maioria dos desenvolvedores de software, sua experiência mais comum de um sistema de origem de eventos é um sistema de controle de versão. O controle de versão permite que muitas pessoas em uma equipe coordenem suas modificações de um sistema interconectado complexo, com a capacidade de explorar estados passados desse sistema e realidades alternativas por meio de ramificação.

Quando pensamos em armazenamento de dados, tendemos a pensar em uma visão de mundo de um único ponto no tempo, o que é muito limitador em comparação à complexidade suportada por um sistema de controle de versão. Portanto, é surpreendente que as ferramentas de armazenamento de dados não tenham emprestado algumas das ideias dos sistemas de controle de versão. Afinal, muitas situações exigem consultas históricas e suporte para múltiplas visões do mundo.

Os sistemas de controle de versão são construídos sobre sistemas de arquivos e, portanto, têm muitas das mesmas limitações para armazenamento de dados de um sistema de arquivos. Eles não são projetados para armazenamento de dados de aplicativos, então são difíceis de usar nesse contexto. No entanto, vale a pena considerá-los para cenários em que seus recursos de linha do tempo são úteis.

14.5 Bancos de dados XML

Na virada do milênio, as pessoas pareciam querer usar XML para tudo, e houve uma onda de interesse em bancos de dados projetados especificamente para

armazenar e consultar documentos XML. Embora essa onda tenha tido tão pouco impacto no domínio relacional quanto as fanfarronices anteriores, os bancos de dados XML ainda estão por aí.

Pensamos em bancos de dados XML como bancos de dados de documentos onde os documentos são armazenados em um modelo de dados compatível com XML, e onde várias tecnologias XML são usadas para manipular o documento. Você pode usar várias formas de definições de esquema XML (DTDs, XML Schema, RelaxNG) para verificar formatos de documentos, executar consultas com XPath e XQuery, e realizar transformações com XSLT.

Os bancos de dados relacionais adotaram o XML e combinaram esses recursos com os relacionais, geralmente incorporando documentos XML como um tipo de coluna e permitindo alguma maneira de combinar linguagens de consulta SQL e XML.

Claro que não há razão para que você não possa usar XML como um mecanismo de estruturação dentro de um armazenamento de chave-valor. XML está menos na moda atualmente do que JSON, mas é igualmente capaz de armazenar agregados complexos, e os recursos de esquema e consulta do XML são maiores do que o que você normalmente consegue para JSON. Usar um banco de dados XML significa que o próprio banco de dados é capaz de tirar vantagem da estrutura XML e não apenas tratar o valor como um blob, mas essa vantagem precisa ser ponderada com as outras características do banco de dados.

14.6 Bancos de dados de objetos

Quando a programação orientada a objetos começou a ganhar popularidade, houve uma onda de interesse em bancos de dados orientados a objetos. O foco aqui era a complexidade do mapeamento de estruturas de dados na memória para tabelas relacionais. A ideia de um banco de dados orientado a objetos é que você evite essa complexidade — o banco de dados gerenciaria automaticamente o armazenamento de estruturas na memória no disco. Você pode pensar nisso como um sistema de memória virtual persistente, permitindo que você programe com persistência, mas sem tomar conhecimento de um banco de dados.

Os bancos de dados de objetos não decolaram. Um dos motivos foi que o benefício da integração próxima com o aplicativo significava que você não conseguia acessar facilmente os dados de outra forma que não fosse com esse aplicativo. Uma mudança de bancos de dados de integração para bancos de dados de aplicativos poderia muito bem tornar os bancos de dados de objetos mais viáveis no futuro.

Uma questão importante com bancos de dados de objetos é como lidar com a migração conforme as estruturas de dados mudam. Aqui, a ligação próxima entre o armazenamento persistente e as estruturas na memória pode se tornar um problema. Alguns bancos de dados de objetos incluem a capacidade de adicionar funções de migração a definições de objetos.

14.7 Pontos-chave

- NoSQL é apenas um conjunto de tecnologias de armazenamento de dados. Como elas aumentam o conforto com persistência poliglota, devemos considerar outras tecnologias de armazenamento de dados, independentemente de terem ou não o rótulo NoSQL.

Capítulo 15

Escolhendo seu banco de dados

Neste ponto do livro, cobrimos muitas das questões gerais que você precisa estar ciente para tomar decisões no novo mundo da persistência poliglota. Agora é hora de falar sobre a escolha de seus bancos de dados para o trabalho de desenvolvimento futuro. Naturalmente, não conhecemos suas circunstâncias particulares, então não podemos lhe dar sua resposta, nem podemos reduzi-la a um simples conjunto de regras a serem seguidas. Além disso, ainda é cedo no uso de produção de sistemas NoSQL, então mesmo o que sabemos é imaturo — em alguns anos, podemos muito bem pensar diferente.

Vemos duas razões amplas para considerar um banco de dados NoSQL: produtividade do programador e desempenho de acesso a dados. Em casos diferentes, essas forças podem se complementar ou se contradizer. Ambas são difíceis de avaliar no início de um projeto, o que é estranho, pois sua escolha de um modelo de armazenamento de dados é difícil de abstrair para permitir que você mude de ideia mais tarde.

15.1 Produtividade do programador

Fale com qualquer desenvolvedor de um aplicativo empresarial, e você sentirá frustração ao trabalhar com bancos de dados relacionais. As informações geralmente são coletadas e exibidas em termos de agregados, mas precisam ser transformadas em relações para persistir. Essa tarefa é mais fácil do que costumava ser; durante a década de 1990, muitos projetos geraram sob o esforço de construir camadas de mapeamento objeto-relacional. Nos anos 2000, vimos frameworks ORM populares como Hibernate, iBATIS e Rails Active Record que reduziram muito desse fardo. Mas isso não fez o problema desaparecer. ORMs são uma abstração com vazamento, sempre há alguns casos que precisam de mais atenção — particularmente para obter um desempenho decente.

Nessa situação, bancos de dados orientados a agregados podem oferecer um negócio tentador. Podemos remover o ORM e persistir agregados naturalmente conforme os usamos. Nós nos deparamos com vários projetos que alegam benefícios palpáveis ao migrar para uma solução orientada a agregados.

Bancos de dados de grafos oferecem uma simplificação diferente. Bancos de dados relacionais não fazem um bom trabalho com dados que têm muitos relacionamentos. Um banco de dados de grafos oferece uma API de armazenamento mais natural para esse tipo de dados e recursos de consulta projetados em torno desses tipos de estruturas.

Todos os tipos de sistemas NoSQL são mais adequados para dados não uniformes. Se você se encontrar lutando com um esquema forte para suportar campos ad-hoc, então os bancos de dados NoSQL sem esquema podem oferecer um alívio considerável.

Essas são as principais razões pelas quais o modelo de programação de bancos de dados NoSQL pode melhorar a produtividade da sua equipe de desenvolvimento. O primeiro passo para avaliar isso para suas circunstâncias é olhar o que seu software precisará fazer. Execute os recursos atuais e veja se e como o uso de dados se encaixa. Ao fazer isso, você pode começar a ver que um modelo de dados específico parece um bom ajuste. Essa proximidade de ajuste sugere que usar esse modelo levará a uma programação mais fácil.

Ao fazer isso, lembre-se de que a persistência poliglota é sobre usar várias soluções de armazenamento de dados. Pode ser que você veja diferentes modelos de armazenamento de dados se encaixarem em diferentes partes dos seus dados. Isso sugeriria usar diferentes bancos de dados para diferentes aspectos dos seus dados. Usar vários bancos de dados é inherentemente mais complexo do que usar um único armazenamento, mas as vantagens de um bom ajuste em cada caso podem ser melhores no geral.

Ao analisar o ajuste do modelo de dados, preste atenção especial aos casos em que há um problema. Você pode ver que a maioria dos seus recursos funcionará bem com um agregado, mas alguns não. Ter alguns recursos que não se ajustam bem ao modelo não é motivo para evitá-lo — as dificuldades do ajuste ruim podem não sobrepujar as vantagens do ajuste bom — mas é útil identificar e destacar esses ajustes ruins casos.

Passar por seus recursos e avaliar suas necessidades de dados deve levá-lo a uma ou mais alternativas de como lidar com suas necessidades de banco de dados. Isso lhe dará um ponto de partida, mas o próximo passo é experimentar as coisas realmente construindo software. Pegue alguns recursos iniciais e construa-os, enquanto presta bastante atenção em quão simples é usar a tecnologia que você está considerando. Nessa situação, pode valer a pena construir os mesmos recursos com alguns bancos de dados diferentes para ver qual funciona melhor. As pessoas geralmente relutam em fazer isso — ninguém gosta de construir software que será descartado. No entanto, esta é uma maneira essencial de julgar o quão eficaz é uma estrutura específica.

Infelizmente, não há como medir adequadamente o quão produtivos são os diferentes designs. Não temos como medir adequadamente a produção. Mesmo se você construir exatamente o mesmo recurso, não é possível comparar verdadeiramente a produtividade porque o conhecimento de como construí-lo uma vez torna mais fácil uma segunda vez, e você não pode construí-los simultaneamente com equipes idênticas. O que você pode fazer é garantir que as pessoas que fizeram o trabalho possam dar uma opinião. A maioria dos desenvolvedores consegue sentir quando são mais produtivos em um ambiente do que em outro. Embora este seja um julgamento subjetivo, e você possa muito bem ter desacordos entre os membros da equipe, esta é a

melhor julgamento que você terá. No final, acreditamos que a equipe que está fazendo o trabalho deve decidir.

Ao testar um banco de dados para julgar a produtividade, é importante também testar alguns dos casos de ajuste ruim que mencionamos anteriormente. Dessa forma, a equipe pode ter uma ideia do caminho feliz e do difícil, para obter uma impressão geral.

Essa abordagem tem suas falhas. Muitas vezes você não consegue ter uma apreciação completa de uma tecnologia sem passar muitos meses usando-a — e executar uma avaliação por tanto tempo raramente é rentável. Mas, como muitas coisas na vida, precisamos fazer a melhor avaliação que pudermos, conhecendo suas falhas, e seguir com isso. O essencial aqui é basear a decisão em tanta programação real quanto possível. Mesmo uma mera semana trabalhando com uma tecnologia pode lhe dizer coisas que você nunca aprenderia com uma centena de apresentações de fornecedores.

15.2 Desempenho de acesso a dados

A preocupação que levou ao crescimento dos bancos de dados NoSQL foi o acesso rápido a muitos dados. Conforme grandes sites surgiram, eles queriam crescer horizontalmente e rodar em grandes clusters. Eles desenvolveram os primeiros bancos de dados NoSQL para ajudá-los a rodar eficientemente em tais arquiteturas. Conforme outros usuários de dados seguem seu exemplo, novamente o foco está em acessar dados rapidamente, frequentemente com grandes volumes envolvidos.

Há muitos fatores que podem determinar o melhor desempenho de um banco de dados do que o padrão relacional em várias circunstâncias. Um banco de dados orientado a agregados pode ser muito rápido para ler ou recuperar agregados em comparação a um banco de dados relacional, onde os dados são distribuídos em muitas tabelas. Fragmentação e replicação mais fáceis em clusters permitem dimensionamento horizontal. Um banco de dados de gráfico pode recuperar dados altamente conectados mais rapidamente do que usar junções relacionais.

Se você estiver investigando bancos de dados NoSQL com base no desempenho, a coisa mais importante que você deve fazer é testar o desempenho deles nos cenários que importam para você. Raciocinar sobre como um banco de dados pode executar pode ajudar você a construir uma lista curta, mas a única maneira de avaliar o desempenho corretamente é construir algo, executá-lo e medi-lo.

Ao construir uma avaliação de desempenho, a coisa mais difícil geralmente é obter um conjunto realista de testes de desempenho. Você não pode construir seu sistema real, então precisa construir um subconjunto representativo. É importante, no entanto, que esse subconjunto seja o mais fiel possível como representante. Não adianta pegar um banco de dados que se destina a atender centenas de usuários simultâneos e avaliar seu desempenho com um único usuário. Você precisará construir cargas e volumes de dados representativos.

Particularmente se você estiver construindo um site público, pode ser difícil construir um testbed de alta carga. Aqui, um bom argumento pode ser feito para usar recursos de computação em nuvem tanto para gerar carga quanto para construir um cluster de teste. A natureza elástica do provisionamento em nuvem é muito útil para trabalhos de avaliação de desempenho de curta duração.

Você não vai conseguir testar todas as maneiras em que seu aplicativo será usado, então você precisa construir um subconjunto representativo. Escolha cenários que sejam os mais comuns, os mais dependentes de desempenho e aqueles que não parecem se encaixar bem no seu modelo de banco de dados. O último pode alertá-lo sobre quaisquer riscos fora dos seus principais casos de uso.

Criar volumes para testar pode ser complicado, especialmente no início de um projeto, quando não está claro quais são os volumes de produção prováveis. Você terá que criar algo para basear seu pensamento, então certifique-se de torná-lo explícito e comunicá-lo a todas as partes interessadas. Tornar isso explícito reduz a chance de que pessoas diferentes tenham ideias variadas sobre o que é uma "carga de leitura pesada". Também permite que você identifique problemas mais facilmente, caso suas descobertas posteriores se desviam de suas suposições originais. Sem tornar suas suposições explícitas, é mais fácil se afastar delas sem perceber que você precisa refazer seu banco de testes à medida que aprende novas informações.

15.3 Mantendo o padrão

Naturalmente, achamos que o NoSQL é uma opção viável em muitas circunstâncias — caso contrário, não teríamos passado vários meses escrevendo este livro. Mas também percebemos que há muitos casos, na verdade a maioria dos casos, em que é melhor ficar com a opção padrão de um banco de dados relacional.

Bancos de dados relacionais são bem conhecidos; você pode facilmente encontrar pessoas com experiência em usá-los. Eles são maduros, então é menos provável que você se depare com as arestas de novas tecnologias. Existem muitas ferramentas que são construídas em tecnologia relacional das quais você pode tirar proveito. Você também não precisa lidar com as questões políticas de fazer uma escolha incomum — escolher uma nova tecnologia sempre introduzirá um risco de problemas caso as coisas se tornem difíceis.

Então, no geral, tendemos a ter uma visão de que, para escolher um banco de dados NoSQL, você precisa mostrar uma vantagem real sobre os bancos de dados relacionais na sua situação. Não há vergonha em fazer as avaliações de programabilidade e desempenho, não encontrar nenhuma vantagem clara e ficar com a opção relacional. Achamos que há muitos casos em que é vantajoso usar bancos de dados NoSQL, mas "muitos" não significa "todos" ou mesmo "a maioria".

15.4 Protegendo suas apostas

Uma das maiores dificuldades que temos em dar conselhos sobre a escolha de uma opção de armazenamento de dados é que não temos muitos dados para nos basear. Enquanto escrevemos isso, estamos vendo apenas os primeiros a adotar discutindo suas experiências com essas tecnologias, então não temos uma imagem clara dos prós e contras reais.

Com a situação tão incerta, há mais argumentos para encapsular sua escolha de banco de dados — mantendo todo o código do banco de dados em uma seção da sua base de código que seja relativamente fácil de substituir caso você decida mudar sua escolha de banco de dados mais tarde. A maneira clássica de fazer isso é por meio de uma camada de armazenamento de dados explícita em seu aplicativo — usando padrões como Data Mapper e Repository [Fowler PoEAA]. Essa camada de encapsulamento tem um custo, particularmente quando você não tem certeza sobre o uso de modelos bem diferentes, como modelos de dados de chave-valor versus gráficos. Pior ainda, ainda não temos experiência com o encapsulamento de camadas de dados entre esses tipos muito diferentes de armazenamentos de dados.

No geral, nosso conselho é encapsular como uma estratégia padrão, mas preste atenção ao custo da camada isolante. Se estiver se tornando um fardo muito grande, por exemplo, tornando mais difícil usar alguns recursos úteis do banco de dados, então é um bom argumento para usar o banco de dados que tem esses recursos. Essas informações podem ser exatamente o que você precisa para fazer uma escolha de banco de dados e, assim, eliminar o encapsulamento.

Este é outro argumento para decompor a camada de banco de dados em serviços que encapsulam o armazenamento de dados (“Uso de serviço sobre uso direto de armazenamento de dados”, p. 136). Além de reduzir o acoplamento entre vários serviços, isso tem a vantagem adicional de facilitar a substituição de um banco de dados caso as coisas não funcionem no futuro. Essa é uma abordagem plausível mesmo se você acabar usando o mesmo banco de dados em todos os lugares — se as coisas derem errado, você pode trocá-lo gradualmente, focando nos serviços mais problemáticos primeiro.

Este conselho de design se aplica tanto quanto se você preferir ficar com uma opção relacional. Ao encapsular segmentos do seu banco de dados em serviços, você pode substituir partes do seu armazenamento de dados por uma tecnologia NoSQL conforme ela amadurece e as vantagens se tornam mais claras.

15.5 Pontos-chave

- Os dois principais motivos para usar a tecnologia NoSQL são:
 - Melhorar a produtividade do programador usando um banco de dados que atenda melhor às necessidades de um aplicativo.
 - Melhorar o desempenho do acesso aos dados por meio de alguma combinação de manipulação de volumes maiores de dados, redução da latência e melhoria do rendimento.
- É essencial testar suas expectativas sobre produtividade e/ou desempenho do programador antes de se comprometer a usar uma tecnologia NoSQL.
- O encapsulamento de serviços oferece suporte a tecnologias de armazenamento de dados em mudança conforme as necessidades e a tecnologia evoluem. Separar partes de aplicativos em serviços também permite que você introduza NoSQL em um aplicativo existente.

- A maioria das aplicações, particularmente as não estratégicas, deve permanecer com a tecnologia relacional — pelo menos até que o ecossistema NoSQL se torne mais maduro.

15.6 Considerações finais

Esperamos que você tenha achado este livro esclarecedor. Quando começamos a escrevê-lo, ficamos frustrados com a falta de algo que nos desse uma visão ampla do mundo NoSQL. Ao escrever este livro, tivemos que fazer essa visão nós mesmos, e achamos que foi uma jornada agradável. Esperamos que sua jornada por este material seja consideravelmente mais rápida, mas não menos agradável.

Neste ponto, você pode estar considerando fazer uso de uma tecnologia NoSQL. Se sim, este livro é apenas um passo inicial na construção de seu entendimento. Nós pedimos que você baixe alguns bancos de dados e trabalhe com eles, pois temos a firme convicção de que você só pode entender uma tecnologia adequadamente trabalhando com ela — encontrando seus pontos fortes e as inevitáveis pegadinhas que nunca chegam à documentação.

Esperamos que a maioria das pessoas, incluindo a maioria dos leitores deste livro, não usem NoSQL por um tempo. É uma tecnologia nova e ainda estamos no começo do processo de entender quando usá-la e como usá-la bem. Mas, como acontece com qualquer coisa no mundo do software, as coisas estão mudando mais rapidamente do que ousamos prever, então fique de olho no que está acontecendo neste campo.

Esperamos que você também encontre outros livros e artigos para ajudá-lo. Acreditamos que o melhor material sobre NoSQL será escrito depois que este livro estiver pronto, então não podemos indicar nenhum lugar em particular enquanto escrevemos isto. Temos uma presença ativa na Web, então para nossos pensamentos mais atualizados sobre o mundo NoSQL, dê uma olhada em www.sadalage.com e <http://martinfowler.com/nosql.html>.

Bibliografia

- [Métodos ágeis] www.agilealliance.org.
- [Dínamo da Amazon] www.allthingsdistributed.com/2007/10/amazons_dynamo.html.
- [Amazon DynamoDB] <http://aws.amazon.com/dynamodb>.
- [Amazon SimpleDB] <http://aws.amazon.com/simpledb>.
- [Ambler e Sadalage] Ambler, Scott e Pramodkumar Sadalage. *Refatoração de bancos de dados: Design de banco de dados evolucionário*. Addison-Wesley. 2006. ISBN 978-0321293534.
- [Berkeley DB] www.oracle.com/us/products/database/berkeley-db.
- [Projetos] <https://github.com/tinkerpop/blueprints/wiki>.
- [Brewer] Brewer, Eric. *Rumo a sistemas distribuídos robustos*. www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.
- [Gaiolas] <http://code.google.com/p/cages>.
- [Cassandra] <http://cassandra.apache.org>.
- [Chang etc.] Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes e Robert E. Gruber. *Bigtable: Um sistema de armazenamento distribuído para dados estruturados*. <http://research.google.com/archive/bigtable-osdi06.pdf>.
- [CouchDB] <http://couchdb.apache.org>.
- [CQL] www.slideshare.net/jericevans/cql-sql-in-cassandra.
- [CQRS] <http://martinfowler.com/bliki/CQRS.html>.
- [C-Store] Stonebraker, Mike, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran e Stan Zdonik. *C-Store: Um SGBD orientado a colunas*. <http://db.csail.mit.edu/projects/cstore/vldb.pdf>.
- [Cifra] <http://docs.neo4j.org/chunked/1.6.1/cypher-query-lang.html>.
- [Daigneau] DAIGNEAU, Robert. *Padrões de design de serviço*. Addison-Wesley. 2012. ISBN 032154420X.
- [DBDeploy] <http://dbdeploy.com>.

- [DBMaintain] www.dbmaintain.org.
- [Dean e Ghemawat] Dean, Jeffrey e Sanjay Ghemawat. *MapReduce: Processamento de dados simplificado em grandes clusters*. http://static.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf.
- [Dijkstra] http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.
- [Evans] Evans, Eric. *Design orientado a domínio*. Addison-Wesley. 2004. ISBN 0321125215.
- [FlockDB] <https://github.com/twitter/flockdb>.
- [Fowler DSL] Fowler, Martin. *Linguagens específicas de domínio*. Addison-Wesley. 2010. ISBN 0321712943.
- [Fowler lmax] Fowler, Martin. *A arquitetura LMAX*. <http://martinfowler.com/articles/lmax.html>.
- [Fowler PoEAA] Fowler, Martin. *Padrões de Arquitetura de Aplicativos Empresariais*. Addison-Wesley. 2003. ISBN 0321127420.
- [Fowler UML] Fowler, Martin. *UML Destilado*. Addison-Wesley. 2003. ISBN 0321193687.
- [Gremlin] <https://github.com/tinkerpop/gremlin/wiki>.
- [Hadoop] <http://hadoop.apache.org/mapreduce>.
- [HamsterDB] <http://hamsterdb.com>.
- [Hbase] <http://hbase.apache.org>.
- [Hector] <https://github.com/rantav/hector>.
- [Colmeia] <http://hive.apache.org>.
- [Hohpe e Woolf] Hohpe, Gregor e Bobby Woolf. *Padrões de integração empresarial*. Português Addison-Wesley. 2003. ISBN 0321200683.
- [HTTP] Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach e T. Berners-Lee. *Protocolo de Transferência de Hipertexto—HTTP/1.1*. www.w3.org/Protocols/rfc2616/rfc2616.html.
- [Hipertable] <http://hypertable.org>.
- [Gráfico infinito] www.infinitegraph.com.
- [JSON] <http://json.org>.
- [LevelDB] <http://code.google.com/p/leveldb>.
- [Liquibase] www.liquibase.org.
- [Lucene] <http://lucene.apache.org>.
- [Lynch e Gilbert] Lynch, Nancy e Seth Gilbert. *Conjectura de Brewer e a viabilidade de serviços web consistentes, disponíveis e tolerantes a partções*. <http://lpd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf>.
- [Memcached] <http://memcached.org>.
- [MongoDB] www.mongodb.org.
- [Monitoramento] www.mongodb.org/display/DOCS/MongoDB+Monitoring+Service.
- [Migrador MyBatis] <http://mybatis.org>.

- [Neo4J] <http://neo4j.org>.
- [Debriefing NoSQL] <http://blog.oskarsson.nu/post/22996140866/nosql-debrief>.
- [Encontro NoSQL] <http://nosql.eventbrite.com>.
- [Instalação de armazenamento de notas] http://en.wikipedia.org/wiki/IBM_Lotus_Domino.
- [OpsCenter] www.datastax.com/products/opscenter.
- [OrientDB] www.orientdb.org.
- [Oskarsson] *Correspondência privada*.
- [Pentaho] www.pentaho.com.
- [Porco] <http://pig.apache.org>.
- [Pritchett] www.infoq.com/interviews/dan-pritchett-ebay-architecture.
- [Projeto Voldemort] <http://project-voldemort.com>.
- [RavenDB] <http://ravendb.net>.
- [Redis] <http://redis.io>.
- [Rekon] <https://github.com/basho/rekon>.
- [Riak] <http://wiki.basho.com/Riak.html>.
- [Solr] <http://lucene.apache.org/solr>.
- [Strozzi NoSQL] www.strozzi.it/cgi-bin/CSA/tw7/l/en_US/NoSQL.
- [Tanenbaum e Van Steen] Tanenbaum, Andrew e Maarten Van Steen. *Distribuído Sistemas*. Prentice-Hall. 2007. ISBN 0132392275.
- [Terrastore] <http://code.google.com/p/terrastore>.
- [Vogels] Vogels, Werner. *Eventualmente Consistente—Revisitado*. www.allthingsdistributed.com/2008/12/eventually_consistent.html.
- [Escalonamento Webber Neo4J] <http://jim.webber.name/2011/03/22/ef4748c3-6459-40b6-bcfa-818960150e0f.aspx>.
- [ZooKeeper] <http://zookeeper.apache.org>.

Esta página foi deixada em branco intencionalmente

Índice

- Transações ACID** (Atomic, Consistent, Isolated e Durable), 19 em bancos de dados de famílias de colunas, 109 em bancos de dados de gráficos, 28, 50, 114–115 em bancos de dados relacionais, 10, 26 vs.
- BASE**, 56 banners de anúncios, 108–109 bancos de dados orientados a agregados, 14, 147 atualizações atômicas em, 50, 61 desvantagens de, 30 sem transações ACID em, 50 desempenho de, 149 vs. bancos de dados gráficos, 28 agregados, 14–23 alterando a estrutura de, 98, 132 modelagem, 31 análise em tempo real com, 33 atualizando, 26 métodos ágeis, 123
- Amazon, 9 *Veja também*
- DynamoDB, SimpleDB análise contando visitantes do site para, 108 de informações históricas, 144 em tempo real, 33, 98 Linguagem Apache Pig, 76 Biblioteca Apache ZooKeeper, 104, 115
- bancos de dados de aplicativos, 7, 146 atualizando visualizações materializadas em, 31
- arcos (bancos de dados de gráficos). *Veja* operações atômicas de documentos cruzados de arestas, 98 rebalanceamento atômico, 58 transações atômicas, 92, 104
- atualizações atômicas, 50, 61 failovers automatizados, 94 fusões automatizadas, 48 reversões automatizadas, 145 fragmentação automática, 39 disponibilidade, 53 em bancos de dados de família de colunas, 104–105 em bancos 10–23 dos documentos, 93 em bancos de dados de gráficos, 115 vs. consistência, 54 *Veja também* médias do teorema CAP, calculando, 72
- B**
- compatibilidade com versões anteriores, 126, 131 BASE (Basicamente disponível, Estado suave, Consistência eventual), 56
- Berkeley DB, 81
- BigTable DB, 9, 21–22 índices mapeados em bits, 106 blogs, 108 Gráfico de propriedades do Blueprints, 115 Brewer, Eric, 53 Conjectura de Brewer. *Veja* baldes de teoremas CAP (Riak), 82 valores padrão para consistência para, 84 domínio, 83 armazenando todos os dados juntos em, 82 transações comerciais, 61
- Desempenho de cache C de, 39, 137 dados obsoletos em, 50
- Biblioteca Cages, 104

- Teorema CAP (Consistência, Disponibilidade e Tolerância de Partição), 53–56 para bancos de dados de documentos, 93 para Riak, 86
- operações CAS (comparar e definir), 62 Cassandra DB, 10, 21–22, 99–109 disponibilidade em, 104–105 famílias de colunas em: comandos para, 105–106 padrão, 101 super, 101–102 colunas em, 100 expirando, 108–109 indexação, 106–107 leitura, 107 super, 101 compactação em, 103 consistência em, 103–104 ferramentas ETL para, 139 transferência sugerida em, 104 espaços de chave em, 102–104 memtables em, 103 consultas em, 105–107 reparos em, 103–104 fator de replicação em, 103 dimensionamento em, 107 SSTables em, 103 carimbos de data/hora em, 100 transações em, 104 linhas largas/finas em, 23 clientes, processamento em, 67 linguagem Clojure, 145 computação em nuvem, 149 aglomeração, 39 clusters, 8–10, 67–72, 76, 149 em sistemas de arquivos, 8 em Riak, 87 resiliência de, 8 bancos de dados de família de colunas, 21–23, 99–109 transações ACID em, 109 colunas para visualizações materializadas em, 31 combinando replicação e fragmentação ponto a ponto em, 43–44 consistência em, 103–104 modelagem para, 34 desempenho em, 103 ausência de esquema de, 28 vs. bancos de dados de valor-chave, 21 largo/fino linhas em, 23 redutores combináveis, 70–71 compactação (Cassandra), 103 compatibilidade, para trás, 126, 131 simultaneidade, 145 em sistemas de arquivos, 141 em bancos de dados relacionais, 4 offline, 62 atualizações condicionais, 48, 62–63 conflitos chave, 82 leitura-escrita, 49–50 resolução, 64 escrita-escrita, 47–48, 64 consistência, 47–59 eventual, 50, 84 em bancos de dados de família de colunas, 103–104 em bancos de dados de gráfico, 114 em replicação mestre-escravo, 52 em MongoDB, 91 lógico, 50 otimista/pessimista, 48 leitura, 49–52, 56 leitura-suas-escritas, 52 relaxante, 52–56 replicação, 50 sessão, 52, 63 negociação, 57 atualização, 47, 56, 61 vs. disponibilidade, 54 escrever, 92 *Veja também o teorema CAP* hashes de conteúdo, 62–63 sistemas de gerenciamento de conteúdo, 98, 108 CouchDB, 10, 91 atualizações condicionais em, 63 conjuntos de réplicas em, 94 contadores, para selos de versão, 62–63 CQL (Linguagem de Consulta Cassandra), 10, 106 CQRS (Comando Consulta Responsabilidade Segregação), 143 operações entre documentos, 98 Loja de conveniência DB, 21 Linguagem cifrada, 115–119
- ## E
- Padrão de Mapeador de Dados e Repositório, 151 modelos de dados, 13, 25 orientados a agregados, 14–23, 30 documentos, 20 chave-valor, 20 relacionais, 13–14

- redundância de dados, 94
 bancos de
 dados escolhendo, 7, 147–152
 implantando, 139
 encapsulando em camada explícita, 151
 NoSQL, definição de, 10–11 integração
 compartilhada de, 4, 6
 Centro de operações Datastax, 139
 Estrutura DBDeploy, 125
 Ferramenta DBMaintain, 126
 deadlocks, 48
 acessos de demonstração, 108
 Padrão de rede de dependência, 77 complexidade
 de implantação, 139
 Algoritmo de Dijkstra, 118 recuperação
 de desastres, 94 sistemas de
 arquivos distribuídos, 76, 141 sistemas de
 controle de versão distribuídos, 48 carimbos de versão,
 64 modelos de distribuição, 37–
 43
Veja também replicações, fragmentação, bancos de
 dados de documentos
 de abordagem de servidor único, 20, 23, 89–98
 disponibilidade em, 93
 incorporação de documentos filhos em, 90 índices
 em, 25 replicação
 mestre-escravo em, 93 desempenho em, 91
 consultas em, 25, 94–95
 conjuntos de réplicas em, 94
 dimensionamento em, 95
 ausência de
 esquema de, 28, 98
 Suporte XML em, 146 buckets
 de domínio (Riak), 83
 Design orientado a domínio, 14
 DTDs (Definições de Tipo de Documento), 146 durabilidade,
 56–57
 DynamoDB, 9, 81, 100 carrinhos
 de compras em, 55
 Dynomite DB, 10
- E**
- primeiros protótipos, 109
 comércio eletrônico
 modelagem de dados para, 14
 esquemas flexíveis para, 98
 persistência poliglota de, 133–138 carrinhos de
 compras em, 55, 85, 87 arestas (bancos
 de dados de gráficos), 26, 111 regras de
 elegibilidade, 26
- empresas
 suporte comercial de NoSQL para, 138–139
- simultaneidade em, 4
 BD como armazenamento de apoio
 para, 4 eventos de registro
 em, 97 integração em,
 4 persistência poliglota em, 138–139 segurança
 de dados em, 139
 tratamento de erros, 4, 145
 etags, 62
 Ferramentas ETL, 139
 Evans, Eric, 10 registro
 de eventos, 97, 107–108 fornecimento
 de eventos, 138, 142, 144 consistência
 eventual, 50 em Riak, 84 uso
 expirado, 108–
 109
- F**
- failovers, automatizados, 94
 sistemas de arquivos,
 141 como armazenamento de apoio para
 RDBMS, 3 com
 reconhecimento de cluster, 8
 simultaneidade em, 141
 distribuído, 76, 141
 desempenho de, 141 consultas em, 141
 FlockDB, 113
 modelo de dados de, 27
 distribuição de nós em, 115
- G**
- Gilbert, Seth, 53 Google,
 9 Google
 BigTable. *Veja* BigTable Google File System,
 141 bancos de dados de gráficos,
 26–28, 111–121, 148
 Transações ACID em, 28, 50, 114–115 agregação–
 ignorância de, 19 disponibilidade em,
 115 consistência em, 114
 criação, 113 arestas (arcos)
 em, 26, 111
 mantido inteiramente na memória,
 119 replicação mestre-escravo em, 115
 migrações em, 131 modelagem para, 35 nós
 em, 26, 111–117
 desempenho de, 149

bancos de dados de gráficos (*continuação*)
 propriedades em, 111
 consultas em, 115–119
 relacionamentos em, 111–121
 dimensionamento
 em, 119 ausência de esquema
 de, 28 configuração de servidor único de, 38
 travessia, 111–117 vs.
 bancos de dados agregados, 28 vs.
 bancos de dados relacionais, 27, 112
 encapsulamento em serviço, 136
Linguagem Gremlin, 115 GUID
 (Identificador Global Único), 62

O

Projeto Hadoop, 67, 76, 141
HamsterDB, 81 tabelas
 hash, 62–63, 81
Banco de dados HBase, 10, 21–22, 99–100
Cliente Hector, 105
 Estrutura do Hibernate, 5, 147 handoff
 sugerido, 104 Hive DB, 76
 hot backup, 40,
 42 reserva de hotel, 4, 55

HTTP (Hypertext Transfer Protocol), 7 interfaces baseadas em, 85 atualizando com, 62

Hipertabela DB, 10, 99–100

I

iBATIS, 5, 147
 incompatibilidade de impedância, 5, 12
 inconsistência em
 carrinhos de compras, 55 de
 leituras, 49 de
 atualizações, 56
 janela de, 50–51, 56 índices
 mapeados
 em bits, 106 em bancos
 de dados de documentos, 25 dados
 obsoletos em, 138
 atualização, 138
Infinite Graph DB, 113 modelos
 de dados, 27 distribuição
 de nós, 114–115 picos de tecnologia
 iniciais, 109 bancos de dados
 de integração, 6, 11 interoperabilidade,

7

Eu

JSON (Notação de Objetos JavaScript), 7, 94–95, 146
Chaves K (bancos de dados de chave-valor), 84
 valor) compostas,
 74 conflitos de, 82
 projetando, 85
 expirando, 85
 agrupando em partições, 70 espaços de chave (Cassandra), 102–104 bancos de dados de chave-valor, 20, 23, 81–88 consistência de, 83–84 modelagem para, 31–33
 nenhuma operação de chave múltipla em, 88 ausência de esquema de, 28 fragmentação em, 86 estrutura de valores em, 86 transações em, 84, 88 vs. bancos de dados de família de colunas, 21 suporte XML em, 146

eu

Ferramenta Liquibase, 126
 serviços baseados em localização, 120

bloqueios

inativos, 48
 offline, 52 atualizações perdidas, 47
Banco de dados Lotus, 91
Biblioteca Lucene, 85, 88, 116
Lynch, Nancy, 53

M

Estrutura MapReduce, 67 padrão map-reduce, 67–77 cálculos com, 72 incremental, 31, 76–77 mapas em, 68 visualizações materializadas em, 76 partições em, 70 reutilização de saídas intermediárias em, 76 estágios para, 73–76 replicação mestre-escravo, 40–42 nomeação de mestres em, 41, 57 combinação com fragmentação, 43 consistência de, 52 em bancos de dados de documentos, 93

- em bancos de dados de gráficos, 115 carimbos de versão, 63
- visualizações materializadas, 30 em map-reduce, 76 atualizações, 31
- Memcached DB, 81, 87 imagens de memória, 144–145 memtables (Cassandra), 103 fusões, automatizadas, 48 migrações, 123–132 durante o desenvolvimento, 124, 126
 - em bancos de dados de gráficos, 131 em projetos legados, 126–128 em bancos de dados orientados a objetos, 146 em bancos de dados sem esquemas, 128–132 incrementais, 130 fase de transição de, 126–128 aplicativos móveis, 131
- MongoDB, 10, 91–97 coleções
 - em, 91 consistência em, 91 bancos de dados em, 91 Ferramentas ETL para, 139 consultas em, 94–95 conjuntos de réplicas em, 91, 93, 96 migrações de esquema em, 128–131 fragmentação em, 96 parâmetro slaveOk em, 91–92, 96 terminologia em, 89 Parâmetro WriteConcern em, 92 Serviço de monitoramento MongoDB, 139 Ferramenta MyBatis Migrator, 126 Banco de dados MySQL, 53, 119
- Não**
 - Banco de dados Neo4J, 113–118
 - Transações ACID em, 114–115 disponibilidade em, 115 criação de gráficos em, 113 modelo de dados de, 27 escravos replicados em, 115 encapsulamento de serviço em, 136 nós (bancos de dados de gráficos), 26, 111 armazenamento distribuído para, 114 localização de caminhos entre, 117 propriedades de indexação de, 115–116
 - dados não uniformes, 10, 28, 30
 - Bancos de dados NoSQL
 - vantagens de, 12 definição de, 10–11
 - falta de suporte para transações em, 10, 61 execução de clusters, 10 ausência de esquema de, 10
- O**
 - bancos de dados orientados a objetos, 5, 146 migrações em, 146 vs. bancos de dados relacionais, 6 simultaneidade offline, 62 bloqueios offline, 52 bloqueio offline otimista, 62 log de refazer do Oracle DB, 104 terminologia em, 81, 89 Servidor Oracle RAC, 8 OrientDB, 91, 113 Estruturas ORM (Mapeamento Objeto-Relacional), 5–6, 147 Oskarsson, Johan, 9
- Tolerância de partição P, 53–54
 - Veja também o teorema CAP*
- particionamento, 69–70
 - replicação ponto a ponto, 42–43 durabilidade de, 58 inconsistência de, 43 carimbos de versão em, 63–64
- Ferramenta Pentaho, 139 desempenho
 - e fragmentação, 39 e transações, 53 protocolos binários para, 7 cache para, 39, 137 acesso a dados, 149–150 em bancos de dados orientados a agregados, 149 em bancos de dados de famílias de colunas, 103 em bancos de dados de documentos, 91 em bancos de dados de gráficos, 149 capacidade de resposta de, 48 testes para, 149 abordagem de pipes e filtros, 73 persistência poliglota, 11, 133–139, 148 e complexidade de implantação, 139 em empresas, 138–139 programação poliglota, 133–134 processamento, em clientes/servidores, 67 produtividade do programador, 147–149 ordens de compra, 25

Q

consultas

- contra estrutura agregada variável, 98 por dados, 88, 94 por chave, 84–86
- para arquivos, 141
- em bancos de dados de família de colunas, 105–107 em bancos de dados de documentos, 25, 94–95 em bancos de dados de gráficos, 115–119 pré-computados e armazenados em cache, 31 por meio de visualizações, 94 quóruns, 57, 59 lidos, 58 gravados, 58, 84

Framework R Rails Active Record, 147

RavenDB, 91

- operações atômicas entre documentos, 98 conjuntos de réplicas, 94 transações, 92 RDBMS.

Veja bancos de dados relacionais lê

consistência de, 49–52, 56, 58

- dimensionamento horizontal para, 94, 96 inconsistente, 49 múltiplos nós para, 143 desempenho de, 52 quóruns de, 58 reparos de, 103 resiliência de, 40–41 separando de gravações, 41 obsoleto,

56 conflitos de leitura-gravação,

49–50 consistência de leitura-suas-

gravações, 52 Real Time

Analytics, 33 Real

Time BI, 33 rebalanceamento,

atômico, 58 mecanismos de recomendação, 26, 35, 121, 138 Redis DB,

81–83 redo log,

104 funções de redução,

- 69 combináveis, 70–71

regiões. *Veja* padrão map-reduce, partições

em

Navegador Rekon para Riak, 139

bancos de dados relacionais (RDBMS), 13, 17

- vantagens de, 3–5, 7–8, 150

agregação-ignorância de, 19

armazenamento de

apoio em, 3 agrupados, 8

colunas em, 13, 90

simultaneidade em, 4

definição de esquemas para,

28 incompatibilidade de impedância

em, 5, 12 custos de

licenciamento de, 8

memória principal em, 3 modificação de vários

registros de uma

vez em, 26 partições

em, 96 persistência em, 3

relações (tabelas) em, 5, 13 esquemas

para, 29–30,

123–128

segurança em, 7 fragmentação em, 8

simplicidade de relacionamentos

em, 112 forte consistência

de, 47 terminologia em, 81, 89

transações em, 4, 26, 92 tuplas

(linhas) em, 5,

13–14 visualizações em, 30 vs.

bancos de dados de gráficos, 27, 112

vs. bancos de dados

orientados a objetos, 6 suporte

XML em, 146

relacionamentos, 25, 111–121

pendurado, 114

direção de, 113, 116, 118

em RDBMS, 112

propriedades de,

113–115 atravessando, 111–

117 RelaxNG, 146

conjuntos de réplicas, 91, 93, 96 fator de replicação, 58

em bancos de

dados de família de colunas, 103

em Riak, 84 replicações,

37 combinando com

fragmentação, 43

consistência de, 42, 50

durabilidade de, 57 em clusters,

149 desempenho de, 39 carimbos de

versão em, 63–64 *Veja**também*

replicação mestre-escravo,

replicação ponto a

ponto resiliência e

fragmentação, 39

leitura, 40–41 capacidade

de resposta, 48 Riak DB, 81–83

clusters em, 87 controlando CAP em, 86 consistência eventual em, 84 int

- link-walking em, 25
 recuperação parcial em, 25
 fator de replicação em, 84
 encapsulamento de serviço em, 136 terminologia em, 81
 transações em, 84
 tolerância de gravação de, 84
Riak Search, 85, 88 modelo
 de domínio rico, 113 reversões, automatizado, 145 roteamento, 120 linhas (RDBMS).
Veja tuplas
- Código de andaime S**, 126
 escalas, 95
 horizontais, 149 para
 leituras, 94, 96 para
 gravações, 96 em
 bancos de dados de famílias de colunas, 107
 em bancos de dados de documentos, 95 em bancos de dados de gráficos, 119 verticais, 8
Padrão Scatter-Gather, 67 bancos de dados sem esquema, 28–30, 148 esquema implícito de, 29 alterações de esquema em, 128–132 esquemas
- compatibilidade com versões anteriores de, 126, 131
 alteração, 128–132
 durante o desenvolvimento, 124, 126
 implícito, 29
 migrações de, 123–132
 mecanismos de busca, 138 segurança, 139 servidores
 manutenção de, 94
 processamento em, 67
 arquitetura orientada a serviços, 7 serviços, 136 e segurança, 139 decompondo a camada de banco de dados em, 151 desacoplamento entre bancos de dados e, 7 sobre HTTP, 7
- sessões
 afinidade, 52
 consistência de, 52, 63 expirar
 chaves para, 85
 gerenciamento de, 133
 pegajoso, 52
 armazenando, 57, 87
 fragmentação, 37–38, 40, 149 e
 desempenho, 39 e resiliência, 39 automático, 39 por
 localização
 do cliente, 97 combinando com replicação, 43 em bancos de dados de chave-valor, 86 em MongoDB, 96 em bancos de dados relacionais, 8 integração de banco de dados compartilhado, 4, 6 carrinhos de compras expiram
 chaves para, 85
 inconsistência em, 55
 persistência de, 133
 armazenando,
 embaralhando, 70
SimpleDB, 99
 janela de inconsistência de, 50
 abordagem de servidor único, 37–38
 consistência de, 53
 nenhuma tolerância de partição em, 54 transações em, 53
 carimbos de versão em, 63
 processadores de eventos de thread único, 145
 instantâneos, 142–143
 redes sociais, 26, 120
 relacionamentos entre nós em, 117
Motor de indexação Solr, 88, 137, 141 situação de cérebro dividido, 53
SQL (Linguagem de Consulta Estruturada), 5
SSTables (Cassandra), 103 dados obsoletos
 em cache, 50
 em índices/mecanismos de busca, 138
 leituras, 56
 famílias de colunas padrão (Cassandra), 101 sessões persistentes, 52 modelos de armazenamento, 13 Strozzi, Carlo, 9 famílias de supercolunas (Cassandra), 101–102 supercolunas (Cassandra), 101 transações de sistema, 61
- Tabelas T. *Veja bancos de dados relacionais, relações em dados telemétricos de dispositivos físicos, 57 Terrastore DB, 91, 94 timestamps noção consistente de tempo para, 64 em bancos de dados de família de colunas, 100 da última atualização, 63*

sistemas de memória transacional, 145

transações, 50

- ACID, 10, 19, 26, 28, 50, 56, 109, 114–115

- em várias

- operações, 92 e desempenho, 53

- atômico, 92, 104 negócios,

- 61 em bancos de

- dados de

- gráficos, 28, 114–115 em bancos de

- dados de chave-valor, 84, 88 em

- RDBMS, 4, 26, 92 em

- sistemas de servidor único, 53

- falta de suporte em NoSQL para, 10, 61

- multioperação, 88 aberto

- durante a interação do usuário, 52

- revertendo, 4

- sistema, 61

estruturas de árvore, 117

gatilhos, 126

TTL (Tempo de Vida), 108–109 tuplas

(RDBMS), 5, 13–14

U

atualiza

- atômico, 50, 61

- condicional, 48, 62–63

- consistência de, 47, 56, 61

- perdido,

- 47 mesclando,

- 48 carimbos de data/hora

- de, 63–64 comentários

- do usuário, 98 preferências

- do usuário, 87 perfis do

- usuário, 87, 98 registros

- do usuário, 98 sessões do usuário, 57

Relógio vetorial V

64 sistemas de controle de versão, 126,

- 145 distribuídos, 48, 64

carimbos de versão, 52, 61–64

vetor de versão, 64

visualizações,

126 colunas virtuais, 126

Voldemort DB, 10, 82

W

serviços web, 7

sites

- distribuindo páginas para, 39

- em grandes clusters, 149

- publicando, 98

- contadores de visitantes para,

108 processadores de

texto, 3 tolerância de

gravação,

- 84 gravações,

- 64 atômicos, 104

- conflitos de, 47–48

- consistência de, 92

- dimensionamento

- horizontal para, 96

- desempenho de, 91 quóruns de,

- 58 separando de leituras, 41 serializando, 47

X

XML (Linguagem de Marcação Extensível), 7, 146

Bancos de dados XML, 145–146

Linguagem de esquema XML, 146

Linguagem XPath, 146

Linguagem XQuery, 146

XSLT (Linguagem de Folha de Estilo Extensível)

- Transformações), 146

Z

ZooKeeper. Veja Apache ZooKeeper

Esta página foi deixada em branco intencionalmente



REGISTER



ESTE PRODUTO

informit.com/register

Registre os produtos Addison-Wesley, Exam Cram, Prentice Hall, Que e Sams que você possui para desbloquear grandes benefícios.

Para iniciar o processo de registro, basta acessar **informit.com/register** para entrar ou criar uma conta. Você será solicitado a inserir o ISBN de 10 ou 13 dígitos que aparece na contracapa do seu produto.

Registrar seus produtos pode desbloquear os seguintes benefícios:

- Acesso a conteúdo suplementar, incluindo capítulos bônus, código-fonte ou arquivos de projeto.
- Um cupom para ser usado em sua próxima compra.

Os benefícios do registro variam de acordo com o produto. Os benefícios serão listados na página da sua conta em Produtos registrados.

Sobre o InformIT — A FONTE DE APRENDIZAGEM DE TECNOLOGIA CONFIÁVEL

A INFORMIT É A CASA DAS PRINCIPAIS IMPRESSÕES DE PUBLICAÇÃO DE TECNOLOGIA Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que e Sams. Aqui você terá acesso a conteúdo e recursos de qualidade e confiáveis dos autores, criadores, inovadores e líderes de tecnologia. Quer esteja procurando um livro sobre uma nova tecnologia, um artigo útil, boletins informativos oportunos ou acesso à biblioteca digital Safari Books Online, a InformIT tem uma solução para você.

informIT.com

A FONTE DE APRENDIZAGEM DE TECNOLOGIA CONFIÁVEL

Addison-Wesley | Cisco Press | Exame Cram
IBM Press | Que | Prentice Hall | Sams

LIVROS SAFARI ONLINE

informIT.com A FONTE DE APRENDIZAGEM DE TECNOLOGIA CONFIÁVEL



InformIT é uma marca da Pearson e a presença online das principais editoras de tecnologia do mundo. É sua fonte de conteúdo e conhecimento confiáveis e qualificados, fornecendo acesso às principais marcas, autores e colaboradores da comunidade de tecnologia.

▲ Addison-Wesley

Cisco Press

EXAM/**CRAM**

IBM
Press.

QUE[®]

PRENTICE HALL

SAMS

Safari
Books Online

Aprenda TI na InformIT

Procurando um livro, eBook ou vídeo de treinamento sobre uma nova tecnologia? Procurando informações e tutoriais oportunos e relevantes? Procurando opiniões, conselhos e dicas de especialistas? **A InformIT tem a solução.**

- Saiba mais sobre novos lançamentos e promoções especiais assinando uma grande variedade de boletins informativos.
Visite informit.com/newsletters.
- Acesse podcasts GRATUITOS de especialistas em informit.com/podcasts.
- Leia os últimos artigos do autor e capítulos de amostra em informit.com/articles.
- Acesse milhares de livros e vídeos no Safari Books
Biblioteca digital on-line em safari.informit.com.
- Obtenha dicas de blogs de especialistas em informit.com/blogs.

Visite informit.com/learn para descobrir todas as maneiras de acessar o conteúdo de tecnologia mais recente.

Você faz parte do pessoal da TI ?

Conecte-se com autores e editores da Pearson via RSS feeds, Facebook, Twitter, YouTube e muito mais! Visite informit.com/socialconnect.

